A Data Model for Object-Oreinted Databases

Manojit Sarkar and Steven P. Reiss

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-56
December 1992

A Data Model for Object-Oriented Databases *

Manojit Sarkar
Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912 USA

Abstract

We present an object-oriented data model. The data model represents all entities as objects. The system uses internally generated *object identifiers* for giving objects their unique identity. An object also has a state, and a behavior. Each object belongs to a class. The class defines the structure of the object's state, and its behavior. Classes are organized in inheritance hierarchy. An object of a class τ can be used where an object of τ's superclass is expected. Inheritance polymorphism is provided through late binding of methods based on actual classes of objects. The data model eliminates the *object-versus-values* dichotomy by representing all entities as objects. This is achieved at no loss of modeling power, or performance. The data model provides for exclusive access of objects by other objects.

1 Introduction

Object-oriented databases [2, 3, 7, 13] have been developed in response to the complex data modeling requirements in various application areas such as CAD, Graphics, and Programming Environments.

We started investigating object-oriented database systems for constructing abstract information about programs, called *program abstractions*, by retrieving information from some form of stored representation of programs, called *program database*. The idea is to generate abstractions by querying the database, the answers to the queries are the target abstractions. These abstractions are subsequently visualized graphically, and our full system is a *program visualization* system [11].

The data model of our full program visualization system is object-oriented, and we wanted a database with a compatible data model. This paper presents the data model. A primary goal of our data model is to combine simplicity, modeling power and performance. Our data model is conceptually simple. It avoids the *objects-versus-values* dichotomy found in other data models such as in the O_2 data model [1, 9], and the EXTRA data model [6]. This is achieved at no extra cost. We also show that it is possible to satisfy a variety of data modeling requirements such as *composite objects* of ORION [3], and *own*, *ref*, and *own ref* objects of EXTRA [6] within the object-oriented paradigm. On the other had, neither O_2, nor EXTRA provides exclusive access to objects with oid and behavior. The conceptual simplicity of the data model does not imply lower performance. It is possible to engineer internal optimizations for good performance.

The data model is presented in Section 2 through Section 4. Section 5 defines what constitutes a database. We compare our data model with some other data models in Scetion 6. Final discussions, remaining issues, and conclusions are in Section 7.

2 Values

We start a formal description of the data model with the following pairwise disjoint sets:

- a finite set of *basic types* **D**
- a countably infinite set of *oids* **O**
- a finite set of *attribute names* **A**
- a finite set of *method names* **M**
- a finite set of *class names* **C**

In our data model each entity is represented by an object. Each object is uniquely identified by an object identifier or oid. The object may have a value, or its value may be undefined. Each object also belongs to a class. Only *structured values* can be values of objects. Structured values are built from *atomic values*.

*Support for this research was provided by NSF grants CCR9111507 and CCR9113226, by ARPA order 8225 and by ONR grant N00014-91-J-4052

2.1 Atomic Values

There are four basic types. These are `Integer`, `Float`, `Boolean`, and `String`. Hence $\mathbf{D} = \{$`Integer`, `Float`, `Boolean`, `String`$\}$. The domains of basic types are called basic domains. Basic domains are pairwise disjoint. Each domain consists of a countably infinite set of atomic values. For example, the domain of type `Integer` consists of the set of all integer values. The atomic values in basic domains are also called *basic values* as opposed to oids.

A *type* is either a class or a basic type. Classes are built from basic types and classes using the `Tuple`, `Set`, and `List` constructors. A class built using the `Tuple` constructor is called a *tuple-class*. Similarly a class built using the `Set`, or the `List` constructor is called a *set-class*, or a *list-class* respectively.

O has three special members. These are `nulltuple`, `nullset`, and `nulllist`. They denote undefined objects of all tuple-classes, set-classes, and list-classes respectively. The value of an undefined object is assumed to be unknown, or an empty-set, or an empty-list depending on the the class of the object. This will be further explained later. Each member of **O** is also an atomic value.

