An Object-Oriented Class Algebra

Daniel J. Buehrer
Institute of Computer Science and Information Engineering
National Chung Cheng University
MinHsiung, Chia Yi, Taiwan 621, R.O.C.

Abstract

Relational databases can be said to be "closed" because relational queries produce relations as answers. This closure property is perhaps the main reason for the popularity of relational databases. In object-oriented terms, a relation is a homogeneous set of objects, where the primary key of the relation is the object identifier (oid). An object-oriented query often results in a homogeneous set of objects, but it may not be clear where this set of objects fits in the ISA hierarchy of the object-oriented database. Therefore, it is not immediately clear what operations apply to the query results, since attributes and methods cannot be inherited. Moreover, the query results cannot be named, stored, and reused in other queries in an efficient manner. This problem, known as the "query closure problem" [11], has plagued many existing implementations of object-oriented databases. The usual solution is to put query results as subclasses of the root "object" class, with the resulting loss of inheritance information and disorganization in the ISA hierarchy. In this paper we develop a natural way to translate relational database queries into homogeneous set operators whose values are also homogeneous sets, and which therefore have a natural place in the ISA hierarchy. This OODB model makes it easy to use results from previous queries to optimize the processing of a similar query. The model also makes automatic knowledge discovery (i.e. data abstraction and classification) much easier.

Keywords: Object-oriented database model, closure, class algebra, query optimization

1. Introduction

Relational databases have fairly simple mathematical models with the closure property. For example, in relational algebra, the results of join, project, select, union, and active complement operators are all relational tables. In relational calculus, the

Journal of Computing and Information
Special Issue: Proceedings of Seventh International Conference of Computing and Information (ICCI'95), Trent University, Peterborough, Ontario, Canada, July 5−8, 1995, pp. 669−685.

value associated with a logical formula is also a relation. In this paper we define such a mathematically closed model for object-oriented classes, where the result of class query operators is a class which has a natural position in the ISA hierarchy. This is in contrast to some existing object-oriented database models (e.g. [11]), which put query results as a subclass of the "any" class, since the query result does not naturally fit into the ISA hierarchy. For instance, a query result may be a union of subsets of existing object sets, but the query result may contain only a subset of the properties (e.g. attributes, fields, methods, functions, tasks) of the classes used in the query. The object-oriented model defined in this paper, can serve as the basis for more refined models which describe various solutions for multiple inheritance with overriding, generic function/task invocation, more advanced control and data structures, and other database functionalities.

In object-oriented programming, classes take the place of relations. When a class is materialized, it results in a set of objects which belong to either that class or one of its descendant subclasses. Notice that an object in this materialization may have been declared to have a "domain" which is a superclass of this class, but if it has the required set of properties and if they satisfy the invariant of this class, this object is included in the materialization. When the object's properties are changed, it may no longer satisfy the invariant of this class (although the assignment operator will make sure that the new value still satisfies the invariant of its declared "domain"). In our model, such a dynamic set is the materialization of a class invariant. New basic classes are defined by using a "<@ where" query operator, which adds the selection criterion onto a superclass's invariant. These basic classes are then combined using union, intersection, and complement operations to form a Boolean algebra of classes. Hiding (i.e. projection) mechanisms are then added into this algebra.

As well as having the above closure properties, the OODB query model developed in this paper can have efficient implementations, and it is well-suited for query optimization and automated knowledge discovery. We first give a brief overview of the query language in Section 2. Then the Boolean algebra for the class union, intersection, and complement operators is presented in Section 3. Section 4 describes how the "<@ where" operator can add new classes into the ISA hierarchy. Section 5 describes how hiding and assignment operators affect the algebra. Finally, a summary is presented in Section 6.

2. An object-oriented query language

There is a simple 1-1 correspondence between a first-order logic query and an object-oriented expression which computes the value of that query. This is similar to the correspondence between relational calculus and relational algebra. However, here we must also consider the question of classifying the answer so that it fits naturally into both the partof and ISA hierarchies.

