SCHEMA VERSIONS AND DAG REARRANGEMENT VIEWS IN OBJECT-ORIENTED DATABASES

Hyoung-Joo Kim and Henry F. Korth

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-05 February 1988

SCHEMA VERSIONS AND DAG REARRANGEMENT VIEWS IN OBJECT-ORIENTED DATABASES

Hyoung-Joo Kim, Henry F. Korth

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

An important requirement of non-traditional database applications such as computer aided design, artificial intelligence, and office information systems with multimedia documents is the support of *application evolution*. Application evolution includes evolution of object schemas as well as evolution of objects in the application.

We provided a framework of schema evolution in [BKKK86, BKKK87] and the framework was realized in an object-oriented database system, ORION, at MCC. In this paper, we extend this schema evolution framework by allowing schema versions and DAG rearrangement views in object-oriented databases. We present a technique that enables users to manipulate schema versions explicitly and maintain schema evolution histories. For completeness, we integrate our model with the object version model formulated by H.T. Chou and W. Kim [CK86].

We identify new types of view, called *DAG rearrangement views*, of composite objects and class hierarchies. We present a set of operators for defining DAG rearrangement views. We identify sets of composite object views with the property that queries on the views are processable on instances of the original composite object schema. We also discuss how instances would be viewed and reorganized in DAG rearrangement views of class hierarchies.

1. Introduction

The successful use of database management systems in data-processing applications has created a substantial amount of interest in applying database techniques to such areas as knowledge bases and artificial intelligence (AI) [SB86], computer-aided design (CAD) [AKMP86], and office information systems (OIS) [IEEE85, Ahls84, WKL86]. In order to provide the additional semantics necessary to model these new applications, many researchers, including those referenced above, have adapted the object-oriented programming paradigm [GR83, BS83, CA84, Symb84] to serve as a data model.

In order to use the object-oriented approach in a database system, it is necessary to add persistence and sharability to the object-oriented programming paradigm. Several database systems based on this approach are under implementation, including GEMSTONE [MOP85], IRIS [Fish87], and ORION [Ban87]. In order to meet the requirements of new database applications, recently developed database systems have advanced features that were not available in conventional database systems such as SQL/DS [IBM81] and INGRES [SWKH76].

An important requirement of non-traditional database applications such as CAD/CAM, AI, and OIS is the support of application evolution. Application evolution includes evolution of object schemas as well as evolution of objects in the application. We provided a framework of schema evolution in [BKKK86, BKKK87] and the framework was realized in an object-oriented database system, ORION, at MCC [Ban87].

[1] Research partially supported by NSF Grant DCR-8507724

In the framework, whenever a schema definition is updated, the previous schema is changed to a new one and existing instances of the previous schema are modified in order to comply with the new schema (i.e., overriding the previous schema and its instances).

In this paper, we extend our schema evolution framework by allowing *schema versions* and *DAG rearrangement views* in object–oriented databases. Although there has been substantial research on object versions [DL85, CK86, KL84, KCB86], the issue of schema versions has not been investigated in the database literature. In non-traditional applications, users will probably define and modify a database schema through trial and error, thus it is important to allow flexibility in the definition of a database schema [WKL86]. The approach to schema versions and views presented in this paper contribute flexibility to the dynamic evolution of non-traditional database applications.

We shall present a technique that enables users to manipulate schema versions explicitly and maintain schema evolution histories in object-oriented databases. Our solution for schema versions is consistent with our schema evolution framework, is designed to minimize *storage redundancy* and allows us to avoid the problem of *update anomaly*. For completeness, we integrate our schema version model with the object version model formulated by H.T. Chou and W. Kim [CK86].

One way of representing variants and alternatives of a schema is to allow views of the schema. Views in object-oriented databases are more versatile than those in relational databases. Two distinct notions in object-oriented data models are are described by directed acyclic graphs (DAG): *composite objects* and *class hierarchies*. Conventional views in relational databases are constructed via combinations of relational operators such as select, project, and join. Views in object-oriented databases include rearrangement of DAG structures (both composite objects and class hierarchies) as well as conventional views in relational databases. We present sets of operators for defining DAG rearrangement views of composite objects and class hierarchies respectively. We identify sets of composite object views with the property that queries on the views can be processed on instances of the original composite object schema. We also discuss how instances would be viewed and reorganized in DAG rearrangement views of class hierarchies.

In section 2, we review the core concepts of object-oriented data models, with emphasis on the ORION data model. Section 3 gives an overview of our schema evolution framework. In section 4 we define the semantics of schema versions and discuss integration of our schema version model with the object version model formulated by [CK86]. Semantics of DAG rearrangement views and operations for constructing DAG rearrangement views are introduced in section 5. Section 6 presents an operational interface (user commands) for manipulating schema versions and DAG rearrangement views. A summary section completes the paper.

2. Object-Oriented Data Models

In this section we review the core concepts of object-oriented data models, which are also the major concepts of the ORION data model. We also indicate some assumptions or conventions in the ORION data model.

2.1 Core Concepts

Object-oriented data models support the usual features of object-oriented languages, including the notions of classes, subclasses, class hierarchies, and objects. Our data model is that of the ORION system [Ban87]. Each entity in an ORION database is an *object*. Objects include *instance variables* that describe the state of the object. Instance variables may themselves be objects with their own internal state, or they may be *primitive objects* such as integers and strings which have no instance variables. Objects also include *methods*

which contain code used to manipulate the object or return part of its state. These methods are invoked from outside the object by means of *messages*. Thus, the public interface of an object is a collection of messages to which the object responds by returning an object.

Although each object has its own set of instance variables and methods, several objects may have the same *types* of instance variables and the same methods. Such objects are grouped into a *class* and are said to be *instances* of the class. Usually each instance of a class has its own instance variables. If, however, all instances must have the same value for some instance variable, that variable is called a *shared-value variable*. A default value can be defined for a variable. This value is assigned to all instances for which a value is not specified. Such variables are called *default-valued variables*. The *domain* of an instance variable is a class. The domain of an instance variable is to be bound to a specific class and all subclasses of the class.

Similar classes are grouped together into *superclasses*. The result is a directed acyclic graph (DAG) containing an edge (C_1, C_2) if class C_1 is a superclass of C_2. A class inherits properties (instance variables and methods) from its immediate superclasses, and thus, inductively, from every class C for which a path exists to it from C. The class-superclass relationship (C_1, C_2) is an "ISA" relationship in the sense that every instance of a class is also in instance of the superclass. Using the terminology of the entity-relationship model (see, e.g., [KS86]), we say that C_1 is a *generalization* of C_2 and C_2 is a *specialization* of C_1. The class OBJECT is defined to be the root of the DAG.

Because we allow the use of a DAG to represent the ISA relationship among classes, it is possible for a class to inherit properties from several superclasses. This is called *multiple inheritance* in [GOLD83, STEF86]. This leads to possible naming conflicts between properties inherited from superclasses. Another source of conflict is the possibility that a locally-defined class variable or method has the same name as an inherited property. These conflicts are resolved by giving the local definition precedence. Other conflicts are resolved based upon a user-supplied total ordering of the superclasses. This ordering can be changed at any time by the user. Furthermore, the user may override the default conflict resolution scheme either by renaming or by explicitly choosing the property to be inherited.

2.2 Instances with One and Only One Type

The inheritance mechanism causes inclusion relationships among sets of instances. For example, if a class S is a subclass of a class C, any instance of S is an instance of C. These inclusion relationships should be maintained to make ISA relationships among classes of a class hierarchy meaningful.

There are two ways of positioning (physically storing) instances in class hierarchies. One can allow an instance to belong to more than one class, or one can require that an instance belong to *one and only one* class. While some object-oriented systems, such as GALILEO [ACO85], ADAPLEX [SFL80], and TAXIS [MBW80], follow the former approach, others, such as ORION [Ban87], GEMSTONE [MOP86], and COMMONLOOPS [Bob85] follow the latter approach. The class hierarchies in Figure 1.a and 1.b illustrate the former and latter approach respectively.

We believe that requiring instances to belong to one and only one class is better in applications that involve many instances (i.e., data-intensive applications) since this reduces data redundancy. By allowing instances to belong to more than one class, storage waste and update costs are increased. As shown in Figure 1.a., if the user deletes an instance (e.g., H.J. Kim) from the UNIVERSITY-PERSON, the system must also delete corresponding instances from all subclasses of UNIVERSITY-PERSON, i.e., GRAD-STUDENT, STAFF, and TA, in order to keep the database in a consistent state. The same argument applies to insert and update operations. One disadvantage of requiring instances to belong to one and only one class is that the

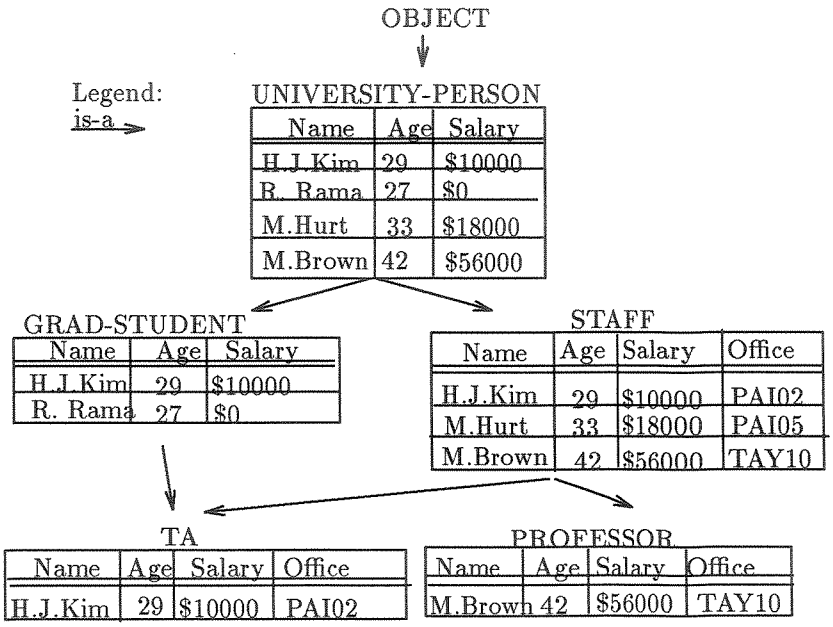


Figure 1.a: Instances belonging to more than one class

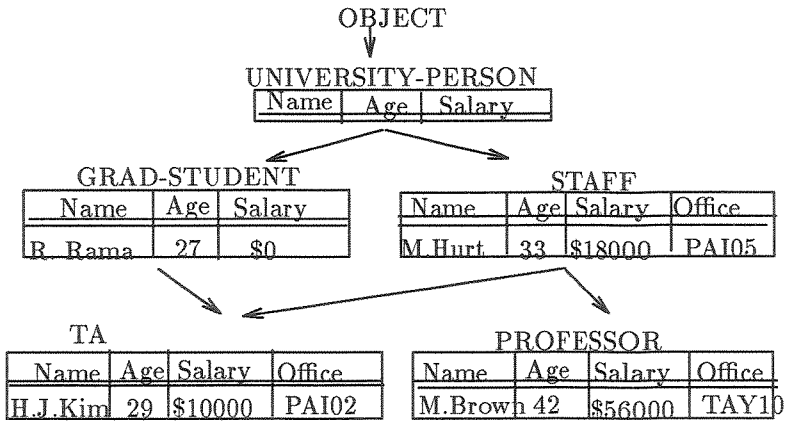


Figure 1.b: Instances belonging to one and only one class

query language is more complicated (addressed in next section); two types retrieval and update operations are needed.

The difference between the two approaches can also be seen in Figure 2. In Figure 2.a, the object 09 can belong to NUCLEARVEHICLE and WATERVEHICLE (and, of course, to VEHICLE), whereas in Figure 2.b, a new class, called NUCLEARWATERVEHICLE, must be created for capturing the object 09 in a single class.

In this paper, we follow the approach in Figure 1.b and Figure 2.b and assume that instances can not belong to more than one class, as in the ORION data model. This assumption influences the semantics of schema versioning.

