Class Algebra as a Description Logic

Daniel J. Buehrer and **Yi-Huang Liu** and **Ting-Yue Hong** and **Jeng-Jong Jou**
Institute of Computer Science and Information Engr.
National Chung Cheng University
Chia Yi, Taiwan, R.O.C.

Abstract

A class algebra has been used to model the query and data manipulation language for a prototype object-oriented database system which is being implemented for the World-Wide Web [Bue95]. The Prolog prototype (i.e. operational semantics, if you will) is located at *http : //www.cs.ccu.edu.tw/ ~ dan/prologProgs/modeloo.txt*. The class algebra includes a Boolean algebra for class union, intersection, and complement (i.e. difference with any). This Boolean algebra is extended with dot (partof), cross product, projection, and selection to make it subsume relational algebra. All classes which are defined with these class expressions have an *extent* which is the set of all objects that are described by the class expression. As such, this class algebra is a kind of description logic. Moreover, subclass definition, variable and function declaration, data assignment, function and relation definition, and other side-effect operators such as insert/delete are defined for this class algebra. A comparison is made between this class algebra and ODMG-93's OQL [Cat96], SQL, Prolog, and first-order predicate logic.

1 Introduction

A *description* is an expression in a formal language that defines a collection. A *description logic* [Mac94] is a language for constructing such descriptions, as well as the semantics that define the collection which corresponds to each description. As such, the class algebra of [Bue95] is a description logic, since it defines the *extent*, or set of instances, of any class expression which is formed out of union, intersection, difference, cross product, project, select, dot [Tsu91], property declaration, assignment, and function definition operators. This class algebra satisfies all of the properties listed by [Yu91] for evaluating algebraic object-oriented query models. That is, the class algebra has ID's and ID manipulation, encapsulation, inheritance, extends relational algebra, includes behavior constructors, invokes behaviors, has dynamic type creation, can query transitive closures, has formal semantics, is a closed algebra, has strong typing, and has optimization strategies. Since the algebra includes invocation of behaviors which may have side-effects, such as assignments and declarations, the algebra should technically be called an *evolving algebra* [Got91] or a calculus. But since at any given time, the database is an algebra with specific constants and functions, we still simply call it a class algebra.

The main feature of this database model is that the algebra involving the classes and typed collections is closed [Kim89] under standard database operations such as union, intersection, difference, select, project, dot, append, and assignment. Collections are homogeneous; i.e. the elements of the collection must all satisfy the invariant of the collection's class. However, unlike other models [Gog92], such as the relational model, in class algebra the collections being unioned, intersected, or differenced do not need to have the same class. (As in the ODMG-93 standard [Cat96], the database collections include bags, sets, arrays, lists, and their subclasses.) The resulting collection's class is calculated using a sorted disjunctive normal form of class names. This model derives the most specific class for the resulting collection, permitting the user to know what attributes and methods are available for each of its elements. For instance, unioning a list of people and a set of students will produce a set of people.

Each of class algebra's *class descriptions* involve a set of superclasses, optional keys, a list of attribute definitions, a list of binary relationship definitions, a prototype, and a class invariant. The ODMG-93 standard ODL language [Cat96] is the same, except that it does not include a prototype or a class invariant.

All attributes in the database are formed by using a declaration *qualified_name* inClass *class_expr*. All relations are binary, and are formed by using a declaration *qualified_name* inverse *id* inClass *class_expr*. This declaration provides necessary conditions, but not sufficient conditions, for values which can be dynamically assigned to the attribute or binary relation. The *class_expr* in assignment statements, on the other hand, provide both necessary and sufficient conditions for the current membership. There are two kinds of assignments in this class algebra. The static assignment

qualified_name := *class_expr* is used to change the existing value of the collection *qualified_name* to the value of *class_expr*. The view assignment (also called *delayed assignment*) *qualified_name* =@ *class_expr* is used to define an implicit function or relation. This function or relation's value may change as assignments change the values of attributes and relations in the database. The *class_expr* in these view assignments cannot contain data assignments or calls to any functions that might have side-effects.

The declaration and assignment statements return the new object referred to by *qualified_name*. Nested environments can be created by using the expression *qualified_name*#(*exprs_separated_by_semicolons*), where the # symbol is used as the *dot* operator. Assignments' left-hand sides are relative to a *current environment* which can be changed either by using the nested environments or by using a Unix-like cd (i.e. change-directory) command. *parent* is a reserved word pointing to the .. object containing this object, and *world* is a reserved word pointing to the / root object. If an object is deleted, all of its subobjects and pointers to those subobjects are also deleted.

