Data Model Issues for Object-Oriented Applications

JAY BANERJEE, HONG-TAI CHOU, JORGE F. GARZA, WON KIM, DARRELL WOELK, and NAT BALLOU
MCC
and
HYOUNG-JOO KIM
University of Texas

Presented in this paper is the data model for ORION, a prototype database system that adds persistence and sharability to objects created and manipulated in object-oriented applications. The ORION data model consolidates and modifies a number of major concepts found in many object-oriented systems, such as objects, classes, class lattice, methods, and inheritance. These concepts are reviewed and three major enhancements to the conventional object-oriented data model, namely, schema evolution, composite objects, and versions, are elaborated upon. Schema evolution is the ability to dynamically make changes to the class definitions and the structure of the class lattice. Composite objects are recursive collections of exclusive components that are treated as units of storage, retrieval, and integrity enforcement. Versions are variations of the same object that are related by the history of their derivation. These enhancements are strongly motivated by the data management requirements of the ORION applications from the domains of artificial intelligence, computer-aided design and manufacturing, and office information systems with multimedia documents.

Categories and Subject Descriptors: H.1.2 [**Models and Principles**]: User/Machine Systems—*human information processing*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.4.1 [**Information Systems Applications**]: Office Automation

General Terms: Design, Theory

Additional Key Words and Phrases: Composite object, object-oriented database, schema evolution, version management

---

1. INTRODUCTION

In recent years, object-oriented programming has gained a tremendous popularity in the design and implementation of emerging data-intensive application systems. These include artificial intelligence (AI), computer-aided design and manufacturing (CAD/CAM), and office information systems (OIS) with multimedia

---

Authors' addresses: J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, and D. Woelk, Database Program, MCC, 3500 West Balcones Center Drive, Austin, TX 78759. N. Ballou, AI/KBS Program, MCC, 3500 West Balcones Center Drive, Austin, TX 78759; H.-J. Kim, Department of Computer Sciences, University of Texas, Austin, TX 78712.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0734-2047/87/0100-0003 $00.75

documents [2, 8, 15]. Object-oriented programming offers a number of important advantages for these applications over traditional control-oriented programming. One is the modeling of all conceptual entities with a single concept, namely, objects. An *object* represents anything from a simple number, say, the number 25, to a complex entity, such as an automobile or an insurance agency. The state of an object is captured in the *instance variables*. The behavior of an object is captured in *messages* to which an object responds. The messages completely define the semantics of an object.

Another advantage of object-oriented programming is the notion of a *class hierarchy* and *inheritance* of properties (instance variables and messages) along the class hierarchy. The class hierarchy captures the IS-A relationship between a class and its *subclass* (equivalently, a class and its *superclass*). All subclasses of a class inherit all properties defined for the class and can have additional properties local to them. The notion of property inheritance along the hierarchy facilitates top-down design of the database, as well as applications.

We are presently prototyping an object-oriented database system, called ORION, to support the data management needs of object-oriented applications from the CAD/CAM, AI, and OIS domains. The intended applications for ORION impose two types of requirements: advanced functionality and high performance. The ORION architecture has been designed to satisfy these requirements. ORION will provide a number of advanced features that conventional commercial database systems do not, including version control and change notification [7], storage and presentation of unstructured multimedia data [31], and dynamic changes to the database schema [4]. For high performance, ORION will support appropriate access paths and techniques for query processing, buffer management, and concurrency control.

To derive an object-oriented application interface to ORION, our initial plan was simply to use a data model from some of the existing object-oriented systems [28] or object-oriented data models [1, 3, 24]. However, two major problems rendered this approach impossible. One was that there is no consensus about the object-oriented model; different object-oriented systems support different notions of objects. We had to extract and consolidate a number of major concepts found in many object-oriented systems and use them as the basis for our data model.

Another problem was that most existing object-oriented systems are programming language systems [6, 13, 20, 21, 22, 29]. As such, their data models completely ignore many important database issues, such as deletions of persistent objects, dynamic changes to the database schema, and predicate-based query capabilities. They also lack concepts that are important to applications, such as *composite objects* and *aggregate objects* for defining and manipulating complex collections of related objects. Further, they do not include version control, which most application systems in the CAD/CAM and OIS domains require. We had to augment the basic set of object concepts with these additional concepts and capabilities.

In Section 2, we provide a review of the fundamental object-oriented concepts, including approaches to the problem of conflict resolution, which arises when a class inherits properties from one or more superclasses, and our own approach to supporting predicate-based queries against the database. In Section 3 we

introduce a formal framework for understanding the taxonomy and semantics of schema change operations that we allow, including changes to the class definitions and the class lattice structure. In Section 4 we define the semantics of composite objects and show their integration into the object-oriented data model. Section 5 shows the integration of our model of versions into the object-oriented data model.

2. OBJECT-ORIENTED CONCEPTS

Existing object-oriented systems exhibit significant differences in their support of the object-oriented paradigm; Stefik and Bobrow [28] provide an excellent account of different variations of the object concepts. In this section we review the basic object concepts, and, where appropriate, show how we have refined them to suit the requirements of our applications in a database environment.

2.1 Basic Concepts

In object-oriented systems, all conceptual entities are modeled as objects. An ordinary integer or string is as much an object as is a complex assembly of parts, such as an aircraft or a submarine. An object consists of some private memory that holds its state. The private memory is made up of the values for a collection of instance variables. The value of an instance variable is itself an object and therefore has its own private memory for its state (i.e., its instance variables). A primitive object, such as an integer or a string, has no instance variables. It only has a value, which itself is an object. More complex objects contain instance variables, which, in turn, contain other instance variables. Further, two objects may have instance variables that refer to a common object. For example, the value of the Manufacturer instance variable of a vehicle may be an object that represents a certain auto company, and that same auto company may also be the value of the Employer instance variable of a person.

The behavior of an object is encapsulated in *methods*. Methods consist of code that manipulates or returns the state of an object. Methods are a part of the definition of the object. Methods, as well as instance variables, however, are not visible from outside of the object. Objects can communicate with one another through messages. Messages, together with any arguments that may be passed with the messages, constitute the public interface of an object. For each message understood by an object, there is a corresponding method that executes the message. An object reacts to a message by executing the corresponding method and returning an object in response.

A program may create and reference a large number of objects. A database may contain an even larger collection of objects. If every object is to carry its own instance variable names and its own methods, the amount of information to be specified and stored can become unmanageably large. For this reason, as well as for conceptual simplicity, "similar" objects are grouped together into a *class*. All objects belonging to the same class are described by the same instance variables and the same methods. They all respond to the same messages. Objects that belong to a class are called *instances* of that class. A class describes the form (instance variables) of its instances and the operations (methods) applicable to

its instances. Thus, when a message is sent to an instance, the method that implements that message is found in the definition of the class.

ORION supports two features to further reduce redundant storage and specification of objects: shared-value and default-value instance variables. For such variables, a value must be specified. For a *shared-value variable* of a class, all instances of the class take on the specified value. This is similar to the class variable concept in Smalltalk [13]. For a *default-value variable*, those instances of a class whose value for the instance variable is not specified take on the specified default value. It is certainly possible for the user to implement the concept of default values through the use of a special-purpose instance creation method for each class. However, for ORION applications that use default values extensively, the provision of the default value concept as a modeling feature makes the creation of classes considerably simpler.

For example, we may define instance variables Medium and TakeoffDistance for the class Aircraft. The instance variable Medium may be shared valued and take on the same value for every aircraft. The instance variable TakeoffDistance, on the other hand, may have a default value of 300. In case a new aircraft is created and its takeoff distance is not specified, the value of that variable is 300.