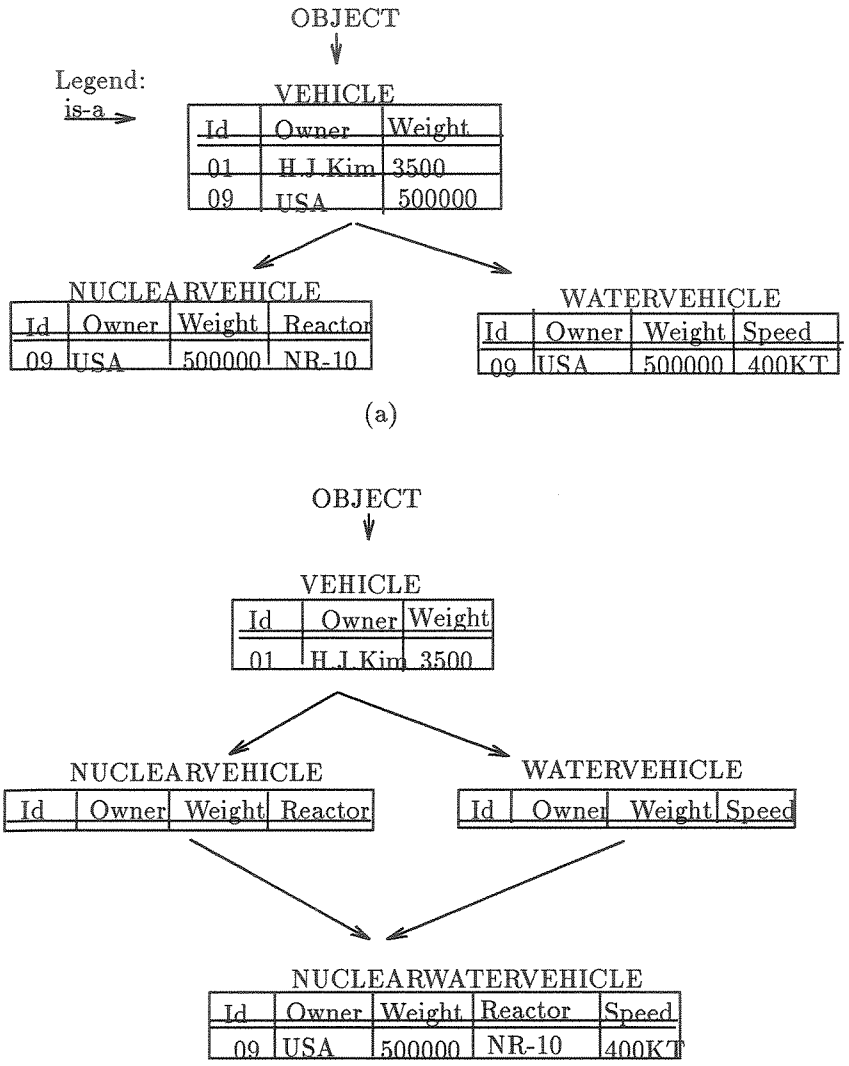


Figure 2: Modeling instances in VEHICLE database

2.3 Operations on Instances

The user's view of a single class is similar to a relational view. SQL-like query languages [IBM81] can be adopted for object-oriented databases. However, the inheritance mechanism and the assumption "an instance can belong to one and only one class" (i.e., the approach in Figure 1.b) force us to provide two kinds of SELECT, DELETE and UPDATE operations. In this section we introduce a set of operations which are necessary under our assumption.

- SELECT-ONLY (instance variables) FROM (classes) WHERE (query predicates): This type of query, when posed to a class C, causes a selection of all instances of C that satisfy the query qualification. Instances of subclasses of C, if any, are ignored. For example, the meaning of the query "SELECT-ONLY * FROM GRAD-STUDENT" is "retrieve all graduate students who are not a TA".
- SELECT-ALL (instance variables) FROM (classes) WHERE (query predicates): In many cases, it may be desirable to retrieve instances of all subclasses of a class as well as its own instances. SELECT-ALL type queries are the same as SELECT-ONLY queries, except all instances of C and C's subclasses are evaluated. For example, the meaning of the query "SELECT-ALL * FROM GRAD-STUDENT" is

"retrieve all graduate students".

- The properties of the SELECT-ONLY operation extend to the DELETE-ONLY and UPDATE-ONLY operations, and the properties of the SELECT-ALL operation extend to the DELETE-ALL and UPDATE-ALL operations.
- INSERT (an instance) TO (a class): Inserting an instance into a particular class is straightforward as long as the type of the instance corresponds to the class.
- MOVE (an instance) FROM (a class) TO (another class): During the lifetime of a database, the role of an instance may evolve. For example, H.J.Kim in Figure 1.a has four roles: T.A., GRADUATE-STUDENT, STAFF, and UNIVERSITY-PERSON. Suppose H.J.Kim gets his PhD degree and takes a teaching job at the same university. Now H.J.Kim has a different set of roles: PROFESSOR, STAFF, and UNIVERSITY-PERSON. The instance H.J.Kim, which belonged to T.A., should be moved from T.A. to PROFESSOR, using the MOVE operation. The MOVE operation is an atomic operation subsuming both the INSERT and DELETE operations.

2.4 Composite Object

The notion of composite objects explicitly captures the IS-PART-OF relationship. A *composite object* is a hierarchical structure of related instances that captures the IS-PART-OF relationships between an object and its parents. (In the literature, what we call composite objects have variously been called complex objects [LP83, KIM85], molecular aggregations [BK85], composite objects [Bobr85], and aggregation hierarchies [Atwo85].) As such, most object-oriented data models support the notion of composite objects.

A composite object has a single root object that references multiple child objects, each through an instance variable. Each child object can in turn reference its own child objects, again through instance variables. A parent object *exclusively owns* its child objects; thus the existence of child objects is predicated on the existence of their parent. Child objects of an object are thus *dependent objects.* The instances that constitute a composite object belong to classes that are organized in a DAG. This 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 the remainder of this paper, we shall use a vehicle class hierarchy and a vehicle composite object as an example for illustrating our ideas. The class hierarchy for vehicles is shown in Figure 3. In Figure 4, we illustrate a composite object schema for vehicles. The classes that are connected by dotted lines form the composite object schema. The root class is the class VEHICLE. Through the instance variables Body and Drivetrain, vehicle instances are linked to their dependent objects, which belong to the classes BODY and DRIVETRAIN. The instances of BODY and DRIVETRAIN, in turn, are connected to other dependent objects.

3. An Overview of Schema Evolution

The ORION schema evolution framework [BKKK86, BKKK87] consists of a set of properties of the consistent 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 are used for defining the semantics of every meaningful schema change. 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. In our previous papers [BKKK86, BKKK87], we showed that how schema transformation rules are applied to maintain the invariants for all types of schema changes in ORION. A more formal treatment of schema evolution is in [KKBK86]. As such, we omit the details of invariants and

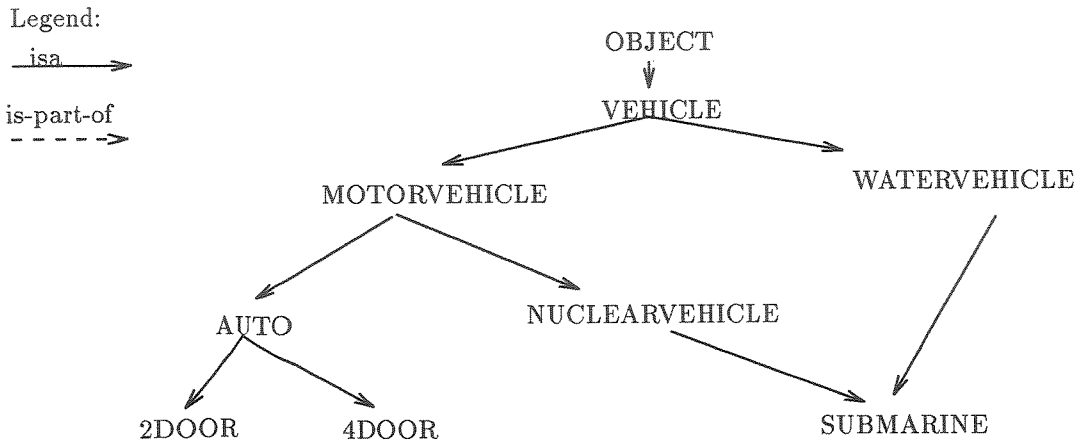


Figure 3: VEHICLE class hierarchy

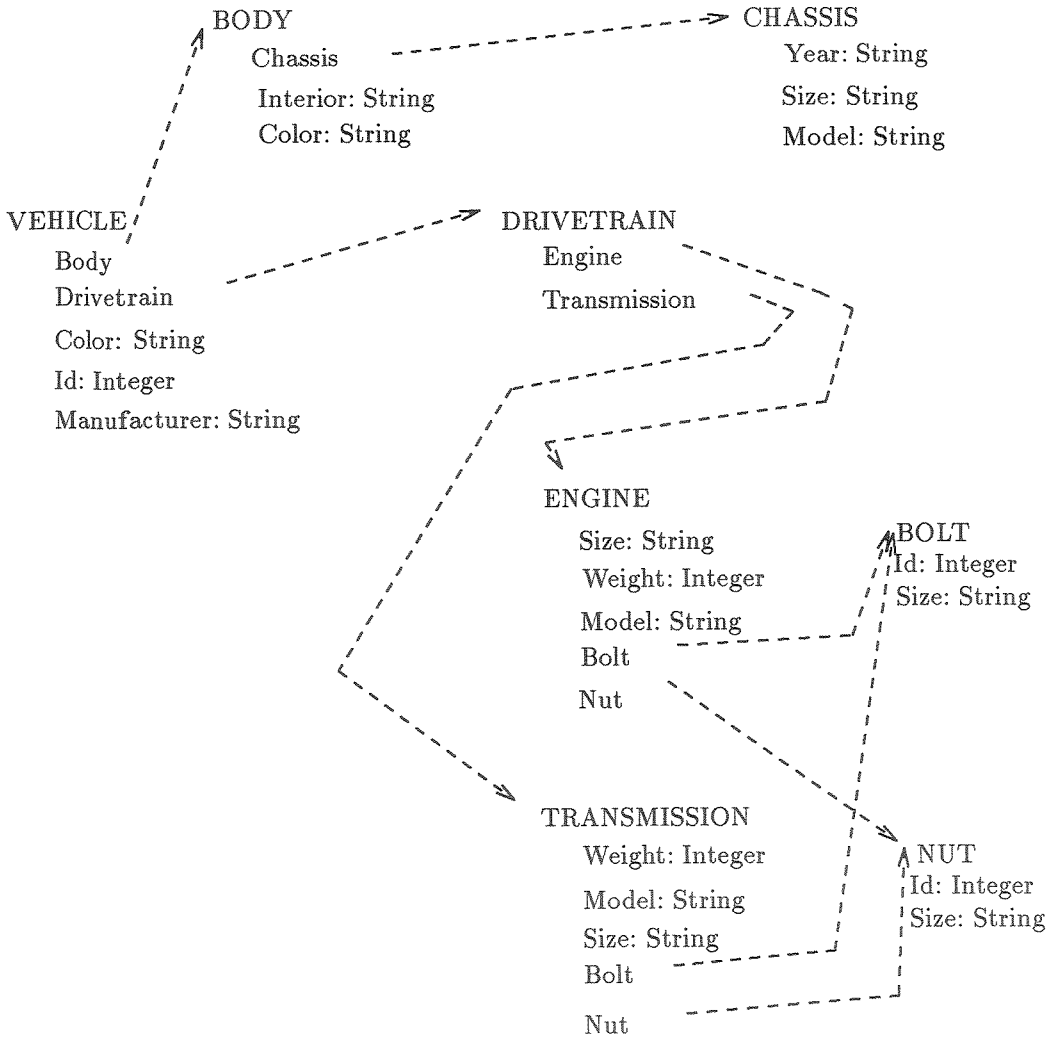


Figure 4: VEHICLE composite object

the schema transformation rules here.