2.2 Structured Values

Structured values are constructed from atomic values using the `Tuple`, `Set`, and `List` constructors. There is one special structured value, [ ] which is used to denote the value of any tuple with zero attributes. Note that [ ] and `nulltuple` are not the same values. The former denotes a structured value, while the later is an atomic value and the oid of an object whose value is undefined. Structured values are defined formally as follows:

- the value [ ] is a structured value
- each tuple of atomic values $[a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n]$ is a *tuple-value*, where $n > 0$, $a_i \in \mathbf{A}$ and v_i for $i = 1, 2, \ldots, n$ are attribute names, and atomic values respectively
- each set of atomic values $\{v_1, v_2, \ldots, v_n\}$ is a *set-value*, where $n > 0$, and v_i for $i = 1, 2, \ldots, n$ are atomic values
- each list of atomic values $< v_1, v_2, \ldots, v_n >$ is a *list-value* where $n > 0$, and v_i for $i = 1, 2, \ldots, n$ are atomic values

There is no order among the elements of a set, but the elements of a list are ordered. Only homogeneous sets (lists) are allowed, *i.e.* the types of all members of any set (list) must be the same. *Types* are discussed in the following section. A set-valued object whose value is an empty-set is equivalent to an undefined object of some set-class. Similarly, a list-valued object whose value is an empty-list is equivalent to an undefined object of some list-class. We denote the set of all the structured values by the countably infinite set **V**.

3 Classes

Each object belongs to a *class*. The class describes the structure of the object's value, and the object's behavior. The syntax for class definitions is as follows:

<*class-decl*> → Class <*class-name*>
→ Class <*class-name*> <*class-def*>

<*class-def*> → [Inherits <*superclass-name*>]
<*class-struc*>
[Methods {<*class-method*>}]

<*class-struc*> → Tuple '(' {<*attr-def*>} ')'
→ Set '(' <*basic-type*> ')'
→ Set '(' [Own] <*class-name*> ')'
→ Set '(' [Own] <*class-struc*> ')'
→ List '(' <*basic-type*> ')'
→ List '(' [Own] <*class-name*> ')'
→ List '(' [Own] <*class-struc*> ')'

<*attr def*> → <*attr-name*> ':' <*basic-type*>
→ [Own] <*attr-name*> ':' <*class-name*>
→ [Own] <*attr-name*> ':' <*class-struc*>

<*class-method*> → <*method-sig*> <*method-body*>

<*method-sig*> → <*method-name*>
'(' {<*param-def*>} ')'
':' <*result-def*>

<*param-def*> → <*basic-type*>
→ <*class-name*>
→ <*class-struc*>

<*result-def*> → <*basic-type*>
→ <*class-name*>
→ <*class-struc*>

<*method-body*> → '{' <*code*> '}'

A class may be defined by specifying a name, a superclass, a structure, and methods. Superclass and methods are optional. It is also possible to define classes *implicitly* without any name, superclass, and methods. Example 3.1 below defines eight explicitly declared classes `Point`, `Vertex`, `Node`, `Edge`, `SegmentedEdge`, CurvedEdge, Network, and Graph. It also implicitly declares some classes with no name, no superclass, no methods. The class associated with the attribute bendpoints of class SegmemtedEdge is such a class with structure List(Point).

It is also possible to define *recursive* classes such class Node in Example 3.1. The syntax allows a class name to be declared and used first, and defined later as class Edge in the same example.