In ORION, as in most object-oriented systems, both classes and instances are viewed as objects. This is necessary mainly for uniformity in the handling of messages. Messages are sent to objects. In most cases messages are sent to instance objects. However, how can one, for example, create an instance object in the first place? Since the instance does not exist, it cannot be sent a message to create itself. This problem is solved by treating a class as a (class) object. To create an instance of a class, a message is sent to the corresponding class object. There are also many other situations in which it is necessary to send messages to class objects, including inquiry of the definition of a class, changing the definition of a class, and so on.

Grouping objects into classes helps avoid the specification and storage of much redundant information. The concept of a *class hierarchy* extends this *information hiding* capability one step further. A class hierarchy is a hierarchy of classes in which an edge between a node and a child node represents the IS-A relationship; that is, the child node is a specialization of the parent node (and conversely, the parent node is a generalization of the child node [27]). For a parent–child pair of nodes on a class hierarchy the parent is called the superclass of the child, and the child is called the subclass of the parent. The instance variables and methods (collectively called properties) specified for a class are shared (inherited) by all its subclasses. Additional properties also may be specified for each of the subclasses. A class needs to inherit properties only from its immediate superclass. Since the latter inherits properties from its own superclass, it follows by induction that a class inherits properties from every class in its *superclass chain*.

Smalltalk [12] originally restricted a class to only a single superclass. In other words, the class hierarchy was limited to being a tree. Most other object-oriented systems, as well as the recent version of Smalltalk, have relaxed this restriction. In these systems (and in ORION) a class can have more than one superclass. Thus the class hierarchy is generalized to a lattice. (We borrow the term *lattice* from the literature on object-oriented systems to mean a directed acyclic graph structure.)

We emphasize, however, that ORION still requires instance objects to belong to only one class. Sometimes, it is useful to allow an instance object to belong to more than one class. That is, an instance object, such as "my-car," may belong to two different classes, say, LandVehicle and PetroleumFueledMotorizedVehicle. We have concluded that the consequences of this generality are lower performance and a large increase in system complexity. This results from the fact that the structure of an instance object is completely variable; since it can belong to any number of classes, its instance variables cannot be determined a priori, and the identifiers of all classes to which an instance belongs must be stored with each and every instance. Only by examining the content of an instance object and determining the classes to which it belongs will it be possible to determine its instance variables and methods. To model "my-car" in the above example, the ORION user must create a new class called Automobile with two superclasses LandVehicle and PetroleumFueledMotorizedVehicle. All instances of cars, including "my-car," then belong to the Automobile class, rather than to two different classes.

It is often desirable not to require the value of an instance variable to belong to a particular class, that is, not to bind the possible values of the instance variable to any single class. This means that two different instances of the same class may reference objects from two different classes, through the same instance variable. For example, the VehicleId of one aircraft may be an integer object, and that of another aircraft may be a string object. In other words, the class definition for Aircraft does not bind the possible values of VehicleId to either the integer class or the string class.

However, for the purposes of integrity control, it is also desirable to bind the domain (called *data type* in conventional programming languages, such as Pascal and C) of an instance variable to a specific class (and therefore implicitly to all subclasses of the class). For example, the Manufacturer instance variable of the Aircraft class may be bound to the class Company. Thus a manufacturer is a company. Further, if the Company class has subclasses, the instance variable Manufacturer may also take on as its value an instance of any subclass of Company. Thus ORION supports both typing and no typing.

2.2 Class Lattice and Conflict Resolution

The class lattice simplifies data modeling and often requires fewer classes to be specified than are required with a class hierarchy. In a class lattice, however, a class has multiple superclasses and thus inherits properties from each of the superclasses. This feature is often referred to as *multiple inheritance* [21, 28]. In a class lattice, two types of conflicts may arise in the names of instance variables and methods. One is the conflict between a class and its superclass (this type of problem also arises in a class hierarchy). Another is between the superclasses of a class; this is the consequence of multiple inheritance. In this section we discuss approaches to resolving these two types of inheritance conflicts.

In all systems we are aware of, name conflicts between a class and its superclasses are resolved by giving precedence to the definition within the class over that in its superclasses. For example, if the class definition for a class Aircraft specifies an instance variable VehicleId, it is the definition used for every

Aircraft instance. This definition overrides any definition that may be inherited from any superclass.

The approach used in many systems to resolve name conflicts among superclasses of a given class is as follows. If an instance variable or a method with the same name appears in more than one superclass of a class C, the one chosen by default is that of the first superclass in the list of (immediate) superclasses for C. For example, as shown in Figure 1, the class Submarine has to inherit an instance variable Size either from the superclass WaterVehicle (which defines Size) or from NuclearPoweredVehicle (which inherits Size from its superclass MotorizedVehicle). If, in the definition of the class Submarine, NuclearPoweredVehicle is specified as the first superclass, Size will be inherited from NuclearPoweredVehicle.

Since this default conflict resolution scheme hinges on the permutation of the superclasses of a class, unlike most other systems, ORION allows the user to explicitly change this permutation at any time. Further, ORION provides two ways in which a user can override the default conflict resolution.

(1) The user may explicitly inherit one instance variable or method from among several conflicting ones. For example, in Figure 1, the user who defines the class Submarine may choose to inherit the instance variable Size from WaterVehicle rather than from NuclearPoweredVehicle, even if NuclearPoweredVehicle is the first superclass in the list of superclasses of Submarine.

(2) The user may explicitly inherit one or more instance variable or method *that have the same name and rename them within the new class definition.* For example, the definer of the class Submarine may specify that the instance variable Size be inherited from WaterVehicle with the new name CrewSize, and also from NuclearPoweredVehicle with the name Size (ORION ensures that all names inherited or defined within a class are distinct).

2.3 ORION Class Lattice and the Set Class

We mentioned earlier that the capability of issuing predicate-based queries against a large database of persistent objects is an important requirement in a database environment. A few operational object-oriented database systems support associative queries [3, 24]. However, most existing object-oriented systems are programming language systems, and as such they do not support associative queries. In this section we provide our simple extension to the existing notion of *a class lattice as a formal basis for allowing queries against unnamed instances* of classes.

As in any object-oriented system, ORION defines a class called OBJECT as the root of the class lattice. The class lattice includes not only all user-defined classes, but also all system-defined classes. Figure 2 shows all ORION-defined classes as subclasses of the OBJECT class. The class PType provides the basis for defining all classes that can be used as primitive domains of instance variables. The class Collection consists of objects that are collections of other objects. A subclass of Collection is the class Set, each of whose instances is a set (a collection of objects with no duplicates) [9, 13]. Whereas the class Collection supplies messages for iterating over the elements in a collection object, the class Set supplies further messages for searching the elements of a set, adding an element

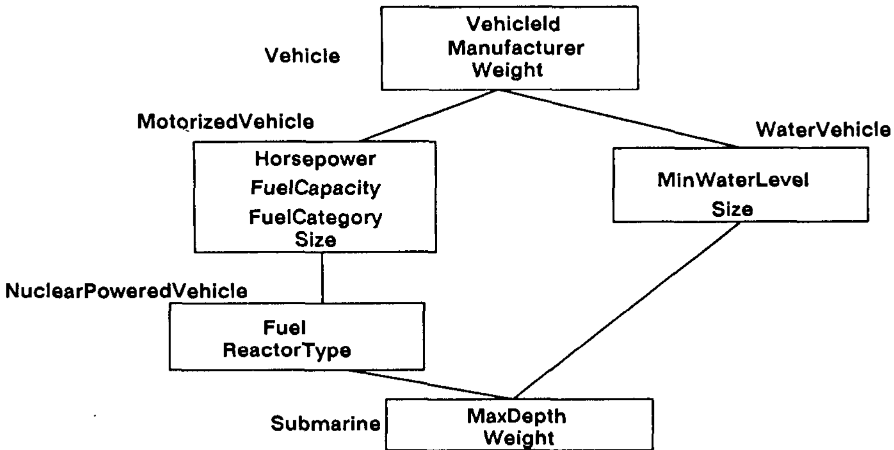

Fig. 1. Resolution of name conflicts among instance variables.

