ISSUES IN OBJECT-ORIENTED DATABASE SCHEMAS

Hyoung Joo Kim

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

TR-88-20 May 1988

To my parents

Jang-Jin Kim and Yun-Haeng Kwon

and

my wife Myoung-So

ISSUES IN OBJECT-ORIENTED DATABASE SCHEMAS

by

Hyoung Joo Kim, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 1988

Abstract

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), computer-aided design (CAD), and office information systems (OIS). In order to provide the additional semantics necessary to model these new applications, many researchers have adopted the object-oriented programming paradigm to serve as a data model. In order to use the object-oriented approach in a database system, it was necessary to add persistence and sharability to the object-oriented programming paradigm. Several database systems based on this approach are under implementation. Therefore, object-oriented database systems combine the strengths of object-oriented programming languages and conventional database systems.

The object-oriented data models are more powerful than conventional data models due to the support of inheritance in class hierarchies, composite objects and new data types. The notions of inheritance and composite objects are enormously important in supporting new database applications of AI, CAD, and OIS. However, the object-oriented database schema is more complicated than conventional database schemas because both class hierarchy and composite object structures are directed acyclic graphs (DAG). In this dissertation, we examine various issues of object-oriented database schemas: *schema evolution, graphics schema editor, schema versioning, DAG rearrangement views, logical design of object-oriented database schema,* and *subclassing.*

The practical applications of object-oriented databases, such as CAD, AI, and OIS, require the ability to dynamically make a wide variety of changes to the database schema. This process is called *schema evolution.* We establish a consistent and complete framework of schema evolution. Based on our framework, the MCC ODBS group implemented a schema manager within their prototype object-oriented database system, ORION. On top of the schema manager of ORION, we implemented *a graphical schema editor,* PSYCHO.

We present a technique that enables users to manipulate *schema versions* explicitly and maintain schema evolution histories in object-oriented database environments. We also examine a new concept, *DAG rearrangement views,* which are rearrangements of DAG structures of composite objects and class hierarchies in object-oriented databases.

Since schema evolution is closely related with the logical design of database schema, we investigate the logical design of object-oriented database schema. We establish *a unified framework for the logical design of object-oriented database schema* by synthesizing research results of the areas such as AI knowledge representation, database dependency theory, AI theorem proving, and graph algorithms. Finally, we elaborate on *subclassing* which is the most frequently used schema change operation.

CONTENTS

Chapter 1
Introduction

1.1 Overview and Motivation

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, Weid86], computer-aided design (CAD) [AKMP86, KLW87, Zdo85], and office information systems (OIS) [ABH85, Ahls84, IEEE85, WKL86, WK87a, WK87b]. In order to provide the additional semantics necessary to model these new applications, many researchers, have adopted the object-oriented programming paradigm to serve as a data model [Gold81, GR83, BS83, CA84, Rowe86a, Symb84]. In order to use the object-oriented approach in a database system, it was necessary to add persistence and sharability to the object-oriented programming paradigm. Several database systems based on this approach are under implementation [KTKB88, PSM87], including GEMSTONE [MOP85, MSOP86, Serv86], IRIS [Fish87], and ORION [Ban87]. Therefore, object-oriented database systems combine the strengths of object-oriented programming languages and conventional database systems.

Object-oriented data models are more powerful than conventional data models due to the support of *class hierarchies* and *composite objects*. A class can have one or more superclasses and also one or more subclasses. Thus, the structure of a class hierarchy is essentially a directed acyclic graph (DAG). A class hierarchy captures the IS-A relationship between a class and its subclass. All subclasses of a class inherit all of the 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. The notion of composite object explicitly captures the IS-PART-OF relationship: a composite object is a collection of related objects that form a hierarchical structure that captures the IS-PART-OF relationship among the

objects. As we will show later, the structure of composite object schema is a directed acyclic graph (DAG), but the structure of an instance of composite object is strictly hierarchical (tree). The notions of inheritance and composite object are enormously important in supporting new database applications of AI, CAD, and OIS. However, the object-oriented database schema is more complicated than conventional database schemas because both the class hierarchy and the composite object structure are DAGs.

In this dissertation, we investigate various issues of object-oriented database schemas, such as *schema evolution*, *graphics schema editor*, *schema versioning*, *DAG rearrangement views*, *logical design of object-oriented database schema*, and *subclassing*. Coping with these issues includes solving theoretically and practically non-trivial problems which stem from the complexity of object-oriented database schemas and the requirements of new database applications.

Schema Evolution [BKKK86, BKKK87, Ban87, KK86]: One of the important requirements of non-traditional applications such as CAD/CAM, AI, and OIS is *schema evolution*, that is, the ability to make a wide variety of changes to the database schema dynamically. There are several applications to which schema evolution is essential: (1) in a CAD environment, the users tend to arrive at the schema for design objects through trial and error [WKL86], (2) it is important for a design database to be able to deal with change at all levels including the schema level [SZ86], and (3) in expert or knowledge based systems of AI, when new knowledge is acquired, the new knowledge may require the addition of new relationships, new structures, or revisions to existing structures to maintain a valid representation [IRA83]. We provide a consistent and complete framework of schema evolution in object-oriented databases and the framework was realized in an object-oriented database system, ORION at MCC (Microelectronics and Computer Technology Corporation). We show the soundness and completeness of our schema evolution framework through a formal model, PIG (Property Inheritance Graph).

Graphics Schema Editor [KK88b]: Based on the schema evolution framework, the MCC ODBS group implemented a schema manager within their prototype object-oriented database system, ORION. Schema modifications using line-oriented interaction are difficult for the user to manage if class hierarchies are complicated. The difficulty is even greater because there are a large number of types of schema modifications. We have designed and implemented a graphical interface PSYCHO (*Pictorial Schema-editor Yielding Class Hierarchies* and *Objects*) on top of the ORION system for the following purposes: (1) A user-friendly interface of the schema management of ORION, (2) A tool for empirically validating the correctness and usefulness of the ORION schema evolution framework. We discuss the use of PSYCHO and its implementation.

Schema Versioning [KK88c]: As we mentioned earlier, one of the important requirements of non-traditional database applications such as CAD, 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. In the schema evolution framework, it is assumed that whenever a schema definition is updated, the previous schema is changed to a new one and existing object instances are modified in order to comply with the new schema (i.e., overriding the previous schema and its instances). We extend our schema evolution framework by allowing *schema versions* in object-oriented databases. Even though there has been a substantial amount of research on object versioning, the issue of schema versioning has not been addressed in the database literature. We present a technique that enables users to manipulate schema versions explicitly and maintain schema evolution histories in object-oriented database environment. Our solution for schema versions is consistent with our schema evolution framework, guarantees *minimum storage redundancy* and allows us to get around the problem of *update anomaly*. We define semantics of schema versions. For our schema version model to be complete, we integrate our model with the object version model formulated by H.T. Chou and W. Kim [CK86].

DAG Rearrangement Views [KK88c]: As we mentioned above, two distinct notions in object-oriented data models are *composite objects* and *class hierarchies*. The semantics of the two notions are captured in directed acyclic graphs (DAG). Most of the applications in object-oriented databases assume a group of cooperative workers (i.e., team) who are sharing the same objects. However, users may not need to see the whole database and objects in it. They would like to see only relevant parts of composite objects and relevant classes of class hierarchies which are necessary to their applications. This premise gives rise to the notion of *DAG Rearrangement Views*. We identify new types of DAG rearrangement views of composite objects and class hierarchies. We present sets of useful 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 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 logical design of object-oriented database schema [KK88a]: The logical design of object-oriented database schema has not been fully understood in the database literature. Schema evolution is closely related with the logical design of object-oriented database schema. Class hierarchy design is the main theme of schema design for object-oriented databases. We establish a unified framework for logical database design for object-oriented database schema by synthesizing research results from the areas of AI knowledge representation, database dependency theory, AI theorem proving, and graph algorithms. We revisit the ORION data model and the ORION schema evolution model from the viewpoint of the unified framework of object-oriented database design that we establish.

Subclassing [KK88a]: We pay special attention to *subclassing* (creating a new subclass from a class) which is the most frequently used schema change operation. A new class needs to be created from a class when a new concept, which cannot be accommodated in the existing classes, has to be introduced. Most subclassings that are motivated from imposing restrictions on

instance variables of a parent class are accompanied by associated constraints, called *subclassing conditions*, which are predicate expressions on instance variables. We present the results of our research into various issues of subclassing: First, we present a taxonomy of subclassing and the semantics of each case. Second, we address the issue of subclassing condition management because most subclassings are accompanied by subclassing conditions. As a database and schema grow in size and complexity, it is very difficult to maintain consistent class hierarchies without taking advantage of subclassing conditions. We also consider the inverse operation of subclassing, *desubclassing* (dropping an existing class). Third, subclassing conditions are useful in many applications of object-oriented databases. We identify those applications and introduce the techniques of applying subclassing conditions to the applications.

1.2 Object-Oriented Data Models

In this section we review the core concepts of object-oriented data models. The ORION data model is the base data model for this dissertation. The results of this dissertation can be applied to most mainstream object-oriented data models because the ORION data model includes the usual core concepts of object-oriented data models. Besides the core concepts, we also indicate some assumptions or conventions in the ORION data model which are necessary in the rest of the dissertation. The full description of the ORION data model is available elsewhere [Ban87].

1.2.1 Core Concepts

Object-oriented data models [Ban87, MOP86, Fish87] support the usual features of object-oriented languages, including the notions of classes, subclasses, class hierarchies, and objects. Each entity in a database is an *object*. Objects include *instance variables* that describe the state of the object. The value of an instance variable may itself be an object with its own internal state, or it may be a *primitive object* such as integers and strings which have no instance variables. Objects also include *methods* which contain code used to manipulate

the object or to return part of their 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 a class hierarchy. The result is a directed acyclic graph (DAG) containing an edge from C_1 to C_2, i.e., (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 from which a path exists to it. The class-superclass relationship (C_1, C_2) is an "ISA" relationship in the sense that every instance of a class is also an instance of the superclass. Using the terminology of the extended 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.

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* [GR83, SB86]. 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.

1.2.2 The ORION Data Model

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 also an instance of C. These inclusion relationships should be maintained to make ISA relationships among classes of a class hierarchy meaningful.

Even if an instance is logically a member of a class C and all of its superclasses, the instance is not necessarily stored physically to C and C's superclasses in duplicate. There are two ways of positioning (physically storing) instances in class hierarchies. One can allow an instance to (physically) belong to *more than one* class, or one can require that an instance (physically) belong to *one and only one* class. While some object-oriented systems, such as GALILEO [ACO85], ADAPLEX [SFL83], 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 and meaningful state. The same argument applies to insert and update operations. One disadvantage of requiring instances to belong to one and only

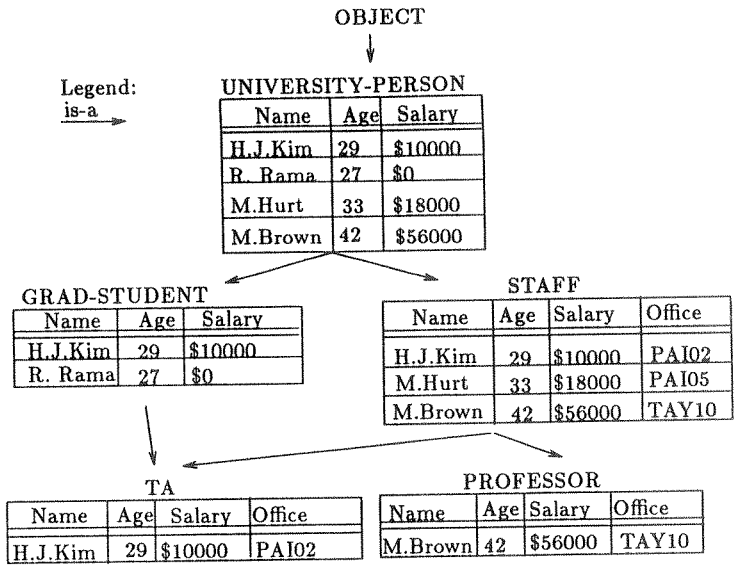


Figure 1.a: Instances belonging to more than one class

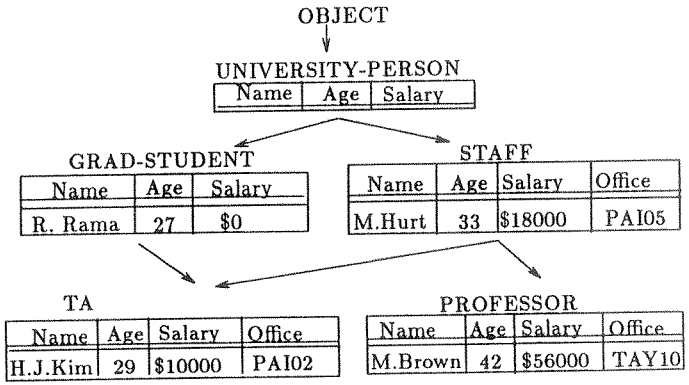
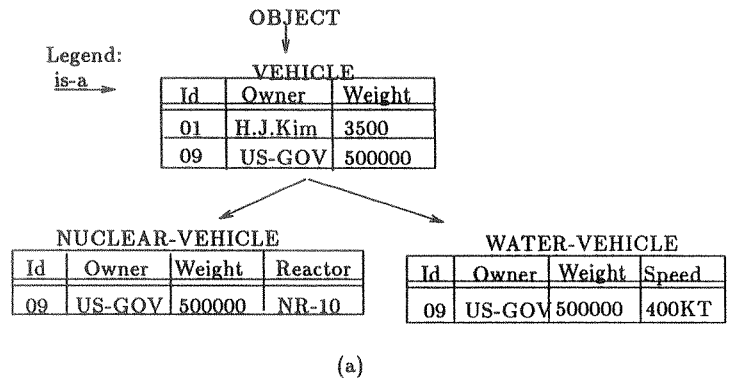
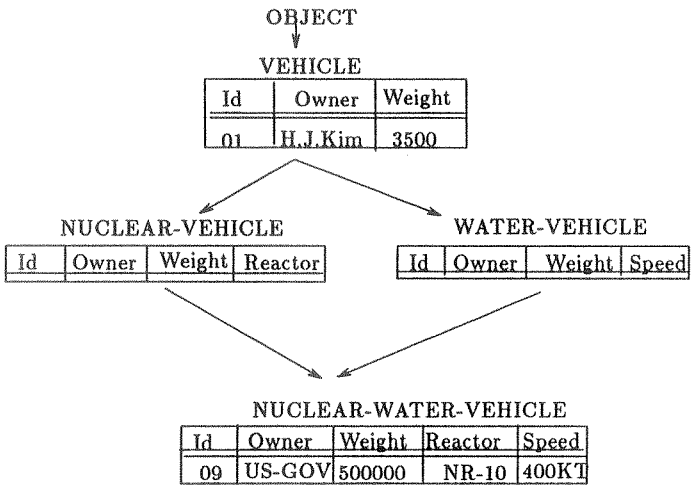


Figure 1.b: Instances belonging to one and only one class



(a)



(b)

Figure 2: Modeling instances in VEHICLE database

one class is that the query language is more complicated (addressed in next section); two types of 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 NUCLEAR-VEHICLE and WATER-VEHICLE (and, of course, to VEHICLE), whereas in Figure 2.b, a new class, called NUCLEAR-WATER-VEHICLE, must be created for storing the object 09 in a single class.

In this dissertation, we follow the approach in Figure 1.b and Figure 2.b and assume that instances cannot (physically) belong to more than one class, as in the ORION data model. In particular, this assumption influences the semantics of schema versioning as we will show in Chapter 4.

Operations on Instances

The user's view of a single class is similar a relational view. SQL-like query languages (SELECT-FROM-WHERE type) can be adapted 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 assumptions.

- 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. When an instance is moved from a class to another class, the user must provide appropriate values for the evolving instance. The MOVE operation is an atomic operation subsuming both the INSERT and DELETE operations.

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, Lor84, Kim85b], molecular aggregations [BK85a], composite objects [Bob85, Kim87], and aggregation hierarchies [Atwo85].) As such, most of the object-oriented data models support the notion of composite objects.

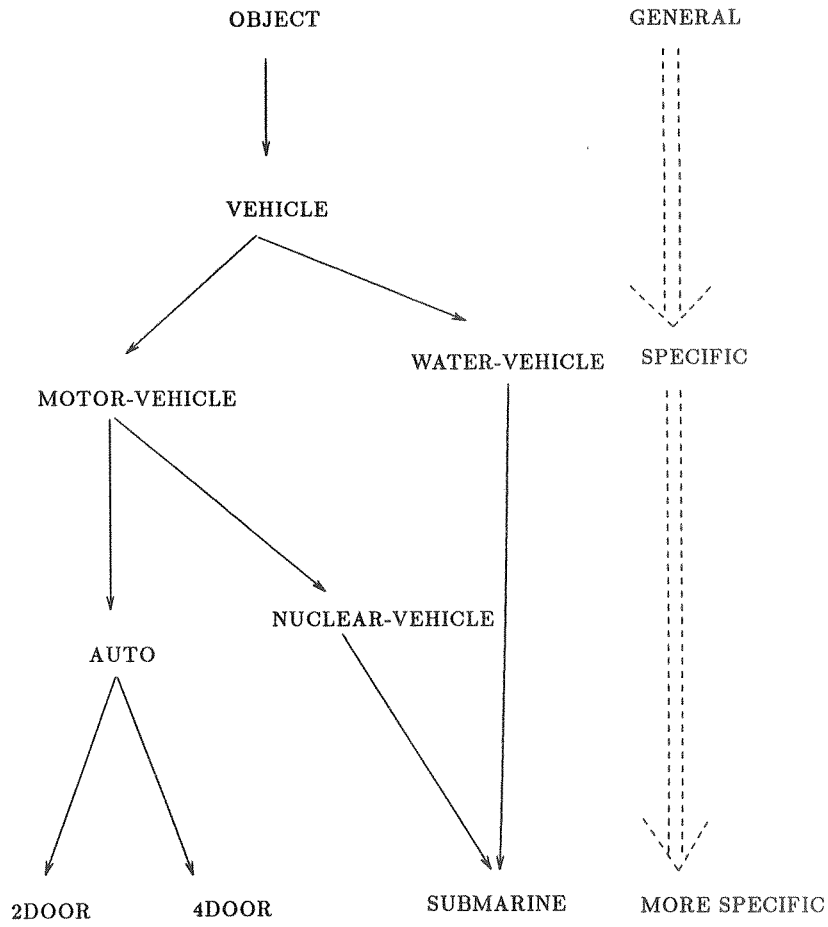


Figure 3: VEHICLE Class Hierarchy

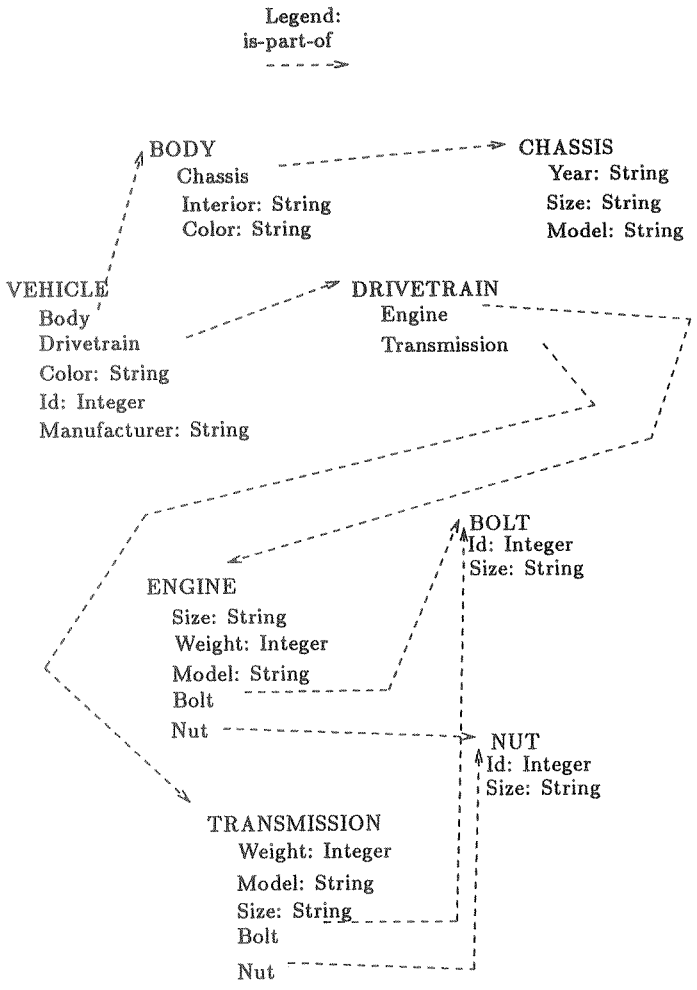


Figure 4: VEHICLE composite object

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* 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 also organized in a DAG. 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*.

We shall use a vehicle class hierarchy and a vehicle composite object as an example in several chapters of the dissertation. The class hierarchy for vehicles is shown in Figure 3: VEHICLE class has two subclasses MOTOR-VEHICLE and WATER-VEHICLE, and, in turn, MOTOR-VEHICLE has AUTO and NUCLEAR-VEHICLE as subclasses, and so on. 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 and so on.

We call instance variables such as Body and Drivetrain *composite instance variables*, that serve as links to dependent classes. The link between a class and the domain of a composite instance variable of a class is called *a composite link*. The dotted lines in Figure 4 are a composite link.

Name Conflict Resolution

SmallTalk [Gold81] originally restricted a class to have only a single superclass, thus limiting the class hierarchy to a tree (called *single inheritance*). Most other object-oriented systems, as well as the recent version of SmallTalk, have relaxed this restriction. In these systems (and in ORION) a class can have more than one superclass, generalizing the class hierarchy to a DAG (directed acyclic

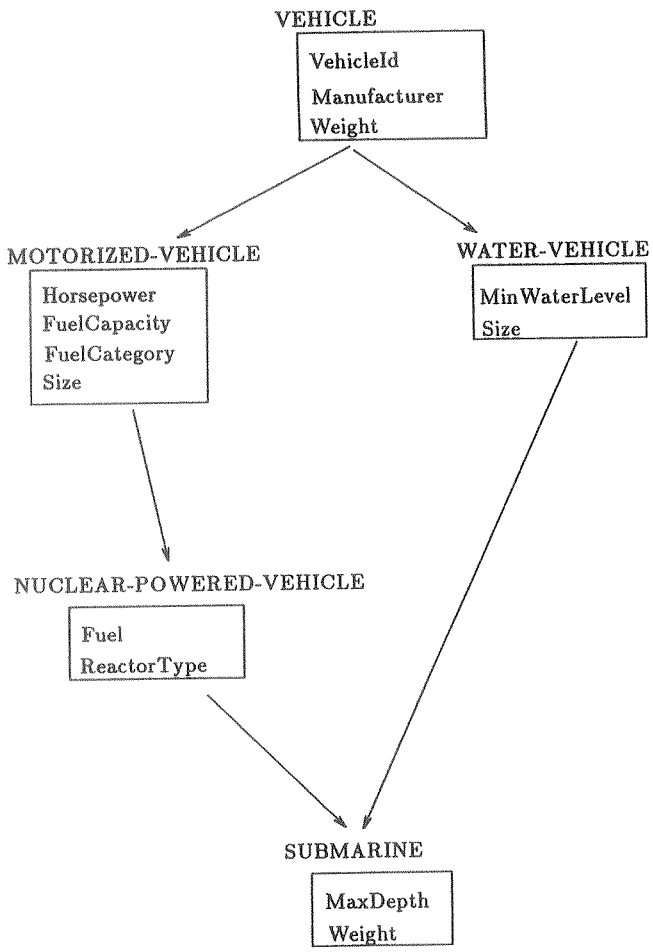


Figure 5: Resolution of name conflicts among instance variables

graph). Since a class has multiple superclasses and thus inherits properties from each of the superclasses. This feature is often referred to as *multiple inheritance* [LMI85, SB86].

Multiple inheritance simplifies data modeling and often requires fewer classes to be specified than with single inheritance. 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 with single inheritance). 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 5, the class SUBMARINE has to inherit an instance variable Size either from the superclass WATER-VEHICLE (which defined Size) or from NUCLEAR-POWERED-VEHICLE (which inherited Size from its superclass MOTORIZED-VEHICLE). If in the definition of the class SUBMARINE, NUCLEAR-POWERED-VEHICLE was specified as the first superclass, Size will be inherited from NUCLEAR-POWERED-VEHICLE.

Since this default conflict resolution scheme hinges on the permutation of the superclasses of a class, ORION allows the user to change this permutation explicitly 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.

1.3 Sequence of Presentation

The remainder of the dissertation is organized as follows. Chapter 2 presents schema evolution. The schema evolution framework for ORION is introduced in Section 2.1. We discuss the soundness and completeness of the schema change operations in our framework via a formal model, called Property Inheritance Graph (PIG) in Section 2.2. In Chapter 3, after briefly mentioning the schema manager of the ORION system, we discuss the use of PSYCHO and its implementation. In Chapter 4, we investigate a technique that enables users to deal with schema versions explicitly and maintain schema evolution histories in object-oriented database environment. We also define semantics of schema versioning operations. In Chapter 5, we study new type of views, called DAG rearrangement views of class hierarchies and composite objects. We introduce a set of operations for defining DAG rearrangement views and address the query processing aspects of DAG rearrangement views. In Chapter 6, we establish a unified framework for the logical design of object-oriented database schema by synthesizing research results from the areas of AI knowledge representation, database dependency theory, AI theorem proving, and graph algorithms. Finally, in Chapter 7, we elaborate on the most frequently used schema change operation, subclassing. We deal with several issues pertaining to subclassing.

Chapter 2
Schema Evolution

In this chapter, we provide a schema evolution framework for object-oriented databases. Portions of section 2.1 are due to Banerjee et al. [BKKK86, BKKK87]. Section 2.2 is also available elsewhere [KK86]. In section 2.3, we present related work pertaining to schema evolution.

2.1 The Schema Evolution Framework

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 in a systematic way. We emphasize that, although our framework has been developed for the ORION data model, we believe that our methodology for the development of the framework is applicable to most mainstream object-oriented programming language systems and database systems. This is because the ORION data model has incorporated all of 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.

Since there are so many schema change operations, we need a framework for defining consistent semantics of each of schema change operations. Our formal framework consists of a set of properties of the schema called *invariants*, and a set of *schema transformation rules*.

The invariants must hold at every stable 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, in defining semantics of some schema change operations, there was more than one way of preserving schema invariants. The set of rules that we have guides the selection of one most meaningful way. Some rules reflect our effort to avoid drastic changes due to schema changes to the database.

Therefore, we could derive globally consistent and meaningful semantics for each of schema change operations by applying the sets of invariants and schema transformation rules.

2.1.1 Invariants of Consistent Schemas

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 hierarchy under our data model.

Class Hierarchy Invariant

The class hierarchy is a *single rooted* and *connected directed acyclic graph* with named nodes and labeled edges. This 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. Each node in the DAG has a unique name. Edges are labeled so that all edges directed toward 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 (class of origin). For example, in Figure 5, the class SUBMARINE can inherit the instance variable Weight from either the class WATER-VEHICLE or NUCLEAR-POWERED-VEHICLE. 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 5, SUBMARINE must inherit Size, whether it is from NUCLEAR-POWERED-VEHICLE or from WATER-VEHICLE, 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 NUCLEAR-POWERED-VEHICLE or WATER-VEHICLE.

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 that 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 MOTORIZED-VEHICLE class, a subclass of VEHICLE, can be COMPANY or a subclass of COMPANY, say, MOTORIZED-VEHICLE-COMPANY.

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.

