Semantics and Implementation of Schema Evolution in Object-Oriented Databases

Jay Banerjee, Won Kim
MCC
3500 West Balcones Center Drive
Austin, Texas 78759

Hyoung-Joo Kim*, Henry F Korth *
Dept of Computer Sciences
University of Texas
Austin, Texas 78712

Abstract

Object-oriented programming is well-suited to such data-intensive application domains as CAD/CAM, AI, and OIS (office information systems) with multimedia documents At MCC we have built a prototype object-oriented database system, called ORION It adds persistence and sharability to objects created and manipulated in applications implemented in an object-oriented programming environment One of the important requirements of these applications is schema evolution, that is, the ability to dynamically make a wide variety of changes to the database schema In this paper, following a brief review of the object-oriented data model that we support in ORION, we establish a framework for supporting schema evolution, define the semantics of schema evolution, and discuss its implementation

1. Introduction

In recent years, object-oriented programming [GOLD81, GOLD83, BOBR83, CURR84, SYMB84, LMI85] has gained a tremendous popularity in the design and implementation of emerging data-intensive application systems These include artificial intelligence (AI) [STEF86], computer-aided design and manufacturing (CAD/CAM) [AFSA86], and office information systems (OIS) with multi-media documents [IEEE85, AHLS84, WOEL86] 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 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

* These authors work was supported in part by NSF Grant DCR-8507224

Within the Database Program at MCC, we have built a prototype object-oriented database system, called ORION, for the purposes of adding persistence and sharability to objects, and supporting various advanced functions that applications from the CAD/CAM, AI, and OIS domains require One of the important advanced functions that these applications require is *schema evolution*, the ability for the users to dynamically change the database schema [KIM85, WOEL86] The types of changes required include creation and deletion of a class, alteration of the IS-A relationship between classes, addition and deletion of instance variables and methods, and so on In object-oriented databases, there are two types of schema evolution changes to the class definitions, and changes to the structure of a class hierarchy Existing object-oriented systems support only a few types of changes to the schema This is a consequence of the fact that existing object-oriented systems are programming language systems We note that even existing database systems allow only a few types of schema changes for example, SQL/DS allows only dynamic creation and deletion of relations (classes) and addition of new columns (instance variables) in a relation [IBM81] This is because the applications they support (conventional record-oriented business applications) do not require more than a few types of schema changes, and also the data models they support are not as rich as object-oriented data models

One of the most important, and innovative, feature of ORION is its support of schema evolution Our research proceeded in three phases First, we established a taxonomy of over 20 useful schema changes under the ORION object-oriented data model Second, we defined the semantics of each schema change To do this, we first identified a set of invariant properties of an object-oriented schema which must be preserved across schema changes Then we defined a set of rules for selecting the most meaningful way to preserve the invariant properties for each of the schema changes where there is more than one way the invariant properties can theoretically be preserved By applying these sets of invariant properties and rules, we derived globally consistent and meaningful semantics of each of the schema changes Third, we developed an implementation strategy for the entire spectrum of schema changes, which does not require database re-organization or system shutdown The objective, and contributions, of this paper is to present the results of all three phases of our research into schema evolution under the ORION data model The entire set of schema changes we defined in our taxonomy has been implemented in ORION

Further a graphics-based schema editor has been implemented on top of ORION, to provide a friendly interface to the end users of ORION and to help us validate the correctness of our schema-evolution framework

Section 2 provides a brief review of the ORION object-oriented data model In Section 3, we provide a formal framework for schema evolution, in terms of a set of invariant properties of an object-oriented schema and rules for selecting a meaningful option in preserving the invariant properties We also define the semantics of each of the schema changes by applying these invariant properties and rules Further, we outline formal proofs of the completeness of our framework for schema evolution, and correctness of the semantics of schema changes In Section 4, we discuss implementation issues, including storage format for ORION objects, impacts of schema changes on instances, and inherited methods with embedded references to instance variables and other methods

2. The ORION Data Model

Existing object-oriented systems exhibit significant differences in their support of the object-oriented paradigm, [STEF86] provides an excellent account of different variations of the object concepts In this section we review briefly the aspects of the ORION data model which are necessary to establish a context for the discussions of schema evolution in Sections 3 and 4 We refer the reader to [BANE87] for a full description of our data model In Section 2 1, we review the basic object concepts which we have selected for our data model In Section 2 2, we discuss the concept of composite objects we have incorporated into our data model as an enhancement to the standard object-oriented data model

2 1 Basic Concepts

objects, instance variables, methods, and messages

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 is the object itself More complex objects contain instance variables, through which they reference other objects, which in turn contain instance variables

The behavior of an object is encapsulated in *methods* Methods consist of code that manipulate or return the state of an object Methods are a part of the definition of the object However, methods, as well as instance variables, are not visible from outside of the object Objects can communicate with one another through messages 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

classes

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 which implements that message is found in the definition of the class

shared-value, and default-valued variables

ORION supports two features to further reduce redundant storage and specification of objects shared-value and default-valued instance variables For a *shared-value variable* of a class, all instances of the class take on the same value This is similar to the class variable concept in Smalltalk [GOLD83] For a *default-valued variable,* those instances of a class whose value for the instance variable is not specified take on a specified default value

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, say 'air', 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 TakeoffDistance is not specified, the value 300 is taken for that instance variable

class hierarchy, and inheritance

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 pair of nodes represents the IS-A relationship, that is, the lower level node is a specialization of the higher level node (and conversely, the higher level node is a generalization of the lower level node) For a pair of classes on a class hierarchy, the higher level class is called a *superclass* of the lower level class, and the lower level class a *subclass* of the higher level class The instance variables and methods (collectively called properties) specified for a class are inherited (shared) by all its subclasses Additional properties may be specified for each of the subclasses A class inherits 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*

domains of instance variables

In object-oriented systems, the domain of an instance variable (which corresponds to data type in conventional programming languages) is a class In ORION we allow the domain of an instance variable to be bound 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

class lattice, multiple inheritance, and name-conflict resolution

Smalltalk [GOLD81] originally restricted a class to have only a single superclass, thus limiting the class hierarchy to a tree Most other object-oriented systems, as well as the recent version of Smalltalk, have relaxed this restriction In these