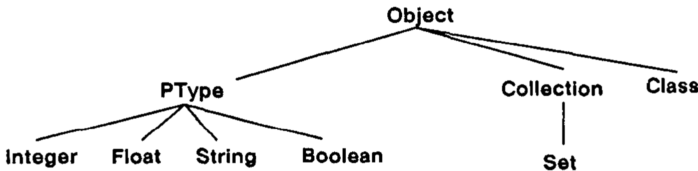

Fig. 2. Primitive class hierarchy in ORION.

to a set, etc. We may define subclasses of the Collection class later in order to add additional capabilities on lists of objects, such as doubly linked lists, stacks, and queues. Each class object belongs to a class, the system-defined class Class. All class objects are instances of this class. To create a new class, a message needs to be sent to the Class class.

For each user-defined class and for the class PType and its subclasses, ORION implicitly defines a corresponding class, a Set-Of class, as a subclass (immediate or indirect) of the Set class. These Set-Of classes form a lattice parallel to the lattice of user-defined classes. For example, the class lattice of Figure 3a is implicitly expanded to the lattice of Figure 3b.

One special instance of the Set-Of class of a user-defined class C is the set of all instances of the class C. Another special instance of the Set-Of class of a user-defined class C is the set of all instances of C or its subclasses. The notion of a set object is particularly important for a database of persistent objects, which outlive programs that created them. While a program is in execution, objects created by a program can be referenced through symbols that point to them. A program's symbol table provides handles into objects. However, a newly started program has direct reference to instances of classes through its symbol table. Instead, the program can refer to the two special set objects in the Set-of class of a class C, thereby referring either to all instances of C, or to all instances of C and its subclasses. A simple naming convention may be used to refer to these

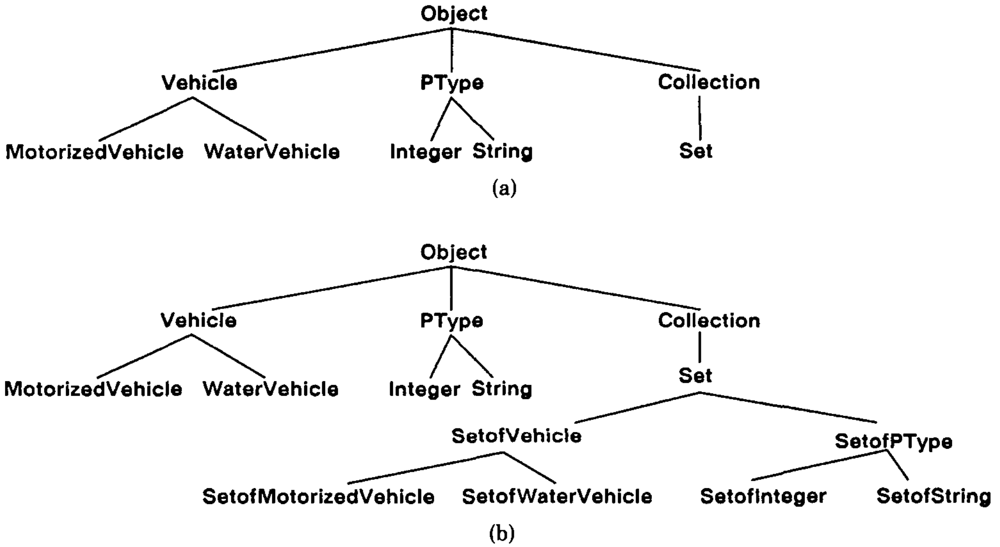

Fig. 3. Expansion of a class hierarchy (a) with set classes (b). (a) Primitive class hierarchy. (b) Class hierarchy expanded with set classes.

objects, for example, the name C to refer to the set object that contains all instances of the class C, and the name C* to refer to the set object that contains all instances of C and its subclasses. Predicate-based queries are messages to these set objects and return subsets of these sets.

For example, for the class MotorizedVehicle, the system defines a class SetofMotorizedVehicle, which has at least one instance, the set object containing all instances of the class MotorizedVehicle. This special set object provides a handle into the instances of the class MotorizedVehicle. Individual instances of MotorizedVehicle may be referenced as elements of this set.

Another motivation for the automatic generation of the Set-Of classes corresponding to user-defined classes is that instance variables often require values that are sets of objects. Just as any other object, set objects must belong to some class. Without the notion of implicitly defined Set-Of classes to which such objects can belong, the user will either have to create a class explicitly to capture the structure and semantics of these objects or treat them as instances of the class OBJECT, thereby losing their semantics.

3. SCHEMA EVOLUTION

ORION applications require considerable flexibility in dynamically defining and modifying the database schema, that is, the class definitions and the inheritance structure of the class lattice [19, 31]. Existing object-oriented systems support only a few types of changes to the schema, without requiring system shutdown. This is the consequence of the fact that existing object-oriented systems are programming language systems. We note that even existing conventional database systems allow only a few types of schema changes: For example, SQL/DS only allows the dynamic creation and deletion of relations (classes) and the

addition of new columns (instance variables) in a relation [14]. This is because the applications they support (conventional record-oriented business applications) do not require more than a few types of schema changes; also the data models they support are not as rich as object-oriented data models.

In this section we first provide a taxonomy of schema change operations supported in ORION. We then present a framework for understanding and enforcing the semantics of each of the schema change operations. The framework consists of the set of properties, which we call *invariants of the class lattice*, and a set of rules for resolving ambiguities in enforcing the invariants. Finally, we describe the semantics of some of the schema change operations to illustrate the application of the schema evolution framework. We refer the reader to our earlier work [4] for more detailed discussions of the contents of this section. A graphics-based schema editor has been implemented to validate the semantics of schema evolution. A detailed presentation of the schema editor will be given in a forthcoming paper.

3.1 Taxonomy of Schema Evolution

Changes to the class lattice can be broadly categorized as (1) changes to the contents of a node, (2) changes to an edge, and (3) changes to a node. ORION allows all three types of changes. These changes can be classified further. In particular, changing the contents of a node implies adding or dropping instance variables or methods, or changing the properties of existing instance variables or methods. The schema change taxonomy is as follows:

(1) Changes to the contents of a node (a class)
- (1.1) Changes to an instance variable
  - (1.1.1) Add a new instance variable to a class
  - (1.1.2) Drop an existing instance variable from a class
  - (1.1.3) Change the Name of an instance variable of a class
  - (1.1.4) Change the Domain of an instance variable of a class
  - (1.1.5) Change the inheritance (Parent) of an instance variable (inherit another instance variable with the same name)
  - (1.1.6) Change the Default value of an instance variable
  - (1.1.7) Manipulate the Shared value of an instance variable
    - (1.1.7.1) Add a Shared value
    - (1.1.7.2) Change the Shared value
    - (1.1.7.3) Drop the Shared value
- (1.2) Changes to a method
  - (1.2.1) Add a new method to a class
  - (1.2.2) Drop an existing method from a class
  - (1.2.3) Change the Name of a method of a class
  - (1.2.4) Change the Code of a method in a class
  - (1.2.5) Change the inheritance (Parent) of a method (inherit another method with the same name)

(2) Changes to an edge
- (2.1) Make a class S a superclass of a class C
- (2.2) Remove a class S from the superclass list of a class C
- (2.3) Change the order of superclasses of a class C

(3) Changes to a node
   (3.1) Add a new class
   (3.2) Drop an existing class
   (3.3) Change the name of a class

3.2 Invariants of Schema Evolution

In this section we summarize the properties of the class lattice that we can extract from our data model. We call these properties *invariants* of the class lattice. Any changes to the class definitions and to the structure of the class lattice must preserve these properties.