Example 3.1: The following are some class declarations:

```
Class Point
  Tuple (x: Float,
         y: Float)
  Methods {distance (Point): Float {···}};

Class Edge;

Class Vertex
  Tuple (Own position: Point,
         label: String,
         radius: Float);

Class Node
  Inherits Vertex
  Tuple (successors: Set (Node));

Class Edge
  Tuple (source: Vertex,
         dest: Vertex,
         label: String)
  Methods {length (): Float {···}};

Class SegmentedEdge
  Inherits Edge
  Tuple (Own bendpoints: List(Point))
  Methods {length (): Float {···}};

Class CurvedEdge
  Inherits Edge
  Tuple (Own controlpoints: List (Point))
  Methods {length (): Float {···}};

Class Network Set (Edge);

Class Graph Set (Node);
```

□

3.1 Structure

All classes must have a *structure*. As shown above, structures are built using the Tuple, Set, and List constructors from basic types, and already declared class names. A class built using the Tuple constructor is called a *tuple-class*. Similarly, a class built using the Set constructor is called a *set-class*, and a class built using the List constructor is called a *list-class*. The value of an object of any tuple-class can only be a tuple-value. Similarly, the value of an object of any set-class can only be a set-value, and the value of an object of any list-class can only be a list-value.

Any object can be potentially referenced by many other objects. Sometimes it is necessary to prohibit such sharing, and allow a class of objects to exclusively refer to other objects. This constraint can be specified using the "Own" qualifier in attribute definitions. An object held by an attribute qualified as Own cannot be referenced by more than one object.

3.2 Methods

A class declaration may have zero or more *methods* as a part of the class definition. Methods define the object's behavior. Each method is a function, it has a *signature* and a *body*. The signature is an expression of the form $c : m(\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau_r$ where c is the *receiver class*, m is the *method name*, $\tau_1, \tau_2, \ldots, \tau_n$ are the types of the *parameters* for some $n \geq 0$, and τ_r is the type of the *return value*. A method's body is a piece of code written in some programming language, and it implements the intended function.

Example 3.2: Example 3.1 defines four methods for the classes Point, Edge, SegmentedEdge, and CurvedEdge. The signatures of these methods respectively are

```
Point: distance (Point) → Float,
Edge: length () → Float, and
SegmentedEdge: length () → Float
CurvedEdge: length () → Float
```

□

3.3 Inheritance

A class declaration may specify a *superclass* as a part of its class definition. Inheritance [4, 5] allows the user to derive new classes from existing classes. Only single-inheritance is allowed in our data model *i.e.*, a class can have at most one superclass [1].

The inheritance relationship is a partial order on the classes, *i.e.*, it is *reflexive*, *antisymmetric* and *transitive*. Since superclasses are optional, the class hierar-

[1]We feel single-inheritance is adequate for our modeling goal. We plan to incorporate multiple-inheritance if our later needs justify the additional complexity.

chy is potentially a forest of many disjoint trees, and not a single tree [2].

A class inherits the attributes and the methods of its superclass by default. The structure and the method suite of the inheriting class therefore include the inherited attributes and methods respectively. The inheriting class can define additional attributes and methods, or redefine inherited attributes and methods. However the resulting structure and method suite of the inheriting class must be compatible to the structure and method suite of its superclass respectively.

3.4 Structural Compatibility

Structural compatibility, denoted by the symbol "$\preceq$" is defined as follows:

- for any type τ, $\tau \preceq \tau$
- for any three types τ_1, τ_2, τ_3, if $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_3$, then $\tau_1 \preceq \tau_3$
- for the types τ_i for $i = 1, 2, \ldots, m+n$, and $\tau_j{}'$ for $j = 1, 2, \ldots, m$, where $m \geq 0$, $n \geq 0$, if $\tau_i \preceq \tau'_i$ for all $i = 1, 2, \ldots, m$, then $\texttt{Tuple(}a_1 : \tau_1, a_2 : \tau_2, \ldots, a_m : \tau_m, \ldots, a_{m+n} : \tau_{m+n}\texttt{)} \preceq \texttt{Tuple(}a_1 : \tau'_1, a_2 : \tau'_2, \ldots, a_m : \tau'_m\texttt{)}$
- for any two types τ and τ', if $\tau \preceq \tau'$, then $\texttt{Set(}\tau\texttt{)} \preceq \texttt{Set(}\tau'\texttt{)}$
- for any two types τ and τ', if $\tau \preceq \tau'$, then $\texttt{List(}\tau\texttt{)} \preceq \texttt{List(}\tau'\texttt{)}$