Relational algebra simply requires that tables which are being unioned or intersected must have the same attribute names (columns). For class operators, there is no such restriction. That is, the class operations which are used to construct the answer must all have a well-defined class for the elements, and the result must be stored in a well-defined location in the ISA hierarchy.

First, consider an example of a mathematical set construction operator such as

{Teacher,Student | teaches(Teacher,Course), taking(Student,Course),
advises(Teacher, Student)}. (1)

This set constructor produces the set of <Teacher, Student> pairs where the Teacher is both teaching and advising the given Student. Here, the variables are untyped, although there may be implicit types due to the signatures of the relations. If so, the resulting type of each variable could be assumed to be in the intersections of the domains of the relations where the variable is used. However, the <Teacher, Student> pairs would not be of any specific type other than 'pair'. On the other hand, a set expression such as

{Student | teaches('John', Student) & Student.age>20} (2)

obviously produces a set of adult students, and we can immediately know that these objects can be enrolled in certain courses, can add and drop courses, etc., just like any other students. Moreover, making a change to the attributes of any of the adult students taught by John should also change that student's attributes globally, throughout the database. Therefore, these single-attribute query responses can be treated in a way which is quite different than sets of n-tuples.

We will see in this section how class queries can always result in what looks like a single-attribute set, where no n-tuples need to be introduced to store query results. This query result involves restricting both the top-level set and its component sets (i.e. partof attributes or inverse references). The nested sets represent the original relations, and the AND/OR/NOT operators of the query are translated into intersection/union/complement operators for these nested attributes.

Once the predicates in a first-order logic query have been arranged in an optimal manner for evaluation, there is a natural way to define class operators which will compute the correct set of n-tuples or the correct set of objects. For example, the set constructor for finding all teachers who teach students that they also advise could be written as

{Teacher | advises(Teacher, Student), teaches(Teacher,Course),
taking(Student, Course)}.

Assuming that T.teaches are the courses taught by teacher T, S.advisor is the teacher advising student S, and S.taking are the courses taken by student S, then a straightforward translation into set operators like intersection (&) might look like the following:

teachers where (teaches.inv(taking) & inv(advisor)) \= empty.

Here, the dot notation is being used to specify the desired sets. All n-ary relations are stored using binary relations. The binary relations have one direction stored explicitly as a set with a name like "attr", and the inverse relation is available by using the attribute name inv(attr). So, for instance, the set of students that have a given advisor t could be given by an expression such as t.inv(advisor). For the above example, we have rather arbitrarily chosen to store the *teaches* relation as attribute of teachers rather than courses. However, since each student has at most one advisor, it is probably better to store the *advises* relation with *student* in order to make it easier for the system to check for invalid assignments of multiple advisors to one student. The *taking* relation could be stored with either students or courses, since it is n-m. Here, we stored it with students.

What should be done for the set constructors with multiple variables? For instance, suppose that we want the set of <Teacher, Student> pairs, or <Teacher, Student, Course> triples where the teacher both advises and teaches the student. The relations "advises", "teaches", and "taking" can be thought of as the materializations of the classes with the same names, and attributes "teacher", "course", and "student" as appropriate. Then the cross product <teacher, student, course> triples can be generated by the following subclass assignment:

advises_teaching_taking <@

(advises X teaches X taking) where (advises.teacher==teaches.teacher and

teaches.course==taking.course and advises.student==taking.student)

It can be seen that such an approach, however, would produce numerous new classes which have no natural place in the ISA hierarchy. Therefore, we prefer that the user store all binary relations as invertible attributes, using "inv(attribute)" to traverse the "attribute" arc in the reverse direction, rather than using global binary predicates. The system should be responsible for maintaining the consistency between the forward attributes and the reverse "inv" attributes.

