Noodle: A Language for Declarative Querying in an Object-Oriented Database

Inderpal Singh Mumick
AT&T Bell Laboratories
mumick@research.att.com

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Abstract

We present a language Noodle in which to specify declarative queries in an object-oriented database system. The language models object-identity, classes, relations, views, inheritance, complex objects, and methods, in addition to logical rules, and permits powerful schema querying. Noodle is intended to be used as a query language in an object-oriented database system, with the benefits of declarativeness, namely increased efficiency and reduced programming time.

Our work bridges the gap between relational, deductive and object-oriented databases. Noodle is being implemented in the SWORD database system: a declarative object-oriented database being built at AT&T Bell Laboratories.

1 Introduction

Motivation

The advance to network database systems from hierarchical systems provided additional data modeling abilities. The advance to relational systems provided a declarative querying capability, and moved optimization of access from the user to the system. The attempted move from relational databases to deductive databases was motivated by the goal of providing a more expressive, potentially complete query language and additional data structuring abilities. Object-oriented databases developed in parallel, offering significantly increased data modeling capabilities, but *moved back* to a procedural query language, and shifted the access optimization problem back to the user [26].

*This research was supported in part by NSF grant IRI-9209029, and by a grant from the AT&T Foundation.

Writing queries in a declarative query language is easier and faster than writing queries in a procedural query language. When using a declarative query language, the amount of code and the time to write the code is significantly reduced. The user does not have to optimize the code for access to the database. As the database characteristics (data organization, indices, and schema) change, the system can re-optimize the query with little or no code rewrite. In a procedural language, the code has to be manually optimized in response to such changes. Applications such as marketing databases, scientific databases, data mining, chemical structure analysis, persistent queries, and adhoc queries need a declarative language support. Consequently, the lack of a declarative query language in object-oriented databases is a drawback. It is also clear, from experience with relational and deductive database systems, that a declarative language by itself is not sufficient for most applications — certain parts of an application are best coded in a procedural manner. Thus, there is a real need for combining declarative and procedural styles of programming in an object-oriented database system.

There has been a lot of work done on object-oriented databases, and several systems have reached the marketplace [4, 5, 9, 20, 22, 29]. Logics of objects have been proposed: O-Logic [18] and F-Logic [17]. Many of the object-oriented systems provide a declarative SQL like query language: OSQL in IRIS [29], CQL++ in ODE [12], Reloop [11] and the O_2 query language ([5],Chapter 11) in O_2, and XSQL in UniSQL [16]. Recursive query languages have also been developed: COL [2], IQL [3], Logres [8], and CDOL [15]. Though reservations about mixing object-orientedness with declarativeness continue [27], we feel that the bridge between the two styles of programming is getting stronger.

We further strengthen this bridge between declarative and object-oriented databases by developing a general declarative query language Noodle for object-oriented databases, and shifting the burden of optimizing access for queries expressed in Noodle[1] to the database system. We are building a declarative object-oriented database system called SWORD[2] at AT&T, integrating Noodle with the O++ procedural language of the ODE database system [4].

Outline

The language Noodle borrows from deductive and object-oriented technology. Noodle is rule based. Relations are supported as objects with object identifiers, but individual tuples of a relation do not have object identifiers. Complex objects (functional and set valued) are supported using HiLog-like constructs [10]. Attributes of objects are referenced by name rather than position. Recursion is permitted.

There are several novel features in our system: (1) Variables can range over all classes, relations, attributes, and objects, in a manner similar to Hilog [10].

[1] Noodle stands for "Nonprocedural Object-Oriented Database LanguagE."

[2] SWORD stands for "System With Objects, Relations and Declarativeness."

This permits queries that are not possible in SQL extensions and other object-oriented proposals. (2) Classes, relations, methods, and objects are referenced in a uniform manner. (3) Queries can do implicit schema querying. Queries such as "find all classes (or find all subclasses of *vehicle* class) whose objects have an attribute *engine_capacity*" can be expressed without reference to the catalog. Queries to find objects for which some attribute value is equal to the string "Stanford University" can also be expressed. (4) Noodle is being integrated with a procedural language in the SWORD system [21]. The integration permits some of the methods of a class to be defined declaratively in Noodle, while others can be defined using the procedural language.

2 The Noodle Language: An Introduction

In this section we introduce the language Noodle. We give a walkthrough, without attempting to give a complete description. The language should be thought of as a declarative *query* language. It is not intended to be an all-purpose database manipulation language; rather, it will be integrated with a procedural data manipulation language.

2.1 Atoms

Noodle has the flavor of Datalog and HiLog, but we name attributes rather than rely on argument positions in fixed-arity relations. For example, here is an atom in our language.

$$p\,[name = N, year = Y]$$

Here p is, in general, an object identifier. Specifically, p may be a symbolic or physical identifier for any object in the database. The meaning of the above atom is a multirelation containing the single tuple corresponding to the name and year of object referred by p. For instance, if one refers to one's car by the name "speedy", then speedy$[name = N, year = Y]$ is an atom defining a multirelation with a single tuple containing the name and the year of the car called "speedy". Relations, classes, and other collections are considered to be objects in Noodle. For instance, if "red_cars" is the name of a relation, then red_cars$[creator = C, version = V]$ is an atom defining a multirelation with the single tuple containing the creator and the version of the relation "red_cars".