3.5 Method Suite Compatibility

The method suite of a class must also be compatible to that of its superclass. If m is a method name and c is a class, then m is defined in c if and only if there exits a method with signature $c : m(\tau_1, \tau_2, \ldots, \tau_l) \rightarrow \tau_r$. A class may override its inherited methods by defining methods with the same names. If c is a superclass of c', and they have two methods with signatures $c : m(\tau_1, \tau_2, \ldots, \tau_l) \rightarrow \tau_r$ and $c' : m(\tau_1{}', \tau_2{}', \ldots, \tau_k{}') \rightarrow \tau_r{}'$, then the following must be satisfied:

- $l = k$
- $\tau'_i = \tau_i$ or τ'_i is a subclass of τ_i for $i = 1, 2, \ldots, l$ where $l \geq 0$
- $\tau_r = \tau'_r$, or τ_r is a subclass of τ'_r

[2] We therefore do not associate systemwide operators with any top-level class. In our model, they are predefined operators applicable to any objects of compatible classes.

3.6 Subclass Relationship

Subclass relationship is denoted by the symbol "<". If c is a subclass of c', then $c < c'$. Structure and method suite compatibility merely allow two classes to be related by subclass-superclass relationship. The actual relationship must be specified by the user by naming the superclass during the class definition of the subclass.

3.7 Substitutibility

Subclass relationships allow an object of class c to be used in any context expecting an object of class c', where c' is a superclass of c. In other words, an object of any class c is also an object of all the superclasses of c.

3.8 Late Binding

Since an object of a certain class can be assigned to a variable of its superclass, given a method call on a variable, sometimes the method to be executed can only be determined at run time based on the object's actual class. This is known as *late binding*.

3.9 Overloading

Method names can also be *overloaded* by defining methods with the same name in more than one class not related by subclass-superclass relationships. Such method name overloading can be resolved at compile time based on method signatures.

3.10 Implicitly Declared Classes

Since implicitly declared classes do not have names, they cannot be declared to be subclass or superclass of any other class. The system therefore can assume that the implicitly declared classes have no subclass, and optimize the representation and access to the objects of such classes to achieve better performance. This will be further explained later.

3.11 Class Equivalence

Two classes are *equivalent* if they have the same name. The system gives internally generated names to classes declared implicitly as the class associated with the attribute `bendpoints` in Example 3.1. Implicitly declared classes with same structures are given the same internally generated name, hence they are equivalent.

3.12 Class Definition Function

We define a partial function σ which maps class names in $\mathbf{C}$ to class definitions. As an example, after the class definitions in Example 3.1 σ(`Point`) = `Tuple(x: Float, y: Float) Methods` {`distance (Point): Float` $\{\cdots\}$}. The function σ is partial to allow for classes that have been declared but not defined. The database schema includes this function.

4 Objects

An object's value may or may not be defined. An *object* whose value is defined is a triple (o, v, c) where $o \in \mathbf{O}$ is the object's oid, $v \in \mathbf{V}$ is the object's value, and $c \in \mathbf{C}$ is the object's class.

If the value of the object is undefined, then its oid must be `nulltuple`, or `nullset`, or `nulllist`. An object whose oid is `nulltuple` is an object of some tuple-class. Similarly, an object with oid `nullset` is an object of some set-class, and an object with oid `nulllist` is an object of some list-class. An object whose value is undefined is called an *undefined object*.

