Design and Implementation of the SWORD*Declarative Object-Oriented Database System

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

Kenneth A. Ross[†]
AT&T Bell Laboratories
kar@cs.columbia.edu

S. Sudarshan
AT&T Bell Laboratories
sudarsha@research.att.com

September 1993

Abstract

SWORD is a declarative object-oriented database being built at AT&T Bell Laboratories.

SWORD provides both procedural and declarative data manipulation languages. SWORD is integrated with the O++ language (an extension of C++) of the ODE database system. The data definition and procedural data manipulation in SWORD are done in the O++ sublanguage. In addition, SWORD provides a declarative sublanguage to express queries. Declarativeness yields benefits in reduced programming time and automatic optimization in response to changing database characteristics. The declarative sublanguage is based on the Noodle language of [MR93], modified for integration with C++/O++ to avoid the impedance mismatch problem. The declarative sublanguage models object-identity, classes, relations, views, inheritance, complex objects, and methods, in addition to logical rules. In this paper we discuss the design and implementation of the declarative sublanguage of SWORD.

Our work bridges the gap between deductive and object-oriented databases. SWORD will permit a wide range of object-oriented queries to be expressed in a declarative way.

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 [Ull88].

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 ad hoc 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

*System With Objects, Relations, and Declarativeness

†Permanent Address: Computer Science Department, Columbia University, New York, NY 10027.

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 [MSOP86, WLH90, AG89, Cat91, BDK92, OHMS92]. Logics of objects have been proposed: O-Logic [KW89] and F-Logic [KLW90]. Many of the object-oriented systems provide a declarative SQL like query language: OSQL in IRIS [WLH90], CQL++ in ODE [DGJ92], Reloop [CDLR89] and the O_2 query language ([BDK92],Chapter 11) in O_2, and XSQL in UniSQL [KKS92]. Recursive query languages have also been developed: COL [AG88], IQL [AK89], Logres [CCCR+90], and CDOL [KU93]. Though reservations about mixing object-orientedness with declarativeness continue [Ull91], 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 declarative query language for object-oriented databases, and shifting the burden of optimizing access for queries expressed in the declarative language to the database system. The declarative language is based on the Noodle language of [MR93], modified for better integration with the O++ procedural language of the ODE database system [AG89]. We describe the design and implementation of SWORD.

Outline

The SWORD declarative language borrows from deductive and object-oriented technology. Our language has recursion and 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 [CKW92]. Attributes of objects are referenced by name rather than position.

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 [CKW92]. 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) The declarative language is integrated into O++. For example, some of the methods of an O++ class can be defined declaratively, others can be defined procedurally using O++. (4) 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 is equal to the string "Stanford University" can also be expressed.

We discuss the implementation of SWORD. We describe how SWORD addresses issues of integration with the *O++* data model, and issues related to compilation into O++, especially with respect to types.

The paper is organized as follows. In Section 3 we give a detailed description of the SWORD declarative language. In Section 6 we present an example application using SWORD. Section 7 describes the implementation. Finally, Section 9 describes future work, and compares our work with related proposals.

2 Terminology

SWORD inherits the typing system of O++. Thus any O++ data-type is also a SWORD data-type. We assume that the reader has at least a passing familiarity with the O++ type system. SWORD enhances the C++ built-in types with a number of built-in classes.

The most important built-in class is the class collection and its descendant classes. These classes are the basis on which set-oriented processing is built. Objects in such classes are called collections. The *multiset* class is the class of collections whose extent is a multiset of elements. The *set* class is a subclass of multisets where each element has a multiplicity of 1. When the element of a multiset is a tuple, the multiset belongs to the subclass *multirelation*. Similarly, the class *relation* consists of sets whose element is a tuple, or equivalently of multirelations whose elements have a multiplicity of 1. We discuss more details of the class hierarchy in Section 3.5.

2.1 Data Items

Each piece of data in a SWORD database or program is called a **data item**. Data items can be partitioned in three different ways: (a) persistent vs. nonpersistent, (b) simple vs. complex, and (c) values vs. objects. We now discuss each of these partitions.

Definition 2.1 Persistent and Nonpersistent Data Items: The data items stored in the database are said to be persistent. Persistent data items outlive the execution of the program that created them, and must be declared persistent at time of creation. In contrast, non persistent data items are ordinary C++ data items that are available only to the application program in which they are created. □

Definition 2.2 Simple and Complex Data Items: A *simple* data item is a data item that is a simple value, and does not have an internal structure. A *complex* data item is one that is not simple. □

Examples of *simple* data items include any of the *C++* built-in types, such as integers, floats, characters, pointers, and enum types. Data items of type struct, class, arrays etc are examples of *complex* data items.

Definition 2.3 Objects and Values: A complex data item created as a member of a class by invocation of the *new* or *pnew* functions is called an *object*. All other data items are called *values*. □

Note that by the above Definition 2.3,

1. All simple data items are values.
2. If C is a class, then a variable of type C is a value. If C is a complex class, that the variable will be a complex value.

An example of a complex value in a database is a nested struct within an object; the object has an object identifier, but the nested struct does not have an independent object identifier. Thus the nested struct is treated as a complex value, not as an object.

All persistent objects have a *persistent object identifier* which serves as a means to identify the object. O++ adds to C++ the notion of persistent classes, i.e., classes that may have objects that are persistent. Each 'persistent' class in O++ has an associated extent, which contains the persistent object identifiers of all persistent objects in the class. Persistent simple values (i.e., of one of the C++ built-in types) can be created in an O++ database, but no extents are maintained for the values. Similarly, a complex value of class *student* can be nested inside a persistent *faculty* object, but the *student* will not be included in the extent for the *student* class. Extents are only maintained for persistent (complex) objects.