3. A Boolean algebra of classes

The basic model of a class mainly involves an invariant, or membership test condition. That is, a class is seen mainly as an implicit set, and its definition involves mainly the class invariant. The class invariant tells what are the requirements for membership in the class. The dynamic class assigned to an object must, of course, be a subclass of its statically declared domain, and it must satisfy the constraints associated with its dynamically assigned subclass, which inherits the constraints from the statically declared class. Here, class expressions involving union, intersection, and complement can be used for either static domain declarations or dynamic set assignments. The thing which makes our model different from many other models [1,8,17] is that it is based on Boolean operations for the normalized invariants of the classes rather than on the set operations for the instances of the classes.

The class invariant states what are the required (i.e. non-optional) properties (i.e. attributes and methods) of each object in the materialization. These properties must then also satisfy the other constraints which are specified in the invariant. Since the set of required properties must satisfy the rules of inheritance, we first develop a class algebra to specify the class/subclass relationships of the ISA hierarchy.

To specify the location of each class expression in the ISA hierarchy, we will assume that there is a global tree of "basic" class nodes, where each node contains a unique class identifier. We will see in the next section how to add leaves to this tree of basic classes by using a selection operator. The root node has the name "any", and it has no required properties and no invariant.

We will first consider class expressions involving only union, intersection, and active complement operators as applied to primitive class names.

Definition 1: A *class expression* is recursively constructed from the following:

A class name

x + y where x and y are class expressions (i.e. class union)

x * y where x and y are class expressions (i.e. class intersection)

~x where x is a class expression. (i.e. class active complement)

Each class expression has a unique sorted, simplified disjunctive normal form. This form corresponds to the set of sorted prime implicants when the class expression is treated like a Boolean algebraic expression. The normal forms of the expressions act as labels for nodes in the ISA hierarchy. We will prove that the normal forms satisfy the laws of Boolean algebra, so the corresponding class operations also form a Boolean algebra.

Definition 2: A *disjunctive normal form* of a class expression is constructed by repeated application of the following simplification rules (in the given order) until getting a disjunction of conjunctions of *literals* (i.e. class names or complemented class names) with a minimal number of operators. Each resulting conjunction is called a *prime implicant*.

```
/*These rules get a disjunctive form */
     x*(y+z) --> x*y+x*z        Distributivity of * over +
     (y+z)*x  --> y*x+z*x
     x+(y*z) --> (x+y)*(x+z) Distributivity of + over *
     (y*z)+x -->( y+x)*(z+x)
     ~(x+y) --> ~x * ~y         DeMorgan's Laws
     ~(x*y)  --> ~x + ~y
     ~ ~ x  -->  x
/*These rules simplify it to a normal form*/
     x*y --> y*x     if y<x  (Use commutativity to sort the conjuncts)
     x+y --> y+x    if y<x   (Use commutativity of + to sort the strings
                              of sorted conjuncts)
     x*x --> x
     x * x * y --> x * y        Factoring
     x + ~x -->  1
     x * ~x -->  0              (Assume that x and ~x are sorted next to each other)
     0 * x -->  0               (Assume 0 and 1 are sorted before other class names)
     1 * x -->  x
     0 + x -->  x
     1 + x -->  1
     x + Z  --> x   if Z is a conjunct containing the literals of x      Subsumption
     x*y + ~x*z --> x*y + ~x*z + y*z     Ordered Resolution
```

The above rules will stop with the prime implicants because the factoring rule will delete any repeated literals within the conjuncts, and the subsumption rule will delete any conjuncts with a superset of its literals contained in another conjunct. There are therefore only a finite number of such conjuncts for a finite number of literals. Once a prime implicant is subsumed, it will never be added back in, since it would still be subsumed in the future, so the replacements will eventually stop.

When the above prime implicants are sorted alphabetically by class name, then we get a *sorted, disjunctive normal form* SDN(x) for a class expression x.

Lemma 1: The sorted, disjunctive normal form of a class expression is unique.