Each class has the attributes *extent, invariant, keys, prototype,* and the relations *superclasses* and *subclasses*. New keys can be assigned to the class by assigning or inserting into *world*#*classes*[*className*].*keys*. The relations *world*#*classes*[*className*].*superclasses* and *world*#*classes*[*className*].*subclasses* are calculated from the *class_expr*. Their calculation involves the use of a sorted-disjunctive normal form which permits the algebra to closed under the above operations. This is described below. Finally, the attribute *world*#*classes*[*className*].*prototype* can be assigned a value which is used for default values of the corresponding attributes.

A subclass declaration *subclass* <@ *superclass* has the effect of inserting *world*#*classes*[*subclass*]. All assignments to new objects (i.e. attributes or relations of other objects) which have been declared to be in the extent of such a class are first checked to make sure that they satisfy the class invariant. Unlike some other description logics, we do not automatically reclassify objects to their *lowest subclass*, although such automatic reclassification by using class invariants is theoretically possible if the invariants are considered to be both necessary and sufficient conditions for membership in the class's extent.

In Section 2 we provide a more formal definition of the class algebra. In Section 3 we describe how the sorted disjunctive normal form permits the algebra to be closed under its operations. A short comparison is also made with the traditional *subsumption* of description logics. In Section 4, we compare class expressions to OQL and SQL. In Section 5, we consider some problems of using class expressions for general first-order queries such as those found in np-complete or np-hard queries. We show how the general-purpose optimization strategies of Prolog and first-order logic theorem proving systems can be included into class algebra. Finally, we summarize the advantages of using such a class algebra in Section 6.

2 An Object-Oriented Class Algebra

In this section, we present a Boolean algebra of classes and their corresponding extents (i.e. the list of all instances of the class). We are mainly concerned with finding the datatype of unions, intersections, and complements of classes, and we show that the resulting classes form a closed, finite ISA hierarchy [Run92]. Then, this Boolean algebra is extended to include the types of collections such as bags, sets, lists, and arrays. For instance, using this model, the concatenation of a bag of flowers and a set of trees produces a bag of plants. This library's closure under its collection operations makes it both user-friendly and easy to optimize.

As much as possible, we try to follow the ODMG-93 standard [Cat96], which includes multiple inheritance. As in the ODMG standard, an *attribute* has a literal value which has no object identifiers, a *relation* is always a binary relation between objects which have object identifiers (so that all references to an object are readily available), and *methods* may involve either operations or functions that may generate exceptions. When we use the word *class*, we are referring to an ODMG *type* where the extent and invariant are required rather than optional.

In our model, a class is mainly defined by its invariant, or its membership test condition. This invariant includes a formal description of the signatures of the required attributes, relations, methods, and exception handlers for elements of this class. The invariant may also describe the input-output behavior of each method or function which the class implements. The invariant can also be used as a membership function. When applied to any object (i.e. an attribute or relation of another object), the invariant will evaluate to either true or false, telling whether or not that object can be added into the *extent* (i.e. the set of all members) of the class. It states the conditions which must be satisfied by each object which can be considered *in* this class. That is, each object has both a *declared* class (from the *signature*) and an *active* class which is a subclass of the declared class. The active class is the union of the classes of the values which are currently assigned to this object.

Such an invariant is a conditional expression which involves one free variable that is assigned to the object which is to be tested for membership. That is, the invariant has the form $\{x \mid \Psi(x)\}$ for some first-order formula Ψ which may itself involve such expressions. In general, when performing theorem-proving with such formulas, the function symbols may become nested arbitrarily deeply, and an arbitrary number of new variables may be introduced. However, in our case, the only rules which are used have the form such as

$$newclassname(x) \rightarrow superclassname(x) \& \Psi(x) \quad (1)$$

We will ignore the proofs which involve the invariants $\Psi(x)$, and will simplify the rules to include only ground

cases for which $\Psi(x)$ is true, so there will not be problems with infinite nesting of functions. The theorem-proving involving these rules then remains finite, and there is an algorithm for simplifying the Boolean combinations of such invariants. The above rules basically form the ISA hierarchy which our model uses for describing both the inheritance of attributes and methods, and for describing *union/intersection/difference* relationships among the extents of the classes.

3 The Sorted Disjunctive-Normal Form

First, let us consider class expressions involving union, intersection and difference operators. Then we will extend this algebra to include other operators.