A second type of atom is available to range over elements of a collection (sets, multisets, classes, relations). If p is an object identifier of a collection object, then the atom

$$p\,\{name = N, year = Y\}$$

defines a binary multirelation with attributes N and Y, containing the *name* and *year* attributes of each tuple in the collection referenced by p. For instance, the atom red_cars$\{name = N, year = Y\}$ defines a multirelation containing the name and year attributes of each car in the relation "red_cars".

The distinction between the $\{\}$ syntax and the $[]$ syntax is important. Given a relation object "red_cars", the attributes in

$$\text{red\_cars}[creator = C, version = V]$$

refer to the attributes of collection object "red_cars", rather than to the attributes of tuples in the relation "red_cars". However, the attributes in

$$\text{red\_cars}\{name = N, year = Y\}$$

refer to the attributes of tuples in the relation "red_cars". Atoms using $\{\}$ will be called *collection atoms*, while atoms using $[]$ will be called *object atoms*. In either case, an object identifier p used as above will be called an object, or a predicate.

There is no notion of *arity* associated with any predicate. An object has a declared signature that indicates its attributes and their types. Irrespective of the signature of an object p,

$$p\,[name = X, age = Y]$$

is a *legal* atom. Further, if p is a collection object, then

$$p\,\{name = X, age = Y\}$$

is also legal atom. The object p (and, in the second case the tuples in p) may simply have two attributes, *name* and *age*. There may be additional attributes, in which case the interpretation of the atoms above is a projection onto *name* and *age*. (Note that the order in which attributes are mentioned is not important.) If the signature of object p does not include one (or both) of the attributes *name* and *age*, then the multirelation defined by the atoms above is always empty.

In the style of HiLog, we allow object identifiers (similar to relation names in relational systems) to be variables that may appear as predicates in atoms. For example,

$$X[name = N, year = Y]$$

is a valid (*object*) atom, defining a multirelation that contains one tuple for each object that has (at least the) attributes *name* and *year*. The tuple contains the identifier of the object, and the *name* and *year* attributes of the object. Similarly,

$$X\{name = N, year = Y\}$$

is a valid (*collection*) atom that matches with every tuple in every collection object whose tuples have (at least the) attributes *name* and *year*. The atom thus defines a multirelation that derives one tuple from each such matching tuple. The tuple in the multirelation contains the identifier of the collection object, and the *name* and *year* attributes of the matching tuple. We also allow X to be bound to sets or classes, for which the atom above is a shorthand for a compound expression that is defined in the next section.

2.2 Classes

A class definition consists of a class name, names and types of attributes, method declarations, and a declaration of the position of the class in the class hierarchy. We use the underlying type system of the procedural language. In this paper we shall use a C++-like type system, with object identifiers represented as pointers. However, an alternative data definition language could be used if desired. A class *vehicle*, containing attributes *name*, *year*, and *manufacturer* may be defined as:

```
class vehicle{ string name;
               date year;
               company* manufacturer; // company is another class in the system };
```

`string` and `date` are assumed to be primitive types in the database system, and *company* is another user defined class. The attribute *manufacturer* is thus the object identifier of a company object.

The collection of all objects in class *vehicle* (objects can be in class *vehicle* directly, or indirectly, by virtue of being in a subclass of *vehicle*) is called its *class extent*. In any database state, the class extent will have zero or more objects. For example, *auto_1*, *auto_4*, and *yacht_6* may be three objects in the class extent.

Within Noodle, we model a class object by a stored unary relation of the same name as the class, with one attribute *member*, that contains the object identifiers of the objects in the class. For example, the class *vehicle* is modeled by a unary relation of the same name, with its attribute *member* being of type *vehicle**. Thus, for the above database state, the class object *vehicle* has the extension:

$$vehicle\{member = auto\_1\},\ vehicle\{member = auto\_4\},\ vehicle\{member = yacht\_6\}$$

We assumed here that all the vehicle objects are included in the relation *vehicle*. However, this default can be overridden. Vehicle objects are created and destroyed from within a procedural language, and we permit the user to omit any or all of the vehicle objects from the default collection of vehicles, say for reasons of security. Further, the user can also choose to maintain other collections of vehicles. These collection can be populated from within the procedural language, and referenced from within Noodle. For the purpose of examples in this paper, we will assume that all the objects of a class are included in the default extension maintained for it. Also note that we do not require that a class extent be stored in any particular fashion. We simply provide a mechanism to *refer* to a class as a relation. The Noodle compiler is responsible for mapping the reference to the implementation for the class.

Thus, the atom $vehicle\{member = X\}$ retrieves members of class *vehicle*, as well as members of all subclasses of *vehicle*. Noodle also provides the syntax

$$@vehicle\{member = X\}$$

to retrieve members of a class *vehicle* that do not belong to any subclass of class *vehicle*. For any class object c, we provide

$$c\{attrib1 = A1, attrib2 = A2\}$$

as a shorthand for

$$c\{member = X\} \ \& \ X[attrib1 = A1, attrib2 = A2].$$

where " & " is the symbol for logical conjunction. Thus, we can use the shorthand atom $vehicle\{name = N, year = Y\}$ to refer to the name and year attributes of each object in the vehicle class.