2.1.2 Schema Transformation Rules

A class hierarchy in a stable 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 rule 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 5, if the domain of Manufacturer of NUCLEAR-POWERED-VEHICLE is COMPANY, and the domain of Manufacturer of WATER-VEHICLE is WATER-VEHICLE-COMPANY which is a subclass of COMPANY, the Manufacturer instance variable from the class SUBMARINE is inherited from the class WATER-VEHICLE.

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 MOTORIZED-VEHICLE (which is a subclass of VEHICLE), then MOTORIZED-VEHICLE will not accept the change. Consequently, the change is not propagated to any subclass of MOTORIZED-VEHICLE that inherited Weight from MOTORIZED-VEHICLE.

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 incompatible 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 MOTORIZED-VEHICLE can be generalized from MOTORIZED-VEHICLE-COMPANY 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 hierarchy. First, the addition of an edge from node A to node B on a class hierarchy 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 hierarchy.

Rule 7 (Edge Addition Rule): If class A is made a superclass of 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 the case that class A is the only superclass of class B. The following rule is necessary to preserve the class hierarchy 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 hierarchy invariant. If the new node has no superclasses, it becomes an isolated node, violating the class hierarchy 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 hierarchy invariant.

Rule 10 (Node Removal Rule): For the deletion of edges from class 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 also to be deleted.

2.1.3 Taxonomy of Schema Change Operation

In this subsection, we classify all schema change operation in our framework, and define the semantics of schema changes, using our schema evolution invariants and rules. Changes to the class hierarchy 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 below provide the semantics of the schema change operations informally. Then, in the next section, we precisely define the semantics of each schema change operation by showing how the schema transformation rules are applied to maintain the schema evolution invariants.

(1) Changes to the contents of a node

(1.1) Changes to an instance variable

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

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

(1.1.2) Drop an instance variable V from a class C: The instance variable V is dropped from the definition (and from the instances) of the class C. C may inherit V from another superclass, if there had been a name conflict involving V. All subclasses of C will also be affected if they had inherited V from C. 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.

(1.1.3) Change the name of an instance variable V to 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 from V of C. The name change is required to be propagated only if V' does not give rise to new conflicts in the subclasses. Further, 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: The domain of an instance variable is itself a class. The domain,

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

(1.1.5) Change the inheritance (parent) of an instance variable: (Inherit a different instance variable with the same name) As discussed earlier, if two or more superclasses of a class C have an identically named variable (either through inheritance or local definition), the system selects only one of them for inheritance by C, based on the order in which the superclasses have become associated with C. The user can explicitly override this default.

If class C has instances, the present values of the conflicting instance variable V must be dropped, and replaced by any default value under the new definition. If C has subclasses which had inherited V, they will now inherit the new definition. Consequently, their instances will be subjected to the same changes as those for the instances of C.

(1.1.6) Change the default value of an instance variable V of a class C: All instances of C that have no value supplied for the variable V already have a default value or nil. They will now get the new default value. If there exists any subclass of C which had inherited V from C, it must also inherit this new default value, unless that subclass has redefined the default value of V.

(1.1.7) Manipulate the shared value of an instance variable

(1.1.7.1) Add the shared value of an instance variable V of a class C: This operation converts a non-shared-value instance variable V to a shared-value instance variable. If V already had a shared value, then all instances of the class C receive the new value. If V was not previously a shared-value variable, it now becomes one, and all instances of C will

take on this new value, dropping any existing values for V in existing instances of C.

If C has subclasses which had inherited V, they will now inherit the new shared value of V, unless they have redefined the value.

(1.1.7.2) Change the shared value of an instance variable V of a class C: This operation replaces the shared value of V with a new one. If C has subclasses which had inherited V, they will now inherit the new shared value of V, unless they have redefined the value.

(1.1.7.3) Drop the shared value of 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. If C has subclasses which had inherited V, they will now drop the shared value of V, unless they have redefined the shared value.

(1.1.8) Drop the composite link property of an instance variable: This operation changes the composite link property of an instance variable V of a class C to non-composite. the change is propagated to the subclasses of C.

(1.2) Changes to a method

(1.2.1) Add a new method to a class: The semantics for this operation is easily inferred from the operation (1.1.1) 'Add a new instance variable to a class'.

(1.2.2) Drop an existing method from a class: The semantics for this operation is easily inferred from the operation (1.1.2) 'Drop an existing method from a class'.

(1.2.3) Change the name of a method of a class: The semantics for this operation is easily inferred from the operation (1.1.3) 'Change the name of an instance variable of a class'.

(1.2.4) Change the code of a method in a class: The semantics for this operation is easily inferred from the operation (1.1.4) 'Change the domain of an instance variable of a class'.

(1.2.5) Change the inheritance (parent) of a method: The semantics for this operation is easily inferred from the operation (1.1.5) 'Change the inheritance of an instance variable'.

(2) Changes to an edge

(2.1) Make a class S a superclass of a class C: Class S is made the last superclass in the list of superclasses of C. C will now inherit the variables and methods from S. If this causes any name conflict, the system will ignore the instance variable (or method) of S in conflict, because C must already have inherited the conflicting variable (or method) from some of its current superclasses. The user may explicitly specify alternate conflict resolution.

If C has subclasses, immediate or indirect, they also inherit instance variables and methods from S. Such inheritance may cause new name conflicts, but they will also be ignored. (Once again, the user may explicitly specify conflict resolutions that will override the default.)

Class C and its subclasses may have existing instances. Since the instance variables they inherit from S are new to these instances, they appear in the instances with the value nil or any default value specified.

(2.2) Remove a class S as a superclass of the class C: The variables (or methods) inherited from S are dropped from the definition of C. C may newly inherit the dropped instance variables (or methods) from other superclasses, if there had been name conflicts involving them. The instances of C are also modified as discussed earlier for the dropping of a class. All subclasses of C will also be affected similarly if they had inherited variables (or methods) from S via C.

(2.3) Change the order of the superclasses of a class C: This alters the default conflict resolution with respect to the class C. Conflicting variables and methods will now be inherited according to the new permutation of superclasses. If the definition of a variable changes because of this new permutation, existing instances of class C will be affected as well. The value that each of the affected variable takes on is the default value under the new definition. Subclasses of C are affected similarly.

(3) Changes to a node

(3.1) Define a new class C: The new class C may be created as a specialization of an existing class or classes. These latter classes can be specified as the superclasses of the new class. The variables (or methods) specified for C will override any conflicting instance variables (or methods) inherited from the superclasses. If there is a name conflict involving the variables (or methods) that C inherits from its superclasses, default conflict resolution is used, unless the user explicitly overrides it.

The class C may also be defined without any superclasses. In this case, C is made a subclass of OBJECT which is a system defined class. Conceptually the OBJECT class is a root node of every class hierarchy. The user may, at a later time, add superclasses for C, in which case OBJECT will no longer be an immediate superclass of C.

(3.2) Drop a class C: Whenever a class definition is dropped, all its instances are deleted automatically, since instances cannot exist outside of a class. However, subclasses of C, if any, are not dropped. Subclasses of C will lose C as their superclass; however, they will gain C's superclasses as their immediate superclasses directly. Further, when a class C is dropped, its subclasses will lose the instance variables and methods they had previously inherited directly from C. If, in the process, a subclass of C loses a variable V (or a method) which was selected over a conflicting variable in another superclass of that subclass, it will now

inherit the alternative definition of V. Consequently, the instances of any such subclass will lose their present values for V, and inherit the default value (or nil) under the new definition of V.

When an instance of the class C is deleted, all objects that reference it will now be referencing a non-existent object. The user will need to modify those references when they are encountered. References to non-existent objects will not be automatically identified because of the performance overhead.

If the class C being dropped is presently the domain of an instance variable $V1$ of some other class, $V1$'s domain becomes the first superclass of the class C. Of course, the user has the choice of specifying a new domain for $V1$.

(3.3) Change the name of a class: If the new name is unique among all class names in the class hierarchy, the name change is allowed. This name change is not propagated.

2.1.4 Semantics of Schema Change Operation

We now define the semantics of each of schema change operation, while showing how the schema transformation rules are applied to maintain the schema evolution invariants. In the next section, we analyze the effect of each schema change operation on existing instances.

(1) Changes to the contents of a node

(1.1) Changes to 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) Manipulate the shared value of an instance variable

(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) Changes to a method

(1.2.1) Add a new method to a class: The rules to apply for deriving the semantics of this operation are easily inferred from the operation (1.1.1) 'Add a new instance variable to a class'.

(1.2.2) Drop an existing method from a class: The rules to apply for deriving the semantics of this operation are easily inferred from the operation (1.1.2) 'Drop an existing method from a class'.

(1.2.3) Change the name of a method of a class: The rules to apply for deriving the semantics of this operation are easily inferred from the operation (1.1.3) 'Change the name of an instance variable of a class'.

(1.2.4) Change the code of a method in a class: The rules to apply for deriving the semantics of this operation are easily inferred from the operation (1.1.4) 'Change the domain of an instance variable of a class'.

(1.2.5) Change the inheritance (parent) of a method: The rules to apply for deriving the semantics of this operation are easily inferred from the operation (1.1.5) 'Change the inheritance of an instance variable'.

(2) Changes to an edge

(2.1) Make a class S a superclass of a class C: To preserve the class hierarchy invariant, the addition of a new edge from S to C must not introduce a cycle in the class hierarchy. 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 hierarchy invariant, the deletion of an edge from S to C must not cause the class hierarchy 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) Changes to 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 hierarchy invariant, it is ensured that the new name is unique among all class names in the class hierarchy.

2.1.5 Effects of Schema Changes on Existing Instances

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

(1) Changes to the contents of a node

(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 projected out.

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 effect 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 a class C: There is no effect 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) Manipulate the shared value of an instance variable

(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 effect 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 effect.

(1.2) Changes to a method of the class C: There is no effect on the existing instances of C. Methods appear only in the definition of the class.

(2) Changes to 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) Changes to a node

(3.1) Add a new class C: Since C is a new class and has no existing instances, there is no effect.

(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 effect on the existing instances of C.

2.2 PIG: The Formal Model

2.2.1 Motivation behind PIG

The taxonomy of schema evolution appears intuitively to capture all 'interesting' types of schema change. One interesting and important question to consider is whether the ORION schema evolution taxonomy indeed captures every possible of schema change (i.e., *completeness*). Another interesting question is whether every ORION schema change operation generates only valid schemas preserving the invariants (i.e., *soundness*).

However, showing the soundness and completeness of 20 ORION operations directly is not a good idea for the two reasons. First, since the ORION model is an implementation (and informal) model rather than a formal model, a soundness and completeness proof is not possible. We note that notions like default value or shared value are all artifacts of implementation. Second, the operations for changes to the contents of a node can be changed anytime depending upon implementation decisions: the set of schema change operations we allow on the contents of a node of a class hierarchy (i.e., the properties of a class) has been determined from the intuition about application requirements and as such, this set may grow or shrink. For example, by adding the notion of *class variables* and *class methods*, we create a need to define another set of operations for manipulating the contents of class variables and class methods. Also if we decide to augment the ORION model with *integrity constraints* (e.g., salary $>$ \$ 0) we need additional operations for changing the integrity constraints of nodes. Therefore, showing the completeness and soundness of all ORION schema change operations appears to be neither feasible or meaningful.

Our approach to showing completeness and soundness is based on a simple formal model [KK86], 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 hierarchy with a *multiple inheritance* mechanism. The abstract model is based on a

single-rooted directed acyclic graph (DAG) corresponding to a class hierarchy. 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 conflict resolution operator*, $\otimes$, over property values which corresponds to name conflict resolution for multiple inheritance in ORION. Using the $\otimes$ operator, we define a PIG to be a pair (V, E), where V is a set of labeled nodes and E is 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 $\otimes$ 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 $\otimes$ operator. We represent an operation in our model as a mapping from the set of all PIGs to itself.

In the formal model, we define 9 operations for manipulating PIGs. They are defined functionally on the data structure of PIGs. The three operations deal with properties of nodes of PIGs and are similar to ORION operations for adding an instance variable (or method), dropping an instance variable (or method), changing the contents of an instance variable (or method). The rest six PIG operations, each of which is a counterpart of ORION DAG operations (2.1, 2.2, 2.3, 3.1, 3.2, and 3.3), operate on DAG structure of PIG.

From the PIG data structures and operations, we can extract important properties which are essential in justifying the semantics of ORION schema change operations.

We below show how we use this model to characterize the power of the ORION schema change framework. We can prove that every legal PIG is achievable using a set of 9 operations (i.e., completeness). We can also prove

that there is only one complete minimal subset of the PIG operation set (i.e., minimality). The complete minimal subset has 4 operations. We also show how to implement operations corresponding to the remaining ORION operations in terms of these 4 basic operations. Furthermore, we show that the basic set of operations cannot generate a DAG that violates the syntactic rules which characterize a PIG (i.e., soundness).

2.2.2 Basic Concepts of PIG

Definition 2.1: (tentative definition) A PIG G is an edge labeled, node-labeled, single-rooted, directed acyclic graph which is a pair (V, E) where V is a set of labeled nodes and E is a set of labeled edges.

The definition of PIG will be elaborated at the end of this section.

Definition 2.2: We associate a set of properties, PSET(a) = $\{p_1, p_2, ..., p_n\}$ with each node a in V. Each p in PSET(a) is a pair (n,v) where n and v represent property name and property value.

Notation: We shall use p[n] and p[v] to represent the property name and the property value of the property p respectively.

Definition 2.3: The root node r of a PIG G is called *OBJECT*. The root node cannot be deleted or renamed, PSET($OBJECT$) = $\emptyset$.

Example 2.1: Suppose we have four classes: VEHICLE, MOTOR-VEHICLE, WATER-VEHICLE, and SUBMARINE. Let VEHICLE have locally defined instance variables: Id (domain: number), Weight (domain: number), and Owner (domain: string). Let VEHICLE be a superclass of MOTOR-VEHICLE and WATER-VEHICLE, and in turn, MOTOR-VEHICLE and WATER-VEHICLE be the first and second superclasses of SUBMARINE. Finally, let MOTOR-VEHICLE have a locally defined instance variable Size (domain: cubic-inches) and WATER-VEHICLE have a locally defined instance variable Size (domain: number-of-people).

PSET(VEHICLE) = { (Id, number), (Weight, number), (Owner, string) }. PSET(MOTOR-VEHICLE) = { (Id, number), (Weight, number), (Owner, string), (Size, cubic-inches) }. PSET(WATER-VEHICLE) = { (Id, number), (Weight, number), (Owner, string), (Size, number-of-people }. PSET(SUBMARINE) = { (Id, number), (Weight, number), (Owner, string), (Size, cubic-inches) }. □

Definition 2.4: An edge e in E of G is a 3-tuple ⟨ parentnode, node, edge-number ⟩ where the edge-number n is a number indicating that the parentnode is the nth parent of the node.

Intuitively, a PIG is an ORION class hierarchy of which the nodes are filled with properties (i.e., we interpret instance variable and methods as properties). Since we shall not consider the operations for changing the contents (such as instance variables, methods, default value) of nodes in a class hierarchy, a PIG is a sufficient data structure for examining the properties of ORION DAG operations.

Definition 2.5: Given a and b in V of a PIG G, b is an immediate descendant of a (denoted $a \Rightarrow b$) if (a, b, n) is in E of G.

Definition 2.6: The function $edge\text{-}label(a, b)$ returns the edge label connecting node a to node b.

Definition 2.7: The function $nth\text{-}parent(a, n)$ returns the node label of the nth parent of a where a has at least n immediate anscestors.

Definition 2.8: Given a and b in V of a PIG G, b is a descendant of a (denoted $a \stackrel{*}{\Rightarrow} b$) if there exists a sequence of edges in E, $(c_1, c_2, edge\text{-}label(c_1, c_2))$, $(c_2, c_3, edge\text{-}label(c_2, c_3))$,..., $(c_{n-1}, c_n, edge\text{-}label(c_{n-1}, c_n))$, $n \geq 2$, where $c_1 = a$ and $c_n = b$.

Observe that if $a \Rightarrow b$ then $a \stackrel{*}{\Rightarrow} b$. This is the case when $n = 2$ in Definition 2.8. Clearly, $\stackrel{*}{\Rightarrow}$ is *transitive*. The definitions of immediate ancestor and ancestor are obvious from the previous four definitions.

Notation: We shall use children(a) and descendants(a) to represent a set of immediate descendants and a set of descendants of a node a respectively. Similarly, we shall also use parents(a) and ancestors(a) for indicating a set of immediate ancestors and a set of ancestors of a.

Definition 2.9: The function $num\text{-}parents(a)$ returns the number of parent nodes (immediate ancestors) of the node a.

We introduce an operator which is useful describing the notions of *name conflict resolution* in ORION schema changes. We assume the definition of the ⊗ operator to be open for simplying the PIG model. Owing to the ⊗ operator, the definitions of PIG operations become more brief. By giving proper semantics to the ⊗ operator, the notion of name conflict resolution in ORION schema changes can be understood within the PIG model. The ⊗ operator satisfies the following three characteristics.

Definition 2.10: The ⊗ operation is a function from VSET × VSET to VSET where VSET is a set of property values (i.e., ⊗: VSET × VSET → VSET). The ⊗ operation satisfies the following characteristics. Let $v1$, $v2$, and $v3$ be property values.

1. (idempotency) $v1 \otimes v1 = v1$
2. (commutativity) $v1 \otimes v2 = v2 \otimes v1$
3. (associativity) $v1 \otimes (v2 \otimes v3) = (v1 \otimes v2) \otimes v3$

Example 2.2: In the previous example, the property value of Size property of SUBMARINE is determined by the operation cubic-inches ⊗ number-of-people. If we honor the superclass ordering as in ORION, the property value of Size property of SUBMARINE is cubic-inches. □

Definition 2.11: Given two property values $v1$ and $v2$, we say $v1 \simeq v2$ if there exists u such that $v1 = u \otimes v2$.

Definition 2.12: Two properties p and q are said to be *equal* if p[n] = q[n] and p[v] = q[v]. Two properties p and q are said to be *similar* if p[n] = q[n] and p[v] ≃ q[v].

Theorem 2.1: $\simeq$ is transitive.

Proof: Let v1,v2, and $v3$ be property values such that $v1 \simeq v2$ and $v2 \simeq v3$. There exists u such that $v1 = u \otimes v2$ and exists u' such that $v2 = u' \otimes v3$. Now we have $v1 = u \otimes (u' \otimes v3)$. By the associativity, $v1 = (u \otimes u') \otimes v3$. Let $w = u \otimes u'$. Then $v1 = w \otimes v3$. Hence $v1 \simeq v3$. □

Theorem 2.2: Given two values $v1$ and $v2$, if $v1 = v2$ then $v1 \simeq v2$.

Proof: Since $v1 = v2$, $v1 = v2 \otimes v2$ by idempotency. Therefore $v1 \simeq v2$. □

Example 2.3: In the previous example, the Size property of SUBMARINE is equal to the Size property of MOTOR-VEHICLE, whereas the Size property of SUBMARINE is similar to the Size property of WATER-VEHICLE, by definition. □

Definition 2.13: Given two nodes a and b in a PIG P, we say PSET(a) $\preceq$ PSET(b) if for each property p in PSET(a), there exist a property q in PSET(b) where p and q are similar.

Now we introduce two rules for property inheritance. Rule 1 is associated with inheritance mechanism and Rule 2 is associated with name conflict resolution. The two rules associate with the full inheritance invariant in the previous section.

Rule 1: If $a \Rightarrow b$ then PSET(a) $\preceq$ PSET(b)

Rule 2: If $a \Rightarrow c$, and $b \Rightarrow c$, and $\exists$ p $\in$ PSET(a), and $\exists$ q $\in$ PSET(b) such that p[n] = q[n], then $\exists$ a property r $\in$ PSET(c) where r is similar to p and r is similar to q.

From Rule 1 and 2, and the fact that if $a \Rightarrow b$ then $a \stackrel{*}{\Rightarrow} b$, the following two rules follow trivially.

Rule 1': If $a \stackrel{*}{\Rightarrow} b$ then PSET(a) $\preceq$ PSET(b).

Rule 2': If $a \stackrel{*}{\Rightarrow} c$, $b \stackrel{*}{\Rightarrow} c$, and $\exists$ p $\in$ PSET(a), $\exists$ q $\in$ PSET(b) such that p[n] = q[n], then $\exists$ a property r $\in$ PSET(c) where r is similar to p and r is similar to q.

Now we strengthen the tentative definition of PIG as follows.

Definition 2.14: A PIG G is an edge labeled, node labeled, single rooted, directed acyclic graph which is a pair (V, E) where V is a set of labeled nodes and E is a set of labeled edges. A PIG G satisfies the two inheritance Rules.

2.2.3 Operations of PIG

Now we define 9 operations for manipulating PIG graphs. The first three operations deal with properties of nodes of PIG graphs and are similar to ORION operations for adding an instance variable (method), dropping an instance variable or a method and changing the contents of an instance variable (method). The remaining six PIG operations, each of which is a counterpart of ORION DAG operations, operate on the PIG DAG. We again stress that we shall ignore most operations for changing the contents of the nodes in ORION class hierarchy, except operations 1.1.1, 1.2.1, 1.1.2, 1.2.2, 1.1.4 and 1.2.4.

- PIG (Op1): Add a new property: This corresponds to ORION schema change operation 1.1.1 or 1.2.1 "Add a new instance variable or a new method."
- PIG (Op2): Drop an existing property: This corresponds to ORION schema change operation 1.1.2 or 1.2.2 "Drop an existing instance variable or an existing method."
- PIG (Op3): Change the contents of a property: This corresponds to ORION schema change operation 1.1.4 or 1.2.4 "Change the domain of an existing instance variable or the code of an existing method."
- PIG (Op4): Add an edge: This corresponds to ORION schema change operation 2.1 "Make a class S a superclass of a class C."

- PIG (Op5): Drop an edge: This corresponds to ORION schema change operation 2.2 "Remove a class S from the superclass list of a class C."
- PIG (Op6): Change the order of two incoming edges of a node: This corresponds to ORION schema change operation 2.3 "Change the order of superclasses of a class C."
- PIG (Op7): Add a new node: This corresponds to ORION schema change operation 3.1 "Add a new class."
- PIG (Op8): Drop an existing node: This corresponds to ORION schema change operation 3.2 "Delete an existing class."
- PIG (Op9): Rename the label of a node: This corresponds to ORION schema change operation 3.3 "Change the name of a class."

We present below the semantics and the functional definition of PIG operations.

(Op1) Add a new property p: The node a and its descendants will inherit the new property p (line 2). If there is a name conflict with an existing property (say q) they have, the property value of q, q[v] is determined by the $\otimes$ operator (line 3). By defining $\otimes$ to be the ORION name conflict resolution mechanism to the $\otimes$ operator, Op1 can function as ORION (1.1.1) or ORION (1.2.1). We note that this operation obeys inheritance rules 1 and 2.

```
function Op1(G, a, p)
/* G is a PIG graph (V,E) */
/* a is a node in V */
/* p is a property to add */
begin
    a-has-p = no;
    foreach q ∈ PSET(a): do
        if (q[n] = p[n])
            then begin
            q[v] = q[v] ⊗ p[v];
            a-has-p = yes;
            end;
    if (a-has-p = no)
        then PSET(a) ← PSET(a) ∪ {p};
    foreach a' ∈ children(a): do Op1(G, a', p);
    return G;
end
```


(Op2) Delete a property p from a node a of a PIG P: The property p is dropped from the node a and those descendents of a that inherited p. If however any ancestor of a has the property p or a property similar to p, p cannot be dropped from a since that would violate the inheritance rules 1 and 2. Instead, p[v] is determined by the $\otimes$ operator.

```
function Op2(G, a, p)
/* G is a PIG graph (V,E) */
/* a is a node in V */
/* p is a property */
begin
    ancestor-has-p = no;
    foreach a' ∈ imm-ans(a): do
        foreach q ∈ PSET(a'): do
```

if (q[n] = p[n])

then begin

p[v] ← p[v] ⊗ q[v];

ancestor-has-p = yes;

end;

if (ancestor-has-p = no)

then PSET(a) ← PSET(a) − {p};

foreach a'' ∈ children(a): *do* Op2(G, a'', p);

return G;

end

(Op3) Change the value of a property p of a node a of a PIG P: The property value of p is changed in the node a. The change is propagated to those descendents of a that inherited p, unless the property value has been locally changed into a value which is different from the original value of p within the descendents.

function Op3(G, a, p, old-value, new-value)

/* G is a PIG graph (V,E) */

/* a is a node in V */

/* p is a property of a */

/* old-value is the old value of p */

/* new-value is the new value for p */

begin

if ∃ q ∈ PSET(a): (p[n] = q[n]) ∧ (q[v]= old-value) *then*

begin

q[v] ← new-value;

foreach a' ∈ children(a): *do*

Op3(G, a', p, old-value, new-value);

end

return G;

end

(Op4) Create a new edge: Since G is an acyclic connected graph, if the graph that results from adding a new edge between nodes a and b is cyclic or the edge is already in G, the addition request is rejected. Otherwise a becomes the n+1 th parent node of b if b has currently n parent nodes where n > 0. The properties of a are inherited to b by the operation Op1.

function Op4(G,a,b)

/* G is a PIG (V,E) */

/* a and b are nodes in V */

begin

if (∄ a sequence of edges (c_1,c_2,edge-label(c_1,c_2)),

(c_2,c_3,edge-label(c_2,c_3)),...,(c_{n-1},c_n,edge-label(c_{n-1},c_n)):

($c_1 = b$) ∧ ($c_n = a$)) ∧ ((a,b,edge-label(a,b)) ∉ E) *then*

begin

E ← E ∪ {(a,b,(count-of-parents(b) + 1))};

foreach p ∈ PSET(a): *do* Op1(G,b,p);

end;

return G;

end

(Op5) Delete an existing edge: Suppose the edge to be deleted is the edge between the nodes a and b. The edge is deleted from the edge set and the inherited properties from the node b are removed from a with Op2. The rest of incoming edges of b except the deleted edge should be re-labeled such that if the edge label is greater than n, the edge label is decreased by 1, otherwise no relabeling is necessary. Since G is an acyclic connected graph, if the graph resulting from deleting an edge (a,b,edge-label(a,b)) is disconnected, b is made an immediate child node of the immediate parent nodes of a. This is governed by Op4.

function Op5(G,a,b)

/* G is a PIG (V,E) */

/* a and b are nodes in V */

begin

E ← E − {(a,b,edge-label(a,b))};

foreach p ∈ PSET(a): *do* Op2(G,b,p);

foreach e ∈ parents(b): *do*

if (edge-label(e,b) > n) *then*

begin

E ← E − {(e,b,edge-label(e,b))};

E ← E ∪ {(e,b,(edge-label(e,b) − 1))};

end;

if (∄ c: (c, b,edge-label(c,b)) ∈ E)

then if a = OBJECT *then* V ← V − {b}

else foreach d ∈ parents(a): *do* Op4(G,d,b);

return G;

end

(Op6) Exchange the order of two incoming edges of a node: This operation causes re-evaluation of the inheritance of properties in the node receiving the edges. The two incoming edges should exist in the edge set before exchanging the labels. Finally, this operation exchanges the labels of two incoming edges of a PIG.

function Op6(G,a,b,c)

/* G is a PIG (V,E) */

/* a,b,c are nodes in V */

begin

if ((a,b,edge-label(a,b)) ∈ E) ∧ ((a,c,edge-label(a,c)) ∈ E) *then*

begin

temp1 ← edge-label(a,b);

temp2 ← edge-label(a,c);

foreach i *from* temp1 *to* temp2: *do*

foreach p ∈ PSET(nth-parent(c,i)): *do*

Op2(G,nth-parent(c,i),p);

foreach p ∈ PSET(nth-parent(c,temp2)): *do*

Op1(G,nth-parent(c,temp2),p);

foreach i *from* temp1 + 1 *to* temp2 − 1: *do*

foreach p ∈ PSET(nth-parent(c,i)): *do*

Op1(G,nth-parent(c,i),p);

foreach p ∈ PSET(nth-parent(c,temp1)): *do*

Op1(G,nth-parent(c,temp2),p);

E ← E − {(a,b,edge-label(a,b)),(a,c,edge-label(a,c))};

E ← E ∪ {(a,b,temp2),(a,c,temp1)};

end;

return G;

end

(Op7) Create a new node: A new node b is created as a descendant of an existing node a. The node a must exist and the node b must not exist before performing this operation. A new edge (a, b,1) is created. The properties of a are inherited by b. This is governed by Op1.

function Op7(G,a,b)

/* G is a PIG (V,E) */

/* a is a node in V */

/* b is a node to be created */

begin

if (a ∈ V) ∧ (b ∉ V) *then*

begin

V ← V ∪ {a};

E ← E ∪ {(a,b,1)};

foreach p ∈ PSET(a): *do* Op1(G,a,p);

end;

return G;

end

(Op8) Delete an existing node: A node a is dropped from the node set and the incoming and outgoing edges of the node are all dropped from the edge set. All properties of a are removed from a's subclasses by the operation Op2. Every immediate descendant of the node now becomes an immediate descendant of immediate ancestors of the node. This is governed by Op4.

function Op8(G,a)

/* G is a PIG (V,E) */

/* a is a node in V */

begin

if (a ∈ V) *then*

begin

V ← V − {a};

temp-set1 ← parents(a);

temp-set2 ← children(a);

foreach b ∈ parents(a): *do*

E ← E − {(b,a,edge-label(b,a))};

foreach c ∈ children(a): *do*

begin

E ← E − {(a,c,edge-label(a,c))};

foreach p ∈ PSET(a): *do* Op2(G,a,p);

end;

```
        foreach e ∈ temp-set1: do
            foreach f ∈ temp-set2: do Op4(G,e,f);
        end
    return G;
end
```

(Op9) Rename a node: The label of a node is updated in the node set. All edges involving the node should be updated in the edge set. There are no effect on properties of nodes or the DAG structure.

```
function Op9(G,a,b)
/* G is a PIG (V,E) */
/* a is a node in V */
/* b is a new label for a */
begin
  if (a ∈ V) ∧ (b ∉ V) then
    begin
      foreach c ∈ children(a): do
        begin
        temp1 ← node-label(a,c);
        E ← E − {(a,c,node-label(a,c))};
        E ← E ∪ {(b,c,temp1)};
        end;
      foreach d ∈ parents(a): do
        begin
```



```
        temp2 ← node-label(d,a);
        E ← E − {(d,a,node-label(d,a))};
        E ← E ∪ {(d,b,temp2)};
        end;
      V ← V − {a};
      V ← V ∪ {b};
    end;
  return G;
end
```

2.2.4 Soundness and Completeness of PIG

The following theorem states that if G is a PIG, G' which is constructed from G by applying an operation in the PIG operation sets {Op1,..., Op9}, is also a PIG.