Assume that every class invariant is written as a Boolean expression involving class names. Such a Boolean expression has a disjunctive normal form, and some theorem-proving techniques can be used to simplify this normal form. For example, subsumption can be used to eliminate any conjuncts which contain more classes than another conjunct. For example, in the disjunction

$$students\&people\&teachers \vee people \tag{2}$$

the *students&people&teachers* conjunct is subsumed by the *people* conjunct (i.e. the union of student-teachers and all people is all people). Another simplification is that superclasses may be eliminated from conjuncts. So, for example, *students&people* simplifies to *students*. A theorem-proving technique such as resolution can generate all of the consequents of the disjunctive normal form, and then subsumption can be used to eliminate unneeded disjuncts and literals. The result is a set of prime implicants. Each conjunct's classes are sorted, and then the disjunction of these conjuncts is also sorted to come up with a unique normal form. For example, although the disjunctive normal forms

$$x\& \sim y \vee y\& \sim z \vee z\& \sim x \tag{3}$$

and

$$x\& \sim z \vee y\& \sim x \vee z\& \sim y \tag{4}$$

both represent the same Boolean value, they have different forms. But, when starting with either expression and deriving all of the unsubsumed consequents, and then sorting, you will arrive at the unique normal form

$$x\& \sim y \vee x\& \sim z \vee y\& \sim x \vee y\& \sim z \vee z\& \sim x \vee z\& \sim y \tag{5}$$

The above normal forms are the labels of nodes in a network structure which is the ISA class hierarchy of the class expressions. Our model uses this ISA hierarchy for both the type hierarchy and the subset hierarchy. The sorted disjunctive normal form is used in a generalization of the subsumption operation which is usually used in description logics (e.g. [Wei94]). That subsumption is only used for conjunctions, which are the datatypes of objects. Description logics do not usually extend the subsumption to include union and complement operators.

A class union operation unions the extents of its subclasses, its invariant is the disjunction of the invariants of its superclasses, and its required properties (i.e. attributes, relations, operators, functions, and exception handlers) are the intersection of the superclasses' required properties. When unioning collections, the signatures of the required attributes and relations are intersected if they have similar names. Similarly, a class intersection operation intersects the extents of its subclasses, its invariant is the conjunction of the superclasses' invariants, and its required properties are the union of the superclasses' required properties. The signatures of attributes or relations with the same name are also unioned in the new class, and the original individual attributes or relations are still available by using C++ like notation Class::attr or Class::reln.

For each class expression involving union, intersection, complement, and difference operators, we can therefore derive a unique node in the ISA hierarchy which describes that class expression. The class expression's extent is the union of the extents of itself and all nodes underneath it in this ISA hierarchy. Its set of required attributes, explicit relations, and methods (i.e. operators, implicit relations, and exception handlers) are the union of its own plus those of the classes above it in the ISA hierarchy.

We now want to extend this definition to arrays, lists, bags, and sets. We follow ODMG's Object Definition Language (ODL) standard [Cat96] for object declarations. The union operation is used to compute the class of the elements in the concatenation of arrays, lists, bags, and sets. For instance, an array of people concatenated with an array of buildings will produce an array of physical_objects, if that is the least upper bound of buildings and people in the ISA hierarchy. Inserting a single element into an array, list, or bag, will also use the union operation to calculate the datatype of the resulting array, list, or bag. Deletion of elements from collections will leave the collection's type unaffected.

Every collection of objects with oids is stored as a binary relation, so each object has inverse relations pointing to all collections in which it is contained. Extents are implemented as lists. Lists have an iterator which can take values from 0 up to the number of elements in the list, and an insert and remove operator which change the locations of elements in the list. A *bag* is a list whose key can be used to identify a set of elements of the extent. A *set* is a bag where the key identifies a unique element rather than a set of elements. If the keys are integers, then the set is called an array. Since arrays are a kind of set, a set is a kind of a bag, and a bag is a kind of a list, append or insert operations which are performed on combinations of collections will result in the more general type. For instance, inserting a $list < A >$ into an $array < B >$ will result in a $list < A \cup B >$. Appending an $array < A >$ to a $set < B >$ will result in a $set < A \cup B >$. Notice that all attributes of $A \cup B$, including the key, also union their types, so the resulting set has both character keys and integer keys intermixed.

Union operations may cause key conflicts, which require the user to use C++ double colon notation to indicate which superclass to use. A *groupby*(*propertynames* ...) method turns a bag into a set with the given propertynames as keys, where each value is itself a set. The *ungroup*() function turns such a set of member relations back into a bag with repeated entries for the keys.