2.3 Rules and Views

Noodle has if-then rules similar to the rules in Datalog extensions, except that the collection and object atoms introduced above replace atoms of first order predicate calculus. The rules are used to write queries, and to define view collections. A view collection is a relation or a multirelation defined in terms of other collections (including classes) and objects. The type of the view collection must be defined in the database's schema definition language. The extent of the view collection is defined using rules. For example, the rule

$(Q1)$: $\quad vehicle\_name\{name = N\}$:- $vehicle\{name = N\}$.

defines a unary view collection *vehicle_name* that contains the names of all vehicle objects, including the names of vehicles in any subclass of *vehicle*. The expression to the left of the :- symbol is called the head of the rule, and the expression to the right of the :- symbol is called the body of the rule. The body can have one or more atoms, each of which defines a multirelation over the variables that appear in the atom. In rule $Q1$, the body atom defines a multirelation over the variable N, with one tuple for every vehicle object. Both collections and object atoms can appear in the body.

The atom $vehicle\_name\{name = N\}$ in the head of the rule defines how new *tuples* in the view collection *vehicle_name* are derived; the tuples are obtained by a projection from the multirelation derived by the rule body. The type of the projection (set or multiset projection) depends on the type declaration for *vehicle_name*. Only collection objects can appear in the rule head.

Atoms in the body may list a subset of the object's attributes; for example object *vehicle* has attributes besides *name*. However, atoms in the head must appear with all attributes in their respective object's signature explicitly mentioned.

The names of vehicles can be listed without assigning them to a view by writing a rule of the form:

$(Q2)$: N : $vehicle\{name = N\}$.

The variable N written to the left of the : symbol indicates that values assigned to variable N by the rule body must be returned as the answer to the query.

Implicit Schema Querying An important feature in Noodle is the ease of expressing views and queries that would require schema querying in other systems. Consider the case where *motor_vehicles* is a subclass representing motorized *vehicles*, with an additional attribute *engine_capacity*. *vehicle* also has other subclasses, some of which have the attribute *engine_capacity*, and others do not. Let *ford_fast_vehicle* be the (view) collection of the vehicles that have an engine_capacity greater than 100, and are manufactured by Ford. The rule

$(Q3)$: $ford\_fast\_vehicle\{vehicle = V\}$:–
$vehicle\{member = V\}$ &
$V[engine\_capacity = E, manufacturer = M]$ & $E > 100$ &
$M[name = \text{“Ford”}]$.

succinctly, and correctly, computes the view regardless of how many subclasses of *vehicle* have the attribute *engine_capacity*. The atom $V[engine\_capacity = E, manufacturer = M]$ is false for all vehicle objects that do not have the attribute *engine_capacity*, and succeeds for those that have the attribute. Also note that the view definition $Q3$ does not have to change in response to changes in the class hierarchy.

We can also retrieve the most specific subclass to which each *ford_fast_vehicle* belongs:

$(Q4)$: C : $ford\_fast\_vehicle\{vehicle = V\}$ & $@C\{member = V\}$.

Recursive Views Noodle allows recursive view definitions. As an example, consider a CAD application where each cell is made up from a set of cells.

```
class cell{ string name;
           set<cell*> components;
           int x, y; // x and y are location attributes };
```

A `set` is a special type of a relation that has one attribute, *element*. The *components* field is defined to be a set of cell object identifiers. We define a view relation *cell_explosion* to associate, with each cell C, all the subcells S that are contained in cell C, directly or indirectly.

```
relation<struct{ cell* supercell; cell* subcell;}> cell_explosion;
```

$(U1)$: $cell\_explosion\{supercell = C, subcell = S\}$:–
$cell\{member = C\}$ & $C[components = Ss]$ & $Ss\{element = S\}$.

$(U2)$: $cell\_explosion\{supercell = C, subcell = S\}$:–
$cell\_explosion\{supercell = C, subcell = I\}$ &
$I[components = Ss]$ & $Ss\{element = S\}$.

Rules $U1$ and $U2$ define a transitive closure of *cell* on its *components* attribute. The atom $Ss\{element = S\}$ is used to refer to elements of a set field.

Aggregation Noodle includes a grouping primitive and several aggregation functions such as **MIN, MAX, SUM, AVG, CNT, VARIANCE, SDEV, MEDIAN**, ...

Composition of Views A view can be used to define other views. As an example, consider a class *employee* with attributes *name*, *salary*, *bonus*, and *spouse*.

Example 2.1: The income of an employee, given by the sum of his *salary* and *bonus*, is defined using the following view:

$(V1)$: $income\_view\{object = E, result = I\}$:–
$employee\{salary = S, bonus = B\}$ & $I = S + B$.

The census bureau will need to compute a relation of family incomes – defined as the sum of incomes of spouses if both spouses are employees, else the income of the working spouse.