Theorem 2.3: [Soundness] The class of PIG graphs is closed under the PIG operations Op1–Op9.

Proof: The proof for each operator is presented below. Let G be a PIG.

- **(Op1) Add a new property:** Since Op1 does not restructure the graph structure of G, the G resulting from applying Op1 is a single–rooted, node–labeled, edge–labeled and connected DAG. By the definition of Op1, every descendant of a node, to which a new property p is added, has a property which is either *equal* or *similar* to p. This preserves the inheritance rule 1 and 2. Therefore the resulting G is a PIG.
- **(Op2) Delete a property:** Similar to the above.
- **(Op3) Change the value of a property:** Similar to the above.

- **(Op4) Create a new edge:** Suppose the new edge is an edge between nodes *a* and *b*. By the definition of Op4, this operation eliminates cases of cyclic graphs or duplicated edges. Thus, the resulting DAG structure is a single–rooted, node–labeled, edge–labeled and connected DAG. For each property in the node *a*, the same property is added to the node *b*. This is governed by Op1. As we already showed the closure property of Op1, the inheritance rule 1 and 2 are preserved in the resulting G. Therefore, the resulting graph G is a PIG.

- **(Op5) Delete an existing edge:** Suppose the edge to be deleted is an edge between the nodes a and b. As shown in the definition of Op5, special care is exercised in case of articulation edge deletion. The structural property of PIG is guaranteed by adding necessary edges where appropriate. This is governed by Op4. The labels of incoming edges of the node receiving the deleted edge are updated where appropriate. All properties which are inherited from a are removed from b by Op2. The closure property of Op4 and Op2 have been already shown. Therefore, the resulting graph is a PIG.

- **(Op6) Exchange the order of two incoming edges of a node:** This operation simply exchanges the labels of two incoming edges of a node. No change is made to the DAG structure of G. As shown in the definition of Op6, updates to properties are governed by Op1 and Op2. We have proved the closure property of Op1 and Op2. Therefore, the resulting graph is a PIG.

- **(Op7) Create a new node:** This operation is allowed only if there are no name conflicts about the node to be created. A new node is created and an incoming edge is created. The new node inherits all properties of the parent node by applying Op1 repeatedly. Clearly adding a node to a PIG results in another PIG.

- **(Op8) Delete an existing node:** As shown in the definition of Op6, special care is exercised in case of articulation node deletion. The structural property of PIG is guaranteed by adding necessary edges where appropriate. Property addition and deletion are governed by Op1 and Op2. Therefore, the resulting graph is a PIG.

- **(Op9) Rename a node:** Node renaming does not affect to the DAG structure of G. Name conflicts are eliminated by rejecting the renaming request. Also node renaming does not make any effect to the contents of nodes in G. The resulting graph is a PIG. □

Lemma 2.4: For any PIG graph, there is a finite sequence of Op8 that reduces the PIG graph to a PIG with one root node.

Proof: Consider the reduction process. Let G be a PIG having a finite number of nodes.

1 For each leaf node of a PIG G, apply Op8.

2 Let the resulting PIG be G'.

3 If G' has other nodes besides the root node, set G to G' and go to step 1.

Clearly we can get a PIG with only root node by applying Op8 repeatedly. □

Lemma 2.5: For any PIG graph, there is a finite sequence of {Op1, Op4, Op7} that generates it from a PIG with only a root node.

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

1 For each node of a PIG G, in breadth–first order, apply Op7 to G'. Set G' to the resulting graph.

2 For each node of a PIG G, in breadth–first order, if the node has more than one incoming edge, apply Op4 to G' for each of the extra incoming edges. Set G' to the resulting graph.

3 For each node of a PIG G, in breadth–first order, apply Op1 to G' for each property of the node.

G is equivalent to G' because G' has all the nodes with the same contents and all the edges in G, and no more nodes and edges. □

Theorem 2.6: Given two arbitrary PIG graphs G1 and G2, there is a finite sequence F of {Op1, Op4, Op7, Op8}, such that F(G1) = G2.

Proof: We can get G2 by (1) reducing G1 to a PIG with only root node by the reduction process in lemma 2.4 using Op 8 only and (2) building G2 from the root node by the construction process in lemma 2.5 using Op1, Op4, and Op7 only. □

Completeness is obvious from theorem 2.6 because {Op1, Op4, Op7, Op8} ⊂ {Op1, Op2, Op3, Op4, Op5, Op6, Op7, Op8, Op9}.

Theorem 2.7: [Completeness] Given two arbitrary PIG graphs G1 and G2, there is a finite sequence of PIG operations Op1–Op9, F such that F(G1) = G2.

Now we shall show that there is only one minimal complete subset of Op1–Op9, the set {Op1, Op4, Op7, Op8}. To prove this, it suffices to show that each operation in the set cannot be achievable using the others in Op1–Op9, and that Op2, Op5, Op6 and Op9 can be achievable using the operations in the set.

Lemma 2.8: Op1 is not achievable using Op2–Op9.

Proof: From the definitions of operations Op1–Op9, it is easy to see that only Op1 can add a property to a node in PIG. □

Lemma 2.9: Op2 is achievable using Op1, Op4, Op7, and Op8.

Proof: Given an arbitrary PIG G, consider another PIG G' after applying Op2 to G. The transformation from G to G' can be simulated from the operations in {Op1, Op4, Op7, Op8} by theorem 2.6. □

Lemma 2.10: Op3 is achievable using Op1, Op4, Op7, and Op8.

Proof: Similar to the proof of lemma 2.9. □

Lemma 2.11: Op4 is not achievable using Op1–Op3, Op5–Op9.

Proof: Since only Op4 (add an edge) can create a graph structure, it is sufficient to show that any finite combination of Op1–Op3, Op5–Op9 always generates a tree structure. First, it is clear that only trees are generated by applying Op5 (add a new node) repeatedly because Op5 creates a node with only one incoming edge by its definition. Second, from the definitions, clearly Op1, Op2 and Op3 do not have any effect on changing tree structures. Third, consider Op6 (change the order of two incoming edges of a node) and Op9 (rename the label of a node). Op6 is of no use in tree structures because each node has only one incoming edge (i.e., unique parent). Op9 is irrelevant because renaming a node cannot change the tree structure. In tree structure, every edge is an articulation edge, every node (except the root node) has a unique parent and every non–leaf node is an articulation node. Therefore, the structure resulting from applying Op6 or Op9 to a tree structure is tree. Fourth, consider Op5 (drop an edge) and Op8 (drop an existing edge). Let a, b, c be nodes such that a is a parent node of b and b is a parent node c. A new edge is created from a to c after deleting the edge between b and c according to the definition of Op5. Similarly a new edge is created from a to c after deleting b according to the definition of Op8. Therefore, the structure resulting from applying Op5 or Op8 to a tree structure is tree. In any cases, nodes with more than one incoming edge (i.e., DAG) cannot be made without Op4. We can conclude it because tree structures are closed under Op1–Op3, Op5–Op9. Therefore Op4 cannot be simulated by Op1–Op3, Op5–Op9. □

Lemma 2.12: Op5 is achievable using Op1, Op4, Op7, and Op8.

Proof: Similar to the proof of lemma 2.9. □

Lemma 2.13: Op6 is achievable using Op1, Op4, Op7, and Op8.

Proof: Similar to the proof of lemma 2.9. □

Lemma 2.14: Op7 is not achievable using Op1–Op6, Op8, and Op9.

Proof: Given an arbitrary PIG G, consider another PIG G' after applying Op7. Since a new node is added, the node set of G' is bigger than that of G. Among Op1–Op9, as shown in the definitions of each operation, only Op7 can add a new node to the node set. Therefore Op7 cannot be simulated by Op1–Op6, Op8, and Op9. □

Lemma 2.15: Op8 is not achievable using Op1–Op7, and Op9.

Proof: Given an arbitrary PIG G, consider another PIG G' after applying Op8. Since an existing node is deleted, the node set of G' is smaller than that of G. Among Op1–Op9, as shown in the definitions of each operation, only Op8 can delete an existing node from the node set. Therefore Op8 cannot be simulated by Op1–Op7 and Op9. □

Lemma 2.16: Op9 is achievable using Op1, Op4, Op7, and Op8.

Proof: Similar to the proof of lemma 2.9. □

As a conclusion, we have established the following.

Theorem 2.17: [Minimality] The only minimal complete subset of the PIG operations Op1–Op9 is {Op1, Op4, Op7, Op8}.

2.3 Related Works

This section is a summary of others' works which are related to schema evolution.

2.3.1 Data Base Restructuring

Schema evolution is more or less related to the area called *Data Base Restructuring* which was one of popular issues during 1970's. Several research groups have addressed the problem of database restructuring in conventional data models, such as hierarchical and network data models. The major issue of database restructuring is to reorganize hierarchical or network data in order

to get good performance in query processing. Most approaches of database restructuring assume off-line database reorganization. Furthermore, dynamic schema changes in the run-rime are not required.

Schema evolution in object-oriented databases and database restructuring in conventional databases might be considered as a similar problem in the sense that the database and its schema are reorganized. However, as we will show, the motivation and complexity are significantly different because applications of conventional databases (business type data processing) do not evolve as dynamically as applications (CAD, AI, OIS) of object-oriented databases. Here is some previous work on database restructuring.

The IBM Research Laboratory developed an experimental prototype database restructuring system EXPRESS for hierarchical databases [Shu77]. Two nonprocedural languages are provided in EXPRESS: DEFINE [Hou75] for data description and CONVERT [SHL75] for data restructuring. CONVERT provides three operators for Form restructuring (Form is a table form of hierarchical databases, that is, a relation-valued relation): SLICE is to transform hierarchical data into a flat file, SELECT is to select the part of a Form which meets the selection criteria, and GRAFT is for joining two hierarchical data records to form a larger hierarchical data record. Besides the three operators, CONVERT supports several useful operations for form restructuring, such as SORT and built-in aggregate functions. The University of Michigan [SDF77, DF75, FS76] and The University of Maryland [ST82] both implemented systems for restructuring network databases. Relational database systems allow only a few types of schema changes: SQL/DS allows only creation of relations, deletion of relations, and addition of new columns [IBM81].

After restructuring a database, the next important issue is to convert existing application programs which were running on the previous database in order to comply with the new database. This is area is called *database program conversion.* Su, et al., at The University of Florida [Su76, SL77], Shneiderman, et al., at The University of Michigan [Shn78, ST82], and Ramirez,

et al., at The University of Pennsylvania [RRP74, Smit71] have considered the problem of database program conversion methodology on hierarchical and network databases.

2.3.2 Penny and Stein

Penny and Stein [PS87] studied class modification in the GEMSTONE system. GEMSTONE is an object-oriented database system that was designed and implemented at Servio Logic Development Corporation. GEMSTONE provides the user with an object model identical to that of SmallTalk-80 [GR83]. As such, the usual features of object-oriented data models are available in the GEMSTONE data model: objects, messages, and classes are supported, and classes are organized into a hierarchy that provides single inheritance of properties.

The origin of schema evolution in GEMSTONE [PS87] is the schema evolution framework of ORION [BKKK86, BKKK87]. Since GEMSTONE assumes class structures to be trees (single inheritance) while ORION supports directed acyclic graphs (DAGs) of class hierarchies, there are fewer class modifying operations in GEMSTONE than in ORION. The class modification operations of GEMSTONE are: (1) renaming a named instance variable, (2) making a class indexable, (3) adding a named instance variable, (4) making an indexable class non-indexable, (5) removing a named instance variable, (6) modifying the constraint on a variable, (7) removing a class, and (8) adding a class. The semantics of these operations can be easily inferred from the ORION schema evolution operations in section 2.2.

2.3.3 Fishman, et al.

IRIS [Fish87] allows limited changes to a class hierarchy. IRIS is a research prototype of object-oriented database system at Hewlett-Packard Laboratories. IRIS has mixed flavors of relational, functional, and object-oriented database systems.

IRIS is composed of two layers: object manager and storage manager. The IRIS object manager serves as the query processor of the DBMS. It translates IRIS queries and operations into an internal relational algebra format, which is then interpreted. In order to process IRIS queries properly, the IRIS object manager must maintain metadata such as class definitions, class hierarchies, rules, and authorization. The IRIS storage manager is a conventional relational storage system which is very similar to the Relational Storage Subsystem (RSS) in System/R [Astr76]. It is responsible for dynamic creation and deletion of relations, transactions, concurrency control, logging and recovery, archiving, indexing and buffer management.

As we mentioned, IRIS supports only a few primitive schema changes in a class hierarchy. New classes may be created, existing classes may be deleted, and objects may be dynamically included or excluded as instances of classes. In the current implementation, a class may be deleted only if it has no subclasses or instances. New subclass/superclass relationships between existing classes may not be created and existing subclass/superclass relationships may not be deleted.

Chapter 3
Schema Manager and Graphics Schema Editor

3.1 Schema Manager of the ORION System

The schema evolution framework in the previous chapter has been fully implemented within the ORION project at MCC. In this section we briefly indicate the key points of the ORION schema manager implementation. The schema manager of the ORION system was implemented by the MCC ODBS team, not by the author. Portions of this section are due to [LEE86, BKKK87]. However the graphics schema editor running on top of the ORION system was designed and implemented by the author.

3.1.1 System-Defined Classes for the Schemas

The schema manager maintain several system-defined classes as major data structures for schema definition. Three major system-defined classes are CLASS, INSTANCEVARIABLE, and METHOD class. Figure 6 shows system-defined classes in ORION.

The definition of a class is stored as an instance of CLASS class, the definition of an instance variable is stored as an instance of INSTANCEVARIABLE class, and the definition of a method is stored as an instance of METHOD class. These are analogous to system catalogs in conventional database systems. Therefore, the schema manager is more or less similar to catalog managers in conventional database systems. When a schema change occurs, the instances of the system-defined classes are updated. For example, if a class is deleted, an instance of CLASS for the deleted class should be deleted and instances of INSTANCEVARIABLE (METHOD) for the instance variables (methods) of the deleted class should be deleted too.

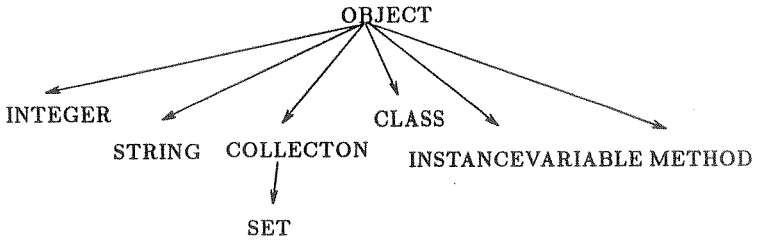


Figure 6: System-Defined Classes in ORION (from [Ban87])

3.1.2 Schema Evolution without 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. [BKKK87; 319]

Storage Format

To support schema evolution without database reorganization, the ORION team designed the storage format for disk-resident

objects as shown in Figure 7. 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 (v_i,o_i) pairs, one for each instance variable for which the object has an explicitly specified value. In each (v_i,o_i) pair, v_i is the identifier of an instance variable, and o_i 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 INSTANCEVARIABLE, 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 INSTANCEVARIABLE as we mentioned in the previous section.

A composite object consists of many instances. Unlike the complex object implementation described in [LP83], ORION does 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. [BKKK87; 319]

3.2 Graphics Schema Editor PSYCHO

PSYCHO is a graphical schema manipulation language which is being used in

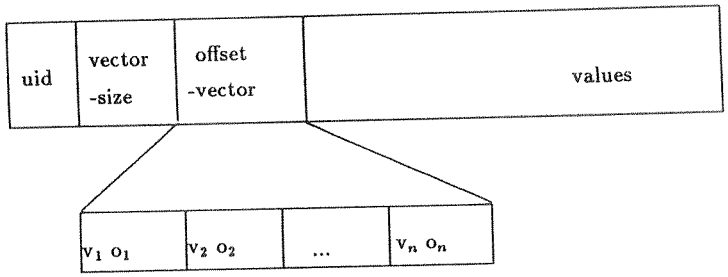


Figure 7: Storage Format of the ORION system (from [BKKK87])

ORION. In this section, we describe the use of PSYCHO and its implementation.

3.2.1 System Structure of PSYCHO/ORION Environment

PSYCHO and ORION have been implemented within the ODBS project at MCC. ORION is written in CommonLisp on a Symbolics 3640. PSYCHO is implemented with Flavors and ZetaLisp on a Symbolics 3640. Figure 8 shows a diagram of the PSYCHO/ORION environment. The user can use the ORION system directly or indirectly through the PSYCHO system. PSYCHO communicates with the schema manager and the transaction manager of ORION.

In the PSYCHO/ORION environment, class hierarchies are represented as DAGs (directed acyclic graphs) on the screen, and the user can manipulate DAGs directly using a mouse and pop-up menus. A schema manipulation session of PSYCHO/ORION environment goes roughly as follows. First, the user chooses "schema-load" option in the command menu, and the system draws a DAG representing the database schema. The user enters schema change commands by using the mouse and pop-up menus to manipulate the DAG. *Transaction Mode* is used to group several schema change operations into a single atomic action. A session is terminated by clicking "Quit" in the command menu.

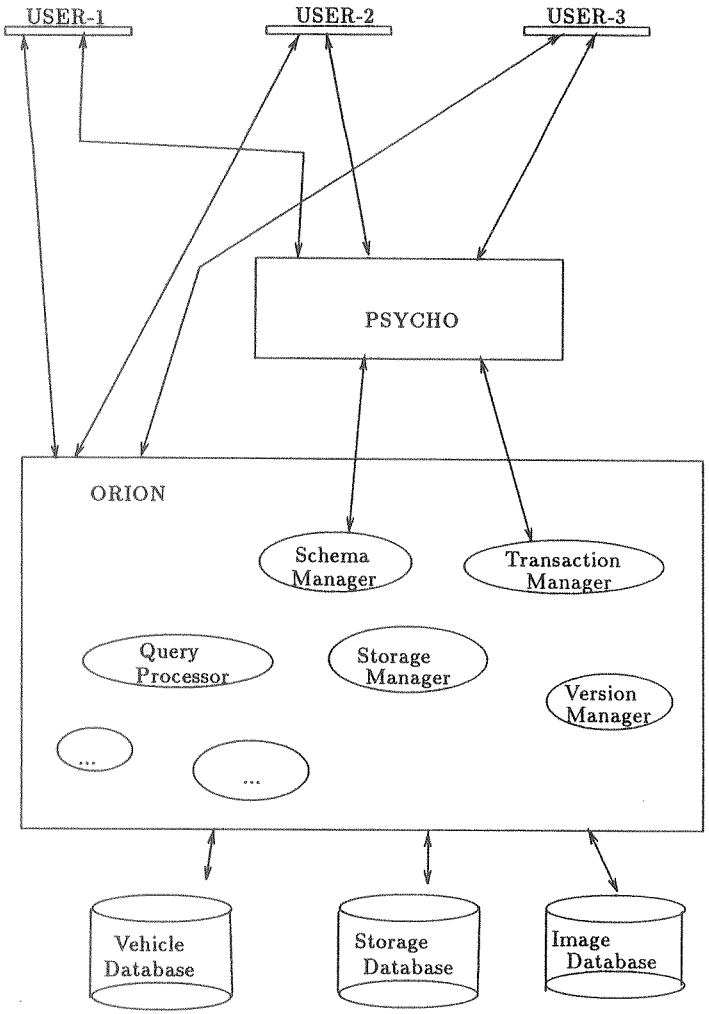


Figure 8: System diagram of PSYCHO/ORION environment

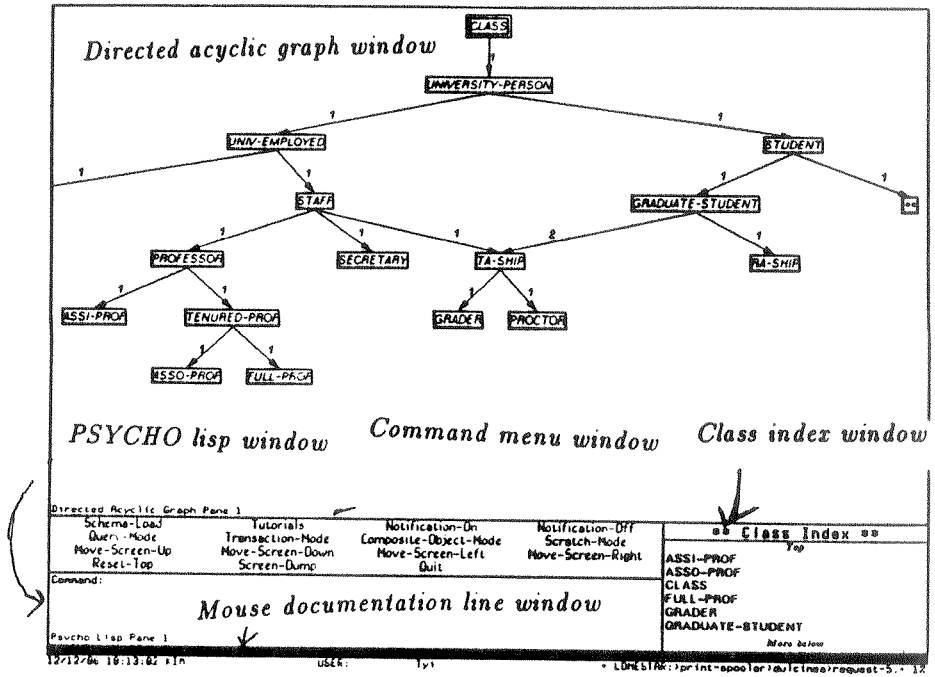


Figure 9: PSYCHO window

We show several examples of PSYCHO sessions in following sections. These examples sessions of PSYCHO demonstrate the power of graphics to provide a simple method stating schema changes that would be complicated to express in the text-oriented command language of the ORION schema manager.

3.2.2 Overall Structure of PSYCHO

PSYCHO provides various facilities to help users in posing schema change commands. In this section, we briefly introduce the structure of PSYCHO system. Figure 9 shows a PSYCHO window which consists of 5 subwindows: a directed acyclic graph window, a PSYCHO LISP window, a command menu window, a class index window and a mouse documentation line window. Besides those subwindows, a method code editor window is provided for editing or modifying code of a method during the schema change session.

Directed Acyclic Graph Window

The directed acyclic graph window is for the graphical manipulation of

the DAG representation of a database schema. The schema of the "University Person" database is shown in Figure 9: the class UNIVERSITY PERSON has two subclasses, UNIV-EMPLOYED and STUDENT, and in turn UNIV-EMPLOYED has two superclasses GRADUATE-STUDENT and UNDERGRADUATE-STUDENT, and so on. We shall use the "University Person" database for illustrating facilities of PSYCHO through sample sessions in section 3.2.3.

PSYCHO Lisp Window

During a schema change session, the user may have to issue ORION commands or queries directly on the LISP top level. The PSYCHO LISP window is a Symbolics LISP window whose size is altered to fit inside PSYCHO system. The user can enter any LISP expressions or any ORION commands in this window.

Command Menu Window

The command menu consists of 15 commands, which govern schema navigation and mode changes. The 15 commands are solely for PSYCHO, and are not transmitted to ORION.

- *Schema-Load:* PSYCHO reads the database schema definition from ORION into internal data structures, and PSYCHO draws a DAG in accordance with the schema definition.
- *Tutorials:* PSYCHO provides the user with a brief introduction to PSYCHO, some example sessions of schema change, and explains the functions of the mouse buttons, pop-up menus and commands.
- *Notification-On:* Various types of system error and warning messages are provided from ORION or PSYCHO. By choosing this command, the user can receive error and warning messages with a beeping sound.
- *Notification-Off:* Error and warning messages are suppressed.

Including the schema change mode (default mode), PSYCHO supports 5 different modes. Three of them are implemented and the remaining two will be implemented in the future. A number of pop-up menus are provided in accordance with the modes. The following 4 commands are for changing the modes of PSYCHO.

- *Query-Mode:* This mode is not implemented currently. We are planning to support a graphical query language facility within PSYCHO.
- *Transaction-Mode:* In this mode, the user can pose a series of schema change operations as an atomic action (i.e., schema change transaction). PSYCHO requests transaction service to PSYCHO by sending a message "Begin Transaction" to ORION. In the middle of a transaction, the user can abort. Then ORION undoes the operations performed between abort point and begin-transaction point. If the user commits a transaction, PSYCHO sends a message "End Transaction." We shall discuss this mode in section 3.2.3.
- *Composite-Object-Mode:* This mode is not implemented.
- *Sketch-Mode:* In this mode, PSYCHO does not communicate with ORION. The user can manipulate the database schema experimentally. We shall discuss this mode in section 3.2.3.
- *Move-Screen-Up (-Down, -Left, -Right):* The DAG window is scrolled up, down, left, or right, respectively. These commands are used for a DAG that is too large to fit on the screen.
- *Reset-Top:* The window is adjusted to show the top of the DAG (i.e., the root appears at the top of the window).
- *Screen-Dump:* Prints a hard copy of the screen on a laser printer.
- *Quit:* Terminates a session.