4 A Comparison to OQL and SQL

The ODMG standard framework includes an Object Query Language (OQL) which has some similarities to SQL, but which is based on the object-oriented paradigm. OQL contains a SELECT statement for asking queries. In our model, the *class_expr* where *Boolean_expr* statement can select from any class expression according to the *where* clause, and it will return either a collection or a single value (e.g. when *class_expr* is an aggregate function or *first*() or *next*() function). The statement *subclass* < @ *class_expr* where *Boolean_expr* is used to define a new subclass whose invariant is the given *Boolean_expr* conjoined with the superclasses' invariants.

The SELECT statement of SQL and OQL also takes the cross product of the FROM relations. In our model, the cross product operation produces a collection of objects whose class is the intersection of the classes being crossed. For example, the cross product of name, address, telephone number, and age objects is a set of objects whose type is name&address&telephoneNum&age. This set is formed in the normal way, by finding the set of all tuples whose i-th element is contained in the i-th input set. Of course, such a set is very large, so algebraic simplification should be used to move the selection operation into the cross-product generation, so that an intelligent method can be used to generate only the desired objects. Our cross product operator is associative, as in [Loi91].

Finally, the SELECT statement of SQL and OQL hides all attributes except those listed after the SELECT keyword. In our model, this projection operation is modeled by overriding the assignment or *getval* operations for the hidden attributes or relations. Similarly, other operators such as update, append, open, delete, etc. can also be made to return errors if the user does not have the appropriate access privileges. That is, both projection and security are handled by overriding corresponding operators' definitions. However, care must be taken to also override the :: operator to first make sure that unqualified users cannot access the old definitions. Since it is understood that all objects may either satisfy the conditions of the invariant or have an error value, such overriding does not invalidate the superclasses' invariants.

The expressions of OQL, the object-oriented query language of the ODMG-93 standard, are very similar to the expressions of the class algebra. OQL contains operators for union, intersection, and complement of sets. OQL does not extend this to the other kinds of collections (i.e. bags, lists, and arrays). Moreover, the result of these operations are simply a set rather than a class. This set's type is not explicitly defined in the ODMG-93 manual, so it could be defined using the sorted disjunctive normal form described in Section 3 above, thus leading to a query language which is closed under these operations. Also, the method of resolving name conflicts, especially among keys, is undefined in the ODMG-93 manual. We use the C++ notation to disambiguate names which occur in more than one class. That is, all property values are accessible for each object, even if several inherited properties have the same name. The properties can be referred to by using a C++ double colon notation class::property_name.

OQL does not define assignments and control. This is left for the binding to a particular language such as C++, Java, or Smalltalk. Our class algebra already includes assignment, function, and relation definition, and simple control structures such as *if* and *while* statements.

5 General First-Order Queries using Class Expressions

First-order logic and Prolog also can serve as a convenient query and database manipulation language. Many current versions of Prolog contain an ODBC interface to relational databases. Predicates can be declared to be database relations. Thereafter, assertions and retractions of these predicates will send the appropriate SQL commands over the network to make the corresponding changes in the database. Prolog queries look the same regardless of whether data is retrieved from the database or from memory.

There has also been a large amount of research on how to optimize Prolog-like queries. For example, predicates may be rearranged and intelligent backtracking can skip over useless backtrackings. The magic-sets algorithm can rewrite a Prolog program into a more efficient program by knowing which arguments are bound in the query. Relaxation algorithms can successively find larger tuples of variables by making use of partial constraints. There is also an algorithm to find all answers of Datalog programs (i.e. Prolog without function symbols) in time $O(n^k)$, where k is the depth of nesting in the constant propagation graph.

The nested views of our class algebra correspond to Prolog programs which have no loops in their *constant propagation graph*, where variables occurring in the same predicate or in unifiable predicates are connected by an edge. That is, the constant propagation graphs form a lattice structure. This is the source of the class algebra's efficiency (as is true for other description logics), but is also a source of limitations. The class algebra expressions can also be optimized (i.e. simplified), but statements with side-effects are not easy to simplify. The functional set and class operations are generally insufficient for programming complicated control such as is found in the magic-sets algorithm. When the class algebra is extended with other control mechanisms, any

algorithm can be expressed, but it is difficult to make use of general-purpose optimization strategies.

Class algebra's *where* operator takes a Prolog-like goal as the second argument, and we can use traditional theorem proving optimization techniques to generate the instances of the first argument which satisfy the goal. This retains the advantages of the declarative nature of the functional operators of the class algebra, permitting use of sophisticated simplification and optimization techniques to find more efficient programs for simply stated problems.