We define a partial function ν from $\mathbf{O}$ to $\mathbf{V}$. The function maps an object's oid to its value for every (defined) object present it the database. The the interpretation of the value depends on the class. Hence assuming that the value v of any object (o, v, c) is defined

- if o is the oid for an object of any tuple-class, then $\nu(o)$ is a tuple-value
- if o is the oid of an object of any set-class, then $\nu(o)$ is a set-value
- if o is the oid of an object of any list-class, then $\nu(o)$ is a list-value

Whenever a new object is created in the database with a defined value the map ν is updated. The instance of a database schema includes the function ν.

4.1 Equality

The system provides three equality operators which test for id-equality, shallow-equality, and deep-equality.

Id-Eqiality: Two basic values *id-equal* if they are the same value. Two objects are *id-equal* when they are defined and they have the same oid. We use the symbol "$\equiv$" to denote id-equality.

Shallow-Equality: Two basic values are *shallow-equal* if they are id-equal. Two objects are shallow-equal if they are id-equal. If they are not id-equal, then they are shallow-equal only if their values are id-equal. A value can be either a tuple-value, a set-value, or a list-value. If the values are tuple-values, then the corresponding attribute names must be the same, and the corresponding attribute values must be id-equal. If the two values are set-values, then they must have the same cardinality, and there must be a one-to-one mapping between the sets such that the corresponding members are id-equal. If the values are list-values, the lengths of the lists must be the same, and corresponding members of the list must be id-equal. Shallow-equality is denoted by the symbol "$=_s$".

Deep-Equality: Any two basic values are *deep-equal* if they are id-equal. Any two objects are deep-equal if they are shallow-equal. If they are not shallow-equal, and then they are deep-equal if their values contain oids and the objects represented by the corresponding oids are deep-equal. Deep-equality is denoted by the symbol "$=_d$" .

The equality operators can be applied to objects of compatible classes or basic types. Class compatibility is defined in Section 3.7.

4.2 Copying

Like the equality operators the system provides two copy operators, shallow-copy and deep-copy.

Shallow-Copy: A shallow-copy of an object is an object with same value and class but a different oid.

Deep-Copy: A deep copy of an object has the same class, a different oid, and possibly a different value created by recursively creating deep-copies of objects till the new object shares no oid with the original object.

Copy operations are useful while assigning values to `Own` attributes by copying existing objects. An object held by such an attribute cannot be assigned to another variable, or any value of any other variable cannot be assigned to such an attribute because of the restrictions on sharing.

4.3 Redefining Equality and Copying

Only shallow-equality and deep-equality operators, and both copy operators can be overridden by methods defined in the classes. This allows the user to define appropriate equality and copy operators for different classes of objects.

5 Database

A database consists of a schema and an instance. A database schema is denoted by S. An instance of the database schema is denoted by I. There is a clear separation between the schema and the instance in our data model. A database schema explicitly declares a set of classes $C \in \mathbf{C}$. The system maintains an extension for each class in C.

5.1 Class Extensions

The extension is the set of all objects of its associated class. We define a function π_d, called the *disjoint oid assignment*, from class names to extensions. If for some $n \geq 0$, $\{o_{1_c}, o_{2_c}, \ldots, o_{n_c}\}$ is the extension of class c, then $\pi_d(c) = \{o_{1_c}, o_{2_c}, \ldots, o_{n_c}\}$. The function π_d is called the disjoint oid assignment because, if c and c' are two different classes in C, then $\pi_d(c) \cap \pi_d(c') = \phi$.