We list all schema changes that are supported in the ORION schema evolution framework below.

```
(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 a shared value
            (1.1.7.3) Drop a 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 the 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
```

Semantics of each operation of the taxonomy are described in [BKKK86, BKKK87]. Although our framework has been developed for the ORION data model, we believe that is applicable to most object-oriented systems.

4. Schema Versions in Object-Oriented Databases

In this section we extend the schema evolution framework by allowing schema versions. We first discuss the semantics of schema versions and then present our approach.

4.1 Schema Version Semantics

In conventional object version models [KL84,DL85], only one schema is shared by multiple object versions of an object. This, however, is rather a strong restriction on object version derivation because an object version can be derived only by changing some values of instance variables of a parent object version, not by

deleting or adding instance variables, nor by changing their domains. The following example illustrates the restriction of object version derivation in conventional object version models.

Example 4.1: Suppose a schema AUTO is **(Id,Engine-size,Color)** and domains of instance variables are **Id: String, Engine-size: 100..500, Color: {red,blue,white}**. Let the AUTO schema be S. Let an object D be (TKW624, 320, blue). Consider the following versions: D_1 (TKW624, 250, blue), D_2 (TKW624, 320, blue, ford), D_3 (TKW624, blue), D_4 (TKW624, 550, blue). In conventional object version models [KL84, KCB86, CK86], D_2, D_3 and D_4 cannot be versions of D because the AUTO schema cannot be shared by them: D_2 has a value (ford) of a new instance variable Manufacturer, D_3 does not have a value for the instance variable Engine-size, and D_4 has an out-of-range value for the instance variable Engine-size. However, suppose we allow object versions D_2, D_3 and D_4 to have their own schema versions of the original AUTO schema S: S_2 **(Id: String, Engine-size: 100..500, Color: {red,blue,white}, Manufacturer: String)**, S_3 **(Id: String, Color: {red,blue,white})**, S_4 **(Id: String, Engine-size: 100..600, Color: {red,blue,white})**. Then, D_2, D_3 and D_4 can be considered as versions of D. □

The next example also illustrates the necessity of a schema versioning facility in a non-traditional database system.

Example 4.2: The system architecture for CAD systems often consists of a public database system connected to a collection of private database systems [CK86]. Users can check out an object version from the public database system and manipulate it in their design workstation (private database). They can create a new object version from the checked-out object version and check the new object version into the public database.

Now suppose that there are several object versions v_1, v_2, ..., v_n of an object schema V in the public database. After user A has checked out v_2, another user B or a database administrator may change the structure of V (i.e., schema change). If the system does not keep track of the previous version of schema V, A cannot check a new object version, say v_9, derived from v_2, into the public database because the new object version v_9 can not be accommodated in the new schema V. □

The restrictions shown in the above two examples can be relaxed by maintaining schema versions. In summary, there are two major benefit of keeping track of schema versions: (1) They provide for more *flexible derivation of object versions*; that is, object version derivation can cross multiple schema versions and (2) They allow the *independence between object evolution and schema evolution* to be maintained.

There are two difficulties (called *storage redundancy* and *update anomaly*) in supporting schema versions, which must be resolved for schema versions to be practical. There are two (naive) approaches to managing schema versions: the snapshot approach and the view approach. In the snapshot approach, whenever a new version of a schema (say B) is derived from a schema (say A), all instances of A are copied, modified in order to comply with B, and stored physically under B. The view approach is that in the same situation above, the instances of A are not copied under B. Instead, whenever necessary, instances of A are viewed under B. Consider the following example to illustrate the difficulties.

Example 4.3: Let us consider a class AUTO-MANUFACTURER with 4 instance variables: **Id (Integer)**, **Name (String)**, **Location (String)**, and **Wagon? (Boolean)**. **Id** is a unique identifier for a company. **Name** is the name of a company. **Location** is a location of each company and **Wagon?** indicates whether the auto-manufacturer produces a wagon or not. Suppose a schema version AUTO-MANUFACTURER.1 is created by dropping the instance variable **Wagon?** from the original schema AUTO-MANUFACTURER. Also suppose another schema version AUTO-MANUFACTURER.2 is created by selecting the instances of

AUTO-MANUFACTURER producing wagons. In this schema version, the value of **Wagon?** in all instances is "Yes".

Figures 5.a and 5.b show the snapshot and view approaches. In the snapshot approach, as Figure 5.a illustrates, many instances are stored in duplicate under the three schemas. If we want to delete auto-manufacturer #02 from the original schema AUTO-MANUFACTURER, we have to delete the same instance from AUTO-MANUFACTURER.1 and AUTO-MANUFACTURER.2. In the view approach, update requests in the views must be translated into updates in the database. If there is more than one translation of a view update request, the view update request is *ambiguous*. Suppose a user requests the deletion of auto-manufacturer #06 from the AUTO-MANUFACTURER.2 view in the Figure 5.b. There are two possible translations of the view update request: one is to delete #06 from AUTO-MANUFACTURER, while another is to replace the **Wagon?** instance variable of #06 with a "No". However, either of two translations may cause an unintended update anomaly.

The view approach has another drawback: suppose another schema version AUTO-MANUFACTURER.3 is created by adding a new instance variable **President (String)**. In the view approach, it is quite clumsy to model AUTO-MANUFACTURER.3: one way is to create an auxiliary class AUX having **Id** and **President** and then form AUTO-MANUFACTURER.3 from a join view of AUX and AUTO-MANUFACTURER. However, the view update ambiguity in a join view is even greater [Kel85].

Our approach is to keep all schema versions in a single class hierarchy. As shown in Figure 5.c, AUTO-MANUFACTURER.1 can be modeled as a superclass of AUTO-MANUFACTURER because the class definition AUTO-MANUFACTURER.1 is more general than that of AUTO-MANUFACTURER. By similar reasoning, AUTO-MANUFACTURER.2 can be modeled as a subclass of AUTO-MANUFACTURER. As we will show, problems in the snapshot and view approaches are nicely resolved in our approach. □

For simplicity, the above example shows the case of one class, i.e., as simple as a single relation. If we adopt either the snapshot or view approach directly in real object-oriented database schemas, the storage redundancy and update anomaly problems are even more serious because the class hierarchy of a real-world object-oriented database schema could be a DAG structure with hundreds of classes.

In summary, we reject the snapshot approach for two reasons. First, storage waste will be enormous if we store instances once for each schema version, and second, database updates will be expensive because every snapshot will be checked. Also, we do not accept the view approach because view updates may cause semantic problems (update anomaly). Hence, we have been seeking a technique that enables users to explicitly deal with schema versions, while guaranteeing *minimum storage redundancy* and allowing us to get around the problem of *update anomaly*.

As shown in section 2, an object instance cannot belong directly to more than one class. However, an object instance I of a class A is also an instance of all superclasses of A. As such, logically I is an instance of A and A's superclasses, but physically I belongs only to A. Storage is saved because the instance I is only stored once under the class A. Therefore, minimum storage redundancy is guaranteed with a class hierarchy.

We make one important assumption in the update semantics of object-oriented data models. In the relational model, if V is a view of a relational schema R, a request for deleting a tuple from the view V causes some update ambiguities because the view tuple deletion may or may not be translated into the database tuple deletion. In object-oriented data models, such update ambiguities are ignored. Suppose a class A is a superclass of B and B is a superclass of C. Let c_1 be an instance of C. Note that c_1 is also an instance of classes A and B, but I is stored only under C. If we follow the convention that is adopted in the

AUTO-MANUFACTURER

Id	Name	Location	Wagon?
01	HyunDai	KOREA	YES
02	Honda	JAPAN	YES
03	Ford	USA	NO
04	Mazda	JAPAN	NO
05	DaeWoo	KOREA	NO
06	Acura	JAPAN	YES

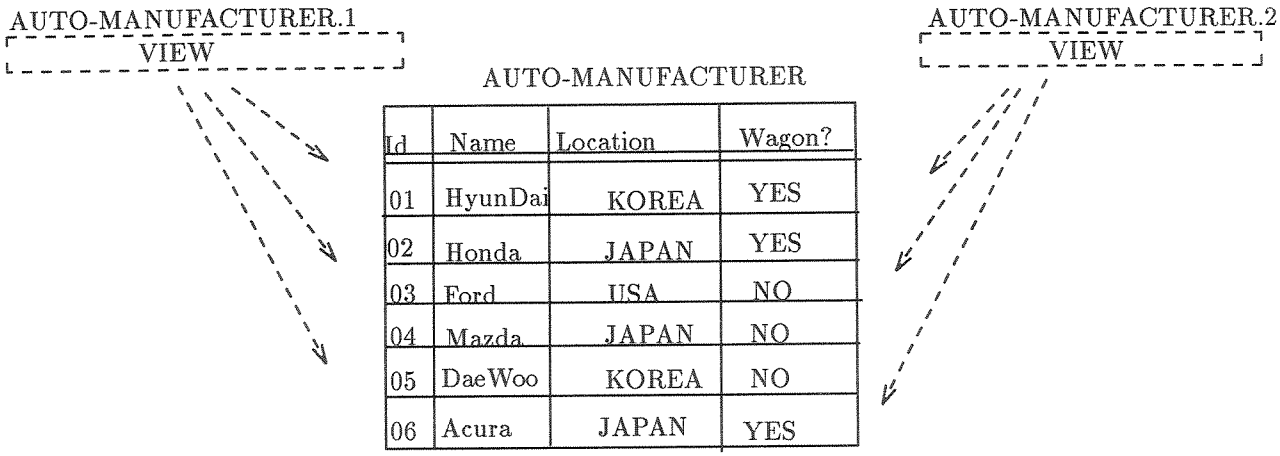
AUTO-MANUFACTURER.1

Id	Name	Location
01	HyunDai	KOREA
02	Honda	JAPAN
03	Ford	USA
04	Mazda	JAPAN
05	DaeWoo	KOREA
06	Acura	JAPAN

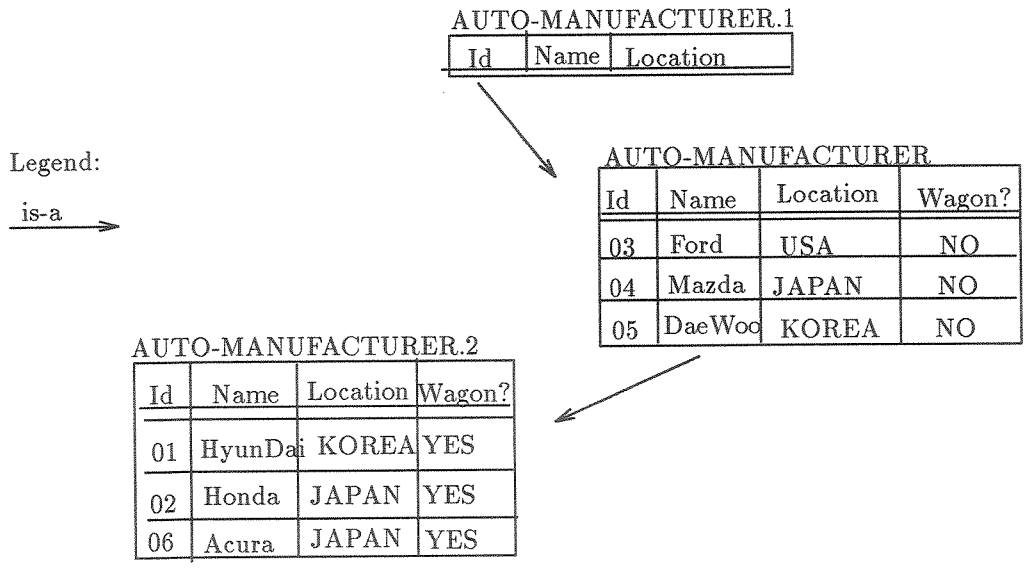
AUTO-MANUFACTURER.2

Id	Name	Location	Wagon?
01	HyunDai	KOREA	YES
02	Honda	JAPAN	YES
06	Acura	JAPAN	YES

(a) Snapshot Approach



(b) View Approach



(c) Our Approach

Figure 5: Approaches for Schema Versioning

relational model, a request for deleting c_1 can be interpreted three ways: (1) dropping from C, but staying in the database as an instance of A and B, (2) dropping from B and C, but staying in the database as an instance of A, (3) dropping from A, B, and C (i.e., dropping from the database). However we assume that (3) is the only deletion semantics in object-oriented data models. Therefore, the issue of update anomaly is ignored within a class hierarchy. Our approach takes advantage of this assumption and the properties of our schema evolution framework and class hierarchies.