Definition 2.4 dbitems, dbobjects, and dbvalues: Since the database only stores persistent data items, we will use the phrases *dbitem* to refer to persistent data items, *dbobject* to refer to persistent (complex) objects, and *dbvalue* to refer to persistent (simple or complex) values. □

Definition 2.5 Collection vs Individual: A data item is a *collection* data item if it belongs to a collection class. A data item is an *individual* data item if it does not belong to a collection class. □

An object is identified uniquely by its object identifier. The values of the attributes of an object can change, but its oid does not change. No other values, such as the value of attributes of an object, can uniquely identify objects in general. Hence a collection of objects is always represented as a collection of object identifiers.

3 The SWORD Language: An Introduction

In this section we describe the declarative sublanguage of the SWORD system. We give a walk-through, without attempting to give a complete description. The language is based on the language Noodle described in [MR93], but with syntactic modifications to make it more compatible with C++/O++. The declarative sublanguage 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 used in conjunction with the O++ procedural data manipulation language.

3.1 Atoms

SWORD has the flavor of SQL, Datalog, and HiLog, and follows the conventions of O++. The language permits logical atoms, but we name attributes rather than rely on argument positions in fixed-arity relations. For example, here is a logical atom in our language.

$p\,[[nameN, yearY]]$

Here p is, in general, a data item that must either be (1) the (symbolic or physical) identifier of a (complex) object, or (2) a complex value. The meaning of the above atom is a multirelation containing the single tuple corresponding to the name and year of data item referred by p. For instance, if one refers to one's car by the name "speedy", then

$$\text{speedy}[[name\ N, year\ Y]]$$

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, with the name of a collection serving as its symbolic object identifier. For instance, if "red_cars" is the name of a relation, then

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

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 logical atom is available to range over elements in the extent of a collection (multisets, sets, classes, multirelations, relations). If p is a collection data item, that is, either (1) the (symbolic or physical) identifier of a collection object, or (2) a a collection value, then the atom

$p\,\{M\}$

defines a multirelation with attribute M, containing the data items in the collection referred by p.

A Shorthand If the extent of p is a relation, a multirelation, or a multiset of object identifiers, we can directly refer to the attributes of tuples/objects in the extent of p. As an example, 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, or the attributes of the object referenced by the object identifier, in the collection referenced by p. The atom above is simply a short hand for

$p\,\{M\}\ ,,\ M[[name\ N, year\ Y]]$

where ,, is the symbol for logical conjunction in SWORD. For instance, the atom

$$\text{red\_cars}\{C\}$$

defines a multirelation with attribute C, containing the tuples in the relation red_cars. The atom

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

defines a multirelation containing the name and year attributes of each tuple in the relation "red_cars".

Object and Collection Atoms The distinction between the { } syntax and the [[ ]] syntax is important. Given a relation object "red_cars", the attributes C and V in the atom

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

refer to the attributes of the relation object "red_cars", rather than to the attributes of tuples in the relation "red_cars". However, the attributes N and Y 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, pointer, or value p used as above will be called a predicate.

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

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

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

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

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

Higher-Order Syntax In the style of HiLog, we allow predicates (similar to relation names in relational systems) to be variables. For example,

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

is a valid (*object*) atom. The variable X ranges over all objects and complex values in the database, (including collection and individual data items). The variable does not range over simple values, such as the integers. Thus, the atom defines a multirelation that contains one tuple for each data item (object and value) that has (at least the) attributes *name* and *year*. The tuple contains the identifier of the object, or the entire complex value, and the *name* and *year* attributes of the data item. Similarly,

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

is a valid (*collection*) atom, with the variable X ranging over all collection objects and collection values in the database. The atom matches with every tuple in every collection data item whose elements 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, or the entire collection value, and the *name* and *year* attributes of the matching tuple.

Atoms where a variable appears in the predicate position are said to use a *higher-order* syntax since the variable can range over "all collections".

The predicate p may be a symbolic identifier for an object, such as *vehicle*, *employee*, and *"john"*. An O++ expression whose type is an object identifier, or a complex value, can also be used as a predicate. For example, if $u \rightarrow v$ has the type *persistent vehicle**, then $u \rightarrow v$ is a valid predicate.

O++ Atoms The final kind of atoms we allow are O++ expressions, such as $X = 3$, or

$$(X! = \text{NULL})\ \&\&\ (X \rightarrow age > min\_drive\_age(\text{"new\_york"})).$$

We require that the type of the expression be any type that can appear as the condition of an *if* statement in O++. Such atoms are called O++ atoms.

3.2 Logical Variables

The variables X, N, Y, ... used in SWORD atoms above are logical variables. A keyword `logical` is provided to declare such variables. As an example, the following are valid declarations:

(1) `logical string` N, C;
(2) `logical int` Y, V;
(3) `logical` X, M;

(1) declares N and C to be logical variables of type `int`, (2) declares Y and V to be logical variables of type `string`, and (3) declares X and M to be untyped logical variables.

3.3 Classes

Classes are defined as in O++. A class *vehicle*, containing attributes *name*, *year*, and *manufacturer* may be defined as:

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

`string` and `date` are assumed to be primitive types in O++, 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 SWORD, we model a class object by a stored set of object identifiers of the objects in the class. The set has the same name as the name of the class; the name functions as the symbolic object identifier of the class object. For example, the class *vehicle* is modeled by a set whose elements have the type *vehicle**, and whose symbolic object identifier is *vehicle*. Thus, for the above database state, the class object *vehicle* has the extension:

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

O++, by default, includes all objects of a class in its class extent. Hence, we assumed here that all the vehicle objects are included in the set *vehicle*. The user can choose to maintain other stored collections of vehicles that do not contain all *vehicles* objects. These collection can be populated at the time of creation of the vehicle objects, by querying an existing collection, or by any other means. 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 set. The SWORD compiler maps a reference to the class onto the implementation for the class.