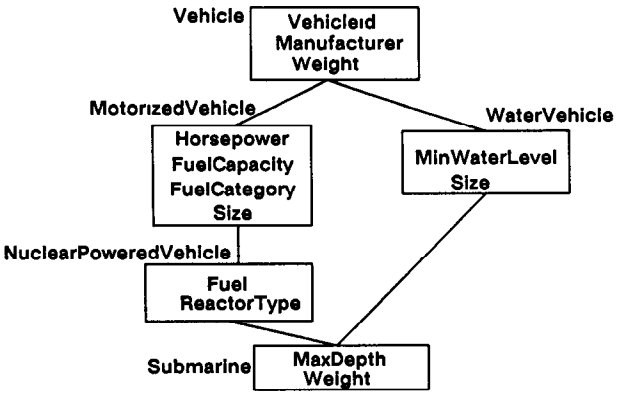


Figure 1. Resolution of name conflicts among instance variables

systems (and in ORION) a class can have more than one superclass, generalizing the class hierarchy to a lattice In a class lattice, a class has multiple superclasses and thus inherits properties from each of the superclasses This feature is often referred to as *multiple inheritance* [LMI85, STEF86] (In this paper, we use the term *lattice*, as in the literature on object-oriented systems, to mean a directed acyclic graph structure)

The class lattice simplifies data modeling and often requires fewer classes to be specified than with a class hierarchy However, it gives rise to two types of conflicts 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 among the superclasses of a class, this is purely a consequence of multiple inheritance

Name conflicts between a class and its superclasses are resolved in all systems we are aware of 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 of C, which the application will have specified For example, as shown in Figure 1, the class Submarine has to inherit an instance variable Size either from the superclass WaterVehicle (which defined Size) or from NuclearPoweredVehicle (which inherited Size from its superclass MotorizedVehicle) If in the definition of the class Submarine, NuclearPoweredVehicle was 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, ORION allows the user to explicitly change this permutation at any time It also provides ways in which the user can override the default conflict resolution, by explicitly inheriting an instance variable or method of the user's choice from a number of conflicting instance variables or methods, or inheriting more than instance variables or methods by first renaming them

2 2 Composite Objects

The object-oriented data model, in its conventional form, is sufficient to represent a collection of related objects As we have seen, it captures the IS-A relationship between a class and its superclass, and it allows an object to reference other objects through its instance variables However, it does not capture the often useful IS-PART-OF relationship between an object and objects it references The notion of composite objects explicitly captures this relationship A *composite object* is a collection of related instances that form a hierarchical structure that captures the IS-PART-OF relationship between an object and its parent (In the literature, what we call composite objects have variously been called complex objects [LORI83, KIM85, KIM87], composite objects [BOBR85], and aggregation hierarchies [ATWO85]) ORION uses the knowledge of composite objects not only to enforce the semantics of composite objects (to be discussed), but also to physically cluster the constituent objects of composite objects, so as to minimize the I/O cost of retrieving composite objects

definitions

A composite object has a single root object, and the root references multiple children objects, each through an instance variable Each child object can in turn reference their own children objects, again through instance variables A parent object *exclusively owns* children objects, and as such the existence of children objects is predicated on the existence of their parent Children objects of an object are thus *dependent objects* The instances 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 composite object schema consists of a single *root class* and a number of *dependent classes*

In Figure 2, 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 The class Vehicle has another instance variable called Manufacturer, but it is not a link to a dependent object 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, AutoDrivetrain, and String (for Color) The brace in

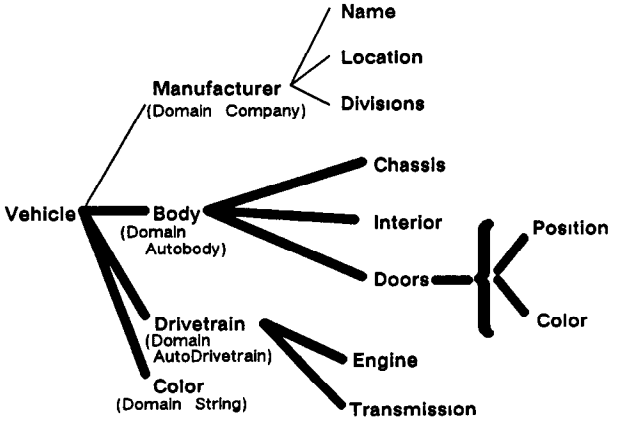


Figure 2 Vehicle modeled as a hierarchy of parts

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

Each non-leaf class on a composite object schema has one or more instance variables, called *composite instance variables*, that serve as links to dependent classes In other words, a composite object schema is created through composite instance variables, which have dependent classes as their domains We will call the link between a class and the domain of a composite instance variable of the class a *composite link* For example, the class Vehicle in Figure 2 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 Drivetrain, whose domain is the class AutoDrivetrain and which is also a composite link The classes AutoBody and AutoDrivetrain, similarly, have composite instance variables

semantics

A composite link has two properties that add to the integrity features of the conventional object-oriented data model One is that a composite link from a class A to a class B, through an instance variable Va of A, constrains an instance of B to be referenced through Va by only one instance of A There can be other objects that can also reference this instance of B, but any such reference cannot be through another composite link

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 a non-composite instance variable

Another property of a composite link is that the object which is referenced through a composite instance variable is a dependent object whose existence depends on the existence of the referencing object For example, the body of a vehicle is not only owned by one specific vehicle, but also cannot exist without the vehicle This means that a dependent object cannot be created if its owner does not already exist As such, a composite object must be instantiated in a top-down fashion, the root object of a composite object 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

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

3. Formal Framework for Schema Evolution

In this section, we present our formal framework for schema evolution under the ORION data model, and then, using the framework, define the semantics of schema evolution we have implemented in ORION We emphasize that, although our framework has been developed for our particular data model, we believe that our methodology for the development of the framework is applicable to most main-stream object-oriented systems This is because our model has incorporated all the basic object concepts for which there is a wide acceptance, and has enhanced the basic object-oriented model with the notion of composite objects

Our formal framework consists of a set of properties of the schema called *invariants*, and a set of *rules* that guide the selection of the most meaningful way of preserving the invariants for those schema changes that allow more than one meaningful way The invariants 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, by ensuring that the change does not leave the schema in an inconsistent state (one that violates any invariant) However, for some schema changes, the schema invariants can be preserved in more than one way The set of rules that we have guides the selection of one most meaningful way