Class Index Window

The small window located in the left bottom of the screen is the class index window. The class index window shows a list of class names in alphabetical order. The user can scroll the class index window up and down by clicking the *top* and *bottom* labels in the screen respectively. If the user clicks on a particular class name, the system draws the sub-DAG whose root is the chosen class. We shall discuss this window in section 3.2.3.

Mouse Documentation Line Window

The mouse documentation line window contains information about what different mouse clicks mean. As the user moves the mouse across different mouse-sensitive items or areas of the screen, the mouse documentation line shows the corresponding documentation to reflect the changing commands available.

Method Code Editor

In ORION, a method is a CommonLisp form. As such, the method definition process is similar to a LISP programming session. Therefore the user needs an Zmacs[1]-like editor window. In the method code editor, the user can test a new method, modify the method and save the definition of the method into ORION.

3.2.3 PSYCHO Facilities

In this section we will illustrate PSYCHO facilities by showing example schema change sessions on the "University Person" database. Before we proceed, we have to clarify the mouse function in PSYCHO. The Symbolics mouse has three buttons. We use only the left and right button, not the middle button. The left button is used for clicking mouse-sensitive items such as nodes in a DAG, commands in the command menu, and class names in the class index window. A mouse sensitive item has its associated action which is supposed to be performed after being clicked by the left button. The right button is used for invoking an pop-up mode which is designed for a particular mode.

[1] Zmacs is Symbolics' version of emacs

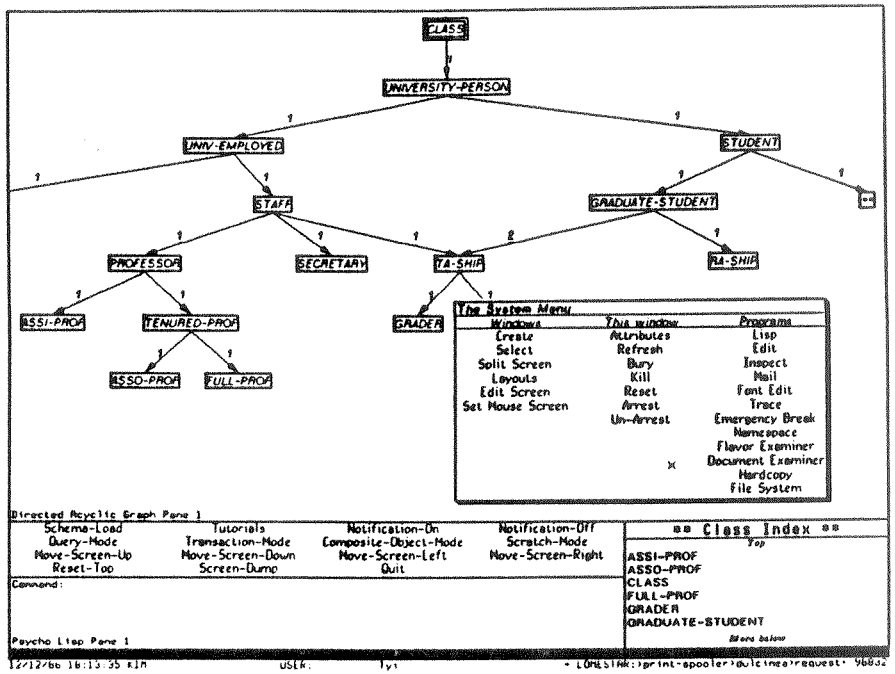


Figure 10: Symbolics system menu

The Symbolics supports a system command menu. We retain the system command menu because the user may want to do various things, such as inspecting directories and files, during a schema change session. The system menu can be invoked by clicking the right button twice as shown in Figure 10.

We also intentionally provide some redundancy in the contents of pop-menus and the command menu, (i.e., some commands are supported in different menus). That is purely for the user's convenience. The pop-up menu in Figure 11 is invoked when the user clicks the right button on an area that is not a mouse-sensitive item. The contents of the pop-up menu is exactly same as the contents of the command menu window.

Schema Manipulation

All schema change operations in the taxonomy in section 2.3 are embedded in the pop-up menu in Figure 12. The pop-up menu (from now on, we call this menu the "basic operation menu") is invoked by clicking the right button upon

nodes of DAG which are mouse sensitive items. The first seven items in the basic operation pop-up menu corresponds to (3.1), (3.2), (3.3), (1), (2.1), (2.2), and (2.3) in the taxonomy of ORION schema change operations, respectively. In this section we will explain the behavior of these seven items by showing examples. The other three items are for schema browsing and user navigation, and are covered in the next section.

Suppose the user clicks the following items on a node of a DAG.

- *Create a New Subclass:* The system will highlight the selected node and display a small box in which the user types a class name as shown in Figure 13. After the user types a class name, the system redraws a DAG including the newly created node as shown in Figure 14.
- *Delete This Class:* The system will drop this class definition and redraw the resulting DAG. Figure 15 shows the resulting DAG after deleting "TENURED-PROF" class. We note that "ASSO-PROF" and "FULL-PROF" now become subclasses of "PROFESSOR" in accordance with the semantics of the ORION schema change operation (3.2).
- *Rename a class:* The system will highlight the selected node and display a small box in which the user types a new name for the selected node. After the user types a class name, the system redraws a DAG including the renamed node.
- *Change the contents of this class:* The system will highlight the selected node and display another pop-up menu showing operation choices for changing the definition of the selected node as shown in Figure 16. Within this pop-up menu, the user can add, delete, or modify instance variables, instance methods, class variables, and class methods.

  Addition: When the user clicks the item "add a new instance variable", the template in Figure 17 is displayed. The user fills the slots in the template menu and creates a new instance variable by clicking "Yes" option in the "** Create It! **" item. Class

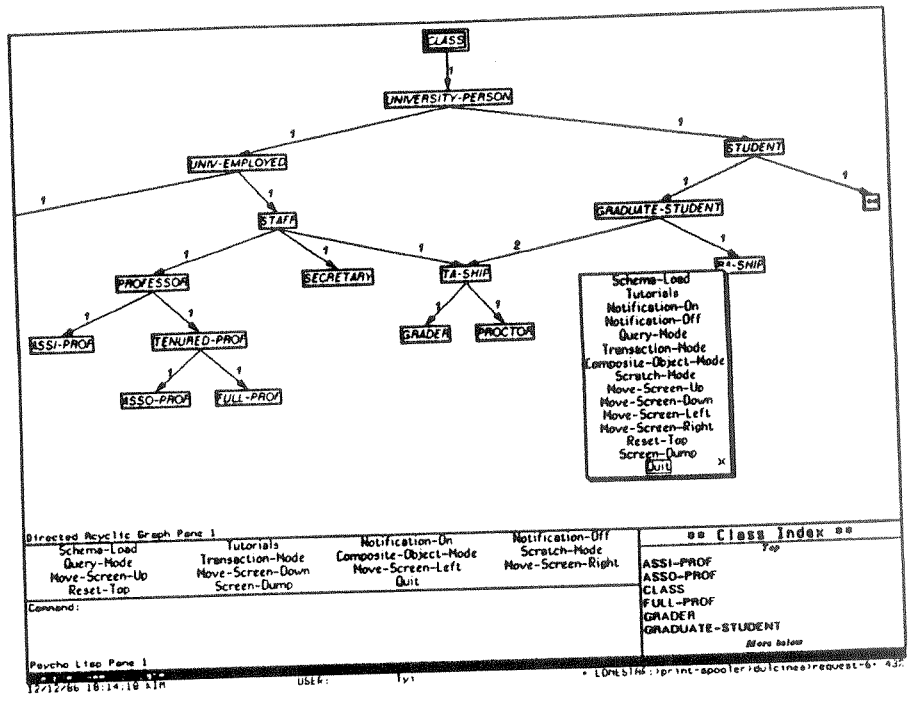


Figure 11: Pop-up menu for non mouse-sensitive items in the DAG window

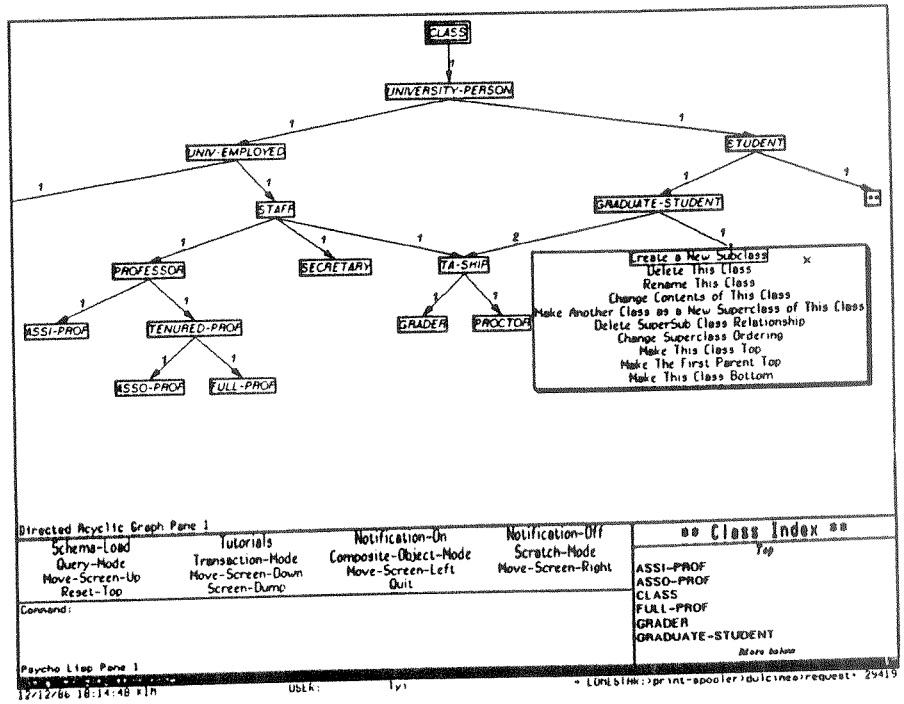


Figure 12: Pop-up menu for mouse-sensitive items in the DAG window

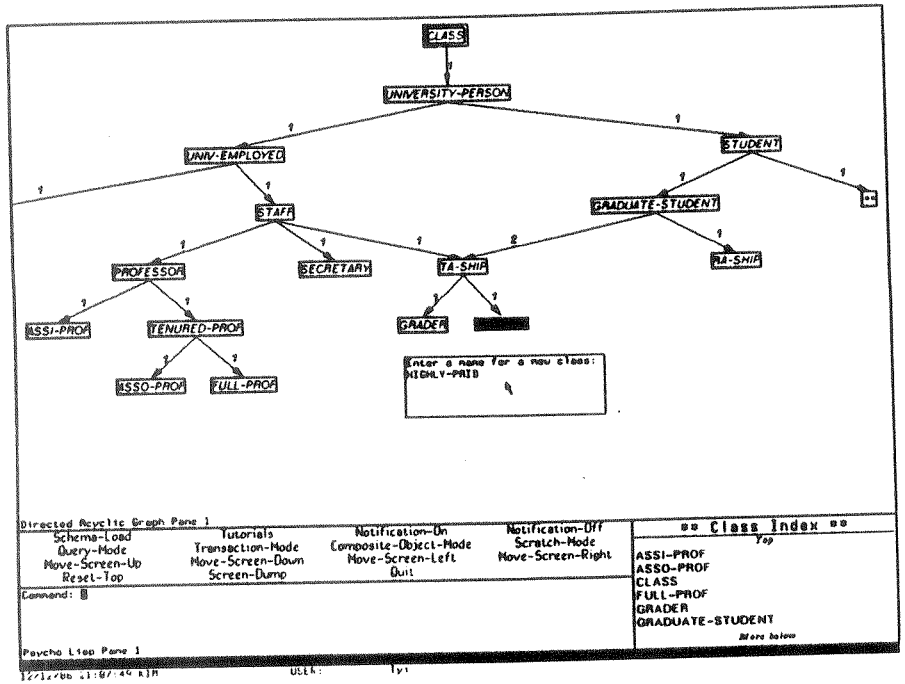


Figure 13: Pop-up window for providing the name of a new class

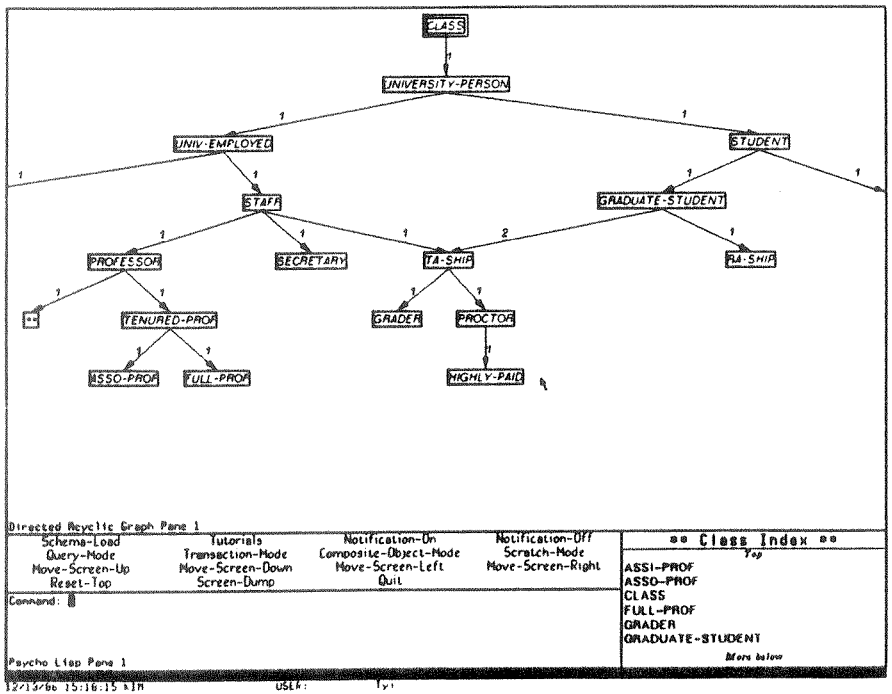


Figure 14: Class hierarchy resulting from creating a new class "HIGHLY-PAID"

variables are dealt with the same manner. If the user clicks "add a new instance method", a method code editor window is created as shown in Figure 18. The user can create a new instance method, test the new instance method, or modify the new instance method in the method code editor. Class methods are dealt with the same manner.

Modification: When the user clicks the item "add a new instance variable", the pop-up menu having the instance variables of the selected class is displayed as shown in Figure 19. The user may choose a particular instance variable in the pop-up menu, say "TACOURSE". Then a template describing the current status of "TACOURSE" pops up as shown in Figure 20. The user can modify the contents of an instance variable by changing the values in slots of the template of Figure 20. Class variables, instance methods, and class methods are dealt with the same way.

Deletion: As shown Figure 20, the template having the contents of an instance variable has the "Drop It!" item. By choosing "Yes" option in the template, the user can delete the instance variable from the selected node. However, if the variable is an inherited one, the request is rejected and an warning message will be displayed. Class variables, instance methods, and class methods are dealt with the same way.

- *Make Another Class a New Superclass of the Class:* The system highlights the selected node and waits for the user to click on another node as shown in Figure 21. The system provides graphical feedback helping the user to choose a valid node. The second selected node will become a new superclass of the first selected node. After the user chooses the second node, the system redraws the DAG to include the new edge. Figure 22 shows the class hierarchy having the new edge between "HIGHLY-PAID" and "RA-SHIP".

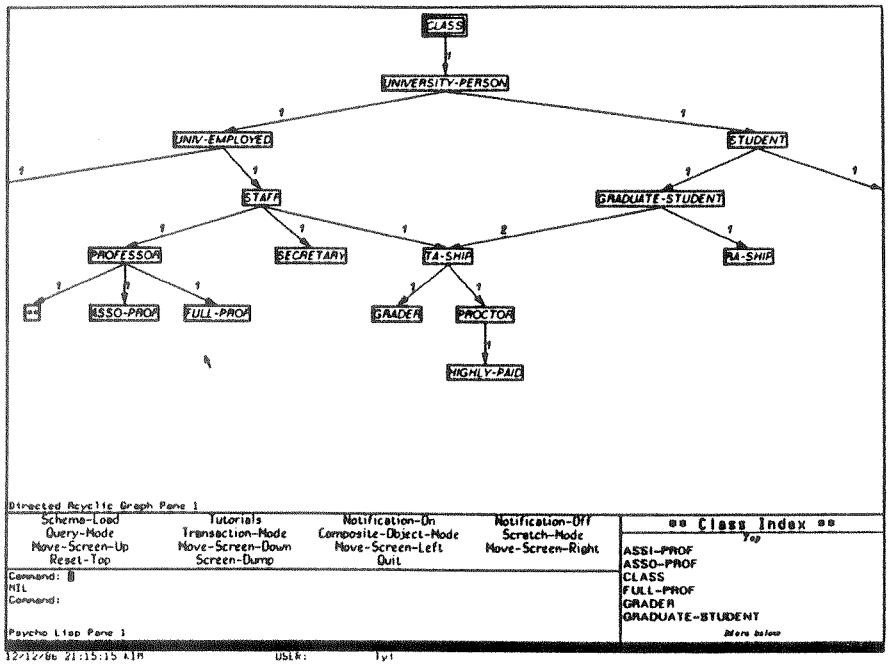


Figure 15: Class hierarchy resulting from deleting the class "TENURED-PROF"

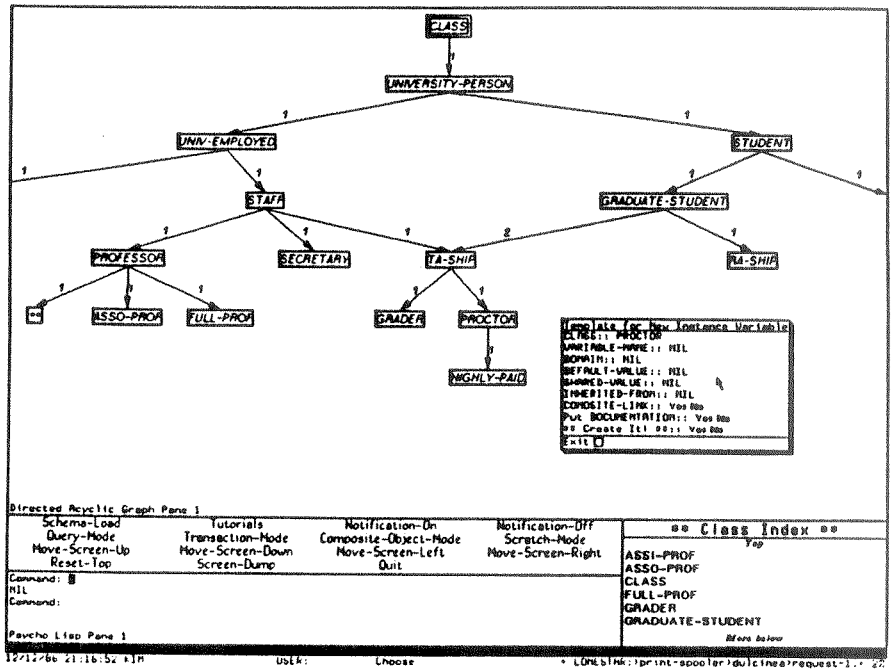


Figure 17: Template for a new instance variable

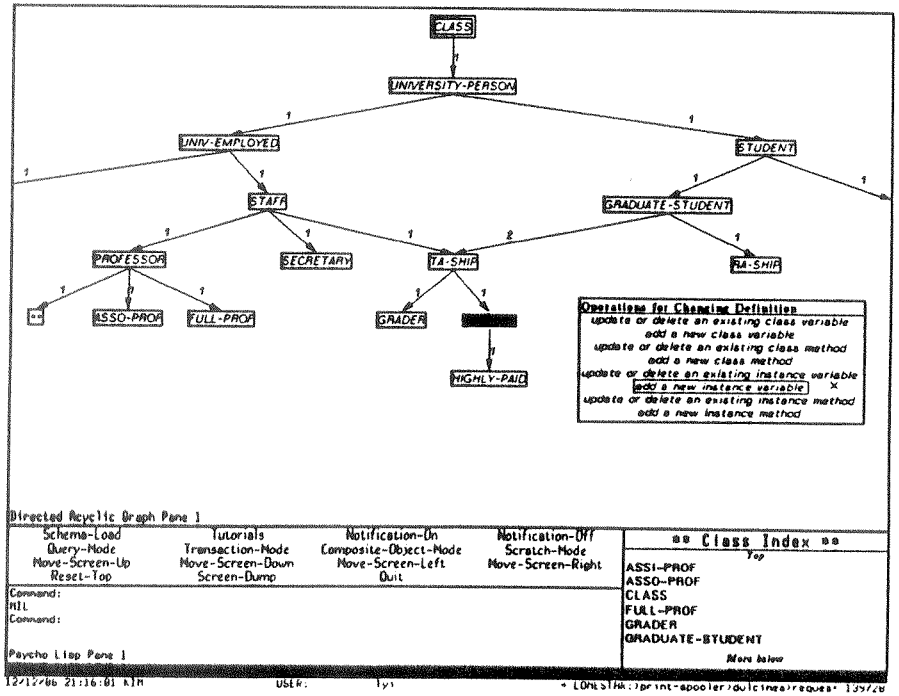


Figure 16: Pop-up menu displaying operations for changing a class definition

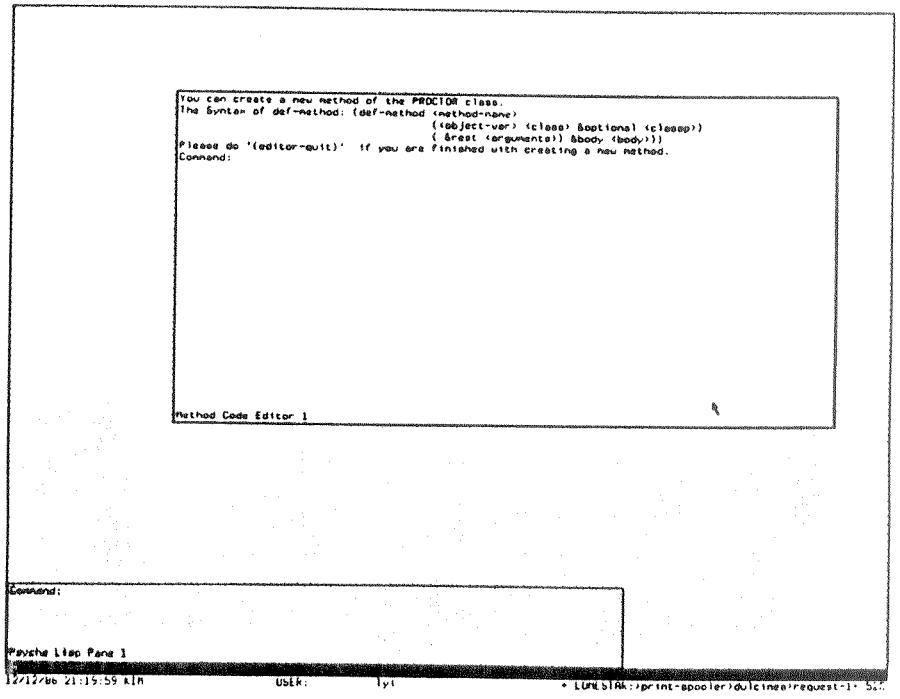


Figure 18: Method code editor

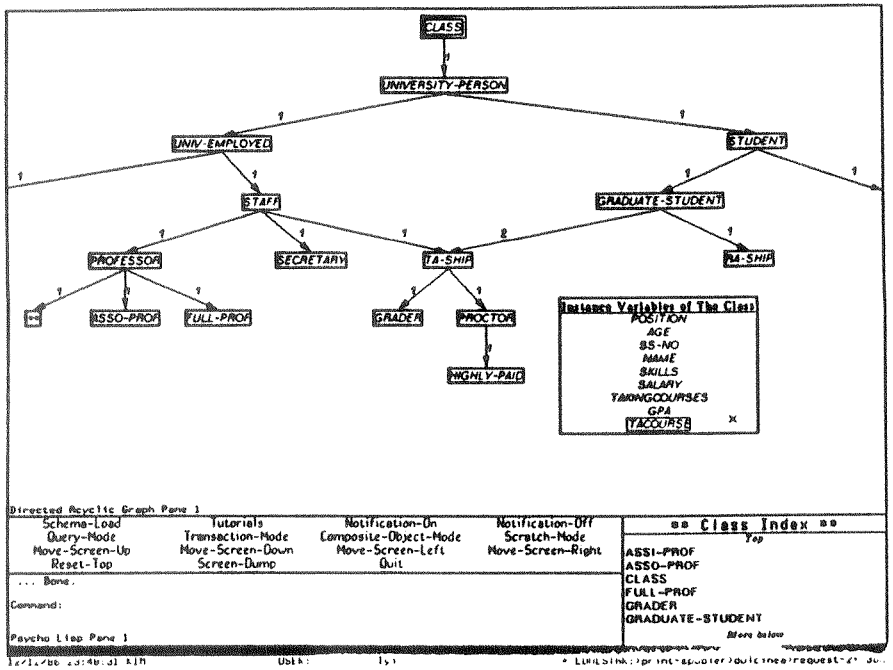


Figure 19: Pop-up menu displaying instance variables of the class, "PROCTOR"

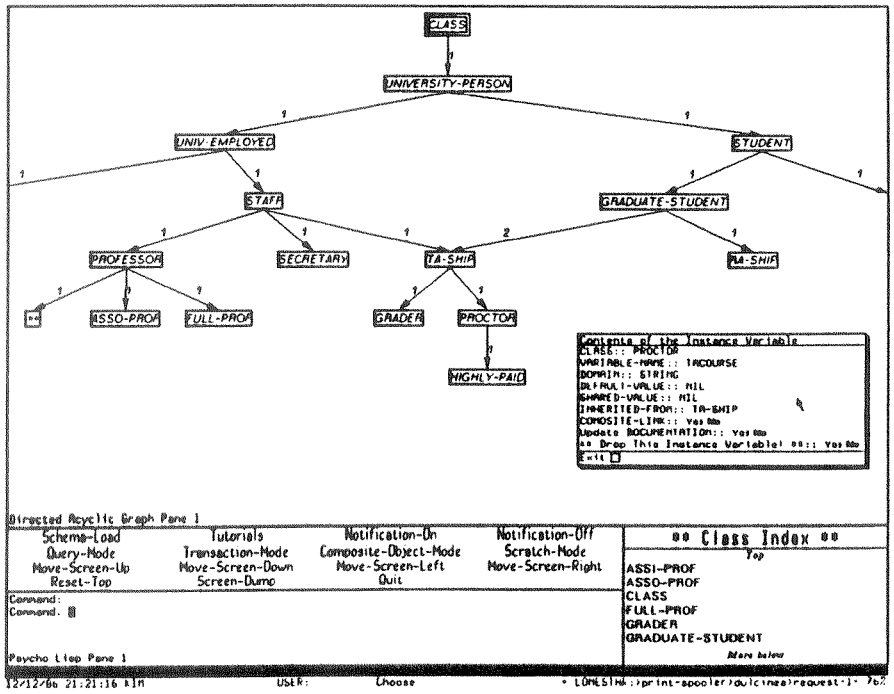


Figure 20: Template describing the selected instance variable, "TACOURSE"

- *Delete SuperSub Class Relationship:* The system highlights the selected node and waits for the user to click on another node as shown in Figure 23. The edge between the first selected node and the second selected node is dropped from the DAG. After the user chooses the second node, the system redraws the DAG to exclude the deleted edge. Figure 24 shows the class hierarchy after dropping the edge between "HIGHLY-PAID" and "RA-SHIP".