$(V2)$: $fam\_inc\{inc = I2\}$:–
$employee\{member = E\}$ & $income\_view\{object = E, result = I1\}$ &
$E[spouse = W]$ & $employee\{member = W\}$ &
$income\_view\{object = W, result = I2\}$ & $I = I1 + I2$.

$(V3)$: $fam\_inc\{inc = I1\}$:–
$employee\{member = E\}$ & $income\_view\{object = E, result = I1\}$ &
$E[spouse = W]$ & $\neg employee\{member = W\}$.

□

2.4 Class Hierarchy

Classes in Noodle are related by a subclass relationship. A class inherits attributes and methods from its superclasses. A class contains all objects contained in subclasses. The subclass relationship is finite and acyclic, so that the class hierarchy is a directed acyclic graph.

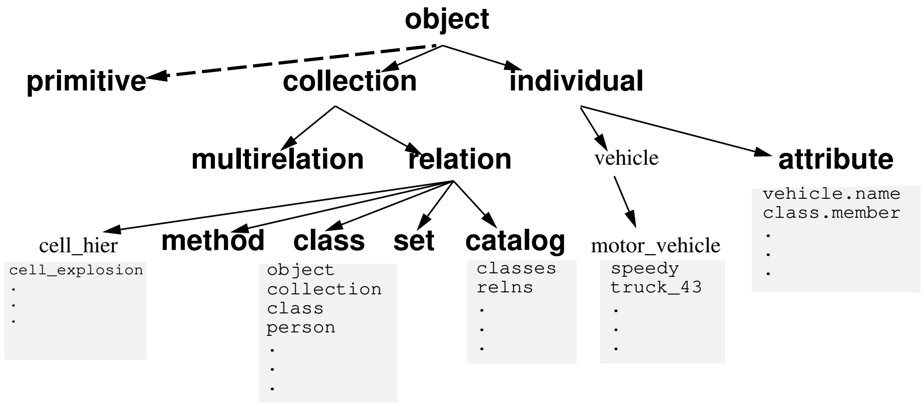


Figure 1: A Partial Class Hierarchy in Noodle.

Noodle has a number of built-in classes to facilitate declarative schema querying. Figure 1 shows the subclass relationship between some of the built-in classes (shown in bold font). Three user-defined classes, *dgraph*, *vehicle* and *motor_vehicle* are shown in regular font for comparison. Example member objects of some classes are listed in shaded rectangles.

The class of all objects is a built-in class (therefore, every object in the database is a member of at least this one class) and is available using

$$object\{member = C\}.$$

Objects belong to a subclass of one of three basic classes, *primitive*, *individual*, and *collection*. The *primitive* class consists of objects such as integers, characters, strings, dates, object identifiers, ..., which don't have a complex structure, and consist of a simple value. Since primitive objects are different from other objects in our language, we draw a dashed arc from *object* to *primitive*.

The class of individuals is the class whose members are individual objects. The *vehicle* class has vehicle objects as members, and is therefore a subclass of *individual*. The class *motor_vehicle* is a subclass of *vehicle*, and contains individual objects *speedy* and *truck_43*.

The class of collections has members that are collections of elements. Each collection of elements is a collection object with an object identifier.

The attributes of various classes are themselves objects, with attributes such as *name, defining_class, default_value*, etc. The attribute objects are members of the built-in class *attribute*.

Collections are of several types: *relations* and *multirelations* are listed, other types of collections (such as lists) are feasible. A *relation* is a set of tuples, and tuples do not have object identifiers. A *multirelation* is a multiset of tuples. Relations with a single attribute called *element* fall into the *set* class. The type of *element* may or may not be an object identifier. When s is a set with elements that are object identifiers, the atom

$$s\{attrib1 = A1, attrib2 = A2\}$$

is permitted as a shorthand for

$$s\{element = X\} \ \& \ X[attrib1 = A1, attrib2 = A2].$$

A user may define subclasses of these basic classes. One may define subclasses of *set*, for example *set-of-employee* and *set-of-dgraph*.

The class of all classes is a built-in class, available as

$$class\{member = C\}.$$

In fact, all relations with a single attribute called *member* fall into the *class* class. The type of *member* is always an object identifier. Class objects such as *vehicle*, *person*, and *object* are members of class *class*. For the hierarchy of Figure 1, the extension of class *class* includes the class objects *object, primitive, collection, multirelation, relation, dgraph, method, class, multiset, set, catalog, individual, vehicle, motor_vehicle*, and *attribute*. Note that the class *class* is necessary in order to ask the query "Tell me all classes" since the alternative

$C : C\{member = X\}.$

lists only the *nonempty* classes.

The class *cell_hier* is a user-defined class containing relations representing hierarchies of cells. Members of this class must have the attributes *cell* and *subcell*. The *cell_explosion* relation of Section 2.3 is a member of this class.

Methods (or, more precisely, method implementations) are also modeled as relations (Section 3.3), and so they form a subclass of *relation*.