3 1 Invariants of Schema Evolution

We have been able to identify five invariants of the object-oriented schema from the ORION data model They define the consistency requirements of the class lattice under our data model

Class Lattice Invariant

The class lattice is a *rooted* and *connected directed acyclic graph* with named nodes and labeled edges The DAG has only one root, a system-defined class called OBJECT The DAG is connected, that is, there are no isolated nodes Every node is reachable from the root Nodes are named, and each node in the DAG has a unique name Edges are labeled such that all edges directed to any given node have distinct labels

Distinct Name Invariant

All instance variables of a class, whether defined or inherited, have distinct names Similarly, all methods of a class, whether defined or inherited, must have distinct names

Distinct Identity (Origin) Invariant

All instance variables, and methods, of a class have distinct identity (origin) For example, in Figure 1, the class Submarine can inherit the instance variable Weight from either the class WaterVehicle or NuclearPoweredVehicle However, in both these superclasses, 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

Full Inheritance Invariant

A class inherits all instance variables and methods from each of its superclasses, except when full inheritance causes a violation of the distinct name and distinct identity invariants In other words, if two instance variables have distinct origin but the same name in two different superclasses, at least one of them must be inherited If two instance variables have the same origin in two different superclasses, only one of them must be inherited For example, in Figure 1, Submarine must inherit Size, whether it is from NuclearPoweredVehicle or from WaterVehicle, or even from both (by assigning new names, in order to maintain the distinct name invariant) Further, Submarine must inherit Weight only once, from either NuclearPoweredVehicle or WaterVehicle

Domain Compatibility Invariant

If an instance variable V2 of a class C is inherited from an instance variable V1 of a superclass of C, then the domain of V2 is either the same as that of V1, or a subclass of V1 For example, if the domain of the instance variable Manufacturer in

the Vehicle class is the Company class, then the domain of Manufacturer in the MotorizedVehicle class, a subclass of Vehicle, can be Company or a subclass of Company, say, MotorizedVehicleCompany

Another aspect of the domain compatibility invariant is that the shared value or default value of an instance variable must be an instance of the class that is the domain of that instance variable

3 2 Rules of Schema Evolution

A class lattice in a quiescent state must preserve all the invariants For some of the schema changes, however, there is more than one way to preserve the invariants For example, if there is a name conflict among instance variables to be inherited from superclasses, the full inheritance invariant requires that at least one of the instance variables be inherited, but it does not say which In order to guide the selection of one option among many in an algorithmic and meaningful way, we have established twelve essential rules, including some which we have adopted from existing object-oriented systems These rules fall into four categories default conflict resolution rules, property propagation rules, DAG manipulation rules, and composite object rules

Default Conflict Resolution Rules

The following three rules permit the selection of 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 ORION user may, however, override these 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 locally defined instance variable is selected over that of the superclass The same applies to methods

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 numbered edge coming into C) among conflicting superclasses If two or more superclasses have methods with the same name, the method inherited is from the first among conflicting superclasses

Inheritance of methods with embedded references to inherited instance variables and methods present an interesting problem We will address this problem in Section 4 2

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 Manufacturer instance variable of the class Submarine is inherited from the class WaterVehicle

Property Propagation and Change 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, shared value, or the composite link property may be changed Also, an instance variable that is not shared-valued can be made shared-valued, 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 inherited them, unless these properties have been re-defined 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 assigned a new default value of 1000 in the class MotorizedVehicle (which is a subclass of Vehicle), then MotorizedVehicle will not accept the change Consequently, the change is not propagated to any subclass of MotorizedVehicle that inherited Weight from MotorizedVehicle

Rule 4 requires that changes to names of instance variables and methods are also propagated However, propagation of name changes or new instance variables and methods in 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 newly added instance variable or method, or a name change to an 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

Requests for changes to instance variables must sometimes be rejected In particular, in ORION the domain of an instance variable, once defined or inherited, can be generalized, that is, changed to one of the superclasses on its superclass chain, but cannot be specialized Otherwise, the domain may be incongruous with that of the shared value, default value, or the values in the instances of the class The domain may be generalized, but only to the extent that the domain compatibility invariant is not violated For example, the domain of Manufacturer in the class MotorizedVehicle can be generalized from MotorizedVehicleCompany to Company (which is the domain of Manufacturer in the class Vehicle), but not to Organization, which is a superclass of Company

Rule 6 **(Domain Change Rule)** The domain of an instance variable can only be generalized Further, the domain of an inherited instance variable cannot be generalized beyond the domain of the original instance variable

DAG Manipulation Rules

We need a set of rules that govern the addition and deletion of nodes and edges from the class lattice First, the addition of an edge from node A to node B on a class lattice means that class A is made a new superclass of class B The following rule ensures that drastic changes are avoided when a new edge is added to a class lattice

Rule 7 **(Edge Addition Rule)** If a class A is made a superclass of a class B, then A becomes the last superclass of B In other words, the edge from A to B is assigned the highest label among all edges directed into B Thus, any name conflicts, that may be triggered by the addition of this superclass, can be ignored However, if a newly inherited instance variable causes an identity conflict, Rule 3 must be applied to resolve it

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

Rule 8 **(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

A corollary to Rule 8 is that, if the root class OBJECT is the only superclass of a class B, any attempt to remove the edge from OBJECT to B is rejected If the edge is removed, node B would become isolated, since OBJECT has no superclass to which B may be linked as a new superclass

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 9 **(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

Rule 10 **(Node Removal Rule)** For the deletion of edges from A to its subclasses, Rule 8 is applied if any of the edges is the only edge to a subclass of A Further, any attempt to delete a system-defined class, such as the class OBJECT, is rejected

Composite Object Rules

A composite instance variable may be changed to a non-composite instance variable, that is, it may lose the composite link property However, we do not allow a non-composite instance variable to later acquire the composite link property The reason is that an object may be referenced by any number of instances of a class through a non-composite instance variable, but a dependent object of a composite object may be referenced by only one instance of a class through a composite instance variable To change a non-composite instance variable to a composite instance variable, it is 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 with each object, one reference count for each instance variable through which the object may be referenced We avoid this complexity in ORION by not permitting a non-composite instance variable to be changed to a composite instance variable