- *Change Superclass Ordering:* The system highlights the selected node and waits for the user to click on two more nodes as shown in Figure 25. The edge between the first selected node and the second selected node and the edge between the first selected node and the third node will be exchanged in the DAG. After the user chooses the third node, the system redraws the DAG resulting from exchanging the two edges. Figure 26 shows the class hierarchy resulting from edge replacement.

Schema Browsing and User Navigation

Consider the basic operation menu in Figure 11 again. The bottom three items in the menu are for navigation.

- *Make This Class Top:* The system places the selected node in the top center of the screen and draws all the subclasses of the selected node. The display that results when the user selects "Make This Class Top" on the node "STAFF", is shown in Figure 27. The purpose of this command is to allow the user to concentrate on the selected node and its *subclasses*.

- *Make The First Parent Top:* The system locates the first superclass P of the selected node in the top center of the screen and draws all the subclasses of P. The display that results when the user selects "Make The First Parent Top" on the node "STAFF", is shown in Figure 28. "UNIV-EMPLOYED" is the first superclass of "STAFF".

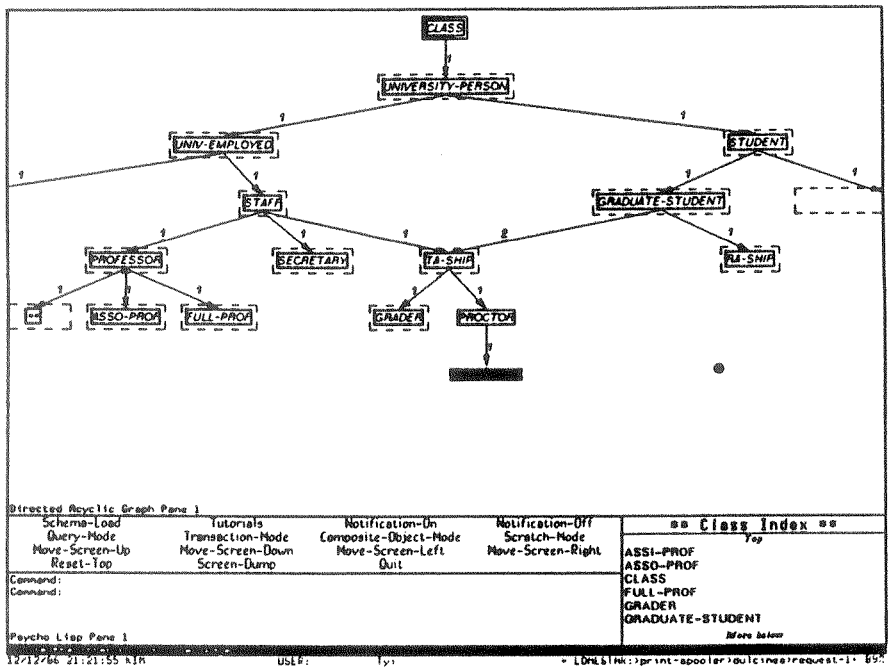


Figure 21: Graphical feedback in the middle of adding a new edge

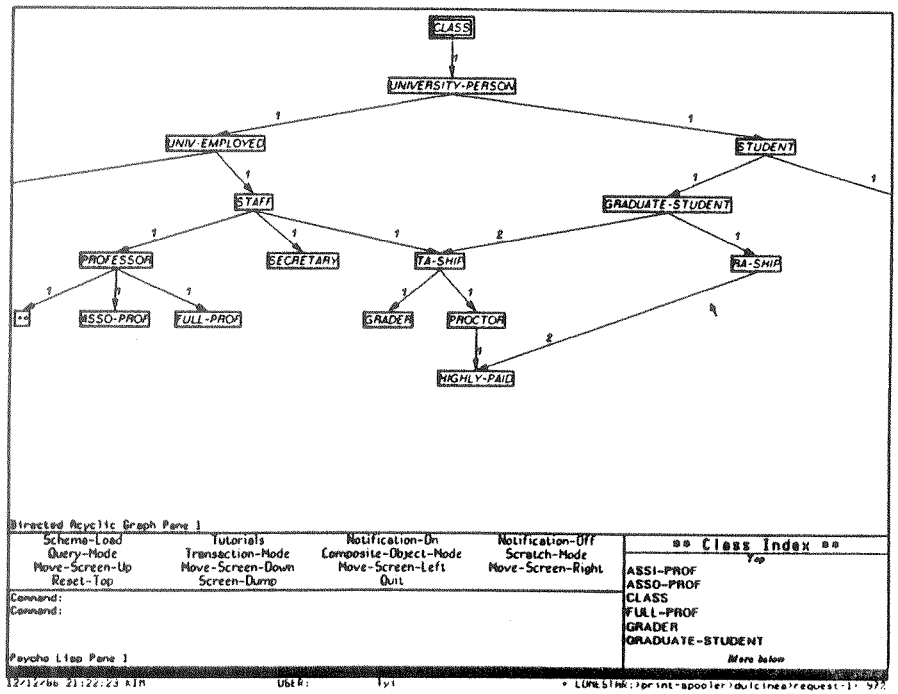


Figure 22: Class hierarchy resulting from adding a new edge

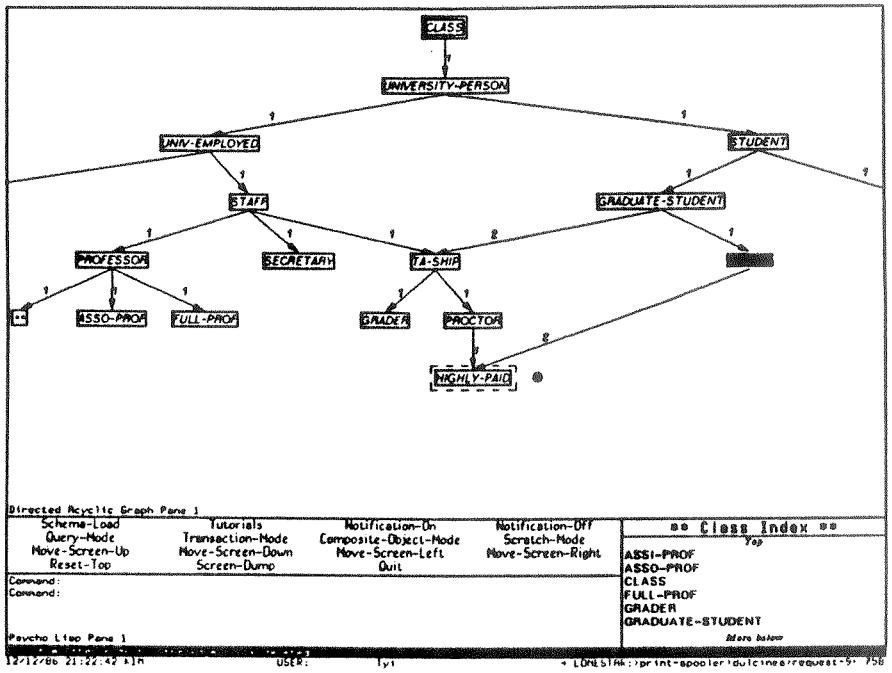


Figure 23: Graphical feedback in the middle of dropping an edge

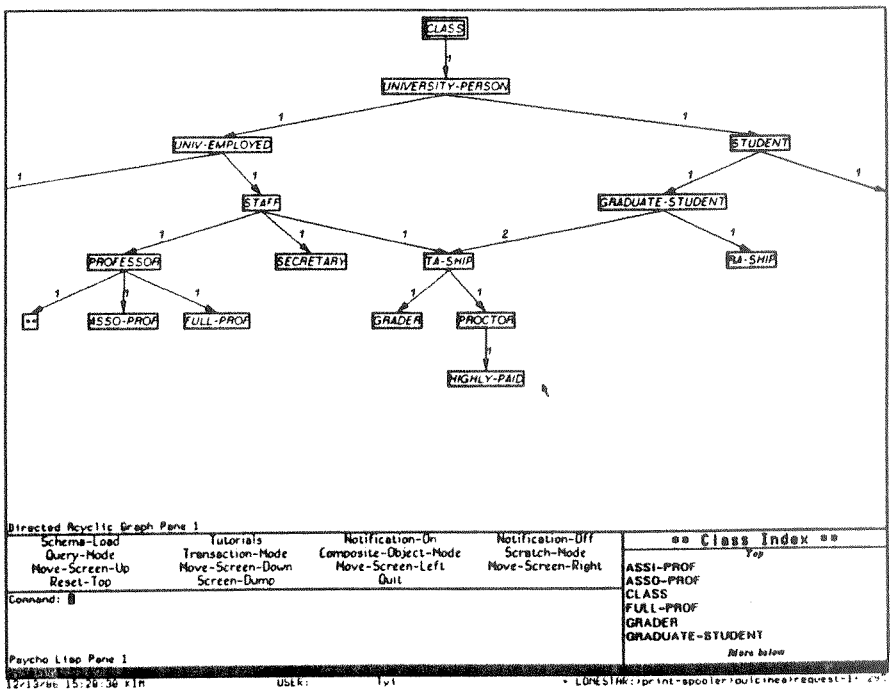


Figure 24: Class hierarchy resulting from dropping an edge

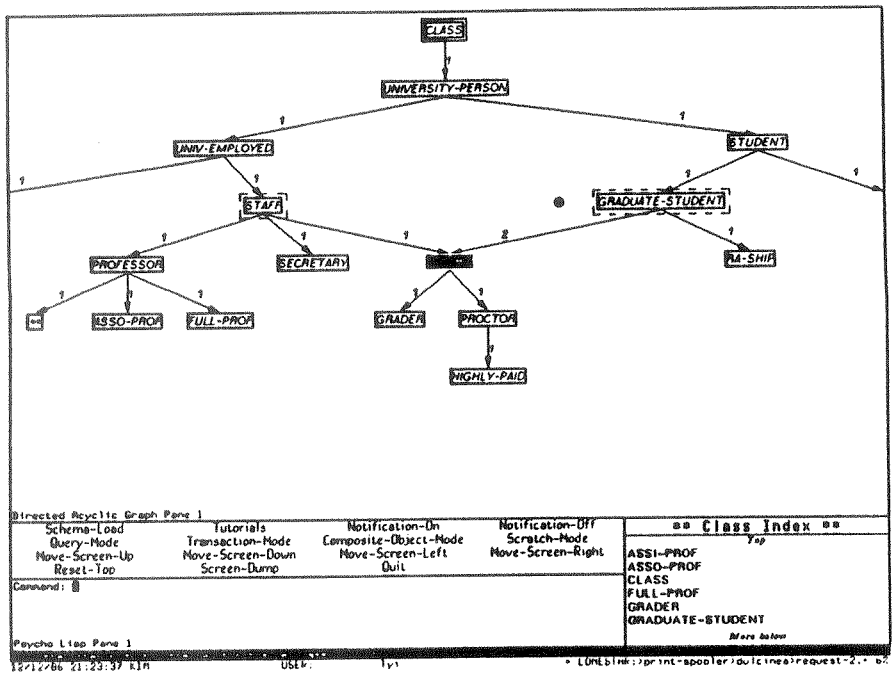


Figure 25: Graphical feedback in the middle of exchanging two existing edges

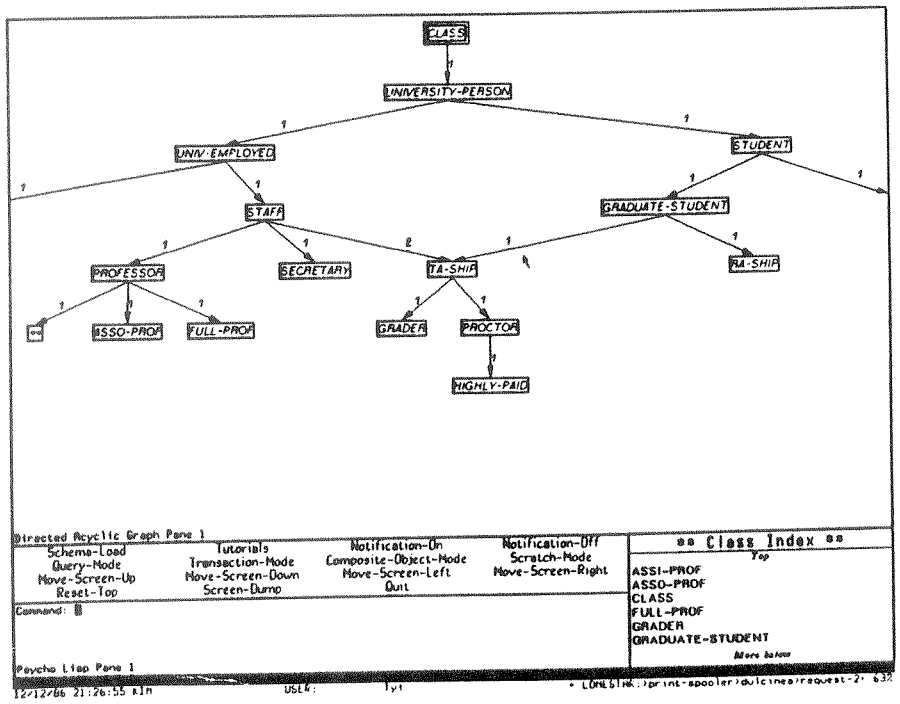


Figure 26: Class hierarchy resulting from exchanging two edges

- *Make This Class Bottom:* The system locates the selected node in the bottom center of the screen and draws all the superclasses of the selected node. The display that results when the user selects "Make This Class Bottom" on the node "TA-SHIP", is shown in Figure 29. The purpose of this command is to allow the user to concentrate on the selected node and its *superclasses*.

We note that PSYCHO supports three types of user navigation facility: (1) The above three commands in the basic operation menu, (2) The 5 navigation commands in the command menu window, and (3) The class index window. In general the commands in category (1) are used when the user wants to reorganize a DAG which is already partially visible on the screen. The commands in the categories (2) and (3) are useful when the user wants to jump to a class which is located far from the current screen position. If the user knows the exact location of a particular class, he can access the class using the 5 navigation commands in the command menu window. Even if the user does not know the exact location of a class on the screen, he can access the class by searching for the class and clicking it in the class index window. Figure 30 illustrates the display that results when the user clicks the class "GRADUATE-STUDENT" from the class index window. Figures 31-35 demonstrate the scrolling capability of PSYCHO.

Graphical Feedback

If the system uses lengthy dialogues to interact with the user, an experienced user may feel frustrated. Rather than tedious dialogues, PSYCHO provides several types of graphical feedback.

In Figure 21, the blinking nodes (i.e., nodes surrounded by dotted lines) are candidates of new superclasses for the highlighted (selected) node. In Figure 23, the blinking nodes indicate that incoming edges from the highlighted (selected) node can be dropped. Also in Figure 15, the blinking nodes indicate that incoming edges from the highlighted (selected) node can be exchanged. Besides those, if the user tries to create an edge (i.e., IS-A relationship) and

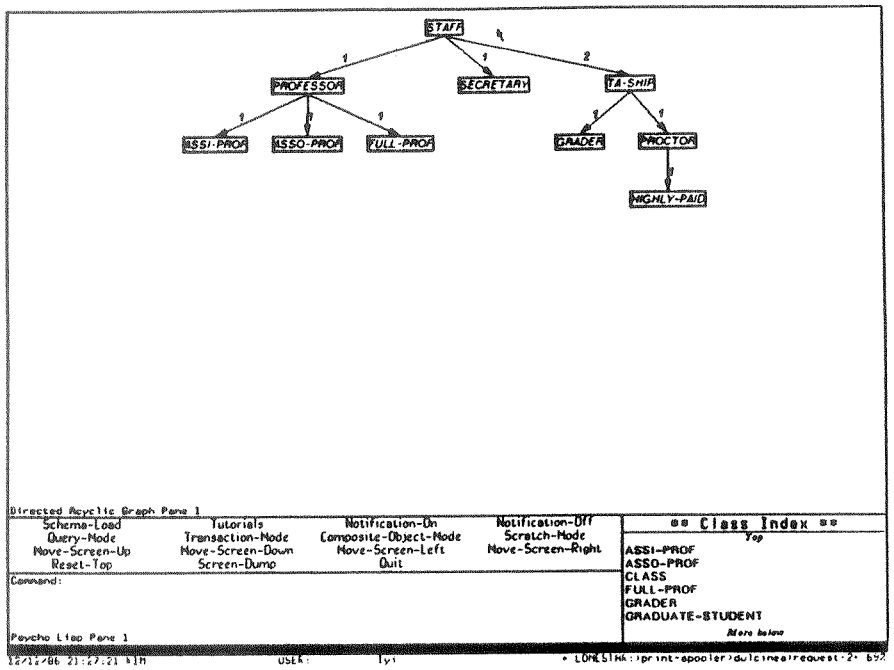


Figure 27: Making the node "STAFF" top

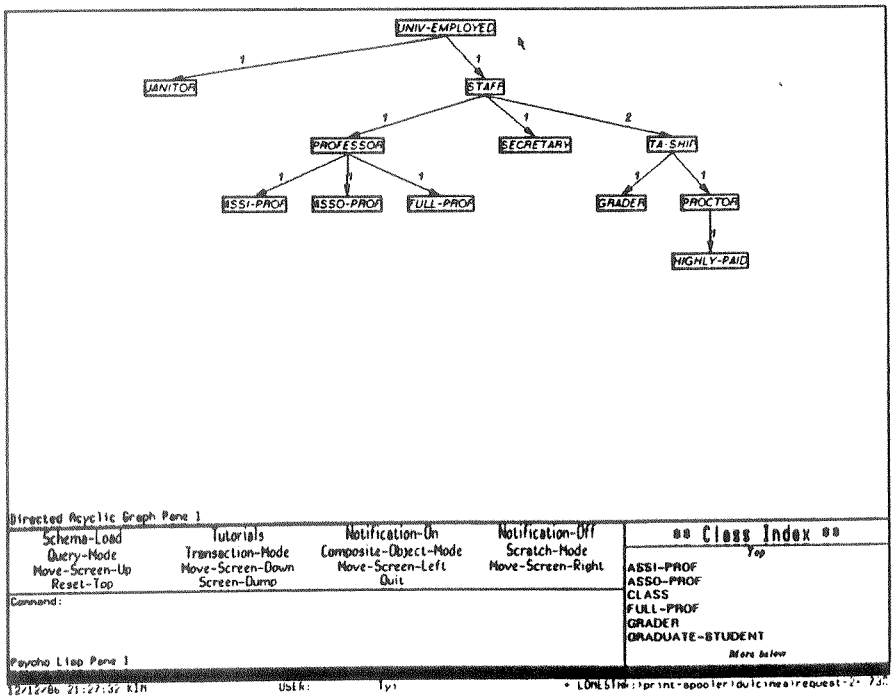


Figure 28: Making the first superclass of the node "STAFF" top

the resulting DAG happens to have a cycle, PSYCHO rejects his request with graphical feedback showing the cycle.

Integrity Checking

The ORION system does not allow schema changes which violate the invariants of the framework [BKKK87]. In case of unacceptable operations, rather than receiving error messages from ORION directly, PSYCHO explains why such requests are not acceptable. PSYCHO eliminates erroneous schema change requests by checking the validity of requests before submitting them to ORION.

The result of validity checking is represented in the form of graphical feedback or pop-up messages. For example, as we mentioned earlier, if the user tries to create an edge and the resulting DAG happens to have a cycle, PSYCHO draws a bold cycle to show the cyclicity resulting from the request. Again in Figure 21, if the user clicks on the node "PROCTOR" as a second node, PSYCHO explains that the second node "PROCTOR" is already a superclass of the first selected node "HIGHLY-PAID".

Name conflicts are also checked by PSYCHO. In Figure 12, suppose the user is trying to rename the node "PROCTOR" into "UNIV-EMPLOYED" which already exists, PSYCHO explains the situation of name conflicts. The same is applied to rename operations such as (1.1.3) (1.2.3) (1.3.3) (1.4.3).

Sketch Mode and Transaction Mode

Besides schema change mode, PSYCHO supports two other useful modes: *sketch mode* and *transaction mode*. In the schema change mode, every graphical action is immediately submitted to ORION and ORION performs the corresponding command. However, in the sketch mode, PSYCHO does not communicate with ORION in the middle of a session. The purpose of this mode is improved performance. Since some of schema change operations are expensive, undoing the previous schema change causes high system overhead. The problem is even greater when the user undoes several previous changes to a schema. In the sketch mode, the user can freely manipulate a schema through trial and