We intend to maintain a history of classes, not a class hierarchy. Therefore, in our approach, schema versioning is achieved through the versioning of classes, i.e., *the granule of schema versioning is a class, not a class hierarchy.* A new class version is derived by changing the definition of the original class. A class definition consists of a set of superclasses, a set of instance variables, and a set of methods. The relationship between S' (a new class version) and S (the original class) is one of the following:

- S' is more generalized than S. (i.e., S' is a *superclass* of S)
- S' is more specialized than S. (i.e., S' is a *subclass* of S)
- S' is neither more generalized, nor more specialized than S, but, S' is somehow related with S. We call S' a *neighborhood class.*

Therefore we view a class version derivation as creating a new class either as a subclass, a superclass, or a neighborhood class of the original class. We shall keep class versions in a single class hierarchy.

4.2 Semantics of Operations in Schema Versioning

In this section we explore our idea by discussing operations in our schema version model. Because of space limitation, we present the key points of semantics of the operations. A more complete description of semantics is in [Kim88].

CREATING A NEW CLASS VERSION

When a new class version C' is derived from an existing class version C, we must first determine the taxonomic location of C' in the class hierarchy. It is possible for the system to find the taxonomic location automatically. However, since in the worst the system must compare the new class version to all existing classes in the class hierarchy in order to determine the taxonomic location, the algorithm is rather expensive. Furthermore, the algorithm involves potentially costly computation to test type subsumption. Thus, we may wish to require the user to provide the taxonomic location of C'.

If SUBSUME(C,C') is true, then C is more general than C', i.e., C is a superclass of C'. The algorithm for determining taxonomic location follows.

TAXONOMIC-LOCATION-DECIDE(C')

/* C' is the new class version */
begin
superclass-set ← {OBJECT};
foreach class C in the class hierarchy *do* mark C;
while there is a marked class C in *superclass-set* *do* *begin*
    unmark C;
    new ← ∅;
    foreach immediate marked subclass C_s of S *do* *begin*
        if SUBSUME(C_s, C') *then* *new* ← *new* ∪ {C_s}
    end;
    if *new* ≠ ∅ *then* *superclass-set* ← (*superclass-set* − {C}) ∪ *new*

end

subclass-set ← ∅;

foreach class C in *superclass-set*

do *subclass-set* ← *subclass-set* ∪ { immediate subclasses of C };

foreach class C in *subclass-set*

do *if* not SUBSUME(C',C) *then* *subclass-set* ← *subclass-set* − {C};

do create a new class C' having the classes in *superclass-set* as immediate superclasses and the classes in *subclass-set* as immediate subclasses;

end

The computational complexity of deciding subsumption between classes depends on the expressive power of the class description language. For example, if the class description language has the expressive power of the first order logic, the type subsumption problem is undecidable [LB86]. But our class description language is simple in that a class definition consists of a set of superclasses, a set of instance variables, and a set of methods. The set of superclasses in a class definition is compiled into a set of inherited instance variables and a set of inherited methods. Therefore, eventually, a class definition is composed of a set of instance variables and a set of methods. Here is the algorithm for SUBSUME:

SUBSUME(C,C')

/* C,C': class descriptions*/

begin

foreach instance variable I of C:

do *if* C' does not have an instance variable subsumed by I *then* return(false);

foreach method M of C:

do *if* C' does not have a method subsumed by M *then* return(false);

return(true); /* C is a superclass of C' */

end

Now we introduce some criteria for deciding subsumption on instance variables and methods.

- Subsumption can be determined between instance variables by simply comparing domains of instance variables: (AGE: 10..100) subsumes (AGE: 10..50) because 10..100 subsumes 10..50, (MANUFACTURER: AUTO-COMPANY) is subsumed by (MANUFACTURER: VEHICLE-COMPANY) because VEHICLE-COMPANY is a superclass of AUTO-COMPANY.
- Suppose a method has a functional specification such as f:(domain of input-parameter-1) × (domain of input-parameter-2) → (domain of output-parameter-1) × (domain of output-parameter-2). Subsumption can be determined between methods by simply comparing domains of parameters in the following way: given methods M: I → O and M': I' → O', if I' is subsumed by I and O is subsumed by O' then M is subsumed by M' [Car83].

After finding the taxonomic location of a new class version, the new class version is created in the class hierarchy. Then, some object instances of the superclasses of the new class version must be repositioned to the new class version. Consider the two class versions AUTO.1 and AUTO.2 in Figure 6.a. When AUTO.2 is derived and created as a subclass of AUTO.1, instances satisfying the AUTO.2 description must be moved from AUTO.1 to AUTO.2 (that is, automobiles whose weight is between 1000 and 4000). That is because of the assumption that an instance can belong to one and only one class.

Next we look in detail at the example in Figure 6. There are 5 class versions of AUTO class as shown in Figure 6.a. Since AUTO.3 subsumes AUTO.1, AUTO.3 is placed in the class hierarchy as a superclass of

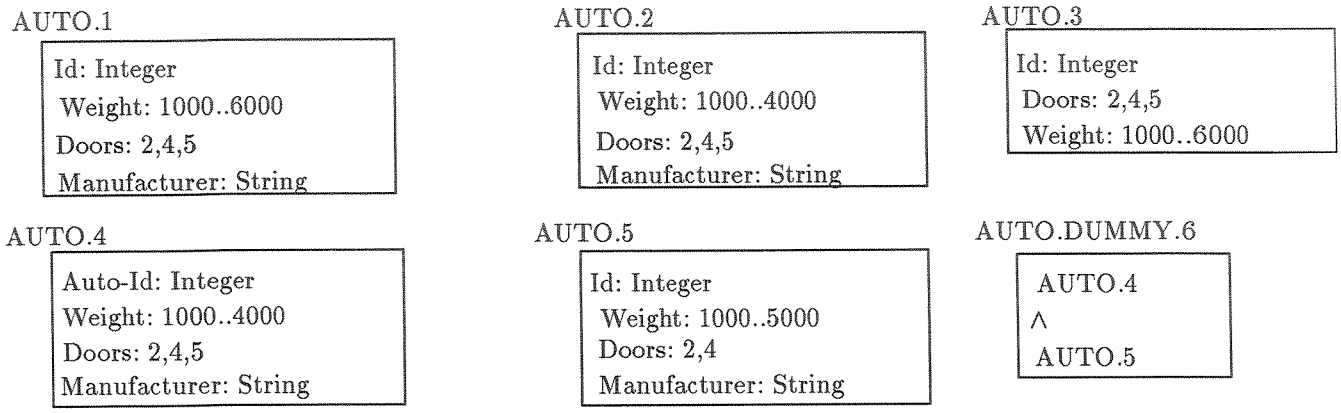


Figure 6.a: Schema Versions of AUTO class

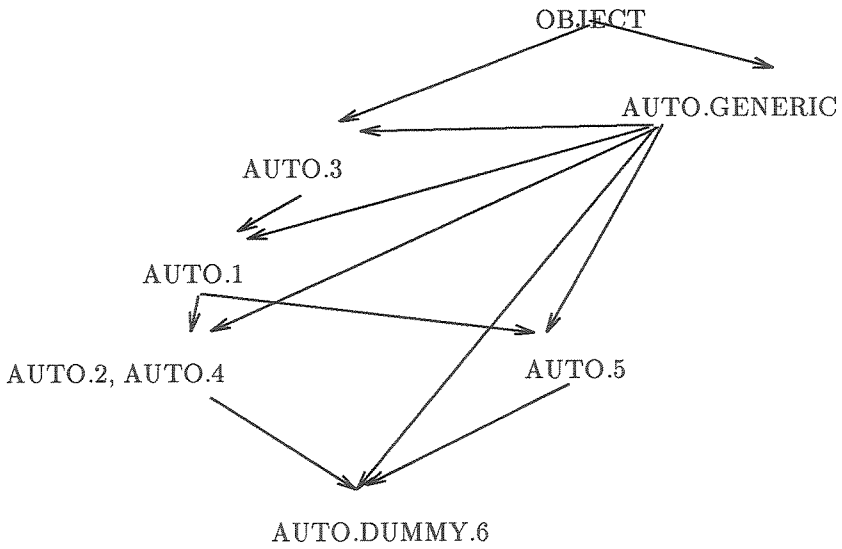


Figure 6.b: Generic, Dummy, and Equivalent classes

AUTO.1, whereas AUTO.2, AUTO.4 and AUTO.5 are placed as subclasses of AUTO.1. We introduce three types of classes: *generic class*, *dummy class*, and *equivalent class*.

- GENERIC CLASS: Retrieval of all object versions of all class versions of a class is a difficult task if there are many class versions created and some of them are already deleted. A generic class is an immediate superclass of all class versions of a particular class and is an immediate subclass of class OBJECT A generic class does not have any instance variables or methods. A generic class is be used for dropping all class versions of a class at once. Another important application of a generic class is that it allows all class versions to be the domain of an instance variable. In the example of Figure 6b, the instance variable Vehicle can have all class versions of AUTO as a domain by specifying AUTO.GENERIC as its domain. When a class is created, its associated generic class is created automatically. If all class versions of a class are deleted, the generic class of the class is deleted automatically.
- DUMMY CLASS: If a new class version is a neighborhood class of the original class version, the new class version cannot be embedded into class hierarchy as a superclass or subclass of the original class version. This makes it impossible to store common instances of the two versions (due to our requirement of an instance belonging to exactly one class). Therefore, we create a dummy class which

can accommodate common instances of the new class version and existing class versions. As shown in Figure 6.b, AUTO.DUMMY.6 must be created in order to accommodate common instances of AUTO.5 and AUTO.4. The class definition of AUTO.DUMMY.6 is just AUTO.5 $\wedge$ AUTO.4.

- EQUIVALENT CLASS: Suppose a new class version S of class C is derived by renaming one of instance variables in C. It does not make sense to store S as a neighborhood class of C and create a dummy class having instances of S and C, because S and C are essentially same schema versions. This gives rise to the notion of *equivalent classes*. S and C can be stored in the same node of a class hierarchy because membership conditions of S and C are the same. Schema change operations (1.1.3) and (1.2) may create semantically equivalent class versions having the same membership condition. As shown in Figure 6.b, AUTO.2 and AUTO.4 are in the same node of the class hierarchy because the definitions of AUTO.2 and AUTO.4 are the same except for the different names of Id and Auto-Id.

DELETING A CLASS VERSION

The semantics of deleting a class version are similar to that of deleting a normal class: all instances and object versions of the class version are deleted and all subclasses of the class version lose inherited instance variables and methods from the class version.

In addition, when a class version is deleted, some corresponding dummy classes may have to be dropped. In Figure 6.b, if AUTO.5 is deleted, AUTO.DUMMY.6 is no longer necessary, and thus it is deleted automatically. If the user deletes a generic class, all class versions of the generic class are deleted as well as instances of them.

UPDATING A CLASS VERSION

The operations in the taxonomy of schema changes (section 3) can be used for changing the contents of a class version.

CREATING A NEW INSTANCE VERSION

An object version can be created under any existing class version of a particular class. In conventional object versioning, object version derivation is allowed only under the same class. In schema versioning, object version derivation can cross two different versions of a class. For example, suppose an object version V1 of an object is created under class version C1. Another object version V2 can be derived from the object version V1 under class version C2. As shown in example 4.1, object version derivation with schema versioning is more flexible than that in conventional object versioning.