Rule 11 **(Composite Link Rule)** The composite link property may be dropped from a composite instance variable, however, it may not be added to a non-composite instance variable

The integrity of a composite object lies in the fact that all dependent objects owe their existence to their parents In particular, if a parent object is deleted, all its dependent objects are deleted, and if a parent object loses a composite instance variable, the dependent object referenced is deleted However, we allow objects to *disown* their dependents, if their composite instance variables are changed to non-composite Disowned objects are not deleted when their previous parents are deleted, since they are no longer dependent on the existence of their previous parents Hence we have the following rule

Rule 12 If a composite instance variable of an object X is changed to a non-composite, X disowns object Y which it references through the instance variable The object X continues to reference the object Y, however, deletion of X will not cause Y to also be deleted

3 3 Taxonomy and Semantics of Schema Evolution

In this subsection, we classify all schema changes that we support in ORION, and define the semantics of schema changes, using our schema evolution invariants and rules 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 Our 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 1 8) Drop the composite link property of an instance variable
- (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

We now define the semantics of each of these schema changes, while showing how the schema evolution rules are applied to maintain the schema evolution invariants

(1 1) Change an instance variable

(1 1 1) Add a new instance variable V to a class C

Suppose first that the new instance variable V causes no new conflicts in the class C or any of its subclasses The full

inheritance invariant requires V to be inherited by all subclasses of C Since the instance variable is new, there can be no new identity conflicts, unless there are two or more paths from C to any of its subclasses, in which case Rule 3 is used to preserve the distinct identity invariant

If the new instance variable causes a name conflict with an inherited instance variable, by Rule 1 the new instance variable will override the inherited instance variable If the old instance variable was also locally defined in C, it is replaced by the new definition In any case, the new instance variable is propagated to all subclasses of C If there is a name conflict in a subclass, the new instance variable is not inherited (Rule 5) This does not violate the full inheritance invariant, since the subclass already contains an instance variable of the same name Once the new instance variable V is added to C or any of its subclasses, existing instances of the class receive the user-specified default value, if there is one, or the nil value (The existing instances are not updated at the time of schema change We will describe our actual implementation in Section 4 1 2)

(1 1 2) Drop an instance variable V from a class C

V must have been defined in the class C, it is not possible to drop an inherited instance variable If V is dropped from C, it must also be dropped recursively from the subclasses that inherited it (Rule 4) If C or any of its subclasses has other superclasses that have instance variables of the same name as that of V, it inherits one of those instance variables This is a consequence of the full inheritance invariant The default conflict resolution rules (Rules 1, 2 and 3) are used to determine which new instance variable to inherit The necessary change in inheritance is handled as in operation 1 1 5 (to be described shortly) In case V must be dropped from C or any of its subclasses without a replacement, existing instances of these classes lose their values for V (Again, the existing instances are not updated at the time of schema change, as we will show in Section 4 1 2)

(1 1 3) Change the name of an instance variable V of a class C

We take the view that name changes are made primarily to resolve conflicts, and as such they should not introduce new conflicts Therefore, if a name change causes any conflict within the class C, the change is rejected If the name change is accepted, it is propagated to subclasses of C that have inherited V from C Rule 5 requires the name change to be propagated only if it does not give rise to new conflicts in the subclasses Further, by Rule 4, name change propagation is inhibited in the subclasses that have explicitly changed the name of their inherited instance variable V

(1 1 4) Change the domain of an instance variable V of a class C

By Rule 6, the domain of an existing instance variable can only be generalized, further, the domain compatibility invariant must not be violated The propagation of the domain change in C to the subclasses of C is governed by Rule 4 Thus, domain change propagation is inhibited in those subclasses that have explicitly changed the domain of their inherited instance variable V

(1 1 5) Change the inheritance of an instance variable V of a class C

This change requires that an instance variable V, presently inherited from a superclass S1, be inherited from another superclass S2 Let us refer to the instance variable in S1 as V1, and that in S2 as V2 Of course, V1 and V2 have the same name or the same origin (or both) If V1 and V2 have distinct origins, the change of inheritance in C results in the dropping of the present instance variable inherited from S1, and the addition of the instance variable from S2 These operations are also propagated to the subclasses of C

If V1 and V2 have the same origin, we need to consider two cases If the domain of V2 is the same as that of V1, or V2 is a superclass of the domain of V1, the properties (domain, parent, default, shared) of V1 are changed, and the changes are propagated according to Rule 4 Otherwise, V1 and V2 are treated as if they have distinct origins, that is, V1 is dropped from C, and V2 is added to C

(1 1 6) Change the default value of an instance variable V of a class C

The default value of every instance variable is either explicitly specified in the schema, or is the nil value Adding a new default value to an instance variable V is equivalent to changing its value from the nil value to a non-nil value Dropping the default value of V is equivalent to changing the value to nil The domain compatibility invariant requires that the changed default value of V should be an instance of the domain of V Propagation of the changed default value to the subclasses of C is governed by Rule 4

(1 1 7 1) Add a new shared value for an instance variable V of a class C

This operation converts a non-shared-value instance variable V to a shared-value instance variable The domain compatibility invariant requires that the shared value of V should be an instance of the domain of V Propagation of the new shared value to the subclasses of C is governed by Rule 4

(1 1 7 2) Change a shared value for an instance variable V of a class C

The new shared value should be within the domain of V so that the domain compatibility invariant is preserved Propagation of this new shared value to the subclasses of C is governed by Rule 4

(1 1 7 3) Drop a shared value for an instance variable V of a class C

This operation changes a shared-value instance variable V to a non-shared one V will now have a default value of nil Propagation of this change to the subclasses of C is governed by Rule 4

(1 1 8) Drop the composite link property of an instance variable

A composite instance variable may be changed to non-composite, but not vice versa (Rule 11) When the composite link property of an instance variable V of a class C is changed to non-composite, the change is propagated to the subclasses of C Further, by Rule 12, instances of C and its subclasses disown the objects they reference through V

(1 2) Change a method

The rules to apply for changes to methods, that is, for operations 1 2 1, 1 2 2, 1 2 3, 1 2 4, and 1 2 5, are easily inferred from operations 1 1 1, 1 1 2, 1 1 3, 1 1 4 and 1 1 5, respectively