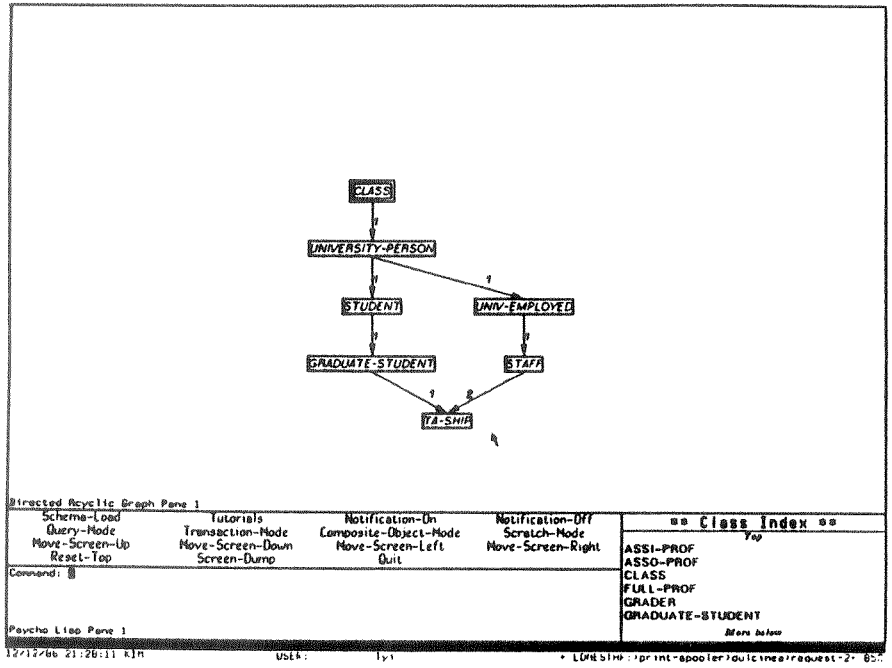


Figure 29: Making the node "TA-SHIP" bottom

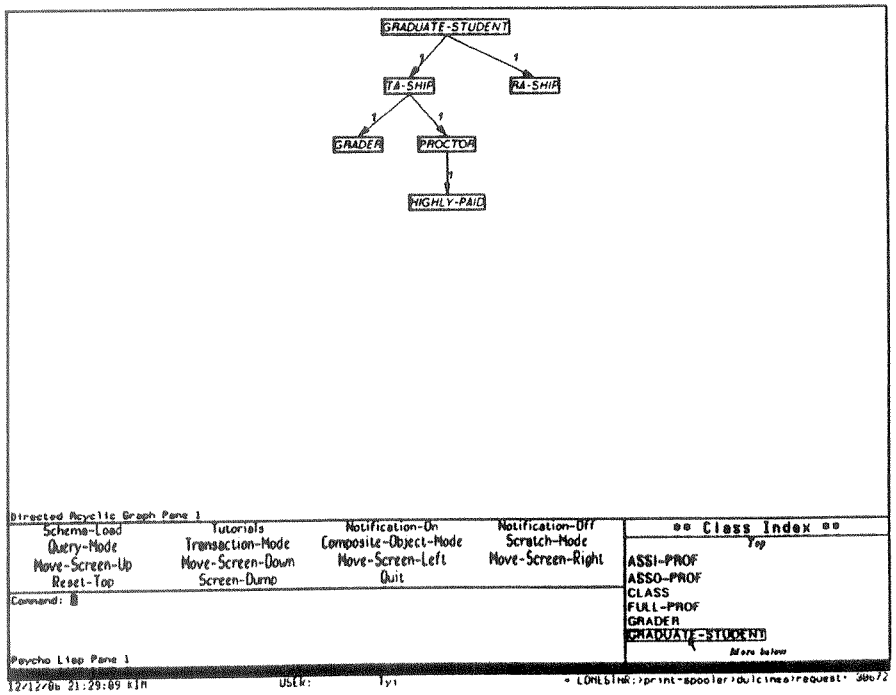


Figure 30: Clicking the class "GRADUATE-STUDENT" from the class index window

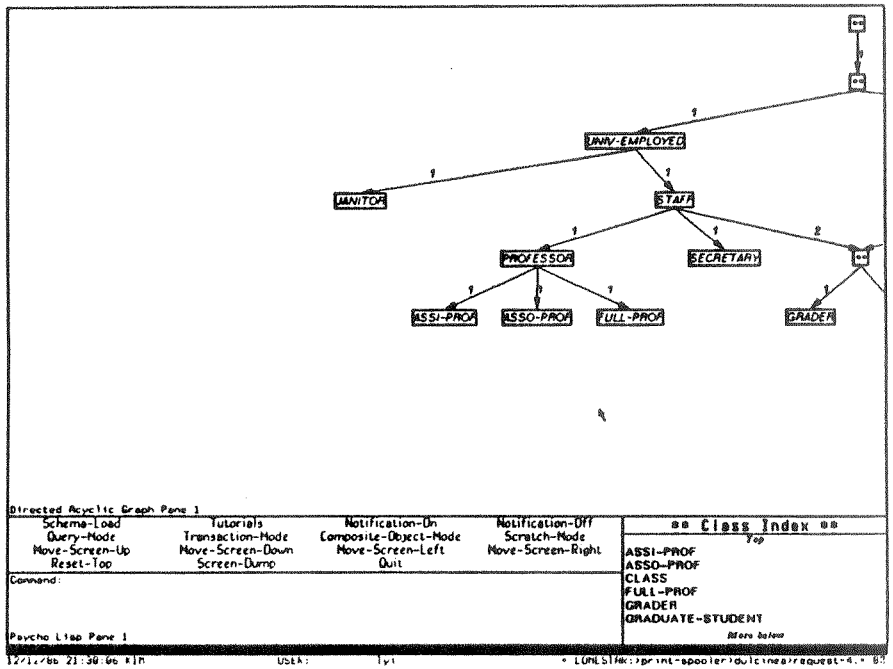


Figure 31: Moving the screen to the left

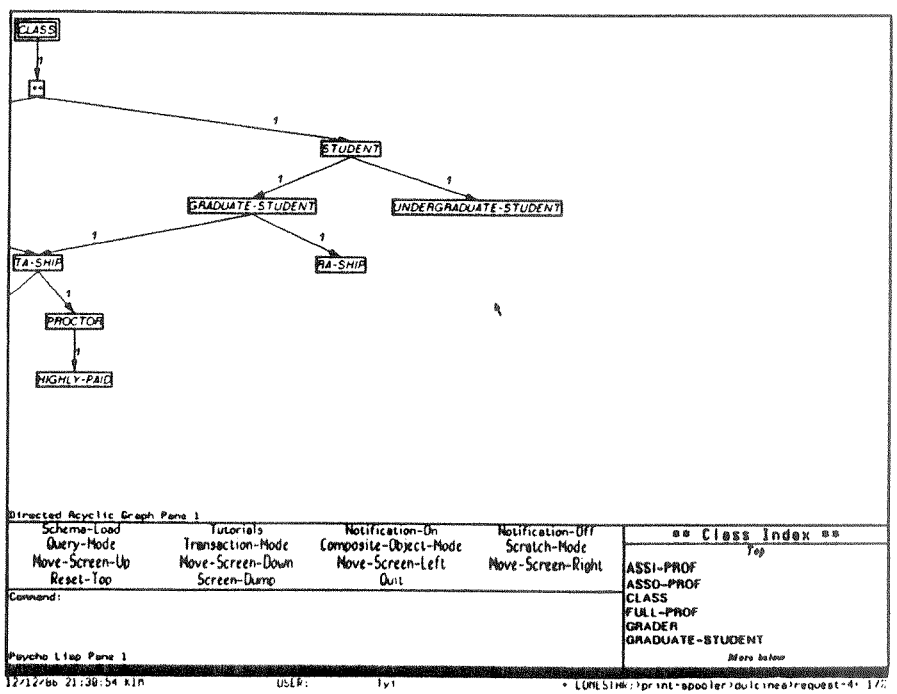


Figure 32: Moving the screen to the right

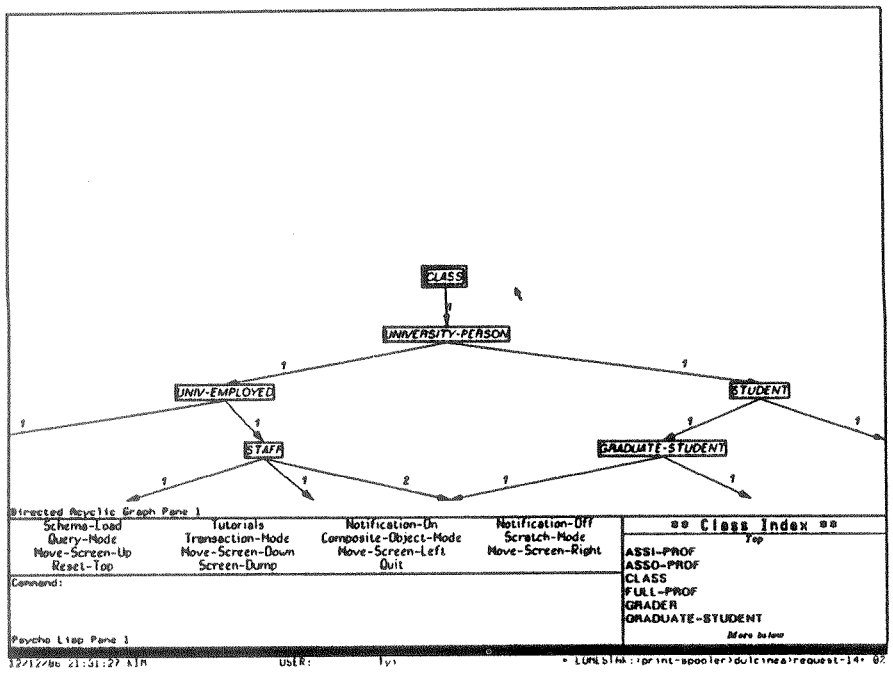


Figure 33: Moving the screen upward

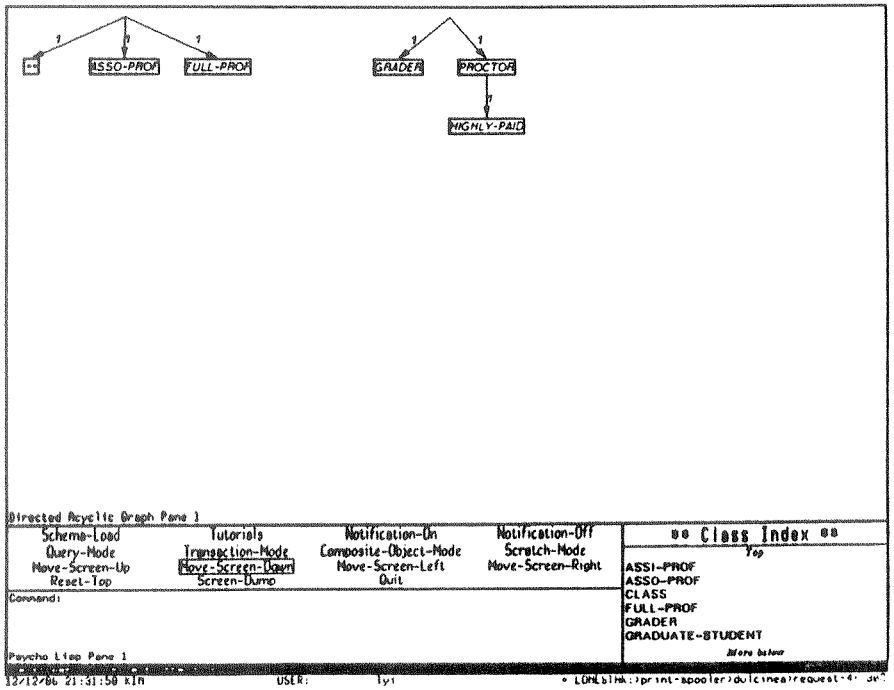


Figure 34: Moving the screen downward

error because schema changes are virtual. The user can give a command "go-ahead-and-do-it" at the end of the session. This sketch mode is also useful in the stage of initial database design when the user does not have a complete understanding of applications.

The motivation of transaction mode is similar to that of sketch mode. In transaction mode, one schema change transaction consists of one more schema change operations. Each graphical action is submitted to the ORION schema manager and to the ORION transaction manager and ORION updates the schema definition with corresponding operations. In case of system failures, the system guarantees recovery. Also the user can abort the schema change transaction any time in the middle of a transaction.

3.2.4 Object-Oriented Implementation

In this section, we describe how PSYCHO is implemented. PSYCHO is implemented in an object-oriented fashion using Flavors, which is an object-oriented programming feature of ZetaLisp.

Flavors and ZetaLisp

The flavor system is the Symbolics machine's mechanism for defining objects. An object can receive messages and act on them. A flavor is a class of active objects, one of which is called an instance of that flavor. A set of instance variables and a set of messages are associated with a flavor. As such, Flavors are similar to ORION classes in several senses. The following example illustrates a sample flavor "automobile", a sample method "old-model-years" which computes the number of years during which a car is used, and instantiation of automobile flavor.

```
(defflavor automobile
        (year-model
        price
```

```
        manufacturer
        mass)
        (vehicle)
    :gettable-instance-variables
    :settable-instance-variables)
(defmethod (automobile :old-model-years) (currentyear)
        (- currentyear year-model))
(setq mycar (make-instance 'automobile
        :year-model 83
        :price 3000
        :manufacturer 'hyundai
        :mass 2500))
```

The Symbolics Flavors system supports hundreds of built-in flavors which are useful in building graphical interfaces. Static menus, dynamic menus, windows, LISP windows, and command menus are all built in flavors.

In PSYCHO, we define a flavor for nodes of DAGs, which is an internal data structure for drawing figures:

```
(defflavor node
        (children
        parents
        name
        x-coord
        y-coord))
        ()
    :gettable-instance-variable
    :settable-instance-variable
    )
```

The instance variables children and parents have a list of immediate subclasses and superclasses respectively. The instance variable name has the name of a class. (x-coord, y-coord) will represent a position where a node is located on the screen. A number of methods are defined on the flavor node, such as add-child, add-parent, and so on.

Besides the node flavors, PSYCHO has over 10 flavors and associated methods, which are used for managing PSYCHO data structures. Methods are written in ZetaLisp. Since PSYCHO is implemented in an object-oriented fashion, it is fairly easy to modify or extend PSYCHO.

Interfacing with ORION Schema Manager

The ORION schema manger is written in CommonLisp because of portability issues. PSYCHO is written in ZetaLisp because graphic functions are only available through Flavors of ZetaLisp in Symbolics. Fortunately, functions in CommonLisp and functions in ZetaLisp can call each other within Symbolics. By attaching "zl:" to ZetaLisp functions and "cl:" to CommonLisp functions, the LISP interpreter can tell whether a function is borrowed from CommonLisp or ZetaLisp. The concept is called *package* concept. When the user represent schema changes graphically on PSYCHO, corresponding ORION functions should be invoked. When PSYCHO calls an ORION function, say delete-class, the corresponding syntax is "orion::delete-class ...". "orion" is the package name.

3.2.5 Discussion

In this section, we discuss several issues related to PSYCHO. First, we introduce

systems similar to that of PSYCHO. Second, we criticize PSYCHO and finally we make some observations on graph representation theory and its relation to graphical schema manipulation.

Related Works

In recent years, many AI tools have based on object oriented programming [LMI85, Symb85]. They also provide visual aids for browsing class hierarchies. PSYCHO is different from the class browsers of AI tools because PSYCHO is designed from a database perspective: transaction mode, sketch mode, query mode, etc. In the visual aids of AI tools, the user can modify the structure of class hierarchy, but only a few operations are allowed. User navigation within the DAG in the visual tools is not as dynamic as PSYCHO. In general the existing AI tools disallow run-time class modification.

Another motivation behind this type of visual tool is that much time is spent learning the large library that is an integral part of most object-oriented language systems. Thousands of built-in classes and methods cannot be easily mastered without a powerful visual tool.

KEE knowledge base browser: KEE is an AI tool for knowledge engineering, from Intellicorp [INTE84]. KEE consists of a set of software tools to assist users in building their own knowledge-based systems. KEE supports various programming paradigms including rule-based programming and object-oriented programming. The object-oriented nature of KEE is based on the *Frame* data structure which can have a number of slots and rules and associated procedures. Slots, rules, and procedures all can be inherited or defined locally. Figures 35 and 36 show the KEE knowledge base browser and sample frame structures respectively.

LOOPS class browser: LOOPS is a knowledge programming language from Xerox PARC [Stef83]. LOOPS class browser is conceptually similar to KEE knowledge base browser. Figure 37 shows an example of a class hierarchy in the LOOPS class browser.

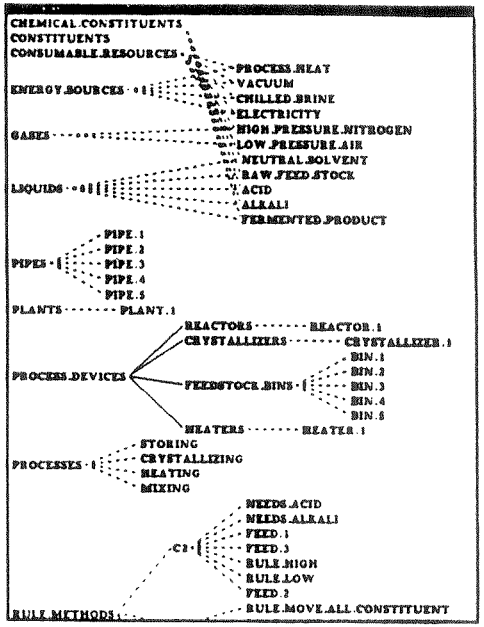


Figure 35: KEE knowledge browser (from [INTE84]

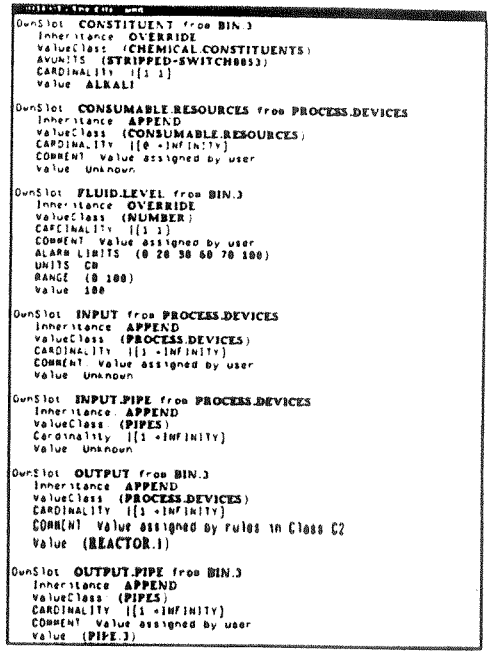


Figure 36: Sample frame definitions in KEE (from [INTE84])

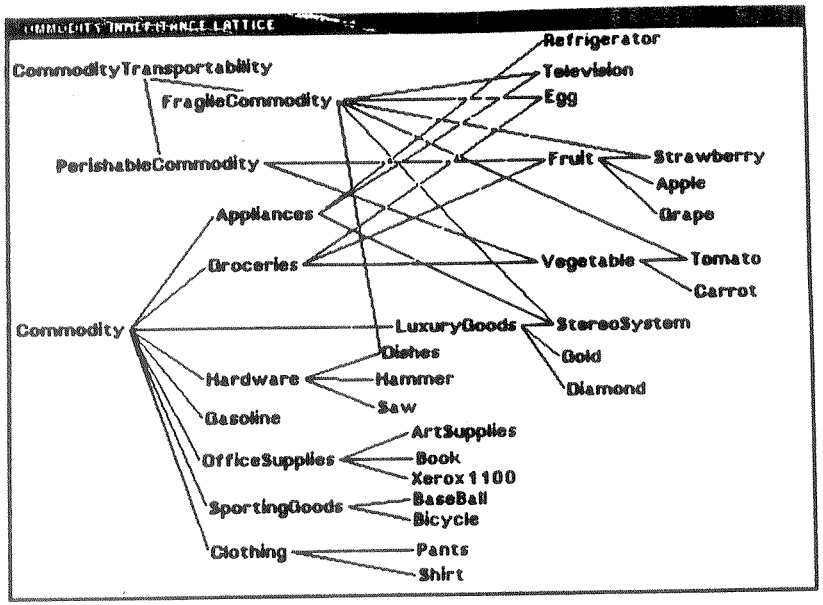


Figure 37: LOOPS class browser (from [Stef83])

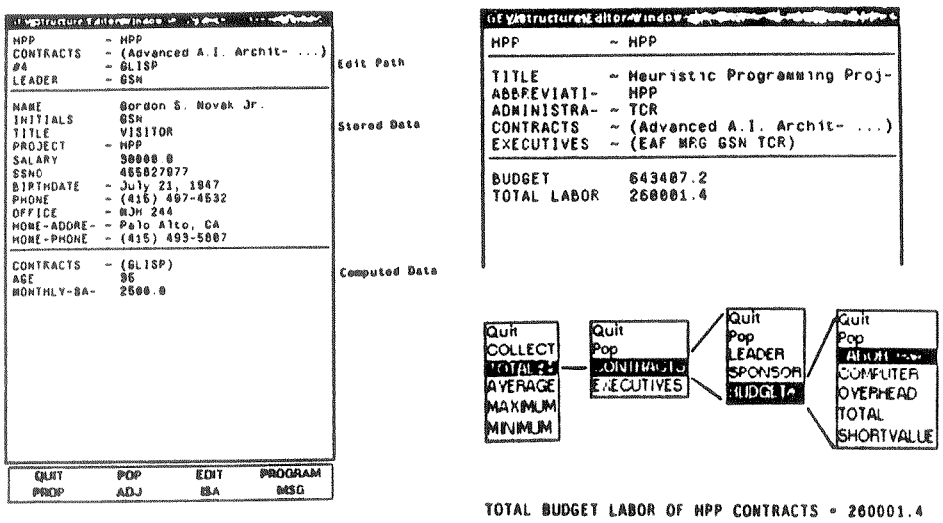


Figure 38: GEV (from [NOV82])

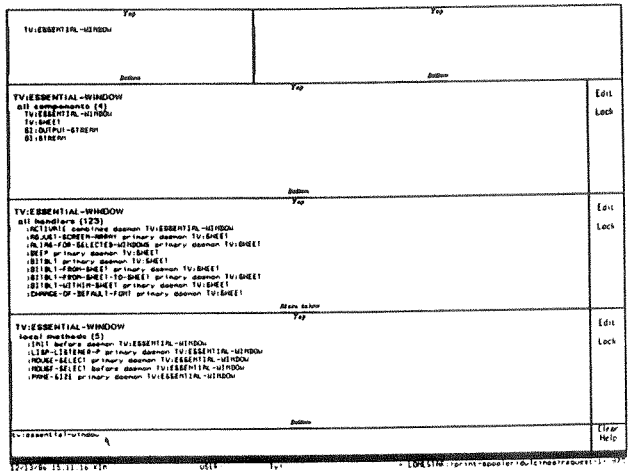


Figure 39: Flavor examiner

GEV: GEV [NOV82] is a tool which allows the user to display, inspect, and edit structured data written in GLISP [NOV83] which is used for knowledge description. GLISP is a LISP dialect that includes abstract data types. GEV allows changes to actual data as well as changes to data types. DAG representation is not supported in GEV. GEV is initiated by giving it a pointer to an object and its type. GEV interprets the data and displays the contents of data on the screen as shown in Figure 38.

Flavor Examiner: This Symbolics utility helps users examine the structure of flavors [Symb84]. The flavor examiner consists of six panes as shown in Figure 39. The top left pane is the flavor history. The top right

pane is the method history. The user enters a flavor name or method name into the bottom pane. The three middle panes are examiner panes that list the answer to a query (i.e., interesting flavors or methods). When the user selects "Edit" in the right end of a pane, the system puts the contents of the pane into a zmacs-like buffer. When the user selects "Lock" in the right end of a pane, the contents in the pane cannot be updated. In Figure 39, the user is browsing the system built-in flavor, "tv:essential-window". The definition of "tv:essential-window" is displayed. This tool is rather text-oriented because the user has to type what he wants. DAG representation is not supported in the flavor examiner.

Self-Criticism

We recognize that PSYCHO has several defects. First PSYCHO is not machine independent, but runs only on top of Symbolics because PSYCHO uses many built-in flavors. Second the DAG representation algorithm of PSYCHO is so primitive that sometimes PSYCHO does not use empty space of the screen properly. However, intelligent DAG representation is computationally hard and may benefit from some heuristic techniques. Most existing graph browsing systems have trivial layout algorithms [Rowe86b]. Third, the user cannot see a total map of the DAG. A zooming facility would allow the user to view the overall DAG structure, but we have not yet incorporated such a feature into PSYCHO.

Graph Representation Theory

Our experience from PSYCHO and our previous experience from PICASSO [Kim85a, Kim86, KKS86], which is a graphical query interface, emphasize the need for further work in *graph representation theory*. In particular *planarity* and *colorability* problems are important in graph representation.

Planar graphs are easier to deal with than non-planar ones. Planar DAGs generally represent a clearer picture of the database than an equivalent non-planar DAGs. Of course, not all DAGs are planar, so we must concern

ourselves with minimizing an appropriate measure of non-planarity, such as the number of crossing edges.

Another useful measure is colorability. We note that it is practical to think in terms of color since color displays are becoming increasingly popular. For black-and-while displays, the number of colors would be restricted to the number of gray tones that are conveniently distinguishable. For example, classes (nodes) and relationships among classes (edges) can be represented in a particular color. A graph representation is k-colorable if using k colors, no intersecting edges have the same color and no crossing edges have the same color. We note that the colorability problem is NP-complete. Thus, heuristic solutions need to be investigated.

Chapter 4
Schema Versions

We provided a framework of schema evolution in chapter 2. 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 chapter, we extend our schema evolution framework by allowing *schema versions* 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. We shall present a technique that enables users to explicitly manipulate schema versions and maintain schema evolution histories in object-oriented databases. Our solution for schema versions is consistent with our schema evolution framework and is designed to minimize *storage redundancy* and *update anomaly*. For completeness, we integrate our schema version model with the object version model formulated by H.T. Chou and W. Kim [CK86].

4.1 Motivation

Benefit

In conventional object version models [KL84, DL85], only one schema is shared by multiple object versions of an object. Suppose D denotes an object and D_1, D_2,..., D_n denote object versions of D respectively. The schema for D, D_1, D_2,..., and D_n should be the same one. 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 called 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 the object version 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, the user 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 cannot 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 benefits 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.

Problems

There are two difficulties (called *storage redundancy* and *update anomaly*) in supporting schema versions, which must be resolved for schema versions to be practical. At first glance, there are two 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, (i.e., all auto-manufacturers in AUTO-MANUFACTURER.2 produce wagons). In this schema version, in all instances the value of **Wagon?** is "Yes".

AUTO-MANUFACTURER

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

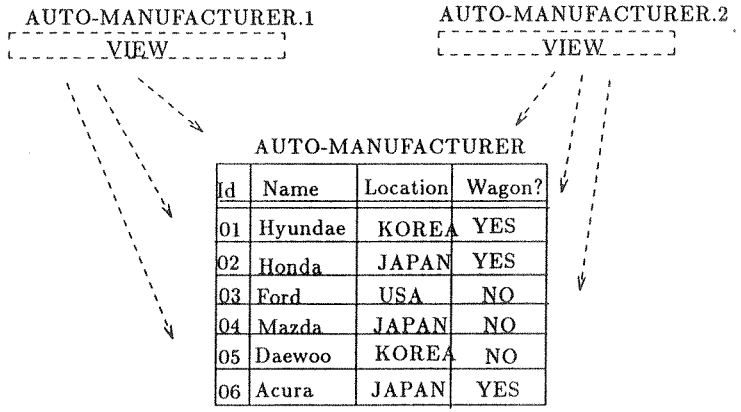
AUTO-MANUFACTURER.1

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

AUTO-MANUFACTURER.2

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

(a) Snapshot Approach



(b) View Approach

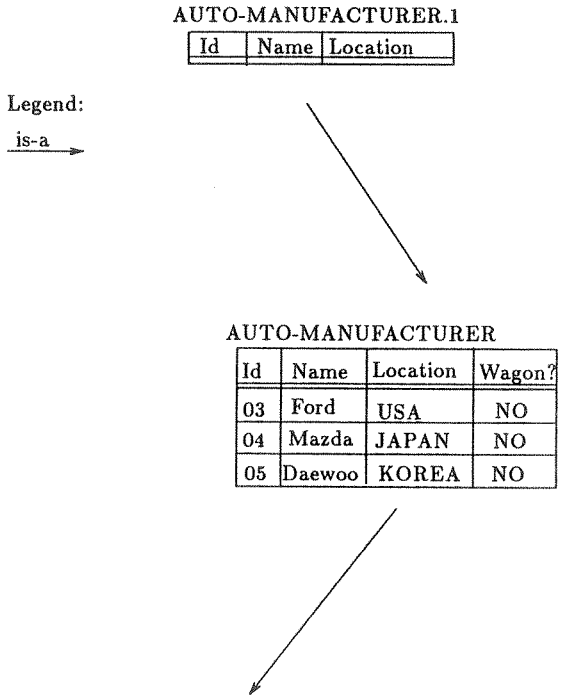
Figure 40: Approaches for Schema Versioning

Figures 40.a and 40.b show the snapshot and view approach respectively. In the snapshot approach, as Figure 40.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 called *ambiguous*. Suppose a user requests the deletion of auto-manufacturer #06 from the AUTO-MANUFACTURER.2 view in the Figure 40.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)** to AUTO-MANUMACTURER. 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 40.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



(c) Our Approach

Figure 40 (Cont'd): Approaches for Schema Versioning

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 per 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*. The approach in Figure 40.c is our solution.

4.2 Introduction to Our Approach

Our approach takes advantage of (1) our schema evolution framework and (2) inherent properties of class hierarchies. By (1) we mean that deriving a new version of a class can be viewed as creating a new class in a class hierarchy (details will be discussed shortly). By (2) we mean two implicit assumptions in object-oriented data models: an assumption that an object instance can belong to one and only one class and an assumption about update semantics (discussed below).

As shown in section 1.2.2, 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.

There is 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 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). We assume that (3) is the only deletion semantics in object-oriented data models. We shall take advantage of this implicit assumption. Therefore, the issue of update anomaly is ignored within a class hierarchy.

We intend to maintain a history of classes, rather than a history of the 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, rather, S' is somehow related with S. We call S' a *neighboring class*.

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

4.3 Schema Version Semantics

In this section we explore our idea in greater detail by discussing operations in our schema version model.

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'.

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

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 C: *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};

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) $\times$ (domain of input-parameter-2) $\rightarrow$ (domain of output-parameter-1) $\times$ (domain of output-parameter-2). Subsumption can be determined between methods by simply comparing domains of parameters in the following way: given methods M: $I \rightarrow O$ and M': $I' \rightarrow O'$, 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 41.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 41. There are 5 class versions of AUTO class as shown in Figure 41.a. Since AUTO.3 subsumes AUTO.1, AUTO.3 is placed in the class hierarchy as a superclass of 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 (note that OBJECT is the root class of a class hierarchy). A generic class does not have any instance variables or methods. A generic class can be used for dropping all class versions of a class at once. Another important application of generic class is that it allows all class versions to be the domain of an instance variable. In the example of Figure 41b, the instance variable Vehicle can have all class versions of AUTO as a domain by specifying AUTO.GENERIC as its

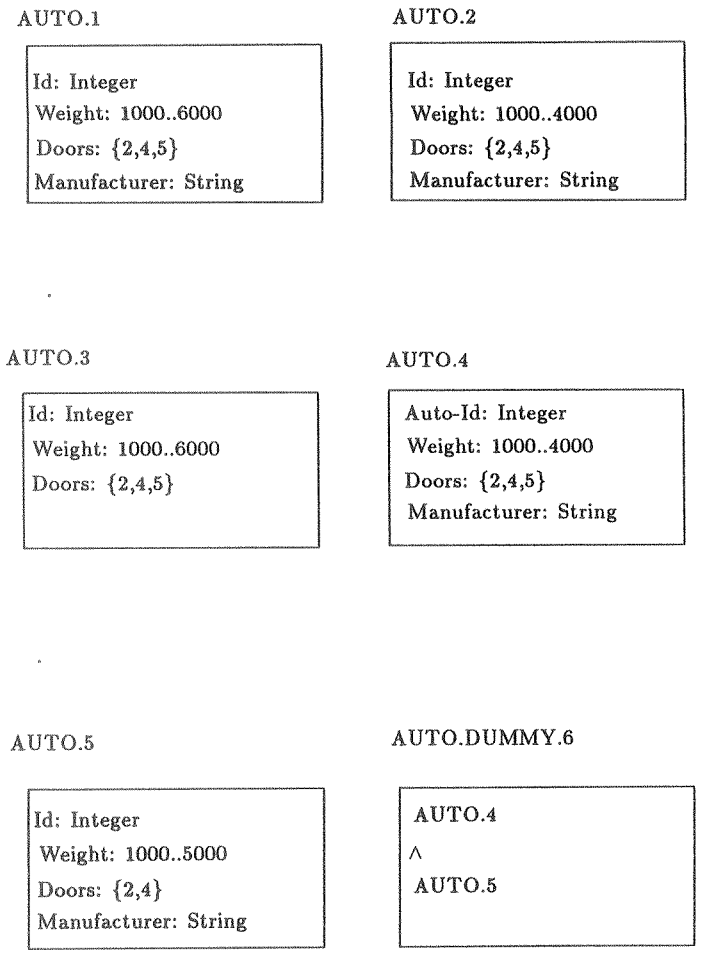


Figure 41.a: Schema Versions of AUTO class

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. In summary, a generic class is a means to access or delete all versions of a particular class as well as a means to bind all versions of a particular class to an instance variable as a domain. Figure 41.b shows the generic class of AUTO.