3.2.1 *Class Lattice Invariant.* The class lattice is a *rooted* and *connected directed acyclic graph* with labeled edges. The directed acyclic graph (DAG) has exactly one root, the class OBJECT. The DAG is connected; that is, there are no isolated nodes. Edges are labeled such that all edges directed to any given node have distinct labels (the edges are used to aid conflict resolution, as we show in Section 3.3).

3.2.2 *Distinct Name Invariant.* All instance variables and methods of a class, whether locally defined or inherited, must have distinct names.

3.2.3 *Distinct Identity Invariant.* All instance variables and methods of a class have distinct origin. For example, referring back to Figure 1, the class Submarine can inherit the instance variable Weight from either the class WaterVehicle or the class NuclearPoweredVehicle. In both these superclasses, however, Weight has the same origin, namely, the instance variable Weight of the class Vehicle, where Weight was originally defined. Therefore, the class Submarine must have only one occurrence of the instance variable Weight.

3.2.4 *Full Inheritance Invariant.* A class must inherit all instance variables and methods from each of its superclasses. There is no selective inheritance, unless the full inheritance invariant should lead to a violation of the distinct name and distinct identity invariants.

3.2.5 *Domain Compatibility Invariant.* If an instance variable V_2 of a class C is inherited from an instance variable V_1 of a superclass of C, then the domain of V_2 must either be the same as that of V_1, or a subclass of V_1. For example, if the domain of instance variable Manufacturer in the Vehicle class is the Company class, then the Manufacturer of a MotorizedVehicle can be a Company or a subclass of Company, for example, a MotorizedVehicleCompany.

3.3 Rules of Schema Evolution

The invariants of the class lattice hold at every quiescent state of the schema, that is, before and after a schema change operation. They guide the definition of the semantics of every meaningful schema change operation by ensuring that the change does not leave the schema in an *inconsistent* state (one that violates an invariant). Occasionally, however, several meaningful ways of interpreting a schema change will result in a consistent schema. To select one that is semantically the most meaningful, we need *rules* that govern our schema change

semantics. The rules fall into three categories: default conflict resolution rules, property propagation rules, and DAG manipulation rules.

3.3.1 *Default Conflict Resolution Rules.* The following three rules select a single inheritance option whenever there is a name or identity conflict. They ensure that the distinct name and distinct identity invariants are satisfied in a deterministic way. The user may override the default conflict resolution rules by explicit requests to resolve conflicts differently.

Rule 1. If an instance variable is defined within a class C, and its name is the same as that of an instance variable of one of its superclasses, the newly defined instance variable is selected over the conflicting instance variable of the superclass.

Rule 2. If two or more superclasses of a class C have instance variables with the same name, but distinct origin, the instance variable selected for inheritance is that from the first superclass (corresponding to the node with the lowest labeled edge coming into C) among conflicting superclasses.

Rule 3. If two or more superclasses of a class C have instance variables with the same origin, the instance variable with the most specialized (restricted) domain is selected for inheritance. However, if the domains are the same, or if one domain is not a superclass of the other, the instance variable inherited is that of the first superclass among conflicting superclasses.

For example, in Figure 1, if the domain of Manufacturer of NuclearPoweredVehicle is Company, and the domain of Manufacturer of WaterVehicle is WaterVehicleCompany, which is a subclass of Company, the instance variable inherited into the class Submarine is the Manufacturer instance variable from the class WaterVehicle.

3.3.2 *Property Propagation Rules.* The properties of an instance variable, once defined or inherited into a class, can be modified in a number of ways. In particular, its name, domain, default value, or shared value may be changed. Also, an instance variable that is not presently a shared value can be made one, or vice versa. Further, the properties of a method belonging to a class may be modified by changing its name or code. The following rule provides guidelines for supporting all changes to the properties of instance variables and methods.

Rule 4. When the properties of an instance variable or method in a class C are changed, the changes are propagated to all subclasses of C that had inherited them, unless these properties were previously redefined within the subclasses.

For example, if the instance variable Weight of the class Vehicle has its default value changed to 2000, then the same must be done to Weight in all subclasses of Vehicle. However, if Weight had earlier been explicitly assigned a default value of 1000 in the class MotorizedVehicle (which is a subclass of Vehicle), then MotorizedVehicle will not accept the change. Consequently, the change will also not be propagated to the subclasses of MotorizedVehicle that had inherited Weight from MotorizedVehicle.

Rule 4 requires that changes to names of instance variables and methods also be propagated. However, the propagation of name changes or of newly added

instance variables or methods of a class may introduce new conflicts in the subclasses. We take the position that name changes are made primarily to resolve conflicts and as such should not introduce new conflicts. By a similar reasoning, we take the view that new instance variables and methods that give rise to new conflicts should not be propagated. Hence we have the following rule, which modifies Rule 4.

Rule 5. A name change or a newly added instance variable or method is propagated to only those subclasses that encounter no new name conflicts as a consequence of this schema modification. A subclass that does not inherit this modification does not propagate it to its own subclasses. For the purposes of propagation of changes to subclasses, Rule 5 overrides Rule 2.

3.3.3 *DAG Manipulation Rules.* We need a set of rules that govern the addition and deletion of nodes and edges from the class lattice. The following rule ensures that drastic changes are avoided when a new edge is added to a class lattice.

Rule 6. (Edge Addition Rule). If a class A is made a superclass of a class B, then A becomes the last superclass of B. Thus any name conflicts that may be triggered by the addition of this superclass will not require any default resolution; that is, name conflicts can be ignored. If a newly inherited instance variable causes an identity conflict, Rule 3 must be applied to resolve the conflict.

The deletion of an edge from node A to node B may cause node B to become isolated in the case in which class A is the only superclass of class B. The following rule is necessary to prevent such a violation of the class lattice invariant, which requires that the DAG be connected.

Rule 7 (Edge Removal Rule). If class A is the only superclass of class B, and A is removed from the superclass list of B, then B is made an immediate subclass of each of A's superclasses. The ordering of these new superclasses of B is the same as the ordering of superclasses of A (clearly, A and B will now have the same collection of superclasses).

The addition of a new node should not violate the class lattice invariant. If the new node has no superclasses, it becomes an isolated node, violating the class lattice invariant. Hence we have the following rule.

Rule 8 (Node Addition Rule). If no superclasses are specified for a newly added class, the root class OBJECT is the default superclass of the new class.

The deletion of a node A is a three-step operation: first the deletion of all edges from A to its subclasses, then the deletion of all edges directed into A from its superclasses, and finally the deletion of node A itself. We need the following rule to ensure the preservation of the class lattice invariant when deleting a node.