Given π_d, the *oid assignment* π (inherited from π_d) is a function mapping each class name to a set of oids such that $\pi(c) = \pi_d(c) \cup \{\pi_d(c') \mid c' \in C, c' < c\}$. In other words, π maps a class name to the set of objects of that class and all of its subclasses. Given π, the interpretation of a class c is defined as follows:

- each basic type `Integer`, `String`, `Float` and `Boolean` has its natural domain
- for each tuple-class c, $domain(c) = \{\mathtt{nulltuple}\} \cup \pi(c)$
- for each set-class c, $domain(c) = \{\mathtt{nullset}\} \cup \pi(c)$
- for each list-class c, $domain(c) = \{\mathtt{nulllist}\} \cup \pi(c)$

Objects of a class can have values from the domain defined by the class's structure. A structure's *domain* is defined as follows:

- $domain(\text{Tuple}(a_1 : \tau_1, a_2 : \tau_2, \ldots, a_n : \tau_n)) = \{[a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n] \mid v_i \in domain(\tau_i), i = 0, 1, \ldots, n\}$
- $domain(\text{Set}(\tau)) = \{\{v_1, v_2, \ldots, v_n\} \mid v_i \in domain(\tau), i = 1, 2, \ldots, n\}$
- $domain(\text{List}(\tau)) = \{< v_1, v_2, \ldots, v_n > \mid v_i \in domain(\tau), i = 1, 2, \ldots, n\}$

5.2 Schema

A database *schema* is a 3-tuple (C, σ, G) where $C \in \mathbf{C}$ is a set of class names, σ is the function mapping class names to class definitions, and G is a set of *global variables* with associated classes. The sets C and G act as the entry points to the database.

Every object with a global name in G is *persistent*. Every object that is part of a persistent object is also persistent.

The function σ maps class names to class definitions. Class hierarchy in C can be constructed from the information available with the class definitions.

5.3 Instances

An *instance* of a database schema consists of a finite set of objects and the four functions π_d, π, ν, and γ. The functions π_d, π, ν, and γ are defined as follows:

- the function π_d is the disjoint oid assignment
- the function π is the oid assignment inherited from π_d
- The function ν maps oids to values for all the defined objects in the database instance
- the function γ maps variable names in G to objects which are the values currently assigned to the variables

6 Other Data Models

The object-oriented paradigm can represent data structures of arbitrary complexity. But it is sometimes convenient to be able to specify additional constrains on the structure of the data. For example, it is sometimes necessary to prohibit sharing of one object by more than one object. This is achieved in O_2 [1] data model by *values*. O_2 values cannot be shared. Similarly, in EXTRA [6] attributes qualified by the *own* qualifier can hold only non-shared data.

Composite objects in ORION [3, 8] imply a different kind of constraint. In EXTRA, the *own ref* qualifier provides the same functionality. In ORION, a reference is either a *composite link* or an *ordinary link*. An object can be referenced by any number of ordinary links, but there can be at most one composite link to the object. A composite link stands for a "ownership" relation. Objects referenced by composite links are "dependent" objects whose existence depends on the existence of their owner objects. When an owner object is deleted all its dependent objects must also deleted (possibly leaving some dangling ordinary references).

We think that *sharing* and *ownership* are orthogonal issues to the issue of *data representation*. The data model should keep these issues separate from each other. Following this philosophy, our data model represents all entities as objects, and provides a separate qualifier for prohibiting sharing. This simplifies the model. It removes any possibility of confusion between objects and values such as in the O_2 data model [10]. The simplification does not necessarily imply lower performance. It is possible to engineer internal optimizations to achieve good performance.

In the O_2 data model, it is not possible to express restrictions on object sharing. Also in EXTRA data model, it is possible to express restrictions on object ownership, but not on object sharing. This is a consequence of the fact that all non-shared data must be *values* which have no behavior, and all the shared data must be objects. In our data model, it is possible to express constraints on object sharing.

Our data model, *does not* provide the functionality of *composite* or *own ref* objects. The data model however can be easily extended with a new qualifier such as *Composite* to express this constraint. We do not include such facility because, we feel at this point our data modeling requirements do not justify the additional complexity which will be required to enforce such constraint.