(2) Change an edge

(2 1) Make a class S a superclass of a class C

To preserve the class lattice invariant, the addition of a new edge from S to C must not introduce a cycle in the class lattice C and its subclasses inherit instance variables and methods from S in accordance with Rule 7 In case of identity conflicts during

the propagation of instance variables to the subclasses of C, Rule 3 is applied Operations 1 1 1 and 1 2 1 are applied, respectively, to add instance variables and methods of S to C

(2 2) Remove a class S from the superclass list of a class C

To preserve the class lattice invariant, the deletion of an edge from S to C must not cause the class lattice DAG to become disconnected In case S is the only superclass of C, Rule 8 is applied, the immediate superclasses of S now become the immediate superclasses of C as well, while the ordering of these superclasses with respect to S remains the same for C Thus, C does not lose any instance variables or methods that were inherited from the superclasses of S C only loses those instance variables and methods that were defined in S If the deletion of the edge from S to C does not leave the DAG disconnected, C is left with one fewer superclasses, and it must drop the instances variables and methods it had inherited from S The operations for dropping an instance variable (operation 1 1 2) and a method (operation 1 2 2) are applied, respectively, for each instance variable and method to be dropped from C

(2 3) Change the order of superclasses of a class C

This operation causes a complete re-evaluation of the inheritance of instance variables and methods in C, in accordance with Rules 1, 2 and 3 In particular, instance variables and methods that partake in name conflicts may have to be replaced by others in accordance with the default conflict resolution rules Any change in inheritance is then handled as in operations 1 1 5 and 1 2 5

(3) Change a node

(3 1) Add a new class C

If no superclasses of C are specified, by Rule 9 the class OBJECT becomes the superclass of C If multiple superclasses are specified, the full inheritance invariant requires all instance variables and methods from all superclasses of C to be inherited, unless there are name or identity conflicts If there are any such conflicts, the default conflict resolution rules (Rules 1, 2, and 3) are used to preserve the distinct name and distinct identity invariants

(3 2) Drop an existing class C

All edges from C to its subclasses are dropped, using operation 2 2 Next, all edges from the superclasses of C into C are removed Finally, the definition of C is dropped, and C is removed from the DAG The subclasses of C continue to exist If the class C was the domain of an instance variable V1 of another class C1, V1 is assigned a new domain, namely the first superclass of the dropped class C This assignment is done when the domain of V1 is actually needed, such as when adding a new instance of C1, changing the value of V1 in some instance of C1, etc

(3 3) Change the name of a class C

To maintain the class lattice invariant, it is ensured that the new name is unique among all class names in the class lattice

3 4 Completeness and Correctness of Schema Evolution

The ORION taxonomy of schema evolution appears intuitively to capture all 'interesting' types of schema changes One interesting, and important, question to consider is whether the ORION schema evolution taxonomy indeed subsumes every possible type of schema change (i e, the completeness issue) Another interesting question is whether every ORION schema change operation generates only valid schemas, that preserve the invariants (i e, the soundness issue) In this section, we outline our proofs for the soundness and completeness of an essential set of ORION schema changes

Our approach to showing completeness is based on a simple formal model [KIM86], called a *property inheritance graph* (PIG), which has only the essential characteristics of the ORION schema evolution model The part of the framework we focus on is the manipulation of the class lattice using our multiple inheritance mechanism In ORION, the set of schema change operations we allow on the contents of a node of a class lattice (i e , the properties of a class) has been determined from our intuition about application requirements As such, this set may grow or shrink Therefore, showing the completeness of all ORION schema changes appears to be neither feasible nor meaningful

The PIG model is based on a single-rooted, directed acyclic graph corresponding to a class lattice Associated with each vertex in this graph are a set of named properties These properties correspond to instance variables and methods We define a *name resolution operator* O over property values, to capture name conflict resolution for multiple inheritance in ORION Using the O operator, we define a PIG to be a pair (V,E), where V is a set of labeled nodes, and E a set of labeled edges The labels on the nodes are the set of properties associated with that node The labels on edges correspond directly to the edge labels in ORION

A PIG must satisfy a pair of syntactic constraints expressed in terms of the DAG and the O operator These constraints enforce relationships among the property sets associated with nodes in the PIG The relationships correspond to inheritance in ORION If there is an edge (a,b) in the DAG, then all properties associated with node a must be associated also with node b Cases corresponding to multiple inheritance are resolved using the O operator We represent an operation in the PIG model as a mapping from the set of all PIGs to itself

In the formal model, we define 8 operations, corresponding to only the essential ORION schema changes We prove that every legal PIG is achievable using a set of 8 operations (i e , completeness) Further, we show that the basic set of operations cannot generate a DAG that violates the syntactic rules which characterize a PIG (i e , soundness) We use the properties of the PIG model to characterize the power of ORION schema change operations

Below we list the 8 PIG operations, without formally defining them The first two operate on the properties of a node in the PIG, and correspond to the ORION operations for adding and dropping an instance variable or a method The remaining six operate on the DAG structure of the PIG, and corresponds to the ORION DAG operations We emphasize that we exclude all operations for changing the contents of the nodes in the ORION class lattice, except operations 1 1 1, 1 2 1,1 1 2, and 1 2 2

1 PIG (Op1) - Add a new property This corresponds to ORION (1 1 1 or 1 2 1) "Add a new instance variable (method)"

2 PIG (Op2) - Drop an existing property This corresponds to ORION (1 1 2 or 1 2 2) "Drop an existing instance variable (method)"

3 PIG (Op3) - Add an edge This corresponds to ORION (2 1) "Make a class S a superclass of a class C"

4 PIG (Op4) - Drop an edge This corresponds to ORION (2 2) "Remove a class S from the superclass list of a class C"

5 PIG (Op5) - Change the order of two incoming edges of a node This corresponds to ORION (2 3) "Change the order of superclasses of a class C"

6 PIG (Op6) - Add a new node This corresponds to ORION (3 1) "Add a new class", however, Op6 can add a new node with only a single superclass

7 PIG (Op7) - Drop an existing node This corresponds to ORION (3 2) "Delete an existing class"

8 PIG (Op8) - Rename the label of a node This corresponds to ORION (3 3) "Change the name of a class"

Theorem 1 guarantees that if G is a PIG and G' is constructed from G by applying an operation in the PIG operation set {Op1, ,Op8}, then G' is also a PIG