Rule 9 (Node Removal Rule). For the deletion of edges from A to its subclasses, Rule 7 is applied if any of the edges is the only edge to a subclass of A. Further, no system-defined classes can be deleted.

3.4 Semantics of Schema Evolution

In this section we provide a description of the semantics of some of the schema change operations, to illustrate the application of the invariants of the class lattice and the schema change rules. We note that there is one very important aspect of schema evolution that the discussions of this section do not properly address. It concerns methods of a class containing references to inherited instance variables. For example, when an instance variable *V* is dropped from a class S, a method defined in a class C, a subclass of S, that references *V* will no longer be operable. It is possible to efficiently detect methods that may become inoperable as a result of schema change operations. However, we defer to a forthcoming paper detailed discussions of how we address this and other problems with methods in the context of schema evolution.

(1) *Define a new class C.* The new class C may be created as a specialization of an existing class or classes. The latter classes can be specified as the superclasses of the new class. As discussed earlier, the instance variables specified for C will override any conflicting instance variables inherited from the superclasses (by rule 1). If there is a name conflict involving the instance variables that C inherits from its superclasses, default conflict resolution rules 2 and 3 are used, unless the user explicitly overrides the default rules.

The class C may also be defined without any superclasses. In this case, C is made a subclass of Object (rule 8). The user may, at a later time, add superclasses for C, in which case Object will no longer be an immediate superclass of C.

(2) *Add a new instance variable to a class C.* The new instance variable, in case of a conflict with an already inherited instance variable, will override the inherited variable (rule 1). In that case, the inherited instance variable must be dropped from C and replaced with the new instance variable, and existing instances of C will take on the value nil or user-specified default for the new instance variable.

If C has subclasses, they will inherit the new instance variable of C. If there is a conflict with an instance variable that they have already defined or inherited, the new variable is ignored (rule 5). If there is no conflict, the subclasses inherit the new instance variable, together with a default value, if any.

(3) *Drop a class C.* Whenever a class definition is dropped, all its instances are deleted automatically, since instances cannot exist outside of a class. However, subclasses of C, if any, are not dropped; subclasses of C will lose C as their superclass. However, if C was their only superclass, they will gain C's superclasses as their immediate superclasses (rule 9).

Further, when a class C is dropped, its subclasses will lose the instance variables (and methods) that they had previously inherited from C. If, in the process, a subclass of C loses an instance variable *V* that was selected over a conflicting instance variable in another superclass of that subclass, it will now inherit the alternate definition of *V* (to maintain the full inheritance invariant). Consequently, the instances of any such subclass will lose their present values for *V* and inherit the default value (or nil) under the new definition of *V*.

When an instance of the class C is dropped, all objects that reference it will now be referencing a nonexistent object. The user will need to modify those references when they are encountered. ORION will not automatically identify references to nonexistent objects because of the performance overhead.

If the class C being dropped is presently the domain of an instance variable V_1 of some other class, V_1's domain becomes the first superclass of the class C. Of course, the user has the choice of specifying a new domain for V_1.

(4) *Drop an instance variable V from a class C.* The instance variable V is dropped from the definition (and from the instances) of the class C. To maintain the full inheritance invariant, C will inherit V from another superclass if there has been a name conflict involving V. All subclasses of C will also be affected if they have inherited V. If C or a subclass of C has methods that refer to V, such methods will now become invalid. The user can either delete these methods or redefine them to make all references consistent with the new definition of C and its subclasses.

(5) *Change the domain of an instance variable V of a class C.* The domain of an instance variable is itself a class. The domain, class D, of an instance variable V of a class C may only be changed to a superclass of D. The values of existing instances of the class C are not affected in any way. If the domain of an instance variable V must be changed in any other way, V must be dropped, and a new instance variable must be added in its place.

4. COMPOSITE OBJECTS

Many applications require the ability to define and manipulate a set of objects as a single logical entity [3, 9, 15, 19, 23, 28, 31]. For example, a vehicle is an object that contains a body object, which has a set of door objects, and each door has a position object and a color object. In other words, a body object exclusively belongs to (is a part of) a vehicle instance, and a set of doors, in turn, belongs to a body, and so on. In general, a complex object, such as a vehicle, forms a hierarchical structure of exclusive component objects. We define a *composite object* as an object with a hierarchy of exclusive component objects, and refer to the hierarchy of classes to which the objects belong as a *composite object hierarchy.*

The object-oriented data model, in its conventional form, is sufficient to represent a collection of related objects. However, it does not capture the IS-PART-OF relationship between objects; one object simply *references,* but does not own, other objects. A composite object hierarchy captures the IS-PART-OF relationship between a parent class and its component classes, whereas a class hierarchy represents the IS-A relationship between a superclass and its subclasses.

Composite objects add to the integrity features of an object-oriented data model through the notion of dependent objects. A *dependent object* is one whose existence depends on the existence of other objects and that is owned by exactly one object. For example, the body of a vehicle is owned by one specific vehicle and cannot exit without the vehicle that contains it. As such, a dependent object cannot be created if its owner does not already exist. This means that a composite object hierarchy must be instantiated in a top-down fashion; the root object of a composite object hierarchy must be created first, then the objects at the next

level, and so on. When a constituent object of a composite object is deleted, all its dependent objects must also be deleted.

We note that an object may contain references to both dependent objects and independent objects, or to only dependent or independent objects. We use the term *aggregate object* to refer to such a general collection of objects. A composite object is a special case of an aggregate object.

The definition of a set of objects as a composite object also offers an opportunity for performance improvement. ORION considers a composite object as a unit for clustering related objects on disk. This is because, if an application accesses the root object, it is often likely to access all (or most) dependent objects as well. Thus it is advantageous to store all constituents of a composite object as close to one another as possible on secondary storage.

The notion of composite objects has been investigated by various researchers. It has been called a complex object in IBM's experimental extension to SQL/DS [14] and a composite object in LOOPS [5]. Our contribution in this paper is in showing the integration of the data modeling concept of composite objects into an object-oriented data model. In particular, after a formal definition of composite objects, we specify our semantics of composite objects and relate them to object-oriented concepts. We then illustrate the composite object semantics in terms of schema definition, and creation and deletion of composite objects. We also indicate our approach to implementing composite objects, including enforcement of the semantics of composite objects, and physical clustering.

4.1 Definitions

A composite object can be defined in BNF as follows:

⟨Composite Object⟩ ::= ⟨Composite Object Root⟩ (⟨Linked Dependent⟩*),
⟨Linked Dependent⟩ ::= ⟨Instance Variable⟩ ⟨Dependent Object⟩,
⟨Dependent Object⟩ ::= ⟨Leaf Object⟩
| ⟨Dependent Object Root⟩ (⟨Linked Dependent⟩*)
| {⟨Dependent Object⟩*}.

In the above definition, the * is a metasymbol that denotes an indefinite number of occurrences. A composite object has a special instance object, called the *root object*. The root of the composite object is connected to multiple *dependent objects*, each through an instance variable in the root object. Each dependent object can be a simple object (with no dependent objects), or it can itself be the root of a hierarchical structure. A dependent object can also be a set of objects. In a composite object, the same instance object cannot be referenced more than once. Thus the definition of a composite object is a hierarchy of instance objects (and not a general digraph). However, all instance objects within a composite object can be referenced by instance objects that do not belong to the composite object, and these references can have the complete generality of a digraph, including digraphs with cycles.