- DUMMY CLASS: We require an instance to belong to one and only one class. If a new class version is a neighboring class of the original class version, the new class version cannot be embedded into the class hierarchy as a superclass or subclass of the original class version. In that case, it appears that some instances of existing class versions should be duplicated and stored under the new class version. However, this is not consistent with the assumption that an instance belongs to one and only one class. We get around this problem by creating a dummy class which can accommodate common instances of the new class version and existing class versions. As shown in Figure 41.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 ∧ 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 neighboring 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 in (1.1.3) and (1.2) may create semantically equivalent class versions having the same membership condition. We call this *trivial schema evolution*. As shown in Figure 41.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

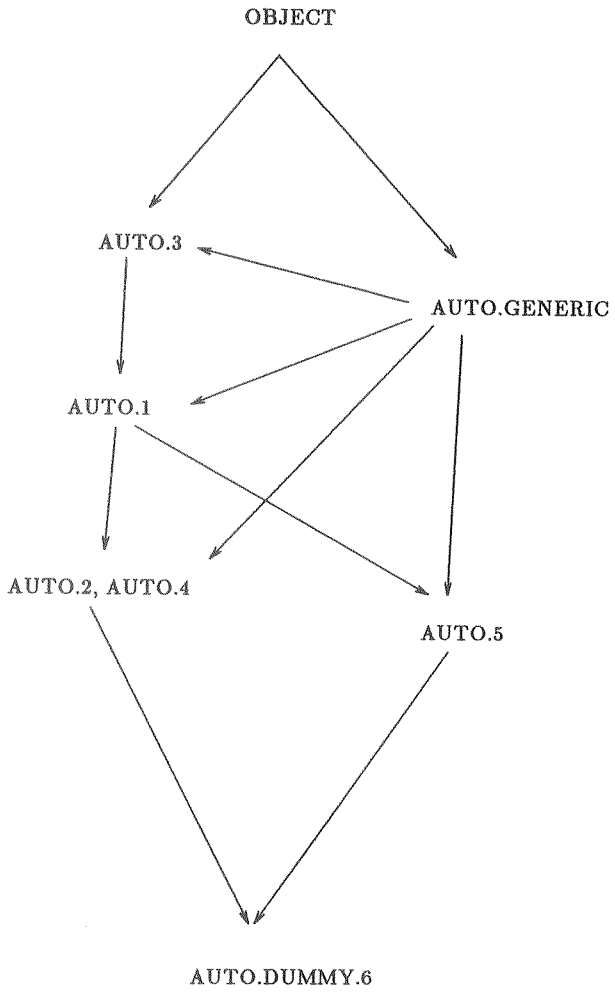


Figure 41.b: Generic, Dummy, and Equivalent Classes

essentially same except the different names of Id and Auto-Id. Equivalent classes are also useful when a new class version is the same as one of its ancestor class versions. We call this *circular schema evolution.* For instance, if a schema version V_1 has three instance variables (X,Y,Z), V_2 has (X,Y) and V_3 has (X,Y,Z), then V_1 and V_3 are stored in the same node of the class hierarchy.

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 too. In Figure 41.b, if AUTO.5 is deleted, AUTO.DUMMY.6 is no longer necessary, and thus it is deleted automatically. The relationship between class versions and their parent class version is maintained in a derivation hierarchy data structure. When a class version is deleted, the new derivation hierarchy data structure must be modified in order to capture the relationship between the child class versions of the deleted class version and the parent of the deleted class version.

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 2.1.3) 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 V_1 of an object is created under class version C_1. Another object version V_2 can be derived from the object version V_1 under class version C_2. As shown in example 4.1, object version derivation in 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 V_i may be different from the class version that is used for an object version V_j which is derived from V_i, the class version for V_j should be identified by either the user or the system. There are three possible cases: (1) only one existing class version can accommodate V_j, (2) more than one existing class version can accommodate V_j, and (3) no existing class version can accommodate V_j.

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

DELETING AN INSTANCE VERSION

Each object version has pointers to its parent object version and child object versions. The user can retrieve parent or child object versions of a particular object version. In order to maintain the derivation hierarchy of object versions, the child object versions of a deleted object version become the child object

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

versions of the parent of the deleted object version. Suppose an object version V_i is the parent of V_j and V_j is the parent of V_k. If V_j is dropped, V_i becomes the new parent of V_k.

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': (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.

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

4.4 Integration with Chou and Kim's Object Version Model

H.T. Chou and W. Kim [CK86] suggested an object version model for distributed CAD databases. Their proposal broadly covers both semantics issues and operational issues in object version control and takes into account the

characteristics of CAD environments, such as the two-level database architecture and the design paradigm. To make our schema version model complete and practical, we integrate our schema version model and their object version model.

PUBLIC DATABASE vs. PRIVATE DATABASE

> 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. [CK86; 3]

TRANSIENT SCHEMA VERSION vs. WORKING SCHEMA VERSION

> There are 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. [CK86; 6-7]

We extend the above scenario in our schema version model. We want to treat class versions and object versions in the same manner in order to provide a consistent user interface.

As in object versions, a transient class version can be updated or deleted by 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**

[3] Some object version models [BK85b] 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; 7].

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 be installed first** [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. That is because the class hierarchy of the private database is different from that of the public database. The same argument is applied to V'.

Consider the example in Figure 42. As shown in Figure 42.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 $f1, g1$, and $c3$ are created in the private database as shown in Figure 42.b. After G and $c3$ are checked into the public database, their respective locations are different in the public and the private database as shown in Figure 42.c if E subsumes G.

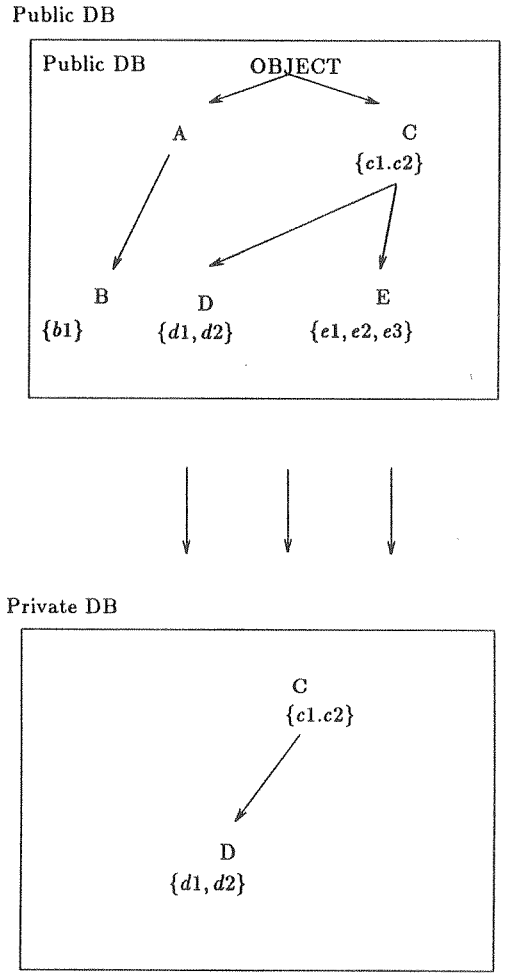
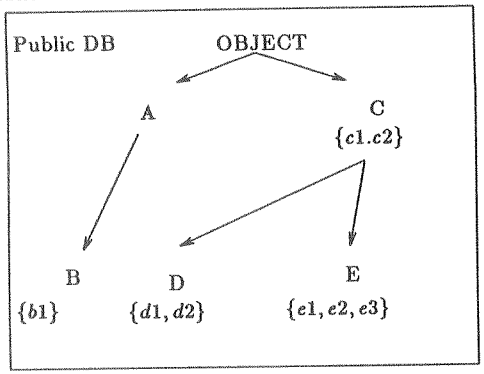


Figure 42.a: Check-Out Process

Public DB



Private DB

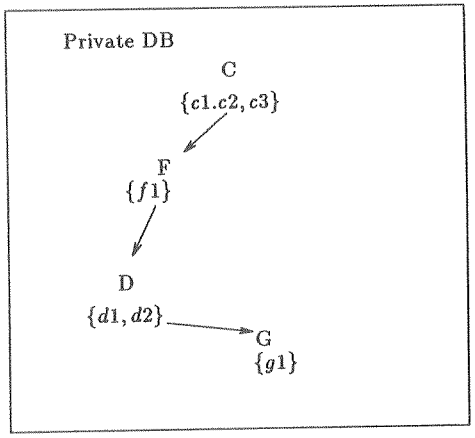


Figure 42.b: Creating New Class Versions and Instances

VERSION NAMING

In the H.T. Chou and W. Kim's object version model 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 database under the TRUCK.2 class version.

NAME BINDING

As in H.T. Chou and W. Kim's object version model, we 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. 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.

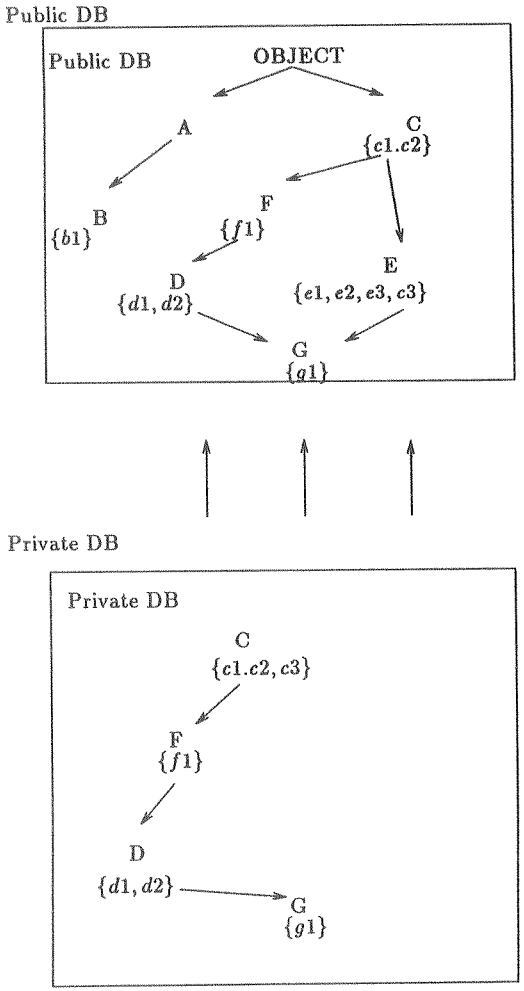


Figure 42.c: Check-In Process

- 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 envision that a class definition is itself an object. The domain of an instance variable is a class. Similarly a method of a class refers to other class names and instance variables of other classes. As such, we view definitions of instance variables and classes as objects to which other classes or methods can refer.

- As domains of instance variables, a transient or working class version in a private database may reference other transient or working class versions in the same private database, or working class versions in the public database. By similar reasoning, as domains of instance variables, a working class version in the public database may reference other working class versions in the public database. 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 of C.

- Additionally we have to consider the impact of each schema change on existing methods. Methods of a transient or working class version in a private database may reference definitions of transient or working class versions in the same private database, or working class versions in the public database. By similar reasoning, methods of a working class version in the public database may reference definitions of other working

class versions in the public database. 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 of C.

4.5 Operational Interface

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

- 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 2.1.3. $< OP >^+$ 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 class version S is promoted to a working class 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 $S.x$ and a class version $S.y$, etc.

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

4.6 Related Work

Skarra and Zdonik [SZ86, Zdo86] addressed the problem of maintaining consistency between a set of persistent objects and a set of type versions that can change. Their objective was to maintain the transparency of a type's change with respect to application programs that use the type.

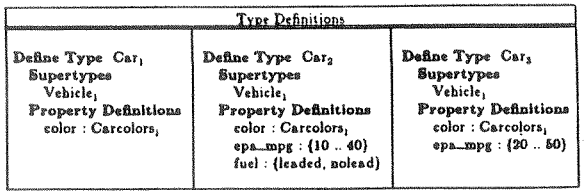


Figure 43.a: Three Type Versions of Car (from [SZ86])

Supertypes

Vehicle1

Property Definitions

color: Carcolors1

epa_mpg: {10..50}

fuel: {leaded, nolead}

Figure 43.b: The Version Set Interface (from [SZ86])

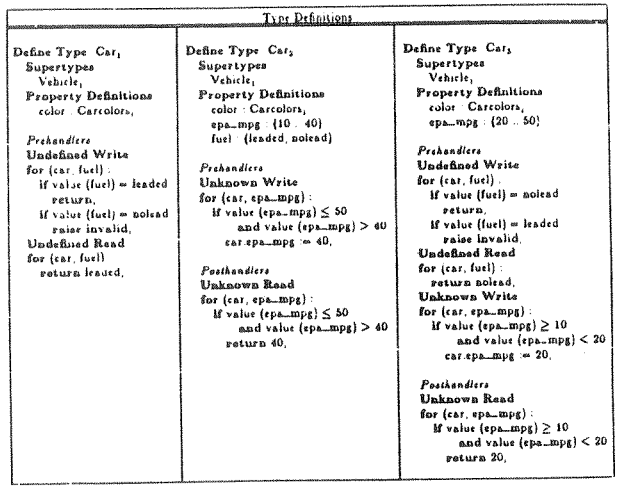


Figure 43.c: Type Versions with Added Error Handlers (from [SZ86])

They define the *version set interface* to be the most general interface to a type by constructing the disjunction of the definitions of all type versions of the type. A type version T_i is a triple $(op(T_i), pr(T_i), con(T_i))$ where $op(T_i)$ is a set of operations, $pr(T_i)$ is a set of properties, and and $con(T_i)$ is a set of constraints. Suppose we have n versions of T: $T_1, T_2, ..., T_n$. Then the version set interface V_T is defined as follows:

$$V_T = (op(V_T), pr(V_T), con(V_T)) =$$
$$(op(T_1) \cup op(T_2) \cup ... \cup op(T_n),$$
$$(pr(T_1) \cup pr(T_2) \cup ... \cup pr(T_n),$$
$$(con(T_1) \cup con(T_2) \cup ... \cup con(T_n))$$

The version set interface V_T contains all the operations, properties, and constraints ever defined by some version of a type T. An instance is physically stored under the type version in which the instance is created, but conceptually all instances are governed by the version set interface.

The basic idea of their approach is to add *error handlers* to each type version in order to allow instances of different type versions to be used uniformly. Error handlers are added to the definition of a type version to provide for behavior defined in the version set interface that is not defined in the type version. Error handlers are to be provided by the users; the users can derive the necessary error handlers by computing the difference between a type version and the version set interface.

The following example in Figure 43 [SK86] illustrates their approach: Figure 43.a and 43.b show the three car type versions and the version set interface of the car type respectively, and Figure 43.c shows the car type versions with the added error handlers. As shown in the Car1 definition of Figure 43.c, error handlers are added to the type versions so that instances of Car1 can be used by application programs expecting an instance of Car2: even if a program P is written for reading and writing instances of Car2, the program P also can

be used for reading and writing instances of Car1 with help of error handlers. There are two kinds of error handlers: *prehandlers* and *posthandlers*. A prehandler is defined on a type version and is used for processing instances of the type version, while a posthandler is used for processing instances of another type versions. They have described the details of error handlers [SZ86, Zdo86].

There are several drawbacks to Skarra and Zdonik's approach. First, they consider their approach as a view mechanism, but never take view update semantics into account. Second, since error handlers must be provided by the users, it may cause serious programming overhead in a complicated application. Third, they do not address the problem from a database perspective. For example, in their approach, when a new type version of a type T is created, new type versions of all subtypes of T are also created. That is not practical in a real world database.

Chapter 5
DAG Rearrangement Views

5.1 Motivation of DAG Rearrangement Views

Schema versioning is a means of maintaining a history of schema changes by keeping versions of schema. Another way to represent variants and alternatives of a schema is to allow views of schema. Views in object-oriented databases are more versatile than those in relational databases. Two distinct notions in object-oriented data models are *composite objects* and *class hierarchies*. The semantics of the two notions are captured in directed acyclic graphs (DAGs). 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 chapter, we present sets of useful 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 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., a 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. A DAG rearrangement view facility is useful for this purpose. 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 chapter, the granularity of schema versions was a single class, not a class hierarchy. With a DAG rearrangement view facility, we can keep several versions of a class hierarchy in an inexpensive way.

5.2 DAG Rearrangement Views on Composite Objects

In this section, we elaborate on DAG rearrangement views of composite objects. As we have seen, 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. One important premise behind DAG rearrangement views of composite objects is that designers (database users) would like to see only those parts of composite objects which are necessary to their applications.

Figure 44.a shows a part hierarchy for vehicle composite objects. Figure 44.a is a simplified form of Figure 4 without description of the instance variables. (In the remainder of this chapter, Figure 3 and 4 are frequently referenced. Therefore, we repeat those figures again here.) Figure 44.b, 44.c, and 44.d show three possible views of the vehicle composite objects of Figure 44.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.

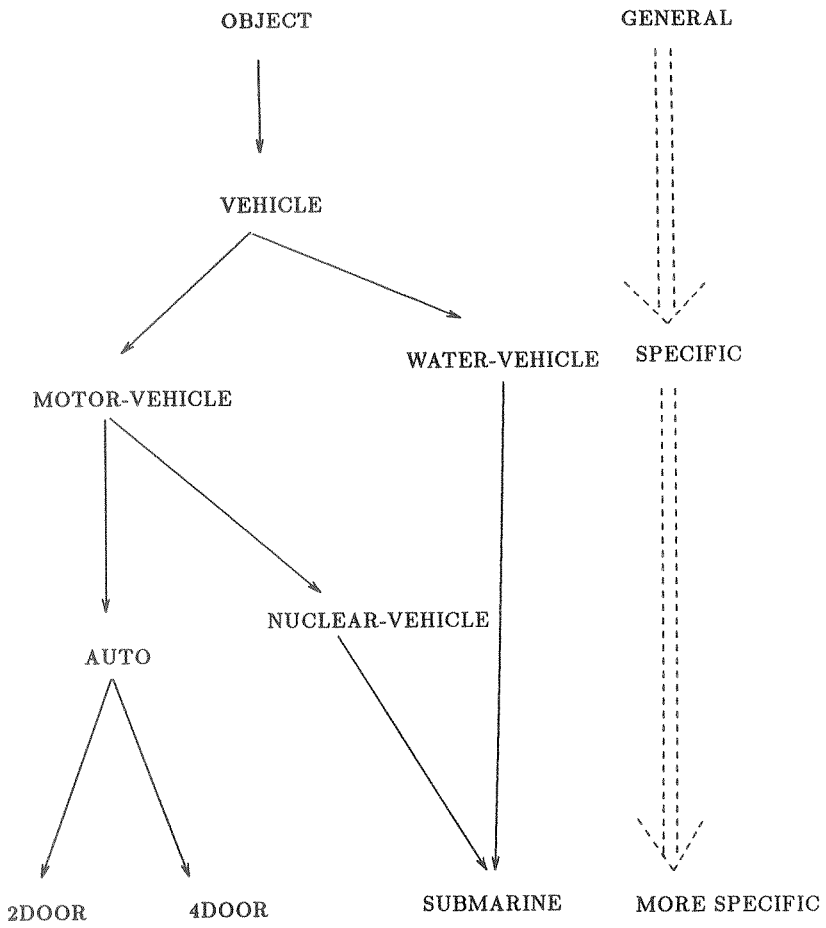


Figure 3: VEHICLE Class Hierarchy

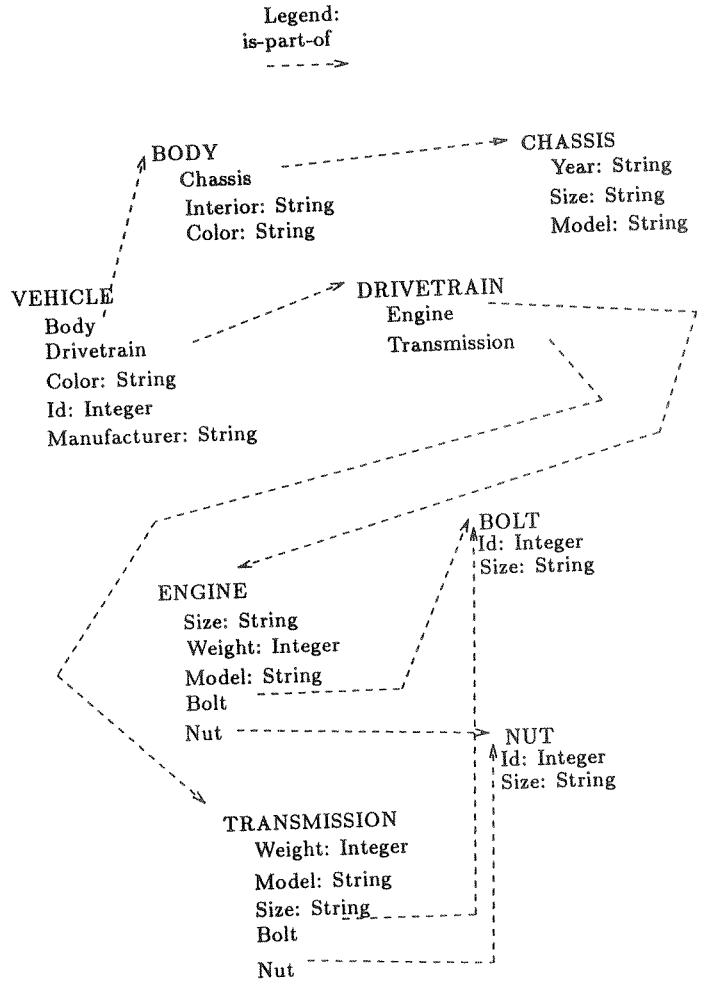


Figure 4: VEHICLE composite object

It turns out that not all possible DAG rearrangement views of composite objects are acceptable because queries to certain DAG rearrangement views are not processable on instances disciplined by the original composite object schema. We shall identify DAG rearrangement views for which queries can be processed 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 which is a hierarchical database system from IBM, query qualification predicates may involve only parent record types of a target record type or the target record type itself (see, e.g., [KS86]). In System 2000, a hierarchical database system from MRI, query qualification predicates may involve only parent record types or child record types of a target record type or the target record type itself [MRI78]. Therefore, the target record type and the record types which are involved in the query qualification predicates must be located in only one leaf-to-root path. The reasons for those restrictions are *query processing overhead* and *query processing ambiguity*. Without those restrictions, there may be ambiguity in evaluating queries because of possible multiple paths between the target record types and the record types of query predicates. Ambiguity can be resolved by the user or the system, but this may lead to processing overhead. One approach is to let the user select the desired path through dialogs. Though not a hierarchical database system, System/U [Kor84] (a universal relational database system) resolves the multiple path problem by returning the union of query results from all possible paths.

The similar restrictions are assumed in querying composite objects in object-oriented data bases because the structure of a composite object is DAG. There are 3 types of access patterns in hierarchical data 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.

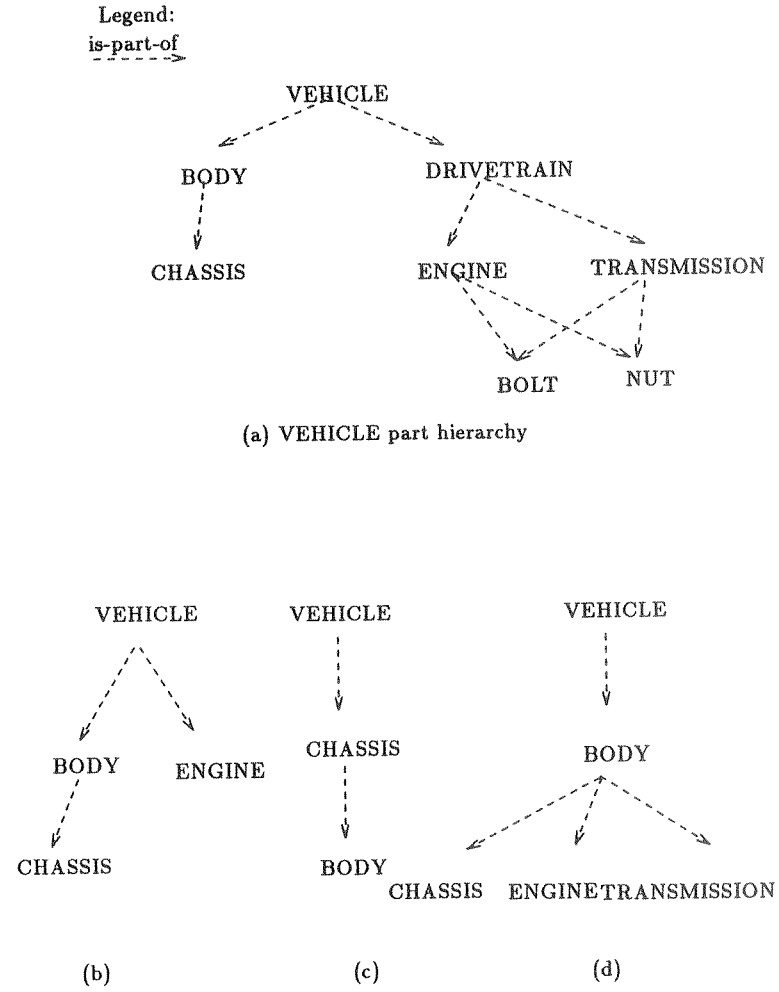


Figure 44: DAG Rearrangement Views of VEHICLE Composite Object

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"*

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 44. As Figure 44.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 44.b because all possible queries on the view are processable on instances disciplined by the vehicle composite object schema. However, childpart-only traversal cannot be used on the view in Figure 44.c because some queries on the view are not processable 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 44.a because CHASSIS is not a parentpart of BODY. By similar reasoning, the view in Figure 44.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 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.

CLAIM: Let S and S' denote the original schema of a composite object and a DAG rearrangement view on S respectively. S' is a PCOV 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.

CLAIM: 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 a PCOV iff S' is a sub-DAG of TR(S).

Now we consider two useful operations for constructing PCOVs.

- (V1) Hide a part P: The part P is not visible and the 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

CLAIM: 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. Note that Q1 and Q2 follow the access pattern rule and Q3 do not. System 2000 uses this childpart-parentpart traversal as its primary access pattern rule.

The childpart-parentpart traversal scheme can be used with the DAG rearrangement views in Figure 44.b and 44.c, but not with the DAG rearrangement view in Figure 44.d. The childpart-parentpart traversal scheme cannot be used with the view of Figure 44.d because some queries on this view cannot be processed in the instances of the original VEHICLE composite object schema. Note that the childpart-only traversal cannot be used on the DAG rearrangement view in Figure 44.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 44.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 44.a.

We characterize PCOVs for childpart-parentpart traversal as follows.

CLAIM: Let S and S' denote the original schema of a composite object and a DAG rearrangement view on S respectively. S' is a PCOV 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 of the previous access pattern.

- (V1) Hide a part P: same as V1 above
- (V2) Make a part P an immediate childpart of one of P's parentparts: same as V2 above
- (V3) Exchange parts: As shown in the view of Figure 44.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. To our knowledge, no existing hierarchical database system allows free traversal because of query processing ambiguity and overhead due to multiple leaf-to-root paths between target attributes and query qualification attributes. However, by supporting disambiguating dialogue or user access path specification, processing of this access pattern is possible. In free traversal, queries Q1, Q2, and Q3 are all allowed. Arbitrary DAG rearrangements views including the views in Figure 44.b, 44.c, and 44.d are allowed. For instance, consider the query GET CHASSIS.Model WHERE ENGINE.Model = "320CI" in the view in Figure 44.d. The query processor tries to find a path from ENGINE to CHASSIS in the original VEHICLE schema. If there is more than one path between ENGINE and CHASSIS, the user is responsible for selecting one access path or the system may return the union of results from each path as System/U does.

CLAIM: Let COT(S) denote the PCOV set of childpart-only traversal of a composite object schema S. Let CPT(S) denote the PCOV set of childpart-parentpart traversal of a composite object schema S. Let FT(S) denote the PCOV set of free traversal of a composite object schema S. The following inclusion relationship is obvious: COT(S) $\subseteq$ CPT(S) $\subseteq$ FT(S).

5.3 DAG Rearrangement Views on Class Hierarchies

In this section we elaborate on DAG rearrangement views of class hierarchies. As mentioned earlier, the notion of class hierarchy explicitly captures ISA relationships and a class hierarchy has a DAG structure. One important premise behind DAG rearrangement views of class hierarchies is that users would like to see only those classes which are relevant to their applications.