Theorem 1 [Soundness] The class of PIGs is closed under the PIG operations Op1 - Op8

Proof We have to show for each PIG operation that when the operation is applied to a PIG, the resulting graph is also a PIG The proof is rather laborious, and is omitted here It is given in [KIM86]

The completeness proof consists of the next two lemmas

Lemma 1 For any PIG, there is a finite sequence of Op7 that reduces the PIG to a PIG with one root node

Proof It is obvious that we can apply Op7 repeatedly, in leaf to root order, to remove all but the root node of a PIG

Lemma 2 There is a finite sequence of {Op1,Op3,Op6} that generates any PIG

Proof Suppose we have two PIGs G and G' Let G be an arbitrary PIG with a finite number of nodes and edges and G' be a PIG with only the root node The following construction process can generate a PIG from G', which is equivalent to G

Traverse G in breadth-first order, and perform the following for each node x of G visited

1 Add a node x' to G', where x' has the same set of properties as that of x, using Op1 and Op6

2 For each incoming edge of x, in the order in which the edges are numbered, add a corresponding incoming edge to x', using Op3

The resulting G' is equivalent to G, since G' has all the nodes and edges in G (and no additional nodes and edges), as well as the same contents for each of the nodes

Theorem 2 Given two arbitrary PIGs, G1 and G2, there is a finite sequence F of {Op1,Op3,Op6,Op7}, such that F(G1) = G2

Proof We can get G2 by first reducing G1 to a PIG with only root node by lemma 1, and then building G2 from the root node by lemma 2

Since {Op1,Op3,Op6,Op7} is a subset of {Op1,Op2,Op3,Op4,Op5,Op6,Op7,Op8}, completeness is obvious from theorem 2

Theorem 3 (Completeness) Given two arbitrary PIGs, G1 and G2, there is a finite sequence F of PIG operations Op1 - Op8, such that F(G1) = G2

4. Implementation of Schema Evolution

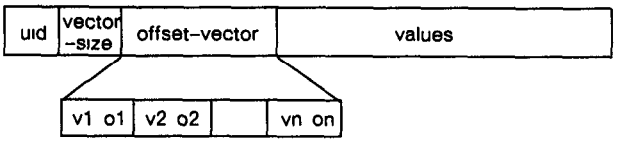
In this section, we will consider implementation of schema evolution in ORION In Section 4 1, we analyze each of the schema change operations to determine its impact on the instances that exist at the time of schema change This analysis makes it clear that some types of schema change, for example, dropping of an instance variable, makes it necessary to update existing instances We identify all schema change operations that logically require updates to existing instances, that is, database reorganization, and describe our implementation which completely avoids database reorganization In Section 4 2 we bring out a problem that multiple inheritance causes on the validity of inherited methods, and describe our solution

4 1 Schema Evolution without a Database Reorganization

It is obvious that such schema change operations as adding or dropping an instance variable logically require existing instances to be updated For example, if an instance variable V of a class C is dropped, the values of V must be removed from all existing instances of C Since a class can have many instances, this is potentially a very expensive undertaking Further, when it takes a long time to update all instances of a class, the class and its instances become unavailable for a long time Hence, it makes sense to avoid updating the instances when the class definition is changed, and instead to merely reflect the changes in the instances presented to the user For example, when the instance variable V of a class C is dropped, existing instances of C are not updated However, when instances of C are later fetched, the values of V are screened from the instances that existed before the schema change

4 1.1 storage format

To support schema evolution without a database reorganization, we designed the storage format for disk-resident objects as shown below



The uid is the globally unique identifier of an object It consists of two parts the unique identifier of the class to which the object belongs, and the unique identifier of the object within the class The vector-size is the number of pairs in the offset-vector The offset vector consists of (vi, oi) pairs, one for each instance variable for which the object has an explicitly specified value In each (vi, oi) pair, vi is the identifier of an instance variable, and oi is the offset of its value in the values part of the object storage format (An instance variable is itself an instance of an ORION system class Instance Variable (similar to a system catalog in conventional database system), and, as such, has a unique identifier) A value can be a primitive value (such as an integer, string, etc), or a reference to another instance, namely, the uid of the referenced object If an object has a default value for an instance variable, or if the instance variable is shared valued, that instance variable does not appear in the storage format The shared value and the default value of shared-value and default-valued instance variables are stored in the system class Instance Variable in which we maintain information about all instance variables in the database

A composite object consists of many instances Unlike the complex object implementation described in [LORI83], in ORION

we do not include in instances the uid of their parent object or that of the root of the composite object This not only saves storage space, but also simplifies the operation of dropping the composite link property from an instance variable (i e , disowning dependent objects) Since dependent objects do not carry the uid of the root of the composite object, there is no need to update them when they are disowned

4 1 2 Impacts of schema changes on existing instances

We now analyze the *impact* of each schema change on existing instances, that is, whether it makes it logically necessary to update any instances For those schema changes which impact the instances, we will describe how ORION avoids actually updating the instances

(1 1) Change an instance variable

(1 1 1) Add a new instance variable V to a class C

For each existing instance in C and subclasses of C that inherit V, there is no explicitly specified value for V When an instance is fetched into memory (on a user request), the system fills in the appropriate default value or nil for the new instance variable Since every instance variable is assigned a unique identifier, there is no possibility that the disk-resident value of a deleted instance variable will be incorrectly presented as the value of the new instance variable

(1 1 2) Drop an instance variable V from a class C

In each existing instance of C and subclasses of C that have to drop V outright, as we have already seen, there is a value for V in the disk format of that instance (unless the value is a default) These instances are left untouched When such an instance is fetched into memory, the value for V is screened

In case the dropped instance variable V has the composite link property, the objects referenced (owned) by instances of C or any subclass of C through V are deleted If other objects are recursively dependent on these objects, they too are deleted

(1 1 3) Change the name of an instance variable V of a class C

There is no impact on the instances of C

(1 1 4) Change the domain of an instance variable V of a class C

We have seen earlier that the domain of an instance variable can only be generalized Existing values of an instance variable V will, therefore, continue to belong to V's domain even after a change is made to the domain As a consequence, instances of C are not affected at all