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

$$@vehicle\{X\}$$

to retrieve members of a class *vehicle* that do not belong to any subclass of class *vehicle*. As discussed in Section 3.1, 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.

3.4 SWORD Expressions and Views

SWORD has select-where expressions that are similar to the select-from-where expressions in SQL. The *select* clause in SWORD is similar to the SQL `SELECT` clause. Terms that SQL forces to be divided between its `FROM` and `WHERE` clauses can all be placed into the SWORD *where* clause. (Consequently, it is easier to write a program in SWORD). The SWORD expressions are also similar to the if-then rules in Datalog extensions, except that the collection and object atoms introduced above replace atoms of first order predicate calculus. The *select* clause corresponds to the "if" part of a rule, and the *where* clause corresponds to the "then" part of a rule.

The SWORDexpressions are used to write queries, and to define view collections. A view collection is a collection defined in terms of other collections (multirelations, relations, classes, ...) and objects. A view definition consists of two parts — (1) the type of the view, and (2) the extent of the view. The type of a view collection is declared using the keyword view, followed by the type, specified in two parts:

$$view \quad col\_type < element\_type >$$

col_type is the type of the collection (multiset, set, relation, multirelation, ...), and *element_type* is the type of each element in the collection. The extent of the view collection is defined using SWORD expressions. For example,

```
(Q1):   persistent view set<struct{string name; date year;}> vehicle_list {
            logical string N; logical date Y;
            select {name N, year Y}
            where vehicle{name N, year Y}
            }
```

defines a view collection *vehicle_list* that contains the names and years of all vehicle objects, including the vehicles in any subclass of *vehicle*. The view has the type "set of structs with attributes, *name* of type `string`, and *year* of type `date`". The keyword *persistent* declares that objects of this collection type, such as the *vehicle_list* object, can be persistent. The extent definition following the type declaration specifies that for every *vehicle* object whose *name* attribute has value N, and *year* attribute has value Y, there must be a tuple ($name\ N$, $year\ Y$).

The tokens *select*, and *where* are keywords in SWORD, with a functionality similar to that in SQL. The *where* clause of the expression consists of a conjunction of *literals*, where a *literal* is either an atom (a *positive literal*), or the negation of an atom (a *negative literal*). The symbol ,, is used for logical conjunction. (The C++ conjunction symbol && is only used inside an O++ expression). The symbol ,, provides no guarantees about the order of evaluation; The conjuncts may be reordered by the system during optimization. Each atom in the *where* clause defines a multirelation over the variables that appear in the atom. In expressin $Q1$, the atom $vehicle\{name\ N,\ year\ Y\}$ defines a multirelation over the variables N and Y, with one tuple for every vehicle object. Both collections and object atoms can appear in the *where* clause.

The keyword *select* is followed by a list of data items that defines how new *tuples* in the view collection *vehicle_list* are derived; the tuples are obtained by a projection from the multirelation derived by the *where* clause, onto the variables in the *select* list. The type of the projection (set or multiset projection) depends on the type declaration for *vehicle_list*; in our example, *vehicle_list* has been defined to be a set, so a set projection will be taken. Each tuple in the projection derives a value for the attributes to the left of the variables, and thereby derives a tuple in the view. The *select* list is enclosed in braces { } to indicate that a collection of tuples is being constructed.

Atoms in the *where* clause may list a subset of the object's attributes; for example object *vehicle* has attributes besides *name* and *year*. However, the *select* list must contain all attributes in their respective tuple type's signature.

The collection of names and years of vehicles can be assigned to a collection variable *vlist* by writing the SWORD expression:

```
(Q2):   logical string N; logical date Y;
        set<struct{string name; date year;}> vlist;
        vlist = select {name N, year Y}
                where vehicle{name N, year Y}
                }
```

A SWORD expression can thus be used inside a view definition, or outside a view definition, as a query in statements of the form $Q2$.

Implicit Schema Querying An important feature in SWORD 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 statement

```
(Q3):   persistent view set<vehicle*> ford_fast_vehicle {
            logical vehicle* V; logical company* M;
            select {V}
            where vehicle{V},,
                    V[[engine_capacity E, manufacturer M]],,E > 100 ,,
                    M[[name = "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):   logical class* C; logical vehicle* V;
        select {C}
        where ford_fast_vehicle{V},,
                @C{V} ,,
    }           class{C}
```

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

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

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.

```
(T):   persistent view relation<struct{ cell* supercell; cell* subcell;}> cell_explosion {
           logical cell* C, S, I; logical set<cell*> Ss
           select {supercell C, subcell S}
           where cell{C} ,, C[[components Ss]] ,, Ss{S}
       union
           select{supercell C, subcell S}
           where cell_explosion{supercell C, subcell I} ,,
                   I[[components Ss]] ,, Ss{S}
```

Statement T defines a transitive closure of *cell* on its *components* attribute. *union* is a SWORD keyword that takes a union of the results of each select-where expression.

Aggregation SWORD includes a *groupby* clause and several aggregation functions such as `MIN, MAX, SUM, AVG, CNT, VARIANCE, SDEV, MEDIAN`, ...