The instance objects that constitute a composite object belong to classes that are also organized in a hierarchy. This hierarchical collection of classes is called a *composite object schema*. A nonroot class on a composite object schema is called a *component class*. Each nonleaf class on a composite object schema has one or

more instance variables that serve as links, called *composite links*. We call instance variables that serve as composite links *composite instance variables*.

In Figure 4, we illustrate a composite object schema for vehicles. The classes that are connected by bold lines form the composite object schema. The *root class* is the class Vehicle. Through instance variables Body, Drivetrain, and Color, vehicle instances are linked to their dependent objects, which belong to classes AutoBody, AutoDrivetrain, and String. (An instance variable with a primitive domain, such as Integer or String, can always be considered a composite link. A value from a primitive domain can be freely copied; hence every reference to such an object can be exclusive. Thus two vehicles can have the color red because each vehicle refers to a separate string object "red.") The Vehicle class has another instance variable called Manufacturer, but it is not a link to dependent objects. The instances of AutoBody and AutoDrivetrain, in turn, are connected to other dependent objects. A vehicle composite object then is an instance of the class Vehicle, together with an instance of each of the classes AutoBody, Auto-Drivetrain, and String (for Color). The brace in the figure indicates a set object. The instance variable Doors of the class AutoBody represents a set of Door instances, each of which has a Position and Color.

4.2 Semantics of Composite Objects

In this section we define the semantics of composite objects within an object-oriented framework. First, the semantics of a composite link are as follows: If there is a composite link from a class A to a class B through an instance variable V_a of A, an instance of B can be referenced through V_a by only one instance of A. There can be other instance objects that can also reference this instance of B, but any such reference cannot be through another composite link. In other words, if an instance object is referenced through a composite link, it must be the only composite link to the object. For example, an instance of the class Vehicle can have a composite link to an instance of the class AutoBody through the instance variable Body. No other instance of Vehicle can refer to this instance of AutoBody through the instance variable Body. Further, if an instance of some other class, say Inventory, has a reference to this instance of AutoBody, the reference must be through an instance variable that is not a composite link.

The composite link property of an instance variable of a class is inherited by subclasses of that class. For example, if the class Automobile is a subclass of Vehicle, it inherits the instance variable Body from Vehicle. Further, because Body is a composite link in the Vehicle class, it will also be a composite link in the Automobile class.

A composite instance variable may later be changed to a noncomposite instance variable, that is, it may lose the composite link property. If a class A has a composite link to a class B through an instance variable V and V becomes a noncomposite instance variable, then the class B may become the root class of a composite object schema through its composite links to other classes.

However, we do not allow a noncomposite instance variable to acquire the composite link property later. An instance object may be referenced by any number of instances of a class through a noncomposite instance variable. However, a dependent object of a composite object may be referenced by only one

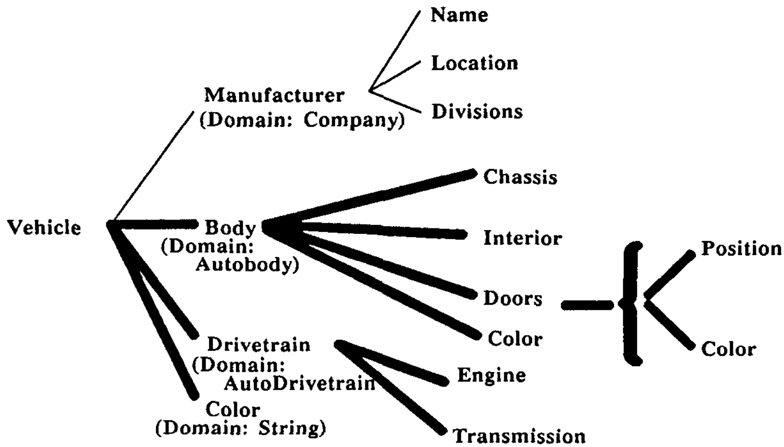

Fig. 4. Vehicle composite hierarchy.

instance of a class through a composite instance variable of the class. Therefore, to change a noncomposite instance variable to a composite instance variable makes it necessary to verify that existing instances are not referenced by more than one instance through the instance variable. This, in turn, makes it necessary to maintain a list of reference counts for each instance object, one reference count for each instance variable through which the instance object may be referenced.

Next, composite objects can further enhance information hiding through the notion of *value propagation* [5]. Default values can be propagated from an instance object to all its dependent objects, thereby simplifying the definition of dependent objects. For example, the color of the body of a vehicle is, by default, the color of the vehicle. We note that value propagation refers to the sharing of the value of an instance variable between instance objects, whereas inheritance is the sharing of the name of an instance variable (and method) between classes.

Values can be propagated only if an object has an instance variable that has the same name as some instance variable of a higher level object. Propagation of a value to a lower level object takes place from the lowest level containing object that has an appropriate value. Further, if the default value of a higher level object is changed, the new value is propagated as the default value of the dependent objects. As an example, in Figure 4 the default color of the doors can be the same as that of the vehicle's body or of the vehicle. If a vehicle's body did not have an instance variable named Color, or (if it did have such an instance variable, but) if the instance variable had no value assigned to it, then every door can assume its default color from the vehicle (bypassing the vehicle's body).

Value propagation is not automatic; it must be specified in the definition of the composite object schema. For example, unless indicated in the definition, the body of a vehicle does not assume the color of the vehicle. Once value propagation is specified, it takes precedence over inheritance from superclasses. For example, let us assume in Figure 4 that the domain of the instance variable Body of the class Vehicle is the class AutoBody, and that AutoBody inherits the instance

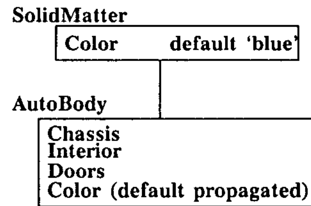

Fig. 5. The class hierarchy of AutoBody.

variable Color from its superclass SolidMatter. This class hierarchy is shown in Figure 5. Let us also assume that the default value of Color in SolidMatter is "blue." The color of a vehicle's body will not be blue; instead, it will assume the color of its containing vehicle.

4.3 Schema Definition, Creation, and Deletion of Composite Objects

A composite object schema is created through composite instance variables. These instance variables have component classes as their domains. For example, the class Vehicle in Figure 4 has a composite link to the class AutoBody through the instance variable Body. The instance variable Body has as domain the class Autobody, and it has the composite link property. The Vehicle class has another instance variable Drivertrain, whose domain is the class AutoDrivetrain, and which is also a composite link. The classes AutoBody and AutoDrivetrain similarly have composite instance variables.

Although Autobody is the domain of the composite instance variable Body of the class Vehicle, it may be used as the domain of other instance variables, including other composite instance variables. In fact, if Vehicle has two subclasses Car and Truck, they both inherit the instance variable Body, along with its domain (AutoBody) and the composite link property. However, an instance of AutoBody can be referenced by only one instance object through a composite instance variable. A particular instance of AutoBody cannot simultaneously be a part of both a Car and a Truck.

An instance object can be made a part of a composite object only at the time of creation of that instance object. The integrity requirement for composite objects is that any instance object within a composite object cannot be referenced through more than one composite link. This integrity requirement is easily enforced. The only way in which a nonnull value can be assigned to a composite instance variable is by simultaneously creating that value (a dependent instance object). Any attempt to assign a nonnull value to a composite instance variable separately is rejected, because in our implementation instance objects within a composite object do not carry the identifier of the composite object to which they belong. Thus, for example, if an existing Body instance were to be separately assigned to a Vehicle instance, it would be prohibitively expensive to determine whether that body is already a part of some other vehicle. If that body were indeed a part of another vehicle, it would then have two parents, violating the integrity constraint on vehicle composite objects.