Proof: From simplification of Boolean circuits, it is well-known that the set of prime implicants is unique up to reordering. The above procedure is a dual form of an ordered resolution theory, which typically uses the conjunctive normal form. Ordered resolution is capable of producing all prime implicants by putting terms into conjunctive normal form and then repeatedly applying the resolution rule, factoring, and subsumption until no new results can be obtained. The above rules simply use the dual form of this procedure. Q.E.D.

The class expressions therefore have a partial order which can be obtained by their SDN form. We first order the prime implicants, and then order the complete SDN forms.

Definition 3: The *basic ISA graph* is a labeled network with each node labeled by a sorted set of literals (i.e. class names and complemented class names). The network has a unique top node labeled "any" which is the 1 of the Boolean algebra, and a unique bottom node labeled "empty" which is the 0 of the Boolean algebra. Any label which contains exactly one more literal than another node is connected as a "son" of that node.

Definition 4: The *complete ISA network* adds in union parent nodes to the basic ISA graph. Each such union parent node's label is an SDN set of prime implicants, and this parent node has son edges connecting it to each of the nodes with one of its prime implicants as a label.

Theorem 1: Class expressions are closed under union, intersection, and complement..

Proof: By the uniqueness of the SDN form, any class expression involving only class names, union, intersection, and complement has a unique corresponding node in the complete ISA network. Q.E.D.

Next, we add a list of properties onto each node. We want all required properties of ancestors to be inherited, as for multiple-inheritance. We therefore give the following definition for the union, intersection, and complement operators for these extended classes, which can now be represented as <SDN_form, properties> pairs.

Definition 5: A *simple class algebra* has constants of the form <SDN_form, properties>. Here, SDN_form is a sorted disjunctive normal form expression involving a given universe of class names. "properties" is a subset of allProperties, the universe of <property-name, arity> pairs. The constants include <any,{}> and <empty,allProperties>. This simple class algebra has the following operators:

x+y = <SDN(x.SDN_form+y.SDN_form), x.properties*y.properties>

x*y = <SDN(x.SDN_form*y.SDN_form), x.properties+y.properties>

~x = < SDN(~(x.SDN_form)), x.properties>

The + and * operators which are used inside the definition correspond to propositional logic OR/AND operators and set union/intersection operators, respectively. As pointed out by Rundensteiner and Bic [16], there are other possible definitions of property inheritance, and the property-inheritance hierarchy is not necessarily the same as the superset/subset hierarchy of objects. However, using the above definition simplifies matters greatly, as well as allowing us to prove that the algebra is a Boolean algebra.

Theorem 2: A simple class algebra is a Boolean algebra.

Proof: The proof is based on the properties of the corresponding set operators and logical operators. A detailed analysis will show that each of the axioms of a Boolean algebra are satisfied by the above operators. Q.E.D..

Two classes are considered to be the same if they have the same SDN_form and the same properties. This definition is sufficient for most cases, but can sometimes be improved by using the SDN of the class invariants rather than the SDN of class names only. Replacing class names by their invariants in the SDN_forms permits the detection of extra class equivalences or subclass relationships, and is also useful in query optimization. The class invariants, however, do not necessarily have a finite, unique SDN_form. If all invariants are in the form of Horn sets of clauses (i.e. at most one non-negated literal), then there would be a unique "initial" model, but since this model may contain infinite terms, some limits would have to be placed on its computation.

The above algebra still satisfies the Boolean algebra axioms for +, *, and ~ when other functions (which are defined using the existing attributes "SDN_form" and "properties") are added into the classes. For instance, we can add functions "ancestors" and "descendants" as follows:

x.ancestors = {<U,V> | UεSDN_form & x.SDN_form |= U
& subset(V, x.properties)}

x.descendants = {<U,V> | UεSDN_form & U |= x.SDN_form
& subset(x.properties, V) & subset(V,allProperties)}

where the "logically implies" operator x|=y means that the SDN_form x being true implies that the SDN_form y must also be true. This may be checked by using truth tables, for example, or by using generalized subsumption to see whether or not x is subsumed by y, which is decidable for propositional calculus or for predicate calculus with at most one variable.