Figure 3 shows the class hierarchy of VEHICLE. Figures 45.a-d show possible DAG rearrangement views of the VEHICLE class hierarchy. Each view in Figure 45 involves several DAG rearrangement operations. 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 through the view 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 2.1.3.

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. Therefore, values of an instance variable of instances of C and

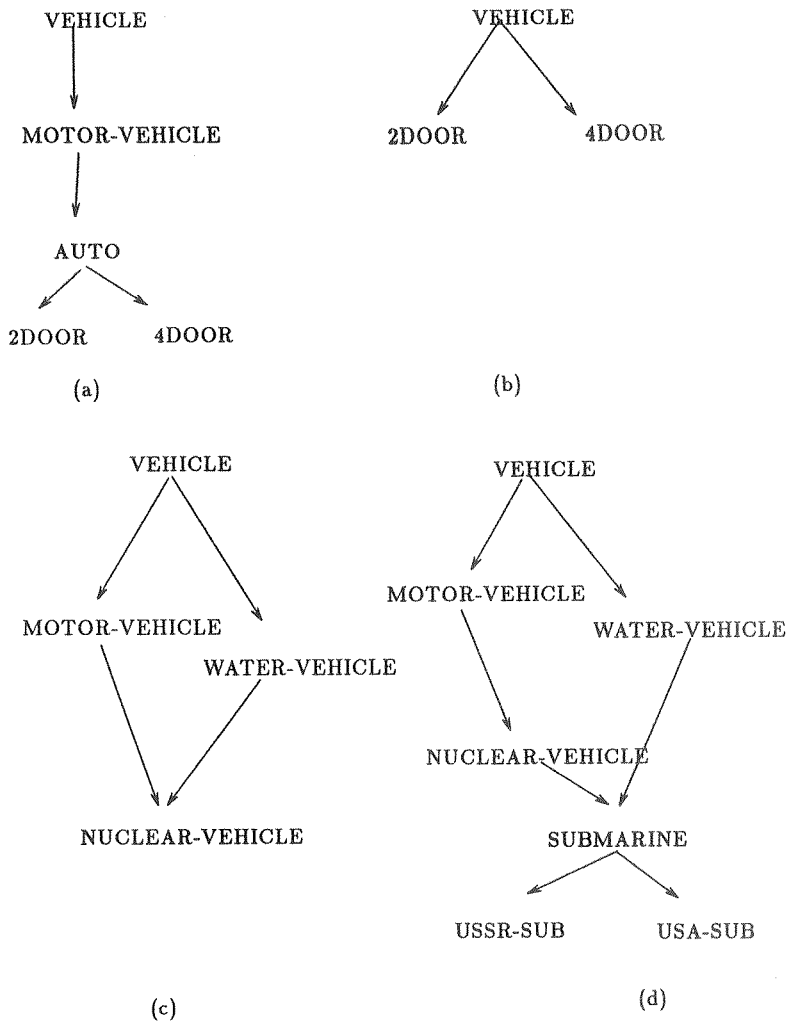


Figure 45: DAG Rearrangement Views of VEHICLE Class Hierarchy

subclasses of C are hidden if the instance variable is inherited from S. Further, instances of C and subclasses of C lose membership (role) 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 2.1.3.

3. Make an ISA relationship explicit: Suppose S is the only superclass of C, and C, in turn, has one subclass 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 2.1.3, 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 2.1.3.

The semantics of the above operations will be clearer after we go through some examples. Now we show how instances are viewed and reorganized in the views which are constructed by using the operations. Suppose classes in the

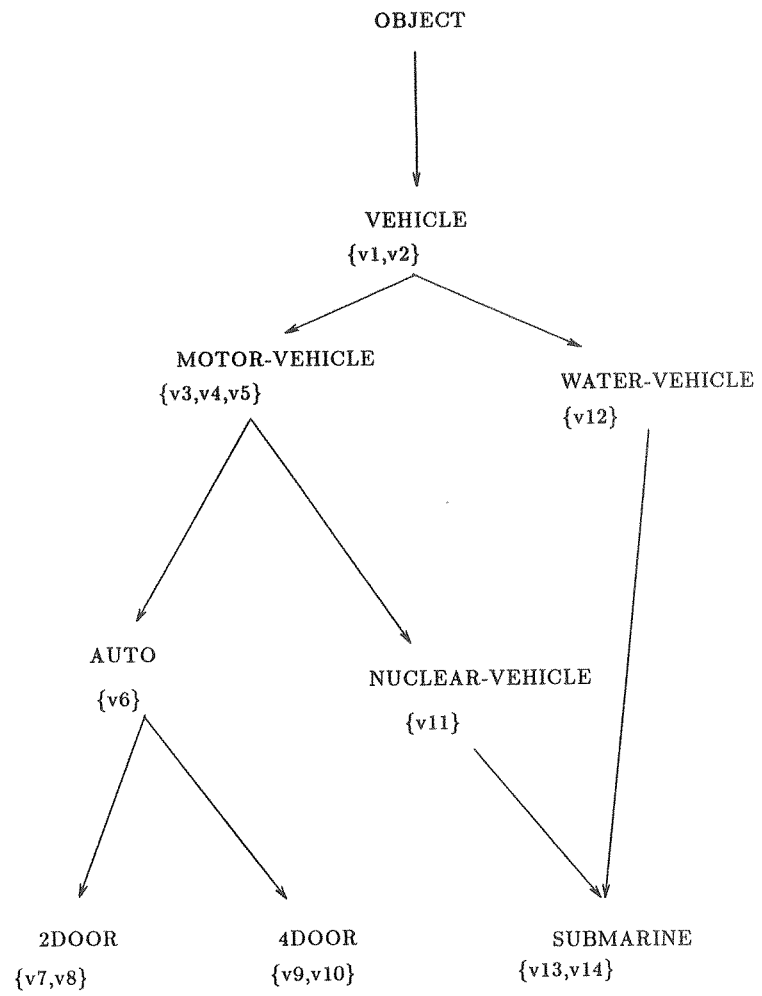
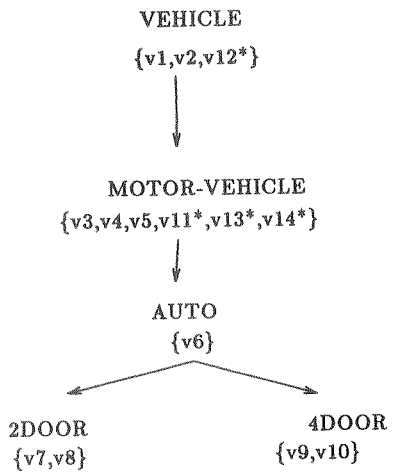
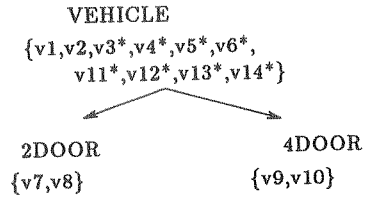


Figure 46: A Sample Database of VEHICLE Class Hierarchy



(a)



(b)

Figure 47: Viewing Instances in the DAG Rearrangement Views of VEHICLE Class Hierarchy

VEHICLE class hierarchy in Figure 3 have the following instances (I_C denotes a set of instances of C) as shown in Figure 46:

- $I_{VEHICLE}$ = { v1, v2 }
- $I_{MOTOR\text{-}VEHICLE}$ = { v3, v4, v5 }
- I_{AUTO} = { v6 }
- I_{2DOOR} = { v7, v8 }
- I_{4DOOR} = { v9, v10 }
- $I_{NUCLEAR\text{-}VEHICLE}$ = { v11 }
- $I_{WATER\text{-}VEHICLE}$ = { v12 }
- $I_{SUBMARINE}$ = { v13, v14 }

• The DAG rearrangement view in Figure 45.a is constructed by the following sequence of operations:

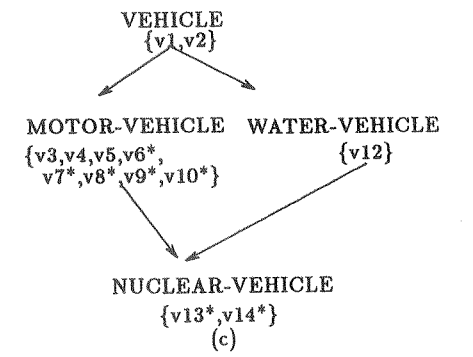
(1) Hide SUBMARINE

(2) Hide NUCLEAR-VEHICLE

(3) Hide WATER-VEHICLE

As shown in Figure 47.a, 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.
- $I_{MOTOR\text{-}VEHICLE}$ = { v3, v4, v5, v11*, v13*, v14* } where v11*, v13*, and v14* are a projected form of v11, v13, and v14 respectively under the MOTOR-VEHICLE class definition.
- I_{AUTO} = { v6 }
- I_{2DOOR} = { v7, v8 }



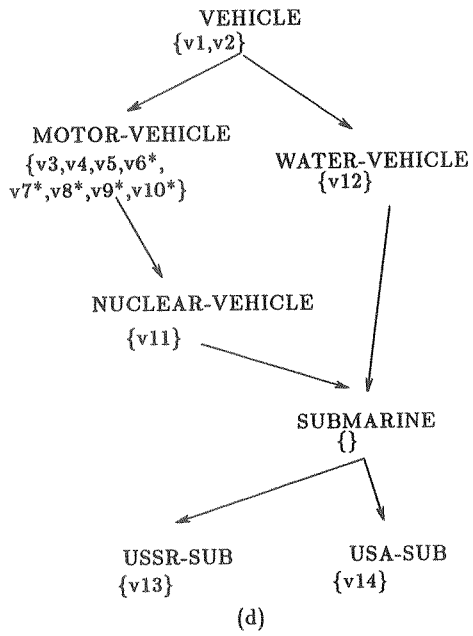


Figure 47 (Cont'd): Viewing Instances in the DAG Rearrangement Views of VEHICLE Class Hierarchy

- I_{4DOOR} = { v9, v10 }

• The DAG rearrangement view in Figure 45.b is constructed by the following sequence of operations:

(1) Hide SUBMARINE

(2) Hide NUCLEAR-VEHICLE

(3) Hide WATER-VEHICLE

(4) Hide MOTOR-VEHICLE

(5) Hide AUTO

(6) Make VEHICLE an immediate superclass of 2DOOR

(7) Make VEHICLE an immediate superclass of 4DOOR

As shown in Figure 47.b, 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 45.c is constructed by the following sequence of operations:

(1) Hide 2DOOR

(2) Hide 4DOOR

(3) Hide AUTO

(4) Hide SUBMARINE

(5) Make NUCLEAR-VEHICLE an immediate superclass of WATER-VEHICLE.

As shown in Figure 47.c, the following instances are visible from each class in this view:

- $I_{VEHICLE}$ = { v1, v2 }
- $I_{MOTOR\text{-}VEHICLE}$ = { 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 MOTOR-VEHICLE class definition.
- $I_{NUCLEAR\text{-}VEHICLE}$ = { v13*, v14* } where v13* and v14* are a projected form of v13 and v14 respectively under the NUCLEAR-VEHICLE class definition. (Note that v11 is dropped from the instance set of NUCLEAR-VEHICLE. In the original class hierarchy, v11 does not have the role of WATER-VEHICLE. As such, v11 should not be visible from the NUCLEAR-VEHICLE class in this view because only instances with the roles of WATER-VEHICLE and NUCLEAR-VEHICLE are qualified as instances of NUCLEAR-VEHICLE)
- $I_{WATER\text{-}VEHICLE}$ = { v12 }

• The DAG rearrangement view in Figure 45.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

As shown in Figure 47.d, the following instances are visible from each class in this view:

- $I_{VEHICLE}$ = { v1, v2 }
- $I_{MOTOR\text{-}VEHICLE}$ = { 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 MOTOR-VEHICLE class definition.
- $I_{NUCLEAR\text{-}VEHICLE}$ = { v11 }
- $I_{WATER\text{-}VEHICLE}$ = { v12 }
- $I_{SUBMARINE}$ = { }
- $I_{USSR\text{-}SUB}$ = { v13 } (assume that v13 is a submarine made in USSR)
- $I_{USA\text{-}SUB}$ = { v14 } (assume that v14 is a submarine made in USA)

5.4 Operational Interface

We present a preliminary operational interface (user commands) for supporting our model DAG rearrangement views in object-oriented databases.

- 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>.

Chapter 6
Logical Design of Object-Oriented Database Schema

In this chapter we establish a unified framework for object-oriented database schema design by synthesizing research results in the areas of AI knowledge representation, database dependency theory, AI theorem proving, and graph algorithms.

6.1 The Unified Framework

The notion of generalization (class hierarchy) is borrowed from the knowledge representation area of AI. In object-oriented databases, objects are classified within class hierarchies. Class hierarchies in knowledge representation schemes may not necessarily have to be accurate due to the heuristic nature of AI, whereas class hierarchies in object-oriented databases must be accurate. Object-oriented database schemas should be *consistent* and *non-redundant* (to be addressed shortly). Object-oriented database schemas tend to be modified frequently during the lifetime of a database and users tend to arrive at a preliminary design through trial and error using the schema change operations [BKKK87]. After the user modifies the class hierarchy, the resulting class hierarchy must also be in a consistent and non-redundant state.

Object-Oriented Database Design Steps

We view object-oriented database schema design as an iterative process. Steps 4, 5, and 6 will be repeated iteratively during the lifetime of a database.

(Step 1) Initial Design: The user specifies a collection of classes and a set of constraints among them. Each class definition consists of a set of superclasses, a set of instance variables, and a set of methods. Constraints we will consider are ISA, Disjointness, and Covering constraints (to be defined shortly).

(Step 2) Class Compilation: Each class definition is compiled so that each class inherits instance variables and methods from its superclasses. During compilation, conflicts among inherited instance variables (and methods) and locally defined ones are resolved by a given set of conflict resolution rules. The compiled version of a class consists of only a set of instance variables and a set of methods, without a superclass declaration.

(Step 3) Schema Verification: The initial design is checked for consistency and non-redundancy. For each declared constraint, an appropriate verification should be performed. Typical verification tasks in Step 3 are: "Is the set of constraints consistent?", "When a user declares A $\stackrel{isa}{\Longrightarrow}$ B, can A really become a subclass of B?", "Are there any equivalent classes or redundant ISA relationships in the set of constraints?", etc.

(Step 4) Schema Querying: During the lifetime of a database, the user can query against a schema and the constraints on the schema. This step is necessary for the user to understand the current class hierarchy and prepare the next schema change operation. Typical queries raised by the user in Step 4 are: "Is A a subclass of B?", "What are subclasses of A?", "Is A disjoint with B?", etc.

(Step 5) Schema Modification: Typical schema change operations are: "create a new class", "create a new ISA relationship among classes", "drop an existing class", etc. Schema change operations in the taxonomy in section 2.1.3 can be all applied against the schema. Constraints on the schema also can be changed.

(Step 6) Schema Verification: The resulting schema from schema change operations and constraint modifications must be checked for consistency and non-redundancy. Typical verification tasks in step 6 are same as those in step 3.

Constraints in Object-Oriented Databases

In object-oriented databases, the following four types of constraints arise. The user declares constraints initially and modifies them later. The system verifies whether the constraints are correctly declared and modified. The user can ask the system if a particular constraint can be derived from a given set of constraints. Let ISET(X) denote a set of instances of a class X.

(1) Single Inheritance Constraint (SIC): Given two classes A and B, A $\stackrel{isa}{\Longrightarrow}$ B means that A is a subclass of B and every instance in A is an instance of B. Consequently, ISET(A) $\subseteq$ ISET(B).

(2) Multiple Inheritance Constraint (MIC): Given three classes A,B, and C A $\stackrel{isa}{\Longrightarrow}$ B $\cap$ C means that A is a subclass of B and also a subclass of C and every object common in B and C is an object of A. A $\stackrel{isa}{\Longrightarrow}$ B $\cap$ C implies (A $\stackrel{isa}{\Longrightarrow}$ B) $\wedge$ (A $\stackrel{isa}{\Longrightarrow}$ C). Consequently ISET(A) $\subseteq$ ISET(B) $\cap$ ISET(C).

(3) Disjointness Constraint (DC): Given two classes A and B, A $\stackrel{disjoint}{\Longleftrightarrow}$ B means that there is no common object in A and B. Therefore, ISET(A) $\cap$ ISET(B) = $\emptyset$.

(4) Covering Constraint (CC): Given three classes, A,B, and C and two SICs, B $\stackrel{isa}{\Longrightarrow}$ A and C $\stackrel{isa}{\Longrightarrow}$ A, A $\stackrel{cover}{\Longrightarrow}$ B $\cup$ C means that every instance of A must be either an instance of B or an instance of C. Therefore, ISET(A) $\subseteq$ ISET(B) $\cup$ ISET(C). From the given SICs, ISET(A) $\supseteq$ ISET(B) and ISET(A) $\supseteq$ ISET(C). Therefore, ISET(A) $\supseteq$ ISET(B) $\cup$ ISET(C). Hence, ISET(A) = ISET(B) $\cup$ ISET(C).

Disjointness constraints and covering constraints are first suggested by Israel and Brachman [IB84] and formal properties of them are investigated by Lenzerini [Len87] and Atzeni and Parker [AP86].

Example 6.1: The VEHICLE class hierarchy in Figure 48 illustrates the above 4 notions. A dotted arrow means a disjointness constraint and a normal arrow means an ISA constraint. An arc means a covering constraint. AUTO, WATER-VEHICLE, and NUCLEAR-VEHICLE are subclasses of VEHICLE

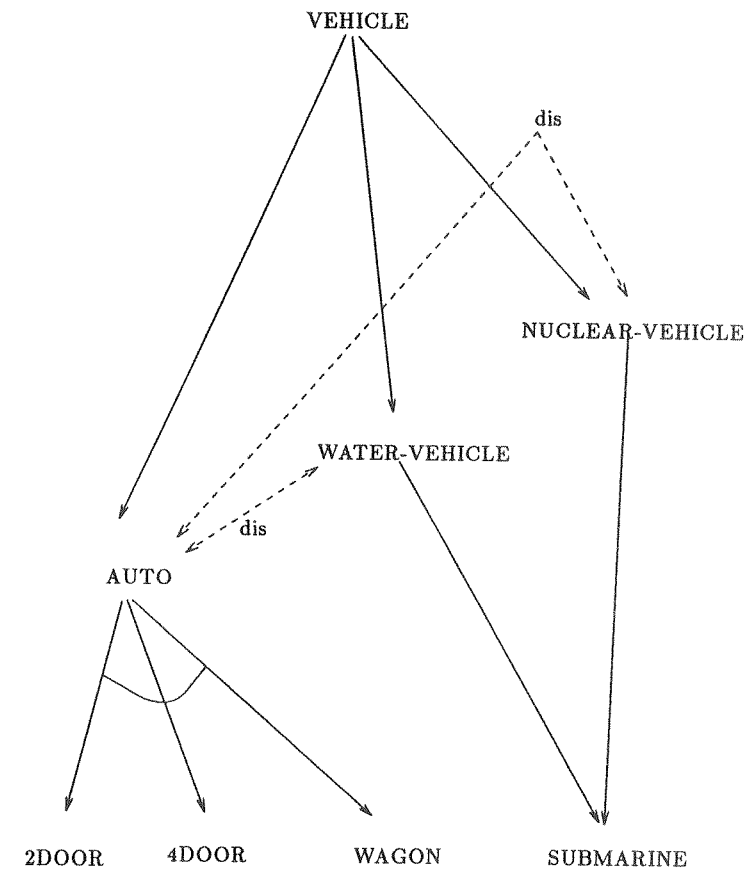


Figure 48: A sample class hierarchy with various constraints

(SICs). SUBMARINE is a subclass of NUCLEAR-VEHICLE and WATER-VEHICLE (MIC). AUTO is disjoint with WATER-VEHICLE and NUCLEAR-VEHICLE (DCs) (i.e., any instance of AUTO must not be an instance of WATER-VEHICLE nor an instance of NUCLEAR-VEHICLE). AUTO covers 2DOOR, 4DOOR, and WAGON (CC) (i.e., every instance of AUTO must be an instance of 2DOOR, 4DOOR, or WAGON). □

Three Fundamental Problems

The following three fundamental problems reside in the core of object-oriented database design.

- **Computation among Type Descriptions**: If the user declares a constraint among classes, the system should make sure that the constraint is really true. We consider three types of computation among type descriptions. We shall use T(C) for denoting type description of a class C. First, given two class definitions T(A) and T(B), if the user declares A $\stackrel{isa}{\Longrightarrow}$ B, then T(A) should be subsumed by T(B): that is, suppose λ(A) and λ(B) are both formulas of a type representation language, a new formula $\lambda(A) \rightarrow \lambda(B)$ should hold where $\rightarrow$ means 'implication'. We call this, the "*type subsumption problem*". Second, given two class definitions T(A) and T(B), if the user declares A $\stackrel{disjoint}{\Longleftrightarrow}$ B, then T(A) and T(B) must be disjoint: that is, suppose λ(A) and λ(B) are both formulas of a type representation language, a new formula $\lambda(A) \rightarrow \neg \lambda(B)$ (or $\lambda(B) \rightarrow \neg \lambda(A)$) should hold. We call this, the "*type disjointness problem*". Third, given three class definitions T(A), T(B) and T(C), if the user declares A $\stackrel{cover}{\Longrightarrow}$ B $\cup$ C, then T(A) must be equivalent to T(B) $\cup$ T(C): that is, suppose λ(A), λ(B), and λ(C) are all formulas of a type representation language, a new formula $(\lambda(A) \rightarrow \lambda(B) \vee \lambda(C)) \wedge (\lambda(B) \vee \lambda(C) \rightarrow \lambda(A))$ should hold. We call this, the "*type covering problem*".

  The common core in the three type computation problems (type subsumption, type disjointness, and type covering) is essentially to *prove a*

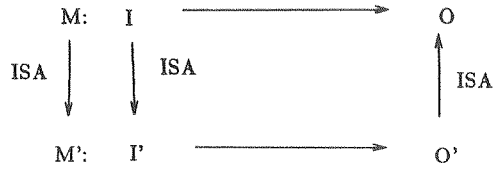


Figure 49: Subsumption between methods

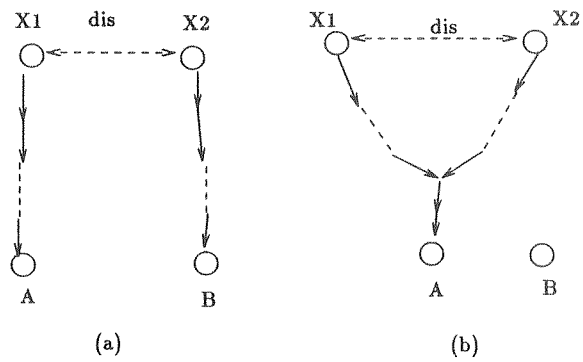


Figure 50: Disjointness constraint derivations (from [AP86])

well formed formula in a type-description language. Further, the type disjointness problem and the type covering problem can be viewed as special cases of the type subsumption problem. Therefore it suffices to consider only the type subsumption problem. As such, we shall give an in-depth review only on the **Type Subsumption Problem** in section 6.2.1.

- **Constraint Membership Problem:** Given a set of constraints provided by the user, new constraints can be derived using the associated inference rules. As mentioned in step 4 of the database design, queries against a class hierarchy and the constraints on the class hierarchy can be posed by the user. The queries are essentially to check if a particular constraint is a member of the closure of a given set of constraints (i.e., to check if the particular constraint can be derived from the given set of constraints). For example, suppose the user asks the system whether SUBMARINE is a subclass of VEHICLE in the VEHICLE class hierarchy of Figure 48. In order to process the query, the system needs to check whether SUBMARINE $\rightarrow$ VEHICLE is a member of the closure of ISA relationships in the VEHICLE class hierarchy of Figure 48.

- **Undesirable Property Detection Problem:** There are three undesirable properties in class hierarchy design. First, consider the situation such that given three classes A, B, and C, the user declares A is a subclass of B, A is also a subclass of C, and B is disjoint with C. Then clearly, the class A is invariably empty. We call such a schema *inconsistent*. Thus a class hierarchy with a class that cannot have an instance is inconsistent. Second, consider the situation such that given three classes A, B, and C, the user declares A is a subclass of B, B is a subclass of C, and C is a subclass of A. This situation is undesirable in that A, B, and C are actually the same class. Since a class hierarchy must be a DAG (directed acyclic graph), cyclic structures must be avoided. We call such classes *redundant classes*. Third, consider the situation such that given three classes A, B, and C, the user declares A is a subclass of B, B is a subclass of C, and finally A is a subclass of C. Since ISA relationship is transitive, the fact "A is a subclass of C" is redundant in the sense that it can be derived from other ISA relationships. We call this a *redundant ISA*. In summary, inconsistent schema, redundant classes, and redundant ISAs should be all detected and avoided for designing a desirable object-oriented database schema.

We note that Constraint Membership Problem is associated with Step 4 whereas Type Subsumption Problem and Undesirable Property Detection Problem are associated with Step 3 and Step 6. We examine below each of the 3 problems in greater depth. We believe that the above 3 problems should be solved automatically by the system, not by the database designer. Fortunately, the solutions for the 3 problems have been scattered in several areas such as AI knowledge representation, database dependency theory, AI theorem proving and graph algorithms.

6.1.1 Type Subsumption Problem

As we mentioned earlier, the type subsumption problem is to prove an implication formula ($\alpha \rightarrow \beta$, where α and β are formulas of a type-description language) written in a type-description language. An important issue here is that there is a fundamental tradeoff between the expressive power of the type representation language and the computational complexity of type subsumption decision of the implication formula.

If a type representation language has a full power of first order logic, the decision problem of type subsumption is ***undecidable*** because deciding the truth value of arbitrary first order logic formula is undecidable. In the knowledge representation area, several approaches have been made to find a type representation language which has both a reasonable expressive power and a reasonable computational overhead.

We introduce a recent result by Levesque and Brachman [LB86]. Levesque and Brachman designed 2 type-description languages that are almost

same at a first glance. However, the computational complexity of type subsumption in one language is $O(n^2)$ whereas the other language has intractable (exponential) time complexity. Here are the BNF forms of the two type representation languages.

(TR-1)

< *type* > ::= < *atom* >

| (AND < *type1* > ... < *type2* >)

| (ALL < *attribute* > < *type* >)

| (SOME < *attribute* >)

< *attribute* > ::= < *atom* >

| (RESTRICT < *attribute* > < *type* >)

(TR-2)

< *type* > ::= < *atom* >

| (AND < *type1* > ... < *type2* >)

| (ALL < *attribute* > < *type* >)

| (SOME < *attribute* >)

< *attribute* > ::= < *atom* >

- Note: Atoms are primitive types. The semantics of AND is that x is an (AND t_1 t_2 ... t_n) iff x is a t_1 and a t_2 and ... and a t_n. ALL and SOME restricts values of an attribute, e.g., (x is an (ALL a t) iff each a of x is a t, that is, the domain of attribute a of the type x is a type t), (x is a (SOME a) iff x has at least one a). RESTRICT constrains attributes by the types of their values, e.g., (y is a (RESTRICT a t) of x iff y is an a of x and y is a t).

Example 6.2: [BL84] Suppose we have five given types: PERSON, MALE, FEMALE, LAWYER, and DOCTOR. Let Child be an attribute of the type PERSON. Then, TR-1 representation of "person with at least one child, and each of whose sons is a lawyer and each of whose daughters is a doctor" is:

```
(AND PERSON
        (SOME Child)
        (ALL (RESTR Child MALE) LAWYER)
        (ALL (RESTR Child FEMALE) DOCTOR))
```

□