An instance object, once it is created as a dependent object, cannot have independent existence. Therefore, if any instance object within a composite object is deleted, it causes a recursive deletion of instances that depend on the

object. The parent of this instance object now has a dangling reference. On a subsequent attempt to access an object through such a dangling reference, the application may choose to replace it with a null value.

A dependent object remains a dependent object throughout its existence, unless a composite link is redefined in the schema as a noncomposite link. The only way in which a composite link between instance objects can be severed is by either deleting the dependent object or making it a part of some other composite object through an ExchangePart message.

4.4 Clustering of Composite Objects

In ORION all instances of the same class are placed in the same storage segment. Thus a class is associated with a single segment, and all its instances reside in that segment. The user does not have to be aware of segments; ORION automatically allocates a separate segment for each class. For clustering composite objects, however, it is often advantageous to store instances of multiple classes in the same segment. User assistance is required to determine which classes should share the same segment.

The user may issue a message, a Cluster message, as a hint for ORION to cluster instances of a class with instances of other classes. A Cluster message specifies a list of class names, ListofClassNames. Instances of classes listed in the ListofClassNames are to be placed in a single segment. The initial size of the segment and any later increments to that size may be specified optionally. The user may sometimes need to cluster a new class C with some existing classes that have already been allocated a segment. In such case the user needs to issue a Cluster message, in which the ListofClassNames is a pair, namely, the class C and any of the existing clases with which C should share a segment. C will then share the same segment with the existing classes.

As we have seen already, a dependent object is linked to its parent when it is created; as such, a dependent object can be stored close to its parent. Ideally, the constituents of a composite object should be stored clustered at all times. In general, this requirement is not a difficult one. A composite object can be stored in a sequence of linked pages. If the composite object increases in size, a new page can be acquired and linked in the manner of a B-tree. If a composite object shrinks in size, pages may be released or compacted. The only difficulty seems to arise with the implementation of the ExchangePart message. When two dependent objects (two subtrees) are exchanged between two parent objects, they really should exchange storage positions as well. For implementation simplicity, however, ORION does not recluster objects in response to an ExchangePart message.

5. VERSIONS

There is a general consensus that version control is one of the most important functions in various data-intensive application domains, such as integrated CAD/CAM systems and OISs dealing with compound documents [3, 10, 11, 16–18, 25, 26, 30]. Users in such environments often need to generate and experiment with multiple versions of an object before selecting one that satisfies their requirements.

In this section, we show our approach to integrating version concepts into an object-oriented data model, including some of the salient implementation issues. A full description of our model of versions, along with a preliminary consideration of its implementation, is given in Chou [7]. The model is appropriate for a federated system of a central server and a number of autonomous workstations sharing objects through the server.

5.1 Version Semantics

In our current prototype we distinguish two types of versions on the basis of the types of operations that may be allowed on them. They are transient versions and working versions.

A *transient version* has the following properties:

(1) It can be updated by the user who created it.
(2) It can be deleted by the user who created it.
(3) A new transient version may be derived from an existing transient version. The existing transient version then is "promoted" to a working version.

A *working version* has the following properties:

(1) It is considered stable and cannot be updated.
(2) It can be deleted by its owner.
(3) A transient version can be derived from a working version.
(4) A transient version can be "promoted" to a working version. Promotion may be explicit (user specified) or implicit (system determined).

We impose the update restriction on the working version because it is considered stable, and thus transient versions can be derived from it. If a working version is to be directly updated after one or more transient versions have been derived from it, we need a set of careful update algorithms (for insert, delete, update) that will ensure that the derived versions will not see the updates in the working version.

5.2 Version Name Binding

There are two ways to *bind* an object with another versioned object: static and dynamic. In *static binding*, the reference to an object includes the full name of the object, the object identifier, and the version number. In *dynamic binding* [3, 11, 18], the reference needs to specify only the object identifier and may leave the version number unspecified. The system selects the default version number. Clearly, dynamic binding is useful, since transient or working versions that are referenced may be deleted and new versions created.

We need to examine the issue of selecting default versions for dynamic binding. In other proposals, the default selected is often the "most recent" version. This simple defaulting scheme is not appropriate in our model. One difficulty is that in our model version history is represented in a hierarchy, the *version-derivation hierarchy*. In particular, we allow more than one transient version to be derived from a working version. In a linear-derivation scheme, where only one version may be derived from any version [10], the most recent version has the implicit meaning that it is the "most correct" or "most complete." However,

a version-derivation hierarchy, in which any number of new versions may be derived from any node on the hierarchy any time, potentially has any number of "most recent" versions in this sense. Therefore, we need to allow the user to specify a particular version on the version-derivation hierarchy as the default version. In the absence of a user-specified default, the system selects the version with the "most recent" timestamp as the default.

5.3 Implementation

Because of the performance overhead in supporting versions, we require the application to indicate whether a class is *versionable.* When an instance of a versionable class is created, a *generic object* for that instance is created, along with the first version of that instance. A generic object is essentially a data structure for the version-derivation hierarchy of an instance of a versionable class. It is deleted when the version-derivation hierarchy for its instance contains no versioned object. A generic object consists of the following system-defined instance variables:

(1) an object identifier,
(2) a default version number,
(3) a next-version number,
(4) a version count, and
(5) a set of version descriptors, one for each existing version on the version-derivation hierarchy of the object.

The default version number determines which existing version on the version-derivation hierarchy should be chosen when a partially specified reference is dynamically bound. The next-version number is the version number to be assigned to the next version of the object that will be created. It is incremented after being assigned to the new version.

A version descriptor contains control information for each version on a version-derivation hierarchy. It includes

(1) the version number of the version,
(2) the version number of the parent version,
(3) the identifier of the versioned object, and
(4) the schema version number associated with the version

The version of schema used for version V_i of an object may be different from that used for version V_j derived from V_i. For example, after a transient version is derived, the user may modify the schema for the transient version. Then the original version and the transient version will use different schemas. This is the reason for including the schema version number for each versioned object. We note, however, that a version of schema for an object X is in general shared by multiple versions of X. For example, if a transient version is derived from a working version, both versions may use the same version of schema. A detailed discussion of our proposal for supporting versions of schemas will be given in a forthcoming paper.

A generic object is also an object and as such has an object identifier. Each version of an instance object of a versionable class contains three system-defined

instance variables. One is the identifier of the generic object. The others are the version number of the version and the version status (transient or working). The generic object identifier is required, so that, given a version of an instance object, any other versions of the instance object may be efficiently found. The version number is needed simply to distinguish a version of an instance object from other versions of the instance object. The version status is necessary so that the system may easily reject an update on working versions.

A versioned object is created initially by the *create* command, which creates the generic-object data structure for the object. The *derive* command is used to derive a new transient version and allocate a new version number for it. If the parent was a transient version, it is automatically promoted to a working version. The *replace* operation causes the contents of a transient version to be replaced by a work-space copy the user specifies. A transient version is explicitly promoted to a working version, making the version nonupdatable, through the *promote* command. The user may delete a version or an entire version-derivation hierarchy using the *delete* command. If the delete is against a generic object, all versions of the instance for which the generic object was created are deleted. If a working version is deleted from which other versions have been derived, the version is deleted, but the fact that the version existed is not deleted from the generic object. The user uses the *set_default* command to specify the default version on a version-derivation hierarchy of an object. A specific version number or the keyword "most-recent" may be specified as the default.