Then we can define the functions "parents" and "sons" as follows:

x.parents = {y | yεx.ancestors &
empty({z | z ε x.ancestors & z ε y descendants}) }

x.sons = {y | yεx.descendants &
empty({z | z ε x.descendants & z ε y.ancestors}) }

4. Selections for defining subclasses

We are now ready to add new basic classes into the ISA hierarchy by using the selection operator. We will represent the introduction of a new class by an assignment of the form

newclassname <@ x where Boolean_condition

for any class expression x, and any Boolean_condition which involves a set of properties which are either inherited from x or else are added as the required properties of newclassname. We have already shown that the class expression x has a unique node in the ISA hierarchy, so the required properties of its ancestors can be computed.

The effect of the assignment is to add the new class <SDN(newclassname & x.SDN_form), union(newproperties, x.properties)> to the class algebra. Then we can compute the above implicit parents/sons functions to get a new ISA network structure. When constructing the new basic ISA network, three copies of the old network are used. One copy is left unchanged. Another copy adds newclassname into each of the interior node's prime implicant labels. The final copy adds ~newclassname into each of the interior node's labels. These three networks are then connected at the top and bottom nodes. Then some edges are added into the resulting network to make the original nodes become parents of that node where either "newclassname" or "~newclassname" added into the label. Most importantly, an edge is added which makes the node with the label "newclassname" a son of the node for x. This edge is due to the fact that the new class's SDN_form logically implies the old class's SDN_form.

The operator *classname1 +<@ classname2* similarly adds an edge into the ISA network, making *classname1* also inherit all properties and invariants from *classname2*.

Actually, in our prototype implementation [4], which is available at the anonymous ftp site www.cs.ccu.edu.tw:/pub/oodb/modeloo.pl, the ISA network is only stored implicitly. It is only necessary to keep the SDN_form for each class and to use subsumption checks to check for ancestors or descendants. A node n with an SDN_form which involves union operators is connected to sons whose SDN_forms are the conjuncts of n's SDN_form. Similarly, nodes with only & operators in their SDN_form are connected to parents whose SDN_forms are the literals involved in the conjunct. The sons are those nodes which & a single new class identifier or its complement into the SDN_form. (The implementation uses "&" and ";" operators instead of "*" and "+" operators, reserving the "+" and "*" operators for standard arithmetic operations on sets of numbers.)

The Boolean_condition is stored with each newclassname node when it is created. The Boolean_conditions are inherited, and the conjunction of all ancestor nodes' Boolean_conditions must be satisfied by any node. The Boolean_conditions specify both what are the required properties (i.e. attributes and methods and their arities), and what conditions these properties must satisfy. The complete inherited invariant for an arbitrary node can be obtained from the SDN_form by replacing each class name in the SDN_form by its Boolean_condition.

5. Assignments to properties and user-defined methods

Any attempt to define a mathematically closed query model which encompasses most of the concepts of object-oriented programming and object-oriented databases [2,6] immediately runs into the problem that there are several different standards for the basic concepts of object-oriented programming, such as inheritance. For example, there are many possible ways to resolve naming conflicts which are due to multiple inheritance. Most of the object-oriented concepts, such as encapsulation, aggregation, generalization/specialization, generic method calls, and message passing between active objects, tend to have various implementations [3,5,7,10,12]. Therefore, this paper attempts to use reasonable definitions of these concepts which are not too far away from currently implemented object-oriented database definitions, but which allow queries to be stored in the appropriate location in the ISA and partof graphs.