EXAMPLE 3.1 The view *avg_salary_view* computes, for each class C whose objects have an attribute *salary*, the average salary value amongst the objects in class C.

```
(U):  persistent view relation<struct{persistent class* cl; double avgSal;}> avg_salary_view{
          logical class* C; logical double S;
          select {cl C, avgSal AVG(S)}
          where class{C},,
                C{salary S}
          groupby C
          }
```

The term groupby C is a grouping clause, and is to be interpreted as follows: Take the multirelation over (C, S) derived from the atoms in the where clause, and group the result by the class identifiers C. The term

avgSal AVG(S)

in the select clause then computes the average S value amongst all the tuples in each group. □

The syntax of the groupby clause is borrowed from SQL. The SWORD language will permit monotonic aggregation [MPR90] and modularly stratified aggregation [Ros90].

Composition of Views and Negation A view can be used to define other views. As an example, consider a class *employee* defined as:

```
persistent class person{
    string name;
    date birthdate;
    persistent person* spouse;
};
```

```
persistent class employee : public person {
    int salary, bonus;
    date year_hired;
};
```

EXAMPLE 3.2 The income of an employee, given by the sum of his *salary* and *bonus*, is defined using the following view:

```
(V1):  persistent view relation<struct{persistent employee* emp; int income;}> income_view{
           logical employee* E; logical int S, B, I;
           select{emp E, income I}
           where employee{E},,
                 E{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):  persistent view relation<int> fam_income{
           logical person* E, W; logical int I1, I2;
           select {I2}
           where employee{E},,
                 E[[spouse W]],,
                 employee{W},,
                 income_view{emp E, income I1},,
                 income_view{emp W, income I2},,
                 I == I1 + I2
       union
           select {I1}
           where employee{E},,
                 E[[spouse W]],,
```

$\neg employee\{W\}$,,
$income\_view\{emp\ E, income\ I1\}$
}

□

The atom $\neg employee\{W\}$ is a negated atom. It succeeds if W is not a member of the collection *employee*. The SWORD language permits modularly stratified negation [Ros90].

C++ Functions

EXAMPLE 3.3 Continuing with the *employee* example, let *manager* be a subclass of employees, defined as:

```
persistent class manager : public manager {
    string mgmt_level;
};
```

We define a view relation of senior employees. All managers are considered senior, and employees who have worked for more than 10 years are senior. For each senior employee, the view contains the employee's name, the number of years of service, and a code "E" or "M", to indicate whether that employee is a manager or not.

```
persistent view relation<struct{string name; int years; char code;}> senior_emp{
          logical N, Y, D;
(W1)      select {name N, years Y, code 'E'}
          where employee{name N, yr_hired D},,
                 Y == today_year() − year(D) ,, Y > 10
     union
(W2)      select {years Y, name N, code 'M'}
          where manager{name N, yr_hired D},,
                 Y == today_year() − year(D)
          };
```

□

today_year() is a C++ function that returns an integer representation of the current year. year is also a C++ function that return an integer representation of the year in a given date. These functions have been used inside inside the equality atom $Y ==$ today_year() $-$ year(D) to obtain the number of years of service. Note that the variables N, Y, and D have been declared as untyped logical variables; their types will be inferred by SWORD. Also note that attributes can appear in the *select* list in an arbitrary order.

3.5 Class Hierarchy

SWORD 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.

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 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

$object\{C\}$.

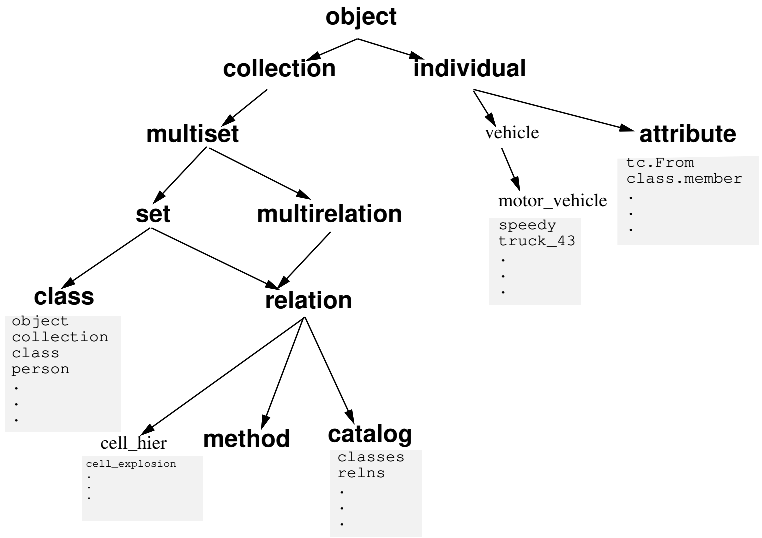

Figure 1: A Partial Class Hierarchy in Noodle.

The class *object* is not available within the O++ language. It is implemented in SWORD by looking up (at run time) the extents of all classes visible to the program at the time of compilation.

The built-in class *collection* is the class whose extent is the set of all collection *objects*. (The class does not contain any collection *values* that are not objects, since it is not feasible to keep track of which or them exist at any point in the evaluation of a program.)

The built-in class *individual* is the class whose members are all the individual *objects*. (As above, it is not practical to include in the extent individual data items that are not objects.) The class is implemented by looking up at run time the extents of all classes that contain individual objects, and are visible to the program at the time of compilation.

In Figure 1, the *person* class is a subclass of *individual*. The class *employee* is a subclass of *person*, and contains the individual objects *clinton* and *gore*.

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 can be of several types: *sets*, *multisets*, *relations* and *multirelations* are listed in Figure 1, but other types of collections (such as lists) are feasible. A *multirelation* is a multiset of tuples; tuples do not have object identifiers. A *relation* is a set of tuples, and is therefore a subclass of set as well as of multirelation.