6. CONCLUDING REMARKS

In this paper we first provided a brief review of the basic object-oriented concepts that we extracted from existing object-oriented systems to form the basis of an object-oriented data model. Then we elaborated on three major enhancements to the conventional object-oriented data model. First was schema evolution, or the capability of making a wide variety of changes to the database schema, including class definitions and the structure of the class lattice, without requiring a database reorganization or system shutdown. We provided a taxonomy for schema changes that an object-oriented database system should allow and introduced a framework for understanding the semantics of the schema changes. Second was the concept of composite objects. A composite object is a collection of objects that recursively captures the IS-PART-OF relationship between pairs of objects. Composite objects should be used for enforcing this IS-PART-OF relationship and, as units of storage clustering and retrieval for improving system performance. We elaborated on the semantics of composite objects and showed their integration into an object-oriented data model in terms of schema definition and creation and clustering of composite objects. Third was version control. We discussed the semantics of versions and showed how they are integrated into the object-oriented data model.

The basic object-oriented concepts and the three enhancements we discussed in this paper, as well as a number of other important features, are presently being incorporated into ORION. ORION is a prototype object-oriented database system under implementation in the Database Program at MCC as a research vehicle for developing a database technology for object-oriented applications from the

CAD/CAM, AI, and OIS domains. The system is intended to directly support some of the applications under development in the AI/KBS (knowledge base system) Program at MCC and to receive feedback about the performance and functionality of the system from them. Because the AI/KBS applications are being implemented in Common LISP, in order to be closely coupled with them, we are implementing ORION in Common LISP to execute on the Symbolics LISP machines. The application interface to ORION then is an object-oriented extension to LISP, much as Flavors [29] and ObjectLISP [22] are, and includes message passing protocol, class lattice, and property inheritance along the class lattice. We are integrating ORION message-passing protocol with LISP function calls so that, to the extent possible, ORION applications can view both ORION objects and LISP structures without having to move from one programming environment to another.

ACKNOWLEDGMENTS

We thank Fred Lochovsky, editor of this issue, and the reviewers for their constructive comments on an earlier draft of this paper. They helped us to correct some technical inaccuracies and improve the overall presentation of the paper.

REFERENCES

1. AFSARMANESH, H., KNAPP, D., MCLEOD, D., AND PARKER, A. An object-oriented approach to VLSI/CAD. In *Proceedings of the 11th International Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.). VLDB Endowment, Saratoga, Calif., 1985.
2. AHLSEN, M., BJORNERSTEDT, A., BRITTS, S., HULTEN, C., AND SODERLUND, L. An architecture for object management in OIS. *ACM Trans. Office Inf. Syst.* 2, 3 (July 1984), 173–196.
3. ATWOOD, T. M. An object-oriented DBMS for design support applications. In *Proceedings of IEEE First International Conference on Computer-Aided Technologies 85* (Montreal, Canada, Sept.). IEEE, New York, 1985, pp. 299–307.
4. BANERJEE, J., KIM, H. J., KIM, W., AND KORTH, H. F. Schema evolution in object-oriented persistent databases. In *Proceedings of the 6th Advanced Database Symposium* (Tokyo, Japan, Aug.). Information Processing Society of Japan's Special Interest Group on Database Systems, 1986, pp. 23–31.
5. BOBROW, D. G., AND STEFIK, M. *The LOOPS Manual.* Xerox PARC, Palo Alto, Calif., 1983.
6. BOBROW, D. G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M., AND ZDYBEL, F. *CommonLoops: Merging Common Lisp and Object-Oriented Programming,* Intelligent Systems Laboratory Series ISL-85-8. Xerox PARC, Palo Alto, Calif., 1985.
7. CHOU, H. T., AND KIM, W. A unifying framework for version control in a CAD environment. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan). VLDB Endowment, Saratoga, Calif., 1986, pp. 336–344.
8. CHRISTODOULAKIS, S., VANDERBROEK, J., LI, J., WAN, S., WANG, Y., PAPA, M., AND BERTINO, E. Development of a multimedia information system for an office environment. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, 1984). VLDB Endowment, Saratoga, Calif., Calif. pp. 261–271.
9. COPELAND, G., AND MAIER, D. Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (June 18–21, Boston, Mass.). ACM, New York, 1984, pp. 316–325.
10. DADAM, P., LUM, V., AND WERNER, H. Integration of time versions into a relational database system. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984). VLDB Endowment, Saratoga, Calif., pp. 509–522.
11. DITTRICH, K., AND LORIE, R. Version support for engineering database systems. IBM Research Rep. RJ4769, IBM Research, San Jose, Calif., July 1985.

12. GOLDBERG, A. Introducing the Smalltalk-80 system. *BYTE 6*, 8 (Aug. 1981), 14–26.
13. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
14. IBM CORPORATION. SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corporation, San Jose, Jan. 1981.
15. IEEE. *Database Eng. 8*, 4 (Dec. 1985), Special Issue on Object-Oriented Systems, F. Lochovsky, Ed. IEEE, New York.
16. KAISER, G., AND HABERMANN, A. An environment for system version control. Tech. Rep., Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., Nov. 1982.
17. KATZ, R., AND LEHMAN, T. Database support for versions and alternatives of large design files. *IEEE Trans. Softw. Eng. SE-10*, 2 (Mar. 1984), 191–200.
18. KATZ, R., CHANG, E., AND BHATEJA, R. Version modeling concepts for computer-aided design databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 28–30). ACM, New York, 1986.
19. KIM, W. CAD database requirements. Tech. Rep., MCC, Austin, Tex., July 1985.
20. KRASNER, G., Ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, Mass., 1983.
21. LMI, INC. *Lisp Machine Manual.* LMI, Cambridge, Mass., 1983.
22. LMI, INC. *ObjectLISP User Manual.* LMI, Cambridge, Mass., 1985.
23. LORIE, R., AND PLOUFFE, W. Complex objects and their use in design transactions. In *Proc. ACM Database Week: Eng. Design Appl.* (May 1983), 115–121.
24. MAIER, D., STEIN, J., OTIS, A., AND PURDY, A. Development of an object-oriented DBMS. Tech. Rep. CS/E-86-005, Oregon Graduate Center, Beaverton, Oreg., Apr. 1986.
25. MCLEOD, D., NARAYANASWAMY, K., AND BAPA RAO, K. An Approach to Information Management for CAD/VLSI Applications. In *Proceedings of the Conference on Databases for Engineering Applications, Database Week 1983* (*ACM*, May). ACM, New York, 1983, pp. 39–50.
26. ROCHKIND, M. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4 (Dec. 1975), 364–370.
27. SMITH, J., AND SMITH, D. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst. 2*, 2 (June 1977), 105–133.
28. STEFIK, M., AND BOBROW, D. G. Object-oriented programming: Themes and variations. *AI Magazine* (Jan. 1986), 40–62.
29. SYMBOLICS, INC. *FLAV Objects, Message Passing, and Flavors.* Symbolics, Cambridge, Mass., 1984.
30. TICHY, W. Design implementation, and evaluation of a revision control system. In *Proceedings of the 6th IEEE International Conference on Software Engineering* (Sept.). IEEE, New York, 1982.
31. WOELK, D., KIM, W., AND LUTHER, W. An object-oriented approach to multimedia databases. In *Proceedings of ACM SIGMOD Conference on the Mangement of Data* (Washington, D.C., May 28–30). ACM, New York, 1986.

Received August 1986; revised November 1986; accepted December 1986