Since object-oriented programming encompasses so many subjects, any mathematical model of an object-oriented database must be very large. An object-oriented database should include an object-oriented query language, which is preferably the same as or similar to an object-oriented programming language for writing the methods, which are an integral part of an extendible object-oriented database. The model for the object-oriented database must therefore include a model for a programming language which at least contains method calls. We will use Prolog-like method calls and will attempt to show informally how this model can be extended to include other kinds of procedure calls and task invocations. Because of space limitations, we will not be able to describe how overriding of the assignment operator can be used to implement many of the features of object-oriented languages, such as persistence, backup/recovery, locking, transactions, versions, security, etc. The Appendix contains an example of the definition of a "singleton" modifier class which overrides the definition of "insert" (which is called by the += operator) to permit at most one element to be added into a singleton set.

Our approach is similar to the evolving algebra approach for EasyOBL [9] in that static functions are used to store sets and dynamic functions are used to define the implicit classes (EasyOBL universes). In our model, however, we prefer to include multiple inheritance. The use of multiple inheritance fits more naturally with the Boolean "intersection" operation. Both our model and EasyOBL have been implemented in Prolog. The interpretational semantics of EasyOBL uses EasyOBL rules which implement a particular kind of subroutine calls with a runtime stack. We prefer to just use a set of Prolog rules as a specification of semantics, rather than using first order logic or mathematical recursive functions, since the semantics of Prolog are fairly standard, and use of assert/retract simplifies the specification of the "hide" operator.

The properties of superclasses are unioned together in our model, and they are all accessible by using the C++ - like notation superclass::propertyname. Therefore, the "hide" operator must be included to implement strong overriding and security features. The assignment operators := and =@ will first hide the old value of the left-hand side and its subparts (including the sdn) before assigning the new value.

The set assignment operator := first deletes the old definitions of the left-side set and its subparts, including the old sdn (i.e. domain of its elements). Then each of the values of the right-hand side is asserted for that set. The immediate value assignment operator := evaluates its second argument, but the delayed assignment operator =@ leaves the expression unevaluated until the left-side function is called. The =@ operator also deletes the old definition of the function. The operators += and +=@ respectively, add new elements to a set and add new rules to a Prolog-like function definition.

We have chosen to evaluate the arguments before calling, as is done in most programming languages. The programmer can always quote the arguments or pass a ^reference to prevent evaluation. Attributes, functions and methods can return multiple values, unlike most programming languages. This enables parallel execution of set-valued arguments. Functions are automatically extended to apply to sets.

If a set is put into the sdn of the current environment, its keys can be used without the need to be preceded by the set name. In our implementation, all objects inherit a set called "class", so the class definitions are available inside of any environment.

There is very little difference between associating a method with a class by using a statement like

someclass.property(This, ...) =@ expression

and associating the method with the elements of a set object, such as someSet, by using

SomeSet.prototype.property(This,...) =@ expression.

Subscripted elements of a set automatically inherit that set's prototype properties unless the keyed element itself has a definition for that property. This is how overriding is implemented. Although elements may override the default sdn, the default sdn is generally an add-only class mod(addOnly,DefaultClassExpression). The addOnly modifier class hides the <@ operator, thus only permitting the addition of extra superclasses and constraints onto the defaults via the +<@ operator.

The elements of a set are represented by a data structure, whereas the elements of a class's materialization can change as assignments are made to objects, adding and deleting this class from their domains, and thus dynamically changing this class's materialization. Since a class name may also be used as a set name, the members of such a set are by default members of the class (unless they have individual sdn's which override this default). Therefore, the materialization of a class C checks each of the following elements to see if they satisfy the invariants of C:

1. All non-prototype objects whose sdn is subsumed by C (i.e. where all of the sdn conjuncts contain C).
2. All of C's elements "C key __" where "C key __ . sdn" is not defined explicitly.
3. All elements of other sets which do not override the default sdn, where the default sdn is subsumed by C.

Notice that all of the superclasses are contained in the sdn. So, for instance, most of the sdn's will contain "world#classes key objects" in each of their conjuncts since each of the union classes usually is a descendant of "objects".