(1 1 5) Change the inheritance of an instance variable V of a class C

Unless the origin of V is the same as that of the new instance variable, and unless the domain of the new instance variable is the same that of V, or it is a superclass of the domain of V, this operation causes the dropping of one inherited instance variable in favor of another Instances need not be modified When an instance is actually accessed (on a user request), the system screens out the deleted instance variable, and supplies the default value of the new instance variable

(1 1 6) Change the default value of an instance variable V of class C

There is no impact on the existing instances of C A default value is always stored in the definition of an instance variable, never in the instances

(1 1 7 1) Add a new shared value for an instance variable V of a class C

This operation changes V from non-shared to shared Existing instances are left untouched When existing instances are later fetched into memory, the values of V are ignored, and replaced with the shared value

(1 1 7 2) Change a shared value for an instance variable V of a class C

There is no impact on the existing instances of C

(1 1 7 3) Drop a shared value for an instance variable V of a class C

This operation changes V from shared to non-shared The default value assigned is nil, and all existing instances must have a nil value for V The existing instances of V do not have to be changed However, if V was non-shared at time t1, then was changed to shared at time t2, and now is changed back to non-shared at time t3, existing instances may have some explicit values for V specified at time t1, and they must be ignored Unfortunately, they will not be ignored, since V is no longer shared-valued Our solution is to assign a new instance variable identifier to V Then, the existing instances of C will have the old (deleted) identifier of V, and will therefore be ignored

(1 1 8) Drop the Composite link property of an instance variable V of a class C

By Rule 12, instances of C and its subclasses disown the objects they reference through V Instances do not carry the identifier of their parents or the composite objects they belong to, hence, there is no impact

(1 2) Change a method of the class C

There is no impact on the existing instances of C Methods appear only in the definition of the class

(2) Change an edge

(2 1) Make a class S a superclass of a class C

This operation requires adding instance variables (operation 1 1 1)

(2 2) Remove a class S from the superclass list of a class C

This operation requires dropping existing instance variables (operation 1 1 2)

(2 3) Change the order of superclasses of a class C

If this operation causes some instance variables to be replaced by others, operation 1 1 1 is used to add instance variables, and operation 1 1 2 is used to drop instance variables

(3) Change a node

(3 1) Add a new class C

Since C is a new class and has no existing instances, there is no impact

(3 2) Drop an existing class C

All instances of C must be dropped If these instances are referenced by existing instances of other classes, the user will have to modify such references upon failing to retrieve the deleted instances of C When a class C is dropped, it may also require some instance variables from subclasses of C to be dropped (operation 1 1 2)

If the dropped class C is a part of a composite object schema, not only should the instances of C be deleted, but also those that depend on those instances Objects that are (recursively) dependent on the instances of C are also deleted

(3 3) Change the name of a class C

There is no impact on the instances of C

4 2 Embedded References in Inherited Methods

A method can have embedded references to other methods and instance variables An embedded reference to a method is analogous to a subroutine call in conventional (control-oriented) programming languages An embedded reference to an instance variable accesses a part of the local state of an object Such references present no special problem in the presence of single-inheritance, that is, when the class lattice is restricted to a hierarchy However, multiple inheritance introduces inevitable name conflicts A method can make inconsistent references to other methods and instance variables In this section, we first describe this problem, and then outline our solution, which we have not yet implemented in ORION

Problem

We will use Figure 3 to illustrate the inconsistent reference problem with inherited methods The class C has two superclasses A and B Class A has locally defined method Ma, and a locally defined instance variable Vx Ma has an embedded reference to Vx Class B has a locally defined method Mb, and a locally defined instance variable Vx Mb also has an embedded reference to Vx However, Vx of A and Vx of B represent two different semantic concepts They only happen to have the same name When the class C is created, it inherits Ma and Mb, but the conflict involving the name Vx forces C to inherit Vx from its first superclass A (due to default conflict resolution) When a message Mb is now sent to an instance of C, the body of the method Mb is invoked Mb has a reference to Vx, but the Vx of that instance is the one inherited from A This is an inconsistent reference, because Mb was intended to function correctly only with respect to the semantics and properties of Vx of B

Solution

We now outline our approach to solving the inconsistent reference problem To illustrate our solution, we again refer to Figure 3 Suppose that C inherits Vx from A, and retains the name Vx C also inherits Vx from B, but renames it Vy When a message Mb is sent to an instance of C, the code defined in B is executed Within that code, there is a reference to Vx However, since Mb is being executed on behalf of an instance of C (rather than B), the reference to Vx must automatically be replaced by a reference to Vy To make this possible, Class C maintains the following information (1) If Vx is referenced from any method defined in A or C, use Vx (2) If Vx is referenced from any method defined in B, use Vy The correct class for an instance

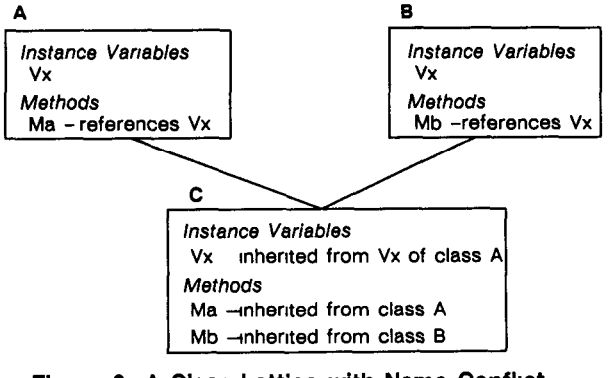


Figure 3 A Class Lattice with Name Conflict

variable referenced in a method is called the *context* of the instance variable The semantics of a reference will be determined by context In our example, Class C will contain the following information for its instance variables

(1) For Vx, a reference to Vx of A, since the properties of Vx are the same as those of Vx of A
For Vx, the following *context lists* are also included
(C Vx) (A Vx) (B Vy)
Each context list is a pair, where the first component is the name of a context, and the second is the name of an instance variable known to the class C The context lists dictate the following If any method to be applied on an instance of C is defined in C, and it has an embedded reference to Vx, use the definition of Vx in C (in this case, a reference to Vx of A) If a method is defined in A, and it references Vx, again use the definition of Vx in C If a method is defined in B, and it references Vx use the definition of Vy in C