6.1 Providing Functionality of Values

Values can be thought of as special objects which are never shared and have no behavior. Since values are never shared it is not necessary to refer to them indirectly from multiple objects. It is therefore possible to avoid indirection to access values. It is also not necessary to assign oids to values, because it is sufficient to assume that no two values are id-equal. Since values have to behavior, they do not carry any type related information at runtime. It is not necessary to perform a late binding of any method on a value based on its actual type. A value of any type can be assigned to a variable of its supertype. The assignment however truncates the value if necessary.

In our data model, a value is an object of an implicitly declared class (which specifies only a structure, but no name, no supertype and no method) and qualified as `Own`. The attribute `bendpoints` of class `SegmentedEdge` in Example 3.1 holds exclusively referenced objects. This provides objects that are never shared and have no behavior. These objects therefore can be stored and accessed like values in O_2 data model. Since the implicitly declared classes have no names, there extensions are also not stored as a part of the database instance. We point out, however, that this is only an internal optimization. To the outside users these special objects are no different than the other objects.

The restriction on sharing is easily enforced. The `Own` qualifier can be used to qualify attributes, members of sets, and members of lists. An object can be assigned to an attribute qualified as `Own` only at the time of the object's creation. Similarly, an object can be made member of a set or list that requires non-shared members only at the time of the object's creation.

6.2 Objects and Basic Values

A basic value can be thought of as an object which is always shared. The *value* of the object is same as the basic value, and the *type* of the object is the type of the basic value. Since these objects are always shared it is not necessary to assign oids to them; it is sufficient to assume that they are id-equal if and only if they are of the same type and they have the same value. For example, the integer `5` can be thought of as the object `(_, 5, Integer)`.

7 Conclusions

We have presented a data model. The operational part of the data model, the query language OQL, and an object algebra is presented in [12]. The data model and OQL have been designed with a practical application in mind, *i.e.*, program visualization. Keeping the data model simple in order to achieve a simple but powerful query language was our primary goal. Shaping the data model and OQL to be an effective program abstraction tool will be our primary research goal in future.

Acknowledgements

Gerd Hilderbrand has helped us to grasp the intricacies of Datalog and IQL. Serge Abiteboul and Paris Kanellakis have patiently explained many aspects of the O_2 data model.

References

[1] S. Abiteboul, and P. Kanellakis. Object identity as a query language primitive. *Proc. of ACM SIGMOD Conf.*, 1989.

[2] F. Bancilhon, C. Delobel and P. Kanellakis. *Building an Object-Oriented Database System The Story of O2. Morgan Kaufman*, pp. 61–75, 1992.

[3] J. Banerjee, H-T Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H-J Kim. Data model issues for object-oriented applications. *ACM Transactions on Information Systems*, 5(1), pp. 3–26, Januray, 1987.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(1), January, 1988.

[5] L. Cardelli, and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4), pp. 471-522, December, 1985.

[6] M. Carey, D. DeWitt and S. Vandenberg. A data model and query language for Exodus. *Proc. of ACM SIGMOD Conf.*, 1988.

[7] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derret, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Transactions on Information Systems*, 5(1), pp. 48–69, Januray, 1987.

[8] W. Kim, J. Banerjee, H-T Chou, J. F. Garza, and D. Woelk. Composite object support in an objecoriented database system. *Proc. of ACM OOPSLA Conf.*, 1987.

[9] C. Lecluse, and P. Richard. Modeling complex structures in object-oriented databases. *Proc. of ACM PODS Symp.*, 1989.

[10] O_2 Technology. The O_2 User Manual, June, 1992.

[11] S. Reiss, and M. Sarkar. Generating abstractions for visualization. Technical Report CS-92-35. *Computer Science Department, Brown University*, September, 1992.

[12] M. Sarkar, and S. Reiss. A query language for object-oriented databases with tuples, sets and lists. Technical Report CS-92-57. *Computer Science Department, Brown University*, December, 1992.

[13] J. D. Ullman. Principles of data and knowledge-based systems. *Computer Science Press*. ISBN0-7167-8158-1, 1988.