6. Conclusions

The object-oriented model described in this paper is fairly complex because the concepts of object-oriented programming are themselves fairly complex. However, the basic idea is fairly simple. The constraints which are used to ask a query can also be used to define a class. All classes thus have both an intentional definition which came from the original query and an extensional definition which is the query's response. Thus, each query and each set have a place in the ISA hierarchy of classes. The ISA organization of the queries can be very useful in query optimization by using theorem proving techniques which we have not described in this paper.

This paper tends to reinforce the idea of Bertrand Meyer [14] that contracts are very important for object-oriented programming. Contracts are basically the same as our class invariants. Moreover, concurrent object-oriented programming's famed "inheritance anomaly" [13] can probably only be overcome by specifying the concurrency control in the class invariant [15,18]. In this way, subclasses which override the method declarations will not be able to override the intended effects of concurrency control constructs such as semaphores. For example, control graphs may be used to specify the order of execution of modules. Control graph expressions are in some ways also easier to program because they are not mixed into the program code, like semaphores. It is also possible that by specifying functional dependencies in the invariant, parallelism may be determined automatically.

References

[1] Alhajj, R; Erol Arkun, M., "An Object Algebra for Object-Oriented Database Systems", *Data Base*, Vol. 24, No. 3, pp.13-22.

[2] Bertino, E., Martino, L., "Object-Oriented Database Management Systems: Concepts and Issues", *Computer*, IEEE Press, Vol. 24, No. 4, April 1991, pp.33-47.

[3] Bobrow D., et al., Common Lisp Object System Specification X3J13 Document 88-002R, reprinted in *ACM SIGPLAN*, Special Issue, Vol. 23, Sept., 1988.

[4] Buehrer, D.J. "Model Solutions to OODB Problems," *Proc. 4th International Hong Kong Computer Society Database Workshop*, Hong Kong, Computer Society, Dec. 12, 1992.

[5] Butterworth P., Otis A., and Stein J., "The GEMSTONE Object Database Management System," *Comm. of the ACM*, Vol. 34, No. 10, October 1991, pp.64-77.

[6] Cattell, R.G.G., Object Data Management: Object-Oriented and Extended Relational Database Systems, Addison Wesley, 1991.

[7] Deux O. et al., "The O2 System," *Comm. of the ACM*, Vol. 34, No. 10, October 1991, pp.34-48.

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

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

[10] Ishikawa, H., Fumio Suzuki, Fumihiko Kozakura, Akifumi Makinouchi, Mika Mi-

yagishima, Yoshio Izumida, Masaaki Aoshima, Yasuo Yamane, "The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System", *ACM Transactions on Database Systems*, Vol. 8, No. 1, March, 1993.

[11] 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.

[12] Lamb C. et al., "The OBJECTSTORE Database System," *Comm. of the ACM*, Vol. 34, No. 10, October 1991, pp. 50-63.

[13] Matsuoka, S. and Yonezawa, A. "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", In *Research Directions in Object-Based Concurrency*, MIT Press, Cambridge, 1993.

[14] Meyer, B., *Eiffel: The Language*, Prentice-Hall, Object-Oriented Series, 1992.

[15] Pandey, R; Browne, J.C., "A Compositional Approach to Concurrent Object-Oriented Programming", Proc. of the 1994 Int'l. Conf. on Computer Languages, IEEE Comput. Soc. Press, May 16-19, 1994, pp.124-135.

[16] 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.

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

[18] Wu S.Y.; Miranker, D.P.; Browne, J.C., "Decomposition Abstraction in Parallel Rule Languages, accepted for publication in *IEEE Transactions on Parallel and Distributed Systems*.

We gratefully acknowledge the support of grants NSC-79-0408-E194-02 and NSC-80-0408-E194-10 by the National Science Council of the Republic of China for partial support of this research.