In order to facilitate querying of the schema, we include a class *catalog*. The elements of *catalog* are relations describing schema-level properties of some fragment of the system. In Figure 1 there are two catalog relations, namely *classes* (describing properties of each class) and *relns* (describing properties of each relation). The properties for *classes* would include *name*, *attribute_list*, *method_list*, *creator*, *stored_or_view*, *cardinality*, *version*, *security*, *triggerSet*, etc.

The subclass relation is built-in. If D is an immediate subclass of C then

$$@subclass\{super = C, sub = D\}$$

holds. We use *subclass* as the transitive closure of @*subclass*, which may be either built in or specified by standard transitive closure rules.

3 Advanced Features of Noodle

3.1 Complex Objects

Noodle models complex objects (sets and functional terms) in the spirit of object-oriented programming: by building an object X for the set or functional term, where X has its own unique object-identifier. For example, the set $\{stud\_1, stud\_2, stud\_3\}$ is modeled as

$$s\{element = stud\_1\},\ s\{element = stud\_2\},\ s\{element = stud\_3\}$$

The name and the object identifier of the set is s, and s will be a member of the class *set*. Unnesting of sets is done simply by referring to the elements of the set, as in

$$\begin{aligned}(r1):\quad & students\_of\{prof = P, stud = S\} \text{ :- } \\ & \quad stud\_set\{prof = P, advisees = Ss\}\ \&\ Ss\{element = S\}.\end{aligned}$$

We allow a limited form of nesting in defining collection views, using *parametric collections* in a fashion similar to HiLog [10]. For example, suppose we had the opposite situation from rule $r1$, i.e., we have a stored relation *students_of* and want to derive a set of students for each teacher. We can use the following rules:

$$\begin{aligned}(r2):\quad & stud\_of[[prof = P]]\{element = S\} \text{ :- } students\_of\{prof = P, stud = S\}. \\ (r3):\quad & stud\_set\{prof = P, advisees = stud\_of[[prof = P]]\} \text{ :- } profs\{member = P\}.\end{aligned}$$

For each *professor* object X, $stud\_of[[prof = X]]$ is the name of a new relation. This name can function as an object identifier, and can appear as an attribute of other relations. In order to avoid some semantic difficulties, such as implicit recursion through negation, and definition of infinitely many relations, we allow variables in parametric object identifiers to be *individual* objects only.

The functional term

$$address\_1[street = \text{"`600 Mountain Ave"}, City = \text{"`Murray Hill"}]$$

has the the object identifier *address_1*. Components of the term can be accesed using object atoms, as in

$$\begin{aligned}(r4):\quad & empCity\{name = N, city = C\} \text{ :- } \\ & \quad employee\{name = N, address = A\}\ \&\ A[city = C].\end{aligned}$$

3.2 Attributes

Classes and relations have attributes. Each attribute has many properties that are of interest to the user (during schema querying) and to the system (during compilation and run-time). For instance, the name of an attribute, the class in which it was defined, the set of classes that inherit it, its default value, whether it is shared across all objects, its shared value, its offset position, its image size (number of distinct values), and whether it is indexed, are properties of interest. The syntax of an atom is extended to include variables in the positions where attribute names were used. Thus

$$vehicle\{A = \mathrm{b}, engine\_capacity = E\}$$

is a valid atom, retrieving engine capacities of vehicles that have the object b as the value of at least one attribute. Variable A ranges over names of all attribute objects. Thus, the names (from the *name* field) of all the attributes that have value b are also retrieved.

3.3 Methods

Noodle allows methods to be defined declaratively (as views). However, not all methods can be expressed declaratively since methods, in general, have side-effects. A method can update the database, print results and in various ways change the state of the database. Hence, Noodle also permits methods to be defined imperatively within the procedural object-oriented language. Methods can be side-effect free, have benign side-effects (such as updating a system-maintained popularity count of all objects accessed by the method), or have real side-effects. Noodle models methods without side-effects, and methods with benign side-effects, as relations, and consequently the class *method* is a subclass of the class *relation* in Figure 1. (This is similar to the approach taken in the NAIL/Glue system [23].) Methods, viewed as a relation, must have an *object* attribute to specify the object to which the method applies, and a *result* attribute through which the method returns a value. Additional attributes may be present. For example, the view *income_view* defined by rule $V1$ in Section 2.3 meets the conditions for being a method. Note that not all views are methods, and not all methods are views. A view is a method if it has the *object* and *result* attributes. A method is a view if it is defined declaratively within Noodle.

Every class has a set of associated methods. A method definition consists of the method name and a pointer to the view definition or the procedural code implementing the method. The association between method names and method implementations is specified in the definition of the corresponding class. Consider the view *income_view* defined by rule $V1$ in Section 2.3 on employees. A method named *income* can be defined, and associated with the implementation *income_view* as follows:

```
class employee { string name;
                 int salary,bonus;
                 int income() = income_view; };
```

We can also replace references to $income\_view$ in the definition of fam_inc (in rules $V2$ and $V3$) with references to the method $income$.