A user may define subclasses of the built-in classes. One may define subclasses of *set*, for example *set-of-employee* and *set-of-dgraph*. The class *cell_hier* shown in the Figure 1 as a subclass of relation is a user-defined class containing objects that are relations representing hierarchies of cells. The definition of the class may specify, for instance, that members of this class must have tuples with the attributes *cell* and *subcell*. The *cell_explosion* relation of Section 3.4 is a member of this class.

Multisets of objects are actually represented by multisets of object identifiers. The extent of class *class* is thus a set of object identifiers, and hence it is a subclass of the *set* class. The user may define other subclasses of *set*, for example *set-of-employee* (that is, set of pointers to employees) and *set-of-dgraph* (that is, set of pointers to dgraphs).

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

$$class\{C\}.$$

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, class, set, catalog, individual, person, employee*, and *attribute*.

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

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

holds. We use *subclass* to denote the direct or indirect subclass relationship (the transitive closure of *@subclass*).

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.

4 Advanced Features of SWORD

4.1 Complex Objects

SWORD models complex objects (sets and functional terms) in the same way as C++: Build an object X for the set or functional term, where X has its own unique object-identifier. For example, an extensional set of talented students, $\{stud\_1, stud\_2, stud\_3\}$, may be modeled by

set<persistent student*> *talented_students* = $\{stud\_1, stud\_2, stud\_3\}$

The name and the object identifier of the set is *talented_students*. Unnesting of sets is done simply by referring to the elements of the set, as in

```
(r3):  persistent view relation<struct{persistent employee* prof; persistent student* advisee;}> students_of {
         select {prof X, advisee Y}
         where stud_set{prof X, students S} ,,
                S{Y}
       }
```

stud_set relates a professor X to the set S of students he advises. The atom $S\{Y\}$ then retrieves the object identifiers Y of students is set S.

Functional terms are modeled as individual objects or complex values. For instance, an address may be represented as the object

address_1[[*street* "600 Mountain Ave", *City* "Murray Hill"]]

with *address_1* being the object identifier. Components of the record are accessed by referring to the elements of the individual object, as in

```
(r6):  persistent view relation<struct{persistent char* empName; persistent char* city;}> empCity {
         select {empName N, city C}
         where employee{name N, address A} ,,
                A[[city C]]
       }
```

4.2 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. These variables must be declared to range over attributes in the rule body. Thus

$attribute\{A\}$,, $vehicle\{A$ b, $engine\_capacity\ E\}$

can be placed in the where clause. The variable A ranges over all attribute objects, and the above clause retrieves engine capacities of vehicles that have the value b as the value of at least one attribute. An alternative to using $attribute\{A\}$ above is to declare A to be of type `logical attribute A`.

4.3 Methods

Methods can be invoked from declarative code; the *this* argument which is implicit for all methods, and the return value of a method must both be referred to explicitly in the declarative code. Suppose class *Employee* had a method *income()* The following is an example of how to use it in declarative code

$$income(this\ Emp, return\ I)$$

However, methods could potentially have side effects, and such methods should not be used in declarative code. Also, methods whose return value depends on the state of some global variables should not be used in declarative code since optimization could result methods being called in a state where the global variables are different. Hence, to be used in declarative code, a method should (a) have no side effects or only benign side effects, and (b) should not depend on the value of any global variable that may vary over time. We are working on annotations similar to the C++ `const` declaration to specify, and to verify, that a method satisfies the above conditions.

Declarative code can be used to define methods since declarative code can be embedded in O++ code. The benefit of a declarative definition is that the optimizer in the SWORD system can be used to optimize evaluation without the intervention of the user. Consider the view *income_view* (Example 3.2); the view takes an object identifier as an argument and returns a value for income. It would be convenient to have syntax that says that the view *income_view* defines the method *income*. We are working on extending the SWORD syntax to allow such a feature.

5 Semantics

5.1 Logical Variables

Within a SWORD statement we may refer to both logical variables and to `O++` variables from an outside scope. We impose a range-restrictedness condition on logical variables. There must be some ordering of the atoms in the where clause such that the leftmost occurrence of each logical variable is either

- The value of an attribute (or member of some collection) in some non-negated and non-O++atom, or
- One side of an "==" expression whose other side contains no unbound logical variables.

There are three reasons for this restriction.

The first reason is efficiency. An atom of the form $X[[nameY]]$ would mean that X is bound to each object and complex value in the database. It would be impractical to find all such complex values. If it is desired that X be in some class *person*, say, an atom $person\{X\}$ can be introduced to ensure range-restrictedness.

The second reason is due to the presence of logical variables in `O++` expressions. The expression $(X < 3)||(X > 7)$ is an `O++` expression that tests whether a particular binding for X is either less than three or greater than seven. It is only a test; the value of X must be bound before this test is reached since it is unreasonable to expect this expression to generate (an infinite number of) possible X values.

The third reason is due to logical variables in negative SWORD literals. We cannot expect a negative literal to generate (a possibly infinite number of) bindings for its logical variables. For safety, we restrict negative literals to be used only as a test, and require that their logical variables be bound before they are evaluated.

The above restriction may be too strong in some circumstances, when only queries that specify some values need to be allowed. We are currently working on language extensions and ways of relaxing the range-restrictedness requirement in order to allow such queries.

5.2 Atoms and Rules

Any `O++` expression is an atom, whose value is the boolean truth value (0 for false, nonzero for true) of the expression, according to the semantics of `O++` expressions. We allow logical variables to appear in an `O++` expression in any place that an `O++` variable would be legal. In addition, we allow atoms of the form $p[[att_1 \;\; val_1, \cdots, att_n \;\; val_n]]$, $q\{X\}$ and $r\{att_1 \;\; val_1, \cdots, att_n \;\; val_n\}$. A *literal* is either an atom or a negated atom.