However, this flexibility leads to a difficult problem: Since the class version used for an object version Vi may be different from the class version that is used for an object version Vj which is derived from Vi, the class version for Vj should be identified by either the user or the system. There are three possible cases: (1) only one existing class version can accommodate Vj, (2) more than one existing class version can accommodate Vj, and (3) no existing class version can accommodate Vj.

In case (1), since only one class version is identified, there is no major problem. The new instance version Vj is created as an instance of that class version. In case (2), since an instance cannot belong to more than one class, a new class version, having class versions which can accommodate Vj as superclasses, needs to be provided by the user [2]. Then Vj is created as an instance of the new class version. In case (3), since there is no appropriate class version for Vj in the class hierarchy, a new class version which can accommodate Vj must be provided by the user and the new schema version needs to be placed in the appropriate place of the class hierarchy. Then Vj is created as an instance of the new class version.

[2] The system can do this. But as we mentioned earlier, it is a potentially expensive task

DELETING AN INSTANCE VERSION

No changes to the class DAG occur as a result of deletion of an instance version. If the last instance of a class version is deleted the class version is not deleted unless the user does so explicitly.

UPDATING AN INSTANCE VERSION

Updating an object version V may require it to belong to a different class version. If the updates violate the membership conditions of the class to which the object version belongs, the object version should be relocated into an appropriate class version in the class hierarchy. The class version for the updated object version V should be identified. The three possible cases are similar to those of CREATING A NEW INSTANCE VERSION, and are handled similarly: (1) only one existing class version can accommodate the updated object version V, (2) more than one existing class version can accommodate the updated object version V, and (3) no existing class version can accommodate the updated object version V.

4.3 Integration with the Object Version Model of H.T. Chou and W. Kim

H.T. Chou and W. Kim [CK86] suggested an object version model for distributed CAD databases. Their proposal includes the broad spectrum of semantics and operational issues in object version control and takes into account the characteristics of CAD environments, such as the system architecture and the way in which users and applications share data and interact among themselves. To make our schema version model complete, we integrate our schema version model and their object version model. The semantics of operations in the previous section are slightly modified because of the integration.

Chou and Kim claim that a CAD environment will consist of intelligent design workstations and central server machines on local-area networks. The central server (mainframe computer) will manage the *public database* of stable design objects and design control data. Each design workstation will have a private database. Users or application programs in a workstation check object versions out of the public database, manipulate them in the workspace of the workstation, and check new object versions into the public database.

TRANSIENT SCHEMA VERSION vs. WORKING SCHEMA VERSION

Chou and Kim identify two types of object versions: transient object versions and working object versions. Working object versions are considered stable and are actively shared by multiple users whereas transient object versions are only manipulated by the users who create them. Transient object versions can be promoted to working object versions if they are considered useful and stable. Only working object versions can reside in the public database while both working and transient object versions can reside in a private database.

A transient object version can be updated or deleted by the user who created it. A new transient object version may be derived from an existing transient object version; when this occurs, the existing transient object version is promoted to a working object version. On the other hand there are update restrictions on working object versions. Since a transient object version can be derived from a working object version, updates to working object version are unnecessary[3]. However deletion of working object versions is allowed.

We extend the above scenario in our schema version model. We want to treat class versions and object versions in the same manner. As in object versions, a transient class version can be updated or deleted by

[3] Some object version models [BK85] allow updates to a working object version. If updates to a working object version are allowed, a set of update propagation algorithms are needed for deriving new object versions (transient or working), from the working object version. This is necessary to enforce consistency between the new object versions and the updated working object version [CK86].

the user who created it. A new transient class version can be derived from an existing transient class version and the existing transient class version is promoted to a working class version. Working class versions cannot be updated. The semantics of deleting working class versions are different from the semantics of deleting working object versions in that, in the class hierarchy, deleting a class version may cause subclasses of the class version to lose inherited instance variables and methods from the class version. Therefore **if any one of the subclasses of the class version is a working class version, deletion of the class version must not be allowed.**

Class versions are also different from object versions in that **if a transient class version has one object version which is promoted to a working object version, the transient class version must also be promoted to a working class version in order to prevent schema changes to the class version which may affect the working object version.**

Only working class versions and working object versions can reside in the public database. Therefore **no updates to working class versions or working object versions are allowed. Deletion of a working class version is allowed if the schema version does not have any subclasses in the class hierarchy.**

In the private database, working class versions and transient class versions are resident in the class hierarchy as are working and transient object versions. **Update or deletion of a class version is allowed if the class version does not have any working subclasses in the class hierarchy.**

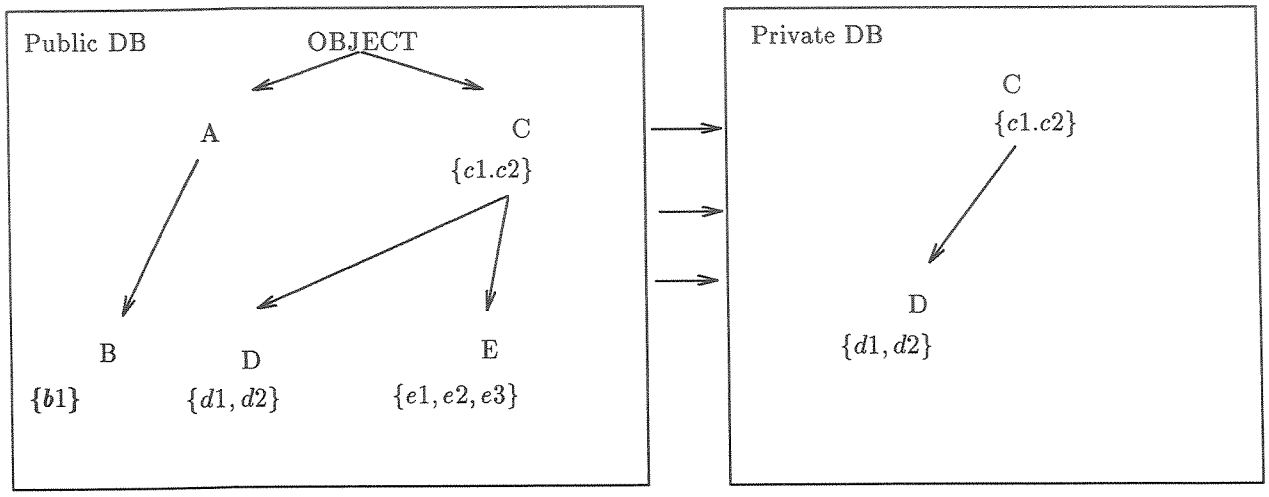
CHECK-IN/CHECK-OUT

When the user checks a working object version V out from the public database into his private database, a copy of V is installed into the private database. The status of the copy is "transient". In this situation, **the schema version S for V must have been checked out previously** [CK86]. If S does not already reside in the private database, the user or the system must check S out from the public database to the private database. The user can check out more than one class version at once (a subset of the class hierarchy). After manipulating V and its schema S in the private database, suppose V' and S' are derived respectively. If the user wants to check V' into the public database, first he must check S' into the public database. After checking S' into the public database, the location of S' in the class hierarchy of the private database might be different from the location of S' in the class hierarchy of the public database, because the class hierarchy of the private database is different from that of the public database.

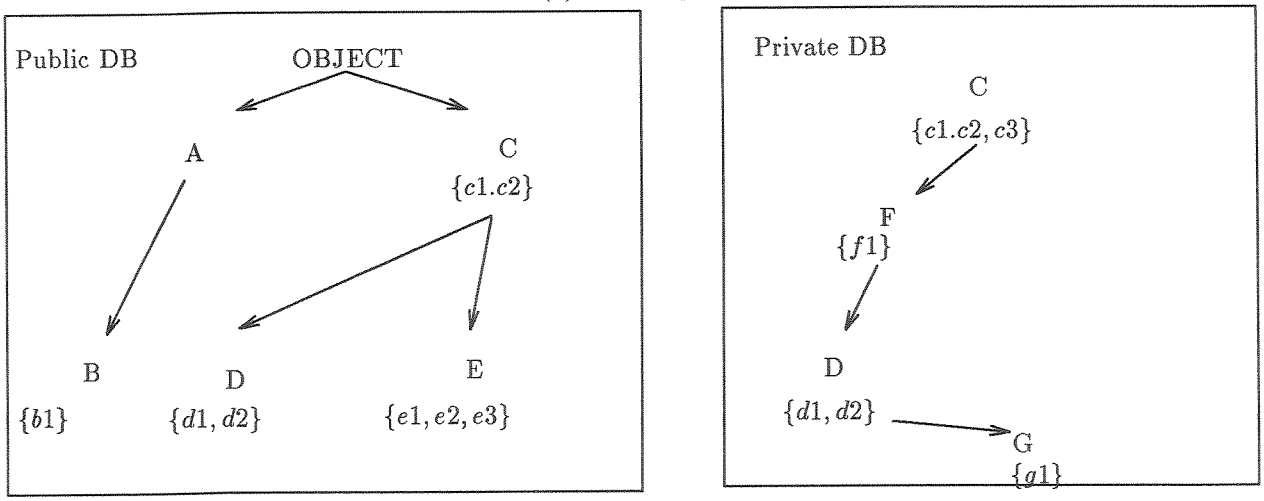
Consider the example in Figure 7. As shown in Figure 7.a, two class versions C, D and four instances c1, c2, d1, and d2 are checked out from the public database to the private database. Suppose two new class versions F, G and three new instances f1, g1, c3 are created in the private database (Figure 7.b). After these instances are checked into the public database, the locations of G and c3 may be different as shown in Figure 7.c if E subsumes G.

VERSION NAMING AND BINDING

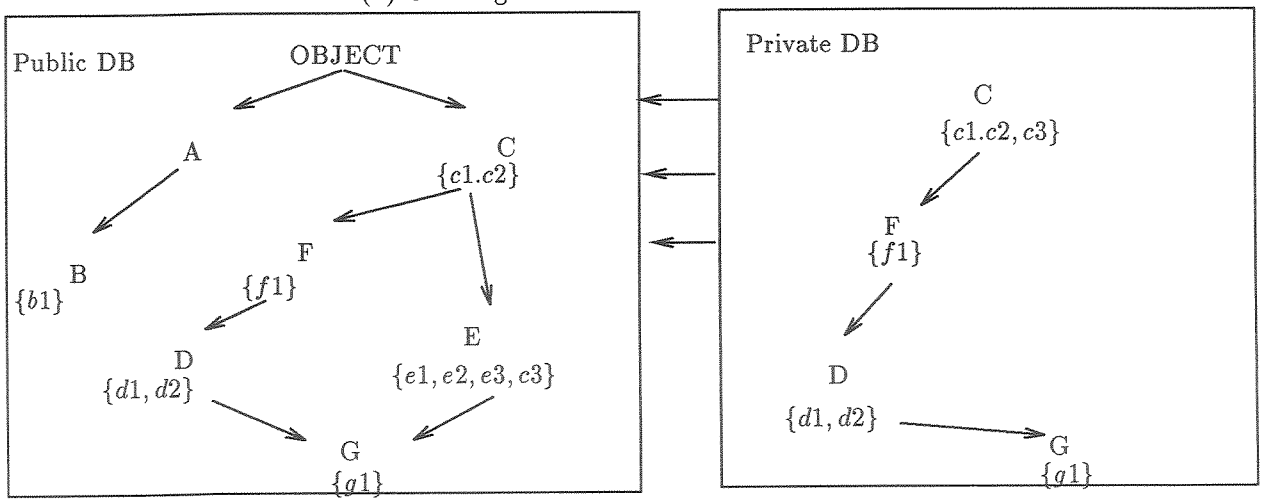
In the object version model [CK86] the full name for each object version is a triple <object name, database name, version number> where 'object name' is the identifier of an object, and 'database name' is the name of a private database, or the public database. However the naming convention is not sufficient for our schema version model because versions of a particular object may cross multiple schema versions. For example a version of an AUTO object may not have the AUTO class as its schema. Now the full name for each object version is a four tuple <object name, class version, database name, version number>. For example <1181, TRUCK.2, KIM's DB, 6> means that object version 6 of the object 1181 is in KIM's private



(a) Checking Out



(b) Creating New Class Versions and Instances



(c) Checking In

Figure 7: Check-In/Check-Out Processes

database under the TRUCK.2 class version.