(2) For Vy, a reference to Vx of B, since the properties of Vy are the same as those of Vx of B
For Vy, there is no context list There is no ambiguity with respect to Vy

(3) If C has locally defined instance variables (not shown in Figure 3), each of them is associated with its list of properties (domain, default value, etc) If the name of such an instance variable V conflicts with that of an instance variable in a superclass, V will also include context lists

Like instance variables, methods must also have context lists in order to handle name conflicts Whenever an instance variable V (or a method) of a class C is referenced from a method M, if V has no context list, the reference is unambiguous since there can be no name conflict that involves V In case V does have context lists, it is necessary to know where M is defined If M is defined in a certain superclass, and one of the context lists of V contains the name of that superclass, then the instance variable name associated with that list must be used (in place of V) If none of the context lists of V contains the name of that class, the reference to V is erroneous, and must be rejected

For the context lists approach to function properly, the system maintains a global workspace called the *context stack*, which is initially empty Whenever a method is invoked, it pushes the name of its class (that is the class in which it is defined) into the context stack Just before the method finishes its execution, it pops the stack If, within the body of that method, a reference is made to an instance variable or method, the class of the referencing method can be known by simply examining (but not popping) the top element of the context stack

The above mechanism relies on the fact that the effect of every push operation done by a method is reversed by a pop operation when the method finishes The user does not write code to pop and push The user is not even aware of the presence of the context stack The user only supplies the body of the method The system fills in the prologue (push) and the epilogue (pop) for the method After a push operation is executed, the system must guarantee that it is reversed when the method completes execution Even if the method should make an exit from the middle of its body (normally or abnormally, due to an error), the system must detect any such exit and execute the pop operation

Algorithms for the generation and maintenance of context lists in the presence of extensive schema changes, along with a number of other solutions we investigated for the problem of inconsistent references in inherited methods, will be reported in a forthcoming paper

5. Summary and Concluding Remarks

In this paper, we presented the results of our research into various issues of schema evolution, that is, dynamic definition and subsequent changes to a database schema in an object-oriented database environment These results have been fully incorporated into ORION, a prototype object-oriented database system developed at MCC After a brief review of the data model that ORION supports, we presented a formal framework for schema evolution, and defined the semantics of schema changes using the framework The framework is based on a graph-theoretic model of the class lattice It consists of five invariants and twelve rules Invariants are those properties of a class lattice that must be preserved before and after any schema change The rules guide the selection of one most meaningful option for preserving the invariants, in cases where more than one option is possible We also outlined proofs of soundness and completeness of our schema evolution framework based on a simple formal model that captures the essential characteristics of a class lattice with multiple inheritance

A number of schema change operations make it logically necessary to update a potentially large number of instances that existed before the schema change Our implementation of schema evolution completely avoids this database reorganization or system shutdown We have also implemented a graphics-based schema editor on top of ORION, as a friendly interface to the schema management features of ORION and as a tool for empirically validating the correctness and usefulness of the ORION schema evolution framework

References

[AFSA86] Afsarmanesh, H , D Knapp, D McLeod, and A Parker "An Object-Oriented Approach to VLSI/CAD," in *Proc Intl Conf on Very Large Data Bases*, August 1985, Stockholm, Sweden

[AHLS84] Ahlsen M , A Bjornerstedt, S Britts, C Hulten, and L Soderlund "An Architecture for Object Management in OIS," *ACM Trans on Office Information Systems*, vol 2, no 3, July 1984, pp 173-196

[ATWO85] Atwood, T M "An Object-Oriented DBMS for Design Support Applications," *Proc IEEE COMPINT 85*, Montreal, Canada,pp 299-307

[BANE87] Banerjee, J , et al "Data Model Issues for Object-Oriented Applications," to appear in *ACM Trans on Office Information Systems*, April 1987

[BOBR83] Bobrow, D G and M Stefik *The LOOPS Manual*, Xerox PARC, Palo Alto, CA , 1983

[BOBR85] Bobrow, D G , K Kahn, G Kiczales, L Masinter, M Stefik, and F Zdybel *CommonLoops Merging Common Lisp and Object-Oriented Programming*, Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC, Palo Alto, CA , 1985

[CURR84] Curry, G A and R M Ayers "Experience with Traits in the Xerox Star Workstation," *IEEE Trans on Software Engineering*, vol SE-10, no 5, September 1984, pp 519-527

[GOLD81] Goldberg, A "Introducing the Smalltalk-80 System," *Byte*, vol 6, no 8, August 1981, pp 14-26

[GOLD83] Goldberg, A and D Robson *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, MA 1983

[IBM81] SQL/Data System Concepts and Facilities GH24-5013-0, File No S370-50, IBM Corporation, Jan 1981

[IEEE85] *Database Engineering*, IEEE Computer Society, vol 8, no 4, December 1985 special issue on Object-Oriented Systems (edited by F Lochovsky)

[KIM85] Kim, W "CAD Database Requirements - Rev 1," *MCC Technical Report DB-058-85*, July 1985

[KIM86] Kim, H J , H F Korth, J Banerjee, and W Kim "Property Inheritance Graph A Formal Model of Multiple Inheritance in Object-Oriented Databases," unpublished memo, Dept of Computer Sciences, University of Texas, Austin, Texas, Dec 1986

[KIM87] Kim, W , H T Chou, and J Banerjee "Operations and Implementation of Composite Objects," to appear in *Proc 3rd Intl Conf on Data Engineering*, Feb 1987, Los Angeles, Calif

[LMI85] *ObjectLISP User Manual*, LMI, Cambridge, MA, 1985

[LORI83] Lorie, R and W Plouffe "Complex Objects and Their Use in Design Transactions," *In Proc Databases for Engineering Applications*, Database Week 1983 (ACM), May 1983, pp 115-121

[STEF86] Stefik, M , and D G Bobrow "Object-Oriented Programming Themes and Variations," *The AI Magazine*, January 1986, pp 40-62

[SYMB84] *FLAV Objects, Message Passing, and Flavors*, Symbolics, Inc , Cambridge, MA, 1984

[WOEL86] Woelk, D , W Kim, and W Luther "An Object-Oriented Approach to Multimedia Databases," in *Proc ACM SIGMOD Conf on the Management of Data*, Washington D C , May 1986