While we do not restrict atoms to be side-effect-free, we make no guarantees about the order in which atoms are processed, and the number of times an atom is processed. Thus the use in rules of methods with side-effects is not encouraged.

Let us suppose, for the moment, that there are no views defined by rules, so that all collection references are to *stored* collections. Then the (operational) semantics of a rule of the form

select $\{att_1 \; X_1, \cdots, att_n \; X_n\}$
    where A_1 ,,$\cdots$,, A_m

is the (multiset) projection onto $X_1, \cdots, X_n$ of the (multiset) join[1] of the atoms $A_1, \cdots, A_m$. The multiplicity of (true) non-multirelation atoms is taken as 1, and the multiplicity of multirelation atom instances (i.e., tuples in the multirelation) corresponds to the number of occurrences of that tuple in the multirelation.

If some of the A_i above were negated atoms rather than atoms, then we would take the multiset join of the positive atoms, and delete all tuples in the result that did not satisfy the negated atoms.

5.3 Views

Now let us define view collections by rules, and let us allow view collections to be mentioned in the body of rules. We allow mutually recursive views to be defined *within the one source file*. The semantics of the view is defined as the least fixpoint of an operator corresponding to the well-founded semantics [VGRS91] extended to handle multirelations. We allow monotonic aggregation [MPR90], modularly stratified aggregation, and modularly stratified negation [Ros90].

Views in one source file may refer to views defined in another file. However, if *mutual* recursion occurs between two or more different source files, then we do not guarantee that the well-founded semantics of the union of the source files will be computed. In order to be able to perform separate compilation, we must assume simply that all external collection references can be treated as if they were extensional database lookups. (A similar assumption is made by the CORAL system [RSS92].)

6 A Composer Example

We illustrate several of the features of SWORD using an example of a composer database originally from [LVZ92]. The database has four classes, namely *person*, *composer*, *composition*, and *instrument*, related as shown in Figure 2.

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.

```
persistent class person { //person is a subclass of individual
                  string name;
                  date birthdate;
};
```

[1] Strictly speaking, if we allow variables as names of objects and collections, then we need to use the α operator of [Ros92] in addition to joins.

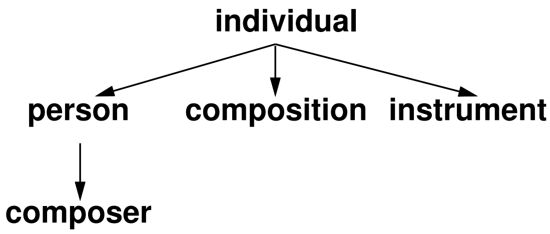

Figure 2: Composer Fragment of the Class Hierarchy

```
persistent class composer : public person { //composer is a subclass of person
            persistent composer* master;
            set<composition*> works;
};
```

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

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

assuming primitive object classes `string`, `int`, and `date`, and an implementation *age_impl* for the method *age*.

The definitions lead to the following entries into built-in classes and relations:

$class\{person\}, \quad class\{composer\}$
$@subclass\{super\ person, sub\ composer\}$
$@subclass\{super\ individual, sub\ person\}$

and a definition of the classes *person* and *composer* with appropriate entries in the *classes* catalog.

The query "Find the title of works by Bach including a harpsichord and flute" can be expressed as:

```
(Q1):  select {T}
    where composer{name "Bach", works Ws} ,,
          Ws{W} ,, W[[title T, instruments Is]] ,,
          Is{I1} ,, I1[[name "harpsichord"]] ,,
          Is{I2} ,, I2[[name "flute"]]
  }
```

Using the shorthand introduced in Section 3.1 for accessing attributes of objects, we can also write:

```
(Q1'):  select {T}
    where composer{name "Bach", works Ws} ,,
          Ws{title T, instruments Is} ,,
          Is{name "harpsichord"} ,,
          Is{name "flute"}
  }
```

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. We define the view *influencer* as follows:

```
(P):  persistent view relation<struct{persistent composer* master; composer disciple; int gen;}> influencer{
      (P1)    select {master M, disciple D, gen 1}
              where composer{D} ,,
                    D[[master M]]
           union
      (P2)    select {master M, disciple D, gen G}
              where influencer{master M, disciple Y, gen N} ,,
                    composer{D} ,,
                    D[[master Y]] ,,
                    G = N + 1
              }
```

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

```
(Q2):  select {N}
     where influencer{master M, disciple D, gen G} ,,
           G ≥ 6 ,,
           D[[name N]] ,,
           M[[works Ws]] ,,
           Ws{W} ,,  W[[instruments Is]] ,,
           Is[[name "harpsichord"]]
  }
```

7 Implementation

The SWORD system integrates the declarative statements described above with the O++ procedural language of the ODE database system. A view definition statement can be placed anywhere a function can be defined in O++. The select expression is treated as any other C++ expression, with a type constructed from the type(s) of the selected elements. An O++ expression can use a view name, or a select expression, wherever it is legal to use an expression of the type of the view or the select expression. Views and select expressions can be used to write methods associated with classes, and to query the database within the trigger language of ODE to check whether certain conditions for the triggering action are true. O++ expressions can be used as conjuncts in the where clause of the select expression. O++ variables, declared outside the scope of the select expression, can be used inside the select expression, just as they are used in any other O++ expression.