For example, the maximum clique problem and the coloring problem can make use of such Prolog and theorem- proving optimizations combined with algebraic simplification techniques:

```
max_clique(Graph) =@ max(size(X where (X in
powerset(Graph.nodes), subset(X, X # otherEnd)))).
min_color(Graph) =@
min(size([A,B,C,D,...] $ asSet)
where next(A,B), next(B,C), next(C,D), ...,
(next(X,Y):- color(X), color(Y), X=/=Y),
color(red), color(blue), color(green), color(orange).
```

6 Advantages of Class Expressions

The class algebra which is described here is an easy to use object-oriented database query and manipulation language. It can be extended to fit into most languages by using a *binding* such as that used by ODMG. However, it also can have a very nice menu-based window interface. A language is usually more convenient for defining *where* conditions and function definitions, but the menu-based interface is still more convenient for programmers who might not know what attributes and functions are available in the database.

The class algebra has its own simplification strategies which can be used in combination with other optimization strategies for Prolog and automatic theorem proving.

More work still needs to be done on using this class algebra in a distributed environment. There seems to be no essential problem since typing is static, garbage collection is well-defined, and transactions and error-recovery are well- defined. The *parent* attribute will permit the retrieval of a unique name for each object, similar to a *url* of the World- Wide Web (WWW). This can be used for communicating with a CORBA-compliant server. There is also essentially no problem to integrate the WWW interface to this system (still under development) into hypertext and spreadsheets. However, we have not yet investigated the problems concerned with multiple caches and multiple versions. So far, such copies of objects and collections must be assigned to new objects. Perhaps a simple solution would be to define a class of *historied* objects whose assignment operators would be overridden to maintain a list of the object's k previous or cached versions. However, configurations of consistent versions would also be needed. The binary relations may point to different objects, depending on the current configuration.

It seems that the class algebra presented in this paper provides a good theoretical basis for investigating these practical distributed object-oriented database problems.

References

[Bue95] Buehrer, D.J., "An Object-Oriented Class Algebra", *Journal of Computers and Information, ICCI95* D116, 1995, pp. 669-685. Available at http:// phoenix. trentu. ca / jci/ papers/ icci95/ D116/ P001. html and http:// www. cs. ccu. edu. tw/ ~dan/ classalg. ps.

[Cat96] Cattell, R.G.G., ed. *The Object Database Standard: ODMG-93, Release 1. 2*, Morgan Kaufmann Publ, SanFrancisco, 1996.

[Gog92] Goguen, J.A.; Diaconescu, R. "Towards and Algebraic Semantics for the Object Paradigm", in *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types*, Joint with the 4th COMPASS workshop, Selected Papers, 1992, pp.1-29.

[Got91] Gottlob, G.; Kappel, G; Schrefl, M., "Semantics of Object-Oriented Data Models–The Evolving Algebra Approach", *Lecture Notes in Computer Science 504*, Springer-Verlag, 1991, pp.144-160.

[Kim89] Kim, W., "A Model of Queries for Object-Oriented Databases", *Proc. of the Fifteenth International Conference on Very Large Databases*, Amsterdam, 1989, pp.423-432.

[Loi91] Loizou, G. and Pouyioutas, P. "A Query Algebra for an Extended Object-Oriented Database Model", *Database Systems for Advanced Applications '91*, Ed. A. Makinouchi, World Scientific Publ., 1991.

[Mac94] MacGregor, R.M. "A Description Classifier for the Predicate Calculus", *Proc. of AAAI '94*, pp.213-220, 1994.

[Run92] Rundensteiner E. and Bic, L., "Set Operations in Object-Based Data Models", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 3, Aug. 1992, pp.382-398.

[Tsu91] Tsukamoto, M.; Nishio, S.; Fujio, M., "DOT: A Term Representation Using DOT Algebra for Knowledge-Bases", *Deductive and Object-Oriented Databases*, Second Intl. Conf., DOOD'91 Proc., 1991, pp.391-410.

[Yu91] Yu, L. and Osborn, S.L., "An Evaluation Framework for Algebraic Object-Oriented Query Models," In *Proc. Int. Conf. on Data Engr.*, Kobe, Japan, Apr. 1991, Vol. 1, pp.670-677.

[Wei94] R. Weida, R. and Litman, D. "Subsumption and Recognition of Heterogeneous Constraint Networks", *The Tenth IEEE Conference on Artificial Intelligence for Applications (CAIA-94).*