[CK86] also consider *static binding* and *dynamic binding*. In static binding, the full name of an object version is specified. In dynamic binding, some portions of the full name are left unspecified. However an 'object name' should be provided explicitly in dynamic binding. For the unspecified parts, the system selects default values. As for the criteria used in selecting default database and default version, refer to [CK86].

CHANGE NOTIFICATION

The scenario of change notification in H.T. Chou and W. Kim's object version model follows: A transient or working object version in a private database may reference other transient or working object versions in the same private database, or working object versions in the public database. A working object version in the public database may reference other working object versions in the public database. Change notification is required when a referenced transient object version is updated, deleted, or a new transient object version of it is created. Change notification is also required when a referenced working object version is deleted or a new transient version of it is derived. Two types of notification techniques are provided: *message-based notification* and *flag-based notification*. The details of these techniques are described in [CK86]. Below we tailor their change notification framework to our schema version model.

We say that class C refers to class C' if

- the domain of an instance variable of C is C'
- a method of C refers to C' or to an instance of C'

Change notification is required when a referenced transient class version is updated, deleted, or a new transient class version of it is created. Similarly, change notification is required when a referenced working class version is deleted or a new transient version of it is derived. The creators of the class versions that refer to a class version C are notified of changes to C.

Additionally we have to consider the impact of each schema change on existing methods. Change notification is required when a referenced transient class version is updated, deleted, or a new transient class version of it is created. Similarly, change notification is required when a referenced working class version is deleted or a new transient version of it is derived. The creators of the methods that refer to a class version C are notified of changes to C.

5. DAG Rearrangement Views

Schema versioning is a means of maintaining a history of schema changes by keeping versions of schema. An alternative method is to allow views of schema. Views in object-oriented databases are more versatile than those in relational databases. Conventional views in relational databases are constructed via combinations of relational operators such as select, project, and join. Views in object-oriented databases include rearrangement of DAG structures (both composite objects and class hierarchies) as well as conventional views in relational databases.

In this section, we present operators for defining DAG rearrangement views of composite objects and class hierarchies. We identify sets of composite object views with the property that queries on the views are processable on instances of the original composite object schema. We also discuss how instances would be viewed and reorganized in DAG rearrangement views of class hierarchies.

Most applications in object-oriented databases assume a group of cooperative workers (i.e., team) are sharing the same objects. However, users may not need to see the whole database and objects in it. They need to see only those parts of composite objects and those classes that are relevant to their applications.

We envision the following applications of a DAG rearrangement view facility:

- Check-out granularity control: In order to work with an object version or create a new object version, users need to check object versions and their classes out of a public database to a private database. In the case of huge design objects with thousands of parts, a DAG rearrangement view facility would allow the user to specify parts of objects to be checked out.
- Authorization: In many situations, the database administrator wishes to control the rights of users to access parts of objects. He may wish to reserve the privilege of modifying a particular part of an object or to make some parts invisible to the user. A DAG rearrangement view facility will provides the database administrator with the means to control access rights.
- Versions of class hierarchies: In the previous section, the granularity of schema versions was a single class, not a class hierarchy. With DAG rearrangement view facility, we can keep several versions of a class hierarchy in an inexpensive way.

5.1 DAG Rearrangement Views on Composite Objects

The notion of composite objects explicitly captures the IS-PART-OF relationship. A composite object is a collection of related instances that form a hierarchical structure, and schema of a composite object is a DAG structure.

Figure 8.a shows a part hierarchy for vehicle composite objects. Figure 8.a is a simplified form of Figure 4 without description of the instance variables. Figure 8.b, 8.c, and 8.d show three possible views of the vehicle composite objects of Figure 8.a.

In relational databases, every possible view (constructed from relational algebra operators) against relational schemas has the property that queries to the view can be processed using the tuples of the underlying relational schemas. It turns out that not all possible DAG rearrangement views of composite objects are acceptable because queries to a certain DAG rearrangement view are not processable on instances disciplined by the original composite object schema.

There are often restrictions on posing queries against tree or DAG type objects (hierarchical structures). In IMS (see, e.g. [KS86]), query qualification predicates may involve only parent record types of a target record type or the target record type itself. In System 2000 [MRI78], query qualification predicates may involve only parent record types or child record types of a target record type or the target record type itself. The reasons behind those restrictions are *query processing overhead.* The same restrictions are assumed in accessing composite objects of object-oriented databases because composite objects are hierarchical structures.

In the remainder of this section, we elaborate on the relationship between an access pattern and the set of DAG rearrangement views for which the given access pattern can be used to process queries on instances disciplined by the original composite object schema. We shall use the following syntax for queries on composite objects and their DAG rearrangement views.

GET *attributes of target record type*

WHERE *predicate*

We shall use the following queries against the schema of the vehicle composite object in Figure 4 for comparing access patterns:

Q1: GET *CHASSIS.Year*

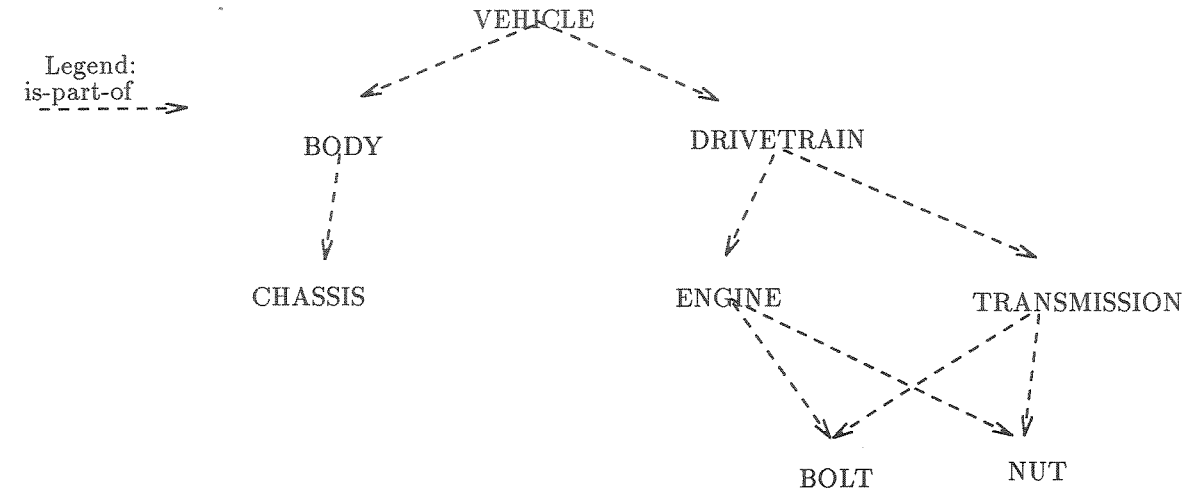
WHERE (*VEHICLE.Manufacturer* = *"HyunDae"*) and (*BODY.Chassis* = *1234*)

Q2: GET *VEHICLE.Color*

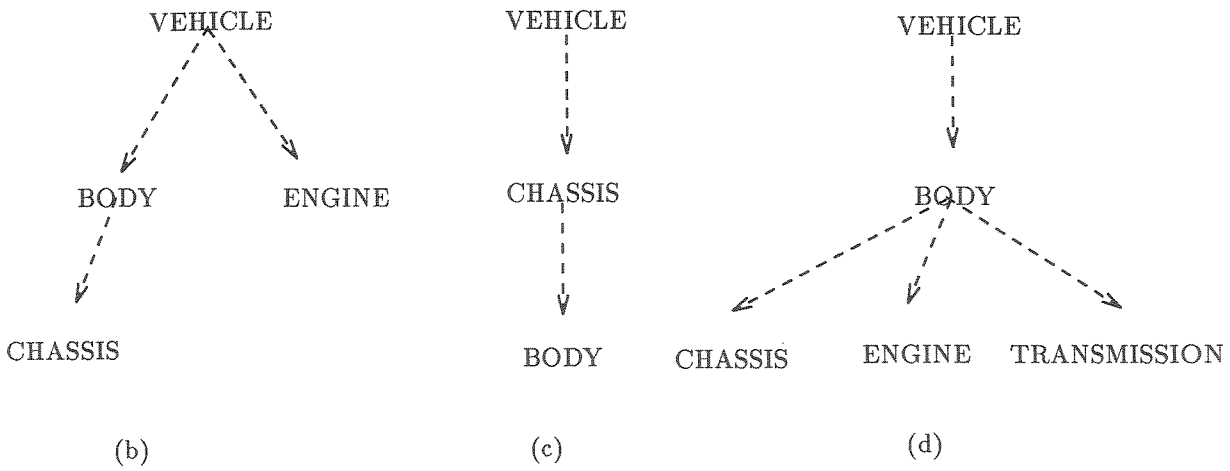
WHERE *BODY.Chassis* = *1234*

Q3: GET *BODY.Interior*

WHERE *ENGINE.Model* = *"320CI"*



(a) VEHICLE part hierarchy



(b) (c) (d)

Figure 8: DAG Rearrangement Views of VEHICLE Composite Object

We shall use terms *parentpart* and *childpart* in what follows: in Figure 4, VEHICLE class has two childparts, BODY and DRIVETRAIN classes, and in turn, VEHICLE class is a parentpart of BODY and DRIVETRAIN classes.

CHILDPART-ONLY TRAVERSAL

In childpart-only traversal, if the target attributes of a query belong to a part P of a composite object, query predicates may contain only attributes of parentparts of P. Query Q1 is allowed in the childpart-only traversal scheme because its target attributes belong to CHASSIS and the query predicates of Q1 involves only attributes of parentparts of CHASSIS. Queries Q2 and Q3 are not allowed in childpart-only traversal because BODY is not a parentpart of VEHICLE and ENGINE is not a parentpart of BODY. IMS uses the childpart-only traversal scheme as its primary access pattern.

Consider the DAG rearrangement views of the VEHICLE composite object in Figure 8. As Figure 8.b illustrates, for each part P in the view, the set of the parentparts of P of the view is a subset of the parentparts of P of the original schema. As such, childpart-only traversal can be used on the view in Figure 8.b because all

possible queries on the view are processable directly on instances disciplined by the vehicle composite object schema. However, childpart-only traversal can not be used on the view in Figure 8.c because some queries on the view are not processable directly on instances disciplined by the original vehicle composite object schema. For instance, the query "GET BODY.Interior WHERE CHASSIS.Year = 1986" is allowable in the composite object view because the query predicate involves attributes of parentparts of BODY. However, the query is not processable in the original composite object schema in Figure 8.a because CHASSIS is not a parentpart of BODY. By similar reasoning, the view in Figure 8.d is not allowed in the childpart-only traversal scheme.

DEFINITION: If every query, which is expressed in a composite object view in accordance with a traversal scheme, is processable directly on the original composite object schema, the composite object view is called a *processable composite object view* (PCOV) in the traversal scheme.

We characterize PCOVs for childpart-only traversal as follows:

- Let S and S' denote the original schema of a composite object and a DAG rearrangement view on S respectively. S' is PCOV in the childpart-only traversal scheme iff for each part P in S', the set of the parentparts of P in S' is a subset of the parentparts of P in S.
- Let S and TR(S) denote the original schema of a composite object and the transitive closure of S. Let S' be a view on S. S' is PCOV iff S' is a sub-DAG of TR(S).

We consider two useful operations for constructing PCOVs:

- (V1) Hide a part P: The part P is not visible and immediate parentparts of P become immediate parentparts of immediate childparts of P.
- (V2) Make a part P an immediate childpart of one of P's parentparts

It is easy to see that given a schema S, the resulting schema S' that is constructed from applying any combination of V1 and V2 is a subset of the transitive closure of S. As such, S' is a PCOV.