Figure 3 describes the architecture of the SWORD compiler for processing SWORD programs. A SWORD program consists of O++ statements, select expressions, and rules, all of which may reference views defined by rules. The SWORD compiler includes the ofront preprocessor of the ODE database system. The ofront module, modified to handle references to views, is used to compile the procedural O++ statements. A declarative statement or expression is processed by a SWORD module that checks for syntactic correctness, does rewrite and plan optimization, and then writes out a procedural computation as a block of O++ statements. This block of O++ statements is immediately processed by ofront. The SWORD modules reference the database schema, stored partly in the database catalog, and partially in the symbol table constructed during compilation.

SWORD views are implemented by deriving O++ methods to populate the view, to look up materialized views, to incrementally maintain views (when possible), and to update the view (when possible). An O++ collection class is declared for each view definition. A collection object, whose symbolic object identifier is the name of the view, is then created in this collection class, and may be populated when needed.

References to views in O++ expressions are translated into either (1) a fresh computation of the view, or (2) a look up of a materialized version of the view, in case materialization is possible and profitable. When

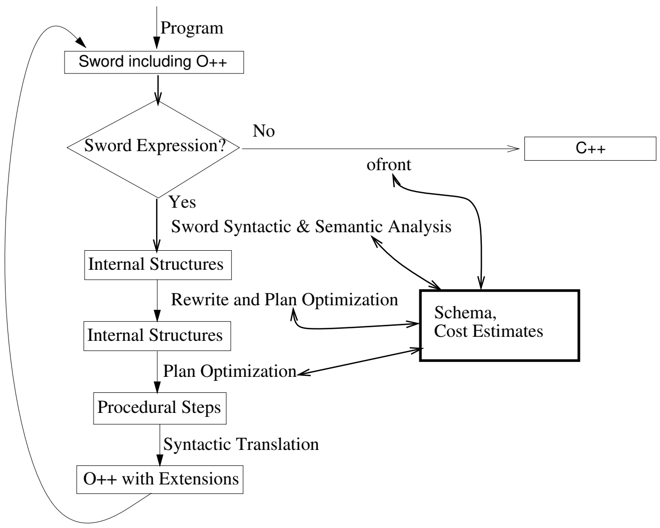

Figure 3: The SWORD Software Module

a view $v1$ is referenced inside another view definition $v2$, we also have the option of in-lining $v1$ to optimize computation of $v2$. The exact choice depends upon the optimizer.

The C++ [Str91] data model, on which the ODE data model is based, does not view classes as objects. However, ODE stores a catalog of all classes, along with attributes of interest for the class. We view each entry in the catalog as some class object, thereby supporting the idea of class objects, and map the ODE catalog to the *catalog* relations in our model. ODE stores objects in a class together in a cluster, permitting an easy mapping from our view of a class as a relation of member objects.

Variables in O++ are strongly typed. With the compiled architecture of SWORD, a type must be inferred for each untyped logical variable in a SWORD rule/expression. We do so by looking up the schema, as visible in the database catalogs and the compilers symbol table, at the time of compilation. Often, a SWORD logical variable can take on two or more types; in case the type of the logical variable has been declared, all of these must be subtypes of the declared type. For instance, in query $Q3$ in Section 3.4, the variable V takes on each subtype of *vehicle* that has the attribute *engine_capacity*. In such cases, the SWORD compiler generates versions of the rule for each possible type of V. A subsequent schema change then requires a recompilation.

Schema Querying The query $Q3$ discussed above introduces the additional burden of schema querying into compilation of queries that have no explicit schema querying. To compile such a query, we look up the system catalog to determine which subclasses of *vehicle* have attributes *engine_capacity* and *name*. We then produce code to access the *engine_capacity* and *name* fields of objects in these subclasses. The code compiles in the offsets for these attributes, and the offsets can be different for each subclass.

If the compiled query is stored, and the schema undergoes changes, the compiled code would produce incorrect answers. If a new subclass gets the *engine_capacity* and *name* attributes, it will not be included in the answer. If a subclass loses one of these attributes, the query may generate an error.

We take the philosophy that queries must be recompiled after schema changes that affect the queries.

8 Related Work

Several efforts to integrate declarative querying in object-oriented database system have been made, and several others are in progress [Abi89, Bee89], COL [AG88], Reloop [CDLR89], IQL [AK89], the O_2 query language ([BDK92], Chapter 11), OSQL in the IRIS system [WLH90], Logres [CCCR+90], F-Logic [KLW90], HiLog [CKW92], Complex [GLR92], XSQL [KKS92], CQL++ [DGJ92], ZQL[C++] [Bla93], Orlog [JL93],

$\mathcal{QUIXOTE}$ [YTM93], CDOL [KU93], A DOOD Ranch [UD93], Coral++ [SRSS93], and Noodle [MR93]. The declarative sublanguage of SWORD is based on Noodle. In this section, we compare the salient features of SWORD with several of the above systems.

SWORD 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 ([Abi89, Bee89], 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.

SWORD is based on HiLog, a logic that provides a first order semantics to apparently second order logical formulae. SWORD extends HiLog to support inheritance and object identity. SWORD 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 SWORD has a clear distinction between subclasses and objects in a class, 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 $Q4$ (Section 3.4) cannot be expressed.

SWORD 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.

SWORD constructs sets and multisets as complex objects, with each set having its own object identifier. In contrast, Reloop, IQL, the O_2 query language, XSQL, Orlog, and Coral++ construct 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.

SWORD, 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. It is our intention to extend SWORD with object creation primitives.

SWORD uses a uniform logic-based syntax to refer to relations, classes, and complex objects. Path traversal in complex structures can be done either (1) through logical atoms that can be understood and optimized by the SWORD compiler, and can be used for schema querying, or (2) through C++path expressions that return a boolean type. 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++], as in SWORD, 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 [Ull91] 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, SWORD is compositional.

9 Conclusions and Further Work

We have designed a declarative object-oriented database system, by integrating a powerful declarative language into the ODE database system. The result is the SWORD database system that combines the data modeling abilities of the object-oriented data model with the querying and collection handling abilities of the relational and deductive data models, combining the procedural C++ language with features from the declarative SQL and other logical languages in a seamless manner.

A preliminary version of SWORD has been implemented, and further work is currently in progress.

There are a number of issues that would be addressed as SWORD develops. An obvious issue is optimization. Other issues include:

View Classes The current system does not create new objects in views. Consequently, view classes cannot be defined. Set and record objects cannot be created. Support for object creation requires the system to generate object identifiers for newly created objects, and to maintain the link between the objects and their object identifiers so as to be able to recreate the association later. How efficiently can such object identifiers be supported?

View Maintenance We would like to maintain views in response to changes to stored data. The algorithms of [GMS93] can be used to maintain view multisets, and some restricted cases of view classes.

View Updates The procedural language should be able to update stored objects through views by defining update methods for the views. The update methods can be written by the user in the procedural language. However, for declaratively specified views, we should in some cases be able to derive update methods, just as incremental view maintenance algorithms can be derived.

Declarative Method Definitions We are currently working on the syntax for declarative implementation of methods, and on annotations to specify that a method is safe for use in declarative code that may be optimized.

Query Forms The range-restrictedness requirement on rules is too strong in some circumstances where only query forms that bind some values need to be allowed. We are working on language extensions and ways of relaxing the range-restrictedness requirement in order to allow such rules.

Lists, Arrays, and Sequences are examples of other collection types. We would like to manipulate these types in a declarative manner, integrating their manipulation with manipulation of sets and multisets.

Acknowledgements

We thank Surajit Chaudhuri, Narain Gehani, Richard Greer, Michael Kifer, Dan Lieuwen, H. V. Jagadish, Nandit Soparkar, Avi Silberschatz, and Jeff Ullman for comments and suggestions.

References

[Abi89] 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.

[AG88] Serge 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.

[AG89] Rakesh Agrawal and Narain 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.

[AK89] Serge Abiteboul and Paris 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.

[BDK92] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, Washington D.C., 1992.

[Bee89] Catriel Beeri. Formal models for object-oriented databases. In *Proceedings of the DOOD89*, pages 370–395, Kyoto, Japan, December 1989.

[Bla93] Jose A. Blakeley. ZQL[C++]: Integrating the C++ language and an object query capability. In Inderpal Singh Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 138–144, Washington, DC, May 29 1993.

[Cat91] R. C. G. Cattell. *Object Data Management.* Addison-Wesley, 1991.

[CCCR+90] F. Cacace, Stefano Ceri, S. Crespi-Reghizi, Letizia Tanca, and Roberto 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.

[CDLR89] Sophie Cluet, Claude Delobel, Christophe Lécluse, and Philippe Richard. Reloop, an algebra-based query language for O_2. In *Proceedings of the DOOD89*, Kyoto, Japan, December 1989.

[CKW92] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 1992.

[DGJ92] Shaul Dar, Narain 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.

[GLR92] Sergio Greco, N. Leone, and P. Rullo. Complex: An object-oriented logic programming system. *ACM Transactions on Knowledge and Data Engineering*, 4(4):344–359, August 1992.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[JL93] M. Hasan Jamil and Laks V. S. Lakshmanan. Realizing Orlog in $\mathcal{LDL}$. In Inderpal Singh Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 45–59, Washington, DC, May 29 1993.

[KKS92] Michael Kifer, Won Kim, and Yehoshua 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.

[KLW90] Michael Kifer., Georg Lausen, and James 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.

[KU93] A. Karadimce and Susan 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, March 1993.

[KW89] Michael Kifer and James 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.

[LVZ92] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed 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.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, Brisbane, Australia, August 13-16 1990.

[MR93] Inderpal Singh Mumick and Kenneth A. Ross. Noodle: A language for declarative querying in an object-oriented database. In *Proceedings of the DOOD93*, Phoenix, AZ, December 1993.

[MR93] Inderpal Singh Mumick and Kenneth A. Ross. Noodle: A language for declarative querying in an object-oriented database. In *Proceedings of the DOOD93*, Phoenix, AZ, December 1993.

[MSOP86] David Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *OOPSLA 1986 Proceedings*, pages 472–482, 1986.

[OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don 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.

[PDR91] Geoff Phipps, Marcia Derr, and Kenneth 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.

[Ros90] Kenneth 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.

[Ros92] Kenneth A. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 346–353, San Diego, CA, June 2-4 1992.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, relations and logic. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB)*, pages 238–250, Vancouver, Canada, August 23-27 1992.

[SRSS93] Divesh Srivastava, Raghu Ramakrishnan, Praveen 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.

[Str91] Bjarne Stroustrup. *The C++ Programming Language (Second Edition)*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[UD93] Susan D. Urban and Suzanne W. Dietrich. A deductive, object-oriented model as a formal framework for active database environments. In Inderpal Singh Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 101–110, Washington, DC, May 29 1993.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1*. Computer Science Press, 1988.

[Ull91] Jeffrey D. Ullman. A comparison between deductive and object-oriented database systems. In *Proceedings of the DOOD91*, pages 263–277, Germany, December 1991.

[VGRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[WLH90] Kevin Wilkinson, Peter Lyngbaek, and Waqar Hasan. The Iris architecture and implementation. *ACM Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

[YTM93] Kazumasa Yokota, Hiroshi Tsuda, and Yukihiro Morita. Specific features of a deductive object-oriented database language $\mathcal{QUIXOTE}$. In Inderpal Singh Mumick, editor, *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 89–99, Washington, DC, May 29 1993.