Appendix

(The current version for a Prolog interpreter for modeloo is stored in
anonymous ftp site www.cs.ccu.edu.tw:/pub/papers/oodb/modeloo.pl
Please send comments and error corrections to dan@cs.ccu.edu.tw)

/* Notation:
b#c Dot notation: the attribute *c* (quoted) of the attribute *b* of the current environment.
a sub expr The exprs allowed for subscripts must eval to primitives or list of primitives.
a key b b is not evaluated; it must be a primitive value or a list of primitives.
a$sdn The sorted disjunctive normal form, used as name of the domain class for set *a.*
a&b a;b ~a Class or set expressions for intersection, union, and active complement.
s inClass ClassExpr withProto assignmnts. Make *s* a null set, s#sdn=normal(ClassExpr).
s+= expr. Add values of *expr* as elements into set *s.*
variable:=expr. If *expr* evaluates to primitive values, similar to
(variable isSet primitiveTypes, variable sub $expr_i$ += $expr_i$).
Env#f(This,X,Y,...) =@ expr. Define functions f(b,X,Y,...) where Env is superclass of b.
Env#f(This,X,Y,...) +=@ expr. Add other values to end of function *Env#f.*
c <@ classExpr where (invariant) Make *c* a subclass of *classExpr*, *and*ing the invariant.
c +<@ className Add *className* as superclass of *c, and*ing invariant.
*/

/* A Sample execution */
?- main.

world#classes key mod(lockable,X) <@ any.

/*Don’t inherit object’s assignment*/
world#classes key mod(lockable,X)#insert(ref(U),W) =@ (U$locked ->

@error(’attempt to assign locked var’, @U);
(insert(X::ref(U),W), U#locked:= @true)).

world#classes key mod(lockable,X)#lock(ref(This))=@(This#locked:= @true).

world#classes key mod(lockable,X)#unlock(ref(This)) =@ (This#locked:= @false).

b<@mod(lockable,primitives).

b$lock.

b+=5.

@ error(attempt to assign locked var , @(world#b))

b$unlock.

```
b+=(7;10).
b+b.
```
14 ; 17 ; 17 ; 20

```
numbers#factorial(This) =@  (This>1 -> This*factorial(This-1); 1).
print factorial(10).
```
3628800

```
world#classes key courses<@ objects, world#classes key people<@ objects,
    world#classes key profs<@ people, world#classes key students<@ people.
world#classes key gradStudents<@ students where
                ((level=phd);(level=masters)).

mod(our,This)<@This where (university := @'Chung Cheng'). ourCourses <@
                mod(our,courses), ourProfs <@ mod(our,profs),
                ourStudents <@ mod(our,students).
ourCourses+= member([@cs100,@cs105,@cs120,@cs200]).
ourProfs+= member([@'John Doe', @'Henry James', @'Superman']).
for X in @[tim,john,mary] do ourStudents key X inClass  mod(our,gradStudents)
                withProto (level:= @phd).
newenvir<@world. for X in [^ourCourses,^ourProfs,^ourStudents,^world#classes]   do
                newenvir#subsets += X.
cd newenvir.
print tim$level.
```
@phd

```
world#classes key mod(singleton,X) <@ X.
world#classes key mod(singleton,X)#insert(ref(U),W) =@  (nonEmpty(U) ->
                @error('inserting in singleton set';X::U:=W);
                insert(X::ref(U),W)).
world#ourStudents#prototype#advisor<@mod(singleton,profs).
mary#taking+=member([cs100,cs200]).
mary#advisor:= 'John Doe'.
```

tim#taking+= member([cs100,cs105]).

tim#advisor+= 'John Doe'.

john#advisor+=member(['Henry James', 'John Doe']).

@error(inserting in singleton set; world#classes key profs :: world#ourStudents key John#advisor := John Doe)

john#taking+= member([cs100,cs120]).

'John Doe'#teaching += member([cs100,cs200]).

'Henry James'#teaching:= cs120.

profs#teachingOwnStudents(ref(This))=@

(nonEmpty(This$teaching$inv(taking) & This$inv(advisor)) -> This).

print mod(our,profs)$asSet$teachingOwnStudents.

world#ourProfs key John Doe

world#ourProfs key Henry James

end.