CHILDPART-PARENTPART TRAVERSAL

In childpart-parentpart traversal, if the target attributes of a query belong to a part P, query predicates may contain only attributes of parentparts or childparts of P, or P itself. Queries Q1 and Q2 are allowed in the childpart-parentpart traversal scheme because CHASSIS is a childpart of VEHICLE and BODY, and VEHICLE is a parentpart of BODY. Query Q3 is not allowed because BODY is neither a parentpart nor a childpart of ENGINE. System 2000 uses the childpart-parentpart traversal as its primary access pattern rule.

The childpart-parentpart traversal scheme can be used with the DAG rearrangement views in Figure 8.b and 8.c, but not with the DAG rearrangement view in Figure 8.d. The childpart-parentpart traversal scheme can not be used with the view of Figure 8.d because some queries on this view can not be processed in the instances of the original VEHICLE composite object schema. Note that the childpart-only traversal can not used on the DAG rearrangement view in Figure 8.c, but childpart-parentpart traversal can. We denote the union of a part P and its child and parent parts of P as CP-SET(P). In Figure 8.b, CP-SET(VEHICLE), CP-SET(BODY), CP-SET(CHASSIS) and CP-SET(ENGINE) are respectively subsets of CP-SET(VEHICLE), CP-SET(BODY), CP-SET(CHASSIS) and CP-SET(ENGINE) in the original VEHICLE schema of Figure 8.a.

We characterize PCOVs for childpart-parentpart traversal as follows:

- Let S and S' denote the original schema of a composite object and a DAG rearrangement view on S respectively. S' is PCOV in the childpart-parentpart traversal scheme iff for each part P in S', CP-SET(P in S) $\supseteq$ CP-SET(P in S')

In childpart-parentpart traversal, there is one more useful view definition operation in addition to V1 and V2.

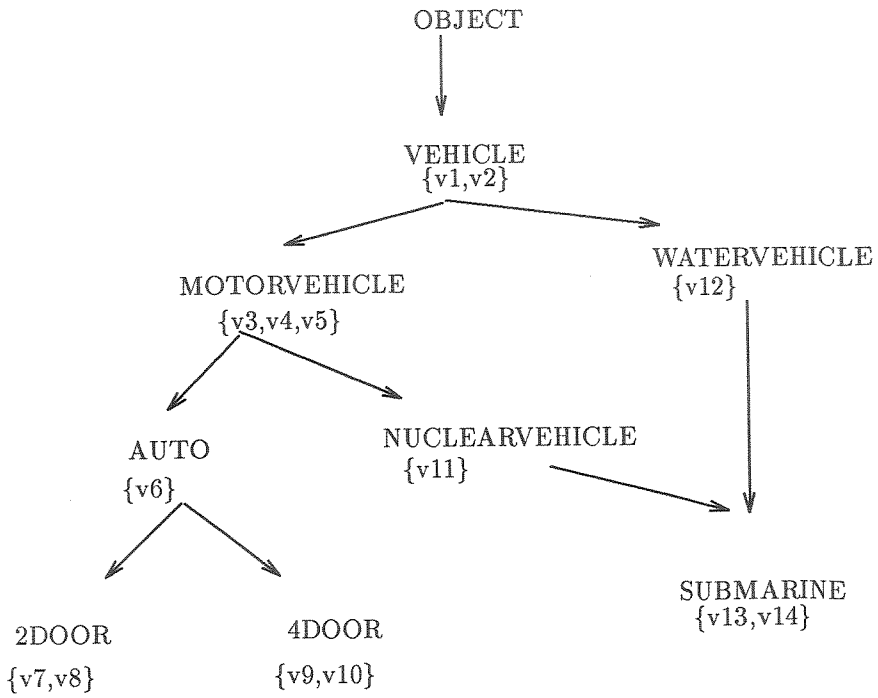


Figure 9: A Sample Database of VEHICLE Class Hierarchy

- (V3) Exchange parts: As shown in the view of Figure 8.c, exchanging parts is allowed. However the exchange of arbitrary two parts can violate the premise of the childpart-parentpart traversal. Thus, exchanging is allowed only when for each part P, CP-SET(P) in the resulting view is a subset of CP-SET(P) in the original schema.

FREE TRAVERSAL

In free traversal, if target attributes of a query are those of a part P, query predicates may contain any parts of the composite object schema. In free traversal, queries Q1, Q2, and Q3 are all allowed. Arbitrary DAG rearrangements views including the views in Figure 8.b, 8.c, and 8.d are allowed. To our knowledge, no existing hierarchical database system allows free traversal because of query processing ambiguity and overhead due to multiple paths between target attributes and query qualification attributes.

5.2 DAG Rearrangement Views on a Class Hierarchy

There are 5 operations which are useful in defining the DAG rearrangement views of a class hierarchy. For each operation, we introduce the semantics of the operation:

1. Hide a class C: The class C is not visible in the view. The instances under C and subclasses of C lose their membership in C. Instance variables or methods which are locally defined in C are not visible from the subclasses of C. However, instances of C are still visible from superclasses of this class. This operation is often followed by the operation "Make an ISA relationship explicit" defined below. The semantics of this operation are similar to those of the schema change operation "(3.2) Drop a class C" in section 3.
2. Hide an ISA relationship: Suppose S is an immediate superclass of C. By hiding the ISA relationship between S and C, instance variables and methods inherited from S are not visible in the view of C and subclasses of C. Further, instances of C and subclasses of C lose the membership in S. The semantics of this operation are more or less similar to those of the schema change operation "(2.2) Remove a class S as a superclass of the class C" in section 3.
3. Make an ISA relationship explicit: Suppose S is the only superclass of C, and C, in turn, has one subclass

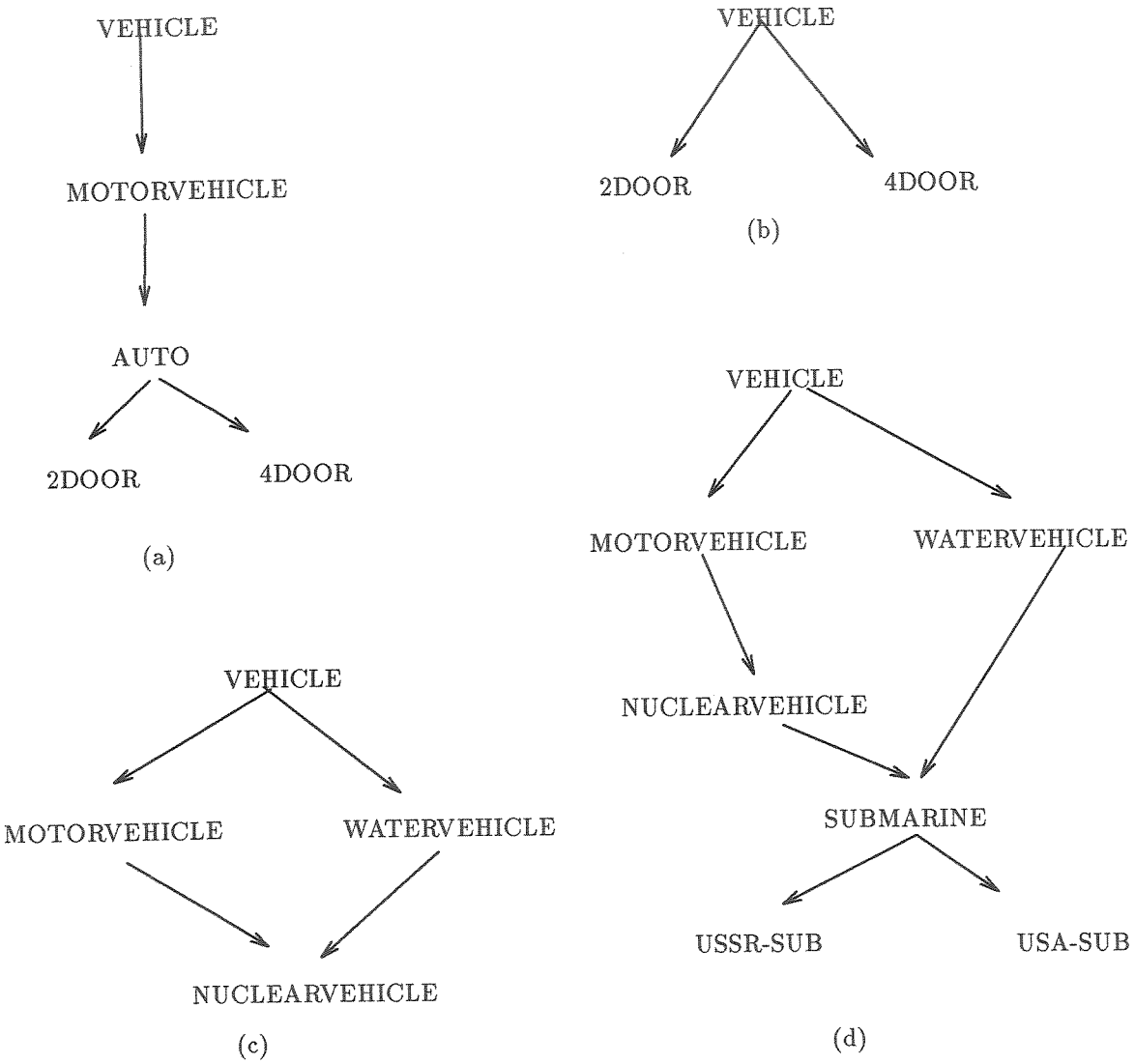


Figure 10: DAG Rearrangement Views of VEHICLE Class Hierarchy

C1. After performing "Hide the class C", S and C1 become disconnected in the view. In that case, this operation can connect S and C1 by making S an immediate superclass of C1. Since S was already a superclass of C1, there is no impact on instances of S or C.

4. Create a new ISA relationship: Suppose S1 and S2 do not have any ISA relationship between them and S1 and S2 have a subclass C in common. After performing "Hide the class C", we can make S2 an immediate superclass of S1 by using this operation. Since S2 was not a superclass of S1 in the original schema, instances of S1 are not qualified as instances of S2 in the view. However instances of C can be viewed as instances of S1 because instances of C were common instances of S1 and S2. This operation is a counterpart of the schema change operation "(2.1) Make a class S a superclass of a class C" in section 2, but has totally different semantics.
5. Create a new subclass: Suppose a new class S is created as a subclass of a class C in a view. Some instances of C may need to be moved in the view from C to S if the instance is qualified as an instance of the new class S. The semantics of this operation are similar to the schema change operation "(3.1) Define a new class C" in section 2.

The semantics of the above operations will be clearer after we go through some examples. Suppose

classes in the VEHICLE class hierarchy in Figure 3 have the instances shown in Figure 9. We show how each of the DAG rearrangement views of Figure 10 is constructed and what instances are visible from each class. In what follows I_C denotes the set of instances of class C.

- The DAG rearrangement view in Figure 10.a is constructed by the following sequence of operations:

(1) Hide SUBMARINE
(2) Hide NUCLEARVEHICLE
(3) Hide WATERVEHICLE

The following instances are visible from each class in this view:

- $I_{VEHICLE}$ = { v1, v2, v12* } where v12* is a projected form of v12 under the VEHICLE class definition.
- $I_{MOTORVEHICLE}$ = { v3, v4, v5, v11*, v13*, v14* } where v11*, v13*, and v14* are a projected form of v11, v13, and v14 respectively under the MOTORVEHICLE class definition.
- I_{AUTO} = { v6 }
- I_{2DOOR} = { v7, v8 }
- I_{4DOOR} = { v9, v10 }

- The DAG rearrangement view in Figure 10.b is constructed by the following sequence of operations:

(1) Hide SUBMARINE
(2) Hide NUCLEARVEHICLE
(3) Hide WATERVEHICLE
(4) Hide MOTORVEHICLE
(5) Hide AUTO
(6) Make VEHICLE an immediate superclass of 2DOOR
(7) Make VEHICLE an immediate superclass of 4DOOR

The following instances are visible from each class in this view:

- $I_{VEHICLE}$ = { v1, v2, v3*, v4*, v5*, v6*, v11*, v12*, v13*, v14* } where v3*, v4*, v5*, v6*, v11*, v12*, v13*, and v14* are a projected form of v3, v4, v5, v6, v11, v12, v13, and v14 respectively under the VEHICLE class definition.
- I_{2DOOR} = { v7, v8 }
- I_{4DOOR} = { v9, v10 }

- The DAG rearrangement view in Figure 10.c is constructed by the following sequence of operations:

(1) Hide 2DOOR
(2) Hide 4DOOR
(3) Hide AUTO
(4) Hide SUBMARINE
(5) Make NUCLEARVEHICLE an immediate superclass of WATERVEHICLE.