Methods associated with superclasses may also be applied to objects in a subclass, unless the subclass has its own definition of a method with the same name. The method is then overridden by associating a different implementation for the same method name in the subclass. For instance, suppose $sales\_employee$ is a subclass of $employee$ with additional attributes $sales$ and $commission\_percent$, declared as

```
class sales_employee : public employee {
      int sales;
      int commission_percent;
      int income() = sales_income_view; };
```

(The keyword `public` makes visible the attributes inherited from class *employee*.) The income of sales employees would be computed as:

$$(V4):\quad sales\_income\_view\{object = E, result = I\} \text{ :- } \\ sales\_employee\{member = E\} \;\&\; \\ E[salary = S, bonus = B, sales = X, commission\_percent = P] \;\&\; \\ I = S + B + X \star P/100.$$

If we now pose the query

$$(Q5):\quad I : employee\{member = E\} \;\&\; income\{object = E, result = I\}.$$

then the effect would be to apply $sales\_income\_view$ to sales employees, and to apply $income\_view$ to the remaining employees. Also, the view fam_inc would automatically apply the correct income method to each employee. We can thus overload method names so that the method used for an object x is the one in the lowest class of the class hierarchy containing x.

3.4 Interchangeability of Attributes and Methods

Noodle provides a built-in method to access every attribute.

For example, since objects in class $employee$ have a $salary$ attribute, Noodle defines a $salary$ method that can be accessed as

$$S : salary\{object = john, result = S\}$$

Conversely, every binary method (with the only attributes being $object$ and $result$) can be referenced either as a method or as an attribute. Thus, the query $Q5$ above could be rewritten as:

$$(Q5'):\quad I : employee\{member = E\} \;\&\; E[income = I].$$

In this way, the user does not have to know whether $salary$ and $income$ are attributes or methods in class $employee$.

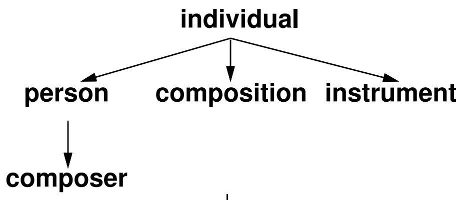
4 Semantics

Noodle permits recursive view definitions, modularly stratified aggregation, and modularly stratified negation [24]. A model-theoretic semantics can be defined for queries and view definitions in Noodle, in a fashion similar to the semantics of HiLog. There are some differences since we must handle explicitly labeled attributes, multiset semantics for multirelations, and variables implicitly bound to *collections* or *objects*. Due to space limitations, we do not present the semantics here.

Given a database, the Noodle rules can be evaluated in a bottom-up fashion to compute the fixed-point semantics. The extent of all *collection* and *individual* objects is assumed available, and may be used to substitute for variables appearing in predicate positions. The schema information is used to prune the variable substitutions that are guaranteed to fail, say, due to the absence of an attribute that is mentioned in the query.

5 An Example

We illustrate several of the features of Noodle using an example of a composer database originally from [19]. The database has four classes, namely *person*, *composer*, *composition*, and *instrument*, related as shown in Figure 2.




```
class person {
      string name;
      date birthdate; };

class composer : public person {
      composer* master;
      set<composition*> works; };
```

```
class composition {
   string title;
   composer* author;
   set<instrument*> instruments; };

class instrument {
      string name;
      string family; };
```

Figure 2: Composer Fragment of the Class Hierarchy

The notation "set<*composition**> *works*" means that the type of attribute *works* is a set of pointers to objects of class *composition*. The *master* attribute of a *composer* gives the identifier of the composer who acted as a mentor, while the *works* attribute gives an identifier for a set of *composition* objects created by the composer. The *instruments* attribute of a *composition* gives an identifier for a set of instruments used in the composition.

The query "Find the title of works by Bach that use a harpsichord and a flute" is expressed as:

$$
\begin{aligned}
(Q6)\text{:}\quad & T : composer\{name = \text{"Bach"}, works = Ws\} \ \& \\
& \quad Ws\{title = T, instruments = Is\} \ \& \\
& \quad Is\{name = \text{"harpsichord"}\} \ \& \ Is\{name = \text{"flute"}\} \ .
\end{aligned}
$$

Let *influencer* be a view such that $influencer\{master = M, disciple = D, gen = N\}$ holds if M is an Nth order ancestor of D in the master/disciple hierarchy.

$$
\begin{aligned}
(P1)\text{:}\quad & influencer\{master = M, disciple = D, gen = 1\} \text{ :- } \\
& \quad composer\{member = D\} \ \& \ D[master = M].
\end{aligned}
$$

$$
\begin{aligned}
(P2)\text{:}\quad & influencer\{master = M, disciple = D, gen = N + 1\} \text{ :- } \\
& \quad influencer\{master = M, disciple = Y, gen = N\} \ \& \\
& \quad composer\{member = D\} \ \& \ D[master = Y].
\end{aligned}
$$

The query "Find names of all composers influenced by composers for harpsichord that lived at least six generations before" can then be expressed as

$$
\begin{aligned}
(Q7)\text{:}\quad & V : influencer\{master = M, disciple = D, gen = N\} \ \& \ N \geq 6 \ \& \\
& \quad X[name = V] \ \& \ Z[works = Ws] \ \& \ Ws\{instruments = Is\} \ \& \\
& \quad Is\{family = \text{"harpsichord"}\}.
\end{aligned}
$$

6 Related Work

Several efforts to integrate declarative querying in object-oriented database system have been made, and several others are in progress [1, 6], COL [2], Reloop [11], IQL [3], the O_2 query language ([5], Chapter 11), OSQL in the IRIS system [29], Logres [8], F-Logic [17], HiLog [10], Complex [13], XSQL [16], CQL++ [12], ZQL[C++] [7], Orlog [14], $\mathcal{QUIXOTE}$ [30], CDOL [15], A DOOD Ranch [28], Coral++ [25], and our own SWORD [21] system based on Noodle. In this section, we compare the salient features of Noodle with several of the above systems.

Noodle supports relations with tuples that do not have object identifiers. Thus, every element of every collection is not forced to be an object with its own object identifier. The need for relations and tuples without object identifiers in an object oriented system has been realized by others ([1, 6], IQL, Logres, the O_2 query language, Orlog, Coral++), but is missing from some of the proposals (CQL++, CDOL, A DOOD Ranch, ZQL[C++]). The latter systems force new object identifiers to be created for each tuple generated in a query, leading to difficulties in duplicate elimination, view maintenance, and taking the difference of views.

Noodle is based on HiLog, a logic that provides a first order semantics to apparently second order logical formulae. Noodle extends HiLog to support inheritance and object identity. Noodle permits implicit schema querying, and permits variables to range over classes, relations, attributes, and methods. Built-in classes are also provided to do explicit schema querying. XSQL is the only other language with the above features; however XSQL is based on F-logic rather than HiLog. While Noodle has a clear distinction between subclasses and class objects, F-Logic does not distinguish between these two concepts. (We understand that, as a result of discussions with us, amongst others, F-logic has been modified to make the above distinction). Orlog has a higher order syntax, but cannot do schema querying. OSQL provides limited implicit schema querying, but queries such as $Q4$ (Page 7) cannot be expressed.

Noodle models sets as complex objects, with each set having its own object identifier. In contrast, Reloop, IQL, the O_2 query language, XSQL, Orlog, and Coral++ model set-valued attributes. In OSQL, the result of every query is a bag (that is, a multirelation), and bags are always treated as values, with references to bags leading to replication of the entire bag.

Noodle permits general recursion, and is more expressive than several SQL based proposals (Reloop, OSQL, CQL++, ZQL[C++]) that do not include recursion. CQL++ does not support taking the difference of the results of two queries (the EXCEPT clause of SQL). Several logic based query languages (COL, IQL, Logres, Complex, $\mathcal{QUIXOTE}$, CDOL, Orlog, Coral++) include recursion, but lack the expressive power of schema querying.

Noodle, as presented here, does not support general object creation. Parametric collection objects are however provided to mimic the behavior of certain types of object creation. COL, IQL, Logres, XSQL, and CDOL provide more powerful object creation primitives, with an associated overhead of maintaining complex object identifiers.

Noodle uses a uniform logic-based syntax to refer to relations, classes, and complex objects. Path traversal in complex structures is done through logical atoms that can be understood and optimized by the Noodle compiler. In contrast, Coral++ and ZQL[C++] use arbitrary C++ expressions to refer to complex structures in classes, and CQL++ uses the SQL dot notation. The user has to understand the path expressions as having a semantics distinct from other atoms in the language - for example, in Coral++ and CQL++, path expression can only serve as arguments to logical predicates, while in ZQL[C++] the path expressions can return a C++ boolean type. XSQL also uses the SQL dot notation for path expressions, but the expressions are in F-logic, and the system can interpret the expressions.

Ullman [27] presents two problems in adding declarativeness to an object-oriented database: (1) Duplicate elimination in presence of object identifiers is tricky. Duplicate elimination is often value-based, not object-based. If every derived object is given a new object identifier, duplicate elimination is not feasible. By explicitly introducing tuples without object identifiers, we resolve the incompatibility. (2) Lack of compositionality. Newly defined (derived) collection objects cannot be used to derive new collection objects, since no real operators are available on the newly derived collections. We address this criticism by modeling a *collection* class that contains all collection objects that are derived using rules. As a result, the relational operators, defined on the class of collections, are applicable to all newly defined collection objects. Consequently, Noodle is compositional.

7 Conclusions and Further Work

We have presented a declarative query language Noodle for an object-oriented database system. The language models object-identity, classes, relations, views, inheritance, complex objects, and methods, in addition to logical rules, and permits powerful schema querying.

A database system called SWORD, based on Noodle, is being implemented at AT&T. Designing an interface with a procedural language is an intricate task, with potential "impedance mismatch" problems. In the SWORD system, Noodle is being integrated with the O++ language of the ODE database system [4].

There are a number of implementation issues that would need to be addressed if this language is to be used in a real system. An obvious issue is optimization. As this language is implemented, we intend to study this question extensively.

Another issue involves schema querying. Some queries perform implicit schema queries. If these queries are *compiled*, then some of the schema information may be compiled into the code. If the schema changes, for example with the addition of a new class, then the query would have to be recompiled. If queries are *interpreted*, then the schema can be queried at run-time.

One might like to define classes using rules; let us call these "view classes." One would need to specify how the object identifiers of each derived object is obtained. In simple cases, derived classes can utilize existing object identifiers; however, in general new object identifiers must be created. Further, the object identifiers must be retained across different computations. The details on how we handle view classes are beyond the scope of this paper, and will be presented separately.

Additionally, we would like to provide support for other collection data types such as sequences and lists. It is not clear, though, what the semantics of lists should be under logical operations such as conjunction.

Acknowledgements

We thank S. Sudarshan for several suggestions and comments that influenced the design of Noodle, and its subsequent implementation in SWORD. We also thank

Narain Gehani, Richard Greer, Michael Kifer, Dan Lieuwen, H. V. Jagadish, Nandit Soparkar, Avi Silberschatz, and Jeff Ullman for comments on earlier drafts of this paper.

References

[1] S. Abiteboul. Towards a deductive object-oriented database language. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 419–438, 1989.

[2] S. Abiteboul and S. Grumbach. Col: a logic-based language for complex objects. In *Proceedings of the International Conference on Extending Data Base Technology*, pages 271–293, March 1988.

[3] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD 1989 International Conference on Management of Data*, pages 159–173, Portland, OR, May 1989.

[4] R. Agrawal and N. Gehani. Ode (object database and environment): the language and the data model. In *Proceedings of ACM SIGMOD 1989 International Conference on Management of Data*, pages 36–45, Portland, OR, May 1989.

[5] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, Washington D.C., 1992.

[6] C. Beeri. Formal models for object-oriented databases. In *Proceedings of the DOOD89*, pages 370–395, Kyoto, Japan, December 1989.

[7] J. A. Blakeley. ZQL[C++]: Integrating the C++ language and an object query capability. In I. S. Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 138–144, Washington, DC, May 29 1993.

[8] F. Cacace, S. Ceri, S. Crespi-Reghizi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 225–236, Atlantic City, NJ, May 23-25 1990.

[9] R. C. G. Cattell. *Object Data Management*. Addison-Wesley, 1991.

[10] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 1992.

[11] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. Reloop, an algebra-based query language for O_2. In *Proceedings of the DOOD89*, Kyoto, Japan, December 1989.

[12] S. Dar, N. Gehani, and H. V. Jagadish. CQL++: A SQL for the Ode object-oriented DBMS. In *Proceedings of the Third International Conference on Extending Data Base Technology*, pages 201–216, Vienna, Austria, March 1992.

[13] S. Greco, N. Leone, and P. Rullo. Complex: An object-oriented logic programming system. *ACM Transactions on Knowledge and Data Engineering*, 4(4):344–359, Aug. 1992.

[14] M. H. Jamil and L. V. S. Lakshmanan. Realizing Orlog in $\mathcal{LDL}$. In I. S. Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 45–59, Washington, DC, May 29 1993.

[15] A. Karadimce and S. Urban. CDOL: A declarative platform for developing OODB applications. In *Proceedings of the International Phoenix Conference on Computers and Communications*, pages 224–230, Tempe, AZ, Mar. 1993.

[16] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 393–402, San Diego, CA, June 2-4 1992.

[17] M. Kifer., G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 90/14 (second revision), SUNY at Stony Brook, 1990. to appear in Journal of the ACM.

[18] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, 1989.

[19] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. Optimization of object-oriented recursive queries using cost-controlled strategies. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data*, pages 256–265, San Diego, CA, June 2-5 1992.

[20] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *OOPSLA 1986 Proceedings*, pages 472–482, 1986.

[21] I. S. Mumick, K. A. Ross, and S. Sudarshan. Design and implementation of the SWORD declarative object-oriented database system, 1993. Unpublished Manuscript.

[22] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the objectstore database system. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data*, pages 403–412, San Diego, CA, June 2-5 1992.

[23] G. Phipps, M. Derr, and K. A. Ross. Glue-Nail: A deductive database system. In *Proceedings of ACM SIGMOD 1991 International Conference on Management of Data*, pages 308–317, Denver, CO, May 29-31 1991.

[24] K. A. Ross. Modular stratification and magic sets for datalog programs with negation. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 161–171, Nashville, TN, April 2-4 1990.

[25] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB)*, Dublin, Ireland, August 24-27 1993.

[26] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1*. Computer Science Press, 1988.

[27] J. D. Ullman. A comparison between deductive and object-oriented database systems. In *Proceedings of the DOOD91*, pages 263–277, Germany, December 1991.

[28] S. D. Urban and S. W. Dietrich. A deductive, object-oriented model as a formal framework for active database environments. In I. S. Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 101–110, Washington, DC, May 29 1993.

[29] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *ACM Transactions on Knowledge and Data Engineering*, 2(1):63–75, Mar. 1990.

[30] K. Yokota, H. Tsuda, and Y. Morita. Specific features of a deductive object-oriented database language $\mathcal{QUIXOTE}$. In I. S. Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 89–99, Washington, DC, May 29 1993.