The following instances are visible from each class in this view:

- $I_{VEHICLE}$ = { v1, v2 }
- $I_{MOTORVEHICLE}$ = { v3, v4, v5, v6*, v7*, v8*, v9*, v10* } where v6*, v7*, v8*, v9*, and v10* are a projected form of v6, v7, v8, v9, and v10 respectively under the MOTORVEHICLE class definition.
- $I_{NUCLEARVEHICLE}$ = { v13*, v14* } where v13* and v14* are a projected form of v13 and v14 respectively under the NUCLEARVEHICLE class definition. (Note that v11 is dropped from the instance set of NUCLEARVEHICLE. In the original class hierarchy, v11 does not have the role of WATERVEHICLE. As such, v11 should not be visible from the NUCLEARVEHICLE class in this view because only instances with the roles of WATERVEHICLE and NUCLEARVEHICLE are qualified as instances of NUCLEARVEHICLE)
- $I_{WATERVEHICLE}$ = { v12 }

- The DAG rearrangement view in Figure 10.d is constructed by the following sequence of operations:
(1) Hide 2DOOR
(2) Hide 4DOOR
(3) Hide AUTO
(4) Create USSR-SUB as a subclass of SUBMARINE
(5) Create USA-SUB as a subclass of SUBMARINE
The following instances are visible from each class in this view:
  - $I_{VEHICLE}$ = { v1, v2 }
  - $I_{MOTORVEHICLE}$ = { v3, v4, v5, v6*, v7*, v8*, v9*, v10* } where v6*, v7*, v8*, v9*, and v10* are a projected form of v6, v7, v8, v9 and v10 respectively under the MOTORVEHICLE class definition.
  - $I_{NUCLEARVEHICLE}$ = { v11 }
  - $I_{WATERVEHICLE}$ = { v12 }
  - $I_{SUBMARINE}$ = { }
  - $I_{USSR-SUB}$ = { v13 } (assume that v13 is a submarine made in USSR)
  - $I_{USA-SUB}$ = { v14 } (assume that v14 is a submarine made in USA)

6. Operational Interface

We present a preliminary operational interface (user commands) for supporting our model of schema versions and DAG rearrangement views in object-oriented databases. Our proposal is a superset of the operational interface in [CK86].

6.1 User Commands for Schema Versions

- DERIVE-OV FROM <O> BY <I,V>$^+$: This command is used to derive a new transient object version from another object version O by replacing the value of the instance variable I with a new value V. <I,V>$^+$ means one or more of <I,V>.
- DERIVE-SV FROM <S> BY <OP>$^+$: This command is used to derive a new transient class version from another class version S by performing one or more schema change operations OP in the taxonomy of section 3. <OP>$^+$ means one or more of <OP>.
- DELETE-OV <O> | <FROM O.x TO O.y>: A series of object versions of O between O.x and O.y are deleted by this command.
- DELETE-SV <S> | <FROM S.x TO S.y> | <S.generic>: A series of class versions of S between S.x and S.y are deleted by this command. If the generic class of a class is deleted, all of its class versions are also deleted.
- PROMOTE-OV <O>: A transient object version O is promoted to a working object version by this command.
- PROMOTE-SV <S>: A transient object version S is promoted to a working object version by this command.
- CHECKIN-OV <O> TO <DB>: The object version O is checked into the target database DB by this command.
- CHECKIN-SV <S> TO <DB>: The class version S is checked into the target database DB by this command.
- CHECKOUT-OV <O> TO <DB>: This command is used to check out an object version O to the target database DB.
- CHECKOUT-SV <S> TO <DB>: This command is used to check out a class version S to the target database DB.

If we allow schema versions and object versions, query languages for object-oriented databases need to be extended for querying and manipulating schema versions as well as object versions. The following situations may require extension of query languages.

1. Queries on schema derivation hierarchy: What are the child class versions of a class version S?, What is the parent class versions of a class version S?, What is the next class version number of a class version S?, etc. The same argument is applied to object versions.
2. Temporal queries over schema versions: Retrieve all instances which are created under a class version S, Retrieve all instances which are created between a class version S.x to a class version S.y, etc.

However, the issue of query language extension is beyond the scope of this paper.

6.2 User Commands for DAG Rearrangement Views

- DEFINE-COV <COV> FROM <S> AS <OP>$^+$: This command creates an composite object view COV by applying composite object view construction operations <OP>$^+$ to a root class S and childpart classes of S.
- DEFINE-CHV <CHV> FROM <S> AS <OP>$^+$: This command creates a class hierarchy view CHV by applying class hierarchy view construction operations <OP>$^+$ to a root class S and subclasses of S.
- DROP-COV <COV>: The composite object view definition of COV is dropped.
- DROP-CHV <CHV>: The class hierarchy view definition of CHV is dropped.

In posing queries to DAG rearrangement views of composite objects and class hierarchies, view names may be indicated with a new query language construct like **FROM** <view name>.

7. Summary

In this paper, we addressed two issues of object-oriented database schemas which were not addressed in the database literature. We presented a model of schema versions and DAG rearrangement views in object-oriented databases. The paper made four contributions.

The first contribution of the paper is in the development of a model which extends schema evolution, by allowing *schema versions* in object–oriented databases. By allowing schema versions as well as object versions, evolution of applications is completely supported by the database system. We defined the semantics of schema versions. We presented a technique that enables users to manipulate schema versions explicitly and maintain schema evolution histories in an object-oriented database environment. Our solution for schema versions is consistent with our previous work on schema evolution [BKKK86,BKKK87], and it guarantees *minimum storage redundancy* and allows us to avoid the problem of *update anomaly*.

The second contribution of the paper is the integration of our schema version model with Chou and Kim's object version model [CK86]. Chou and Kim's object version model is designed for distributed CAD databases. We examined various issues of Chou and Kim's object version model in our context, including working and transient schema versions, schema version check-in and check-out, version naming and binding, and change notification.

The third contribution of the paper is the definition of DAG rearrangement views in object-oriented databases. We presented sets of useful operators for defining DAG rearrangement views of composite objects and class hierarchies respectively. We identified sets of composite object views with the property that queries on the views are processable on instances of the original composite object schema. We also discuss how instances would be viewed and reorganized in DAG rearrangement views of class hierarchies.

The fourth contribution of this paper is an operational interface for manipulating schema versions and constructing DAG rearrangement views.

Implementation aspects of schema versioning and DAG rearrangement views will be addressed in a forthcoming paper.

REFERENCES

[ACO85] Albano, A., L. Cardelli and R. Orisini, "Galileo: A strongly-typed interactive conceptual language," *ACM Transactions on Database Systems*, Vol. 10, No. 2, 1985.

[Ahls84] Ahlsen, M., A. Bjornerstedt, S. Britts, C. Hulten, and L. Soderlund, "An Architecture for Object Management in OIS," *ACM Transactions on Office Information Systems*, Vol. 2, No. 3, July, 1984.

[AKMP86] Afsarmanesh, H., D. Knapp, D. McLeod, and A. Parker, "An Object-Oriented Approach to VLSI/CAD," *Proceedings of International Conference on Very Large Databases*, Stockholm, Sweden, August, 1985.

[Atwo85] Atwood, T.M., "An Object-Oriented DBMS for Design Support Applications," *Proceedings of IEEE COMPINT*, Canada, 1985.

[Ban87] Banerjee, J. et al., "Data Model Issues in Object-Oriented Appplications" *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, March, 1987.

[BK85] Batory, D. and W. Kim, "Modeling Concepts for VLSI CAD Objects", *ACM Transactions on Database Systems*, Vol. 10, No. 3, September, 1985.

[BKKK86] Banerjee, J., H.J. Kim, W. Kim and H.F. Korth, "Schema Evolution in Object-Oriented Persistent Databases," *Proceedings of 6th Advanced Database Symposium*, Tokyo, Japan, 1986.

[BKKK87] Banerjee, J., W. Kim, H.J. Kim and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *Proceedings of ACM SIGMOD Conference on the Management of Data*, San Francisco, CA, May, 1987.

[Bob85] Bobrow, D.G. et al., "CommonLoops: Merging Common Lisp and Object-Oriented Programming," *Intelligent Systems Laboratory Series ISL-85-8*, Xerox PARC, Palo Alto, CA., 1985.

[BS83] Bobrow, D.G. and M. Stefik, "The LOOPS Manual," *Xerox PARC*, Palo Alto, CA., 1983.

[CA84] Curry, G.A. and R.M. Ayers, "Experience with Traits in the Xerox Star Workstation," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September, 1984.

[CK86] Chou, H-T. and W. Kim, "A Unifying Framework for Version Control in a CAD Environment," *Proceedings of International Conference on Very Large Databases*, 1986.

[DL85] Dittrich, K. and R. Lorie, "Version Support for Engineering Database Systems," *IBM Research Report: RJ4769*, IBM San Jose, July, 1985.

[Fish87] Fishman, D.H. et al., "Iris: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems*, Vol. 5., No. 1, 1987.

[GR83] 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, January, 1981.

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

[KCB86] Katz, R., E. Chang, and R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," *Proceedings of ACM SIGMOD Conference on the Management of Data*, Washington, D.C., May, 1986.

[Kel85] Keller, A., "Updating Relational Databases Through Views," *Stanford University*, PhD Thesis, STAN-CS-85-1040, February, 1985.

[Kim88] Kim, H.J., "Issues in Object-Oriented Database Schemas," *The University of Texas at Austin*, PhD Thesis, May, 1988.

[KK86] Kim, H.J. and H.F. Korth, "Property Inheritance Graph: A Formal Model of Multiple Inheritance in Object-Oriented Databases," *Unpublished memo*, Dept. of Computer Science, University of Texas at Austin, Texas, Dec. 1986.

[KL84] Katz, R. and T. Lehman, "Database Support for Versions and Alternatives of Large Design Files," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, March, 1984.

[KS86] Korth, H.F. and Silberschatz, A. "Database System Concepts," *McGraw-Hill Book Company*, 1986.

[LB86] Levesque, H., and Brachman, R., "A Fundamental Tradeoff in Knowledge Representation and Reasoning," *(submitted to) Computational Intelligence*, 1986.

[LP83] Lorie, R. and W. Plouffe, "Complex Objects and Their Use in Design Transactions," *Proceedings of ACM-IEEE Database Week*, 1983.

[MBW80] Mylopoulos, J., P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Transactions on Database Systems*, Vol. 5, No. 2, 1980.

[MOP85] Maier, D., A. Otis and A. Purdy, "Object-Oriented Database Development at Servio Logic," *Database Engineering Bulletin*, Vol. 8, No. 4, 1985.

[MRI78] "System 2000 Reference Manual," *MRI Systems Corp.*, Austin, Texas, 1978.

[SB86] Stefik, M. and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, Winter 1986.

[SFL83] Smith, J.M., S.A. Fox, and T. Landers, "ADAPLEX Rationale and Reference Manual," *Technical Report CCA-83-08, Computer Corporation of America*, Cambridge, MA, May 1983.

[SWKH76] Stonebraker, M., E. Wong, P. Kreps and G. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, Vol. 1., No. 3, 1976.

[Symb84] "FLAV Objects, Message Passing, and Flavors," *Symbolics Inc.*, Cambridge, MA, 1984.

[WKL86] Woelk, D., W. Kim and W. Luther, "An Object-Oriented Approach to Multimedia Databases," *Proceedings of ACM SIGMOD Conference on the Management of Data*, Washington D.C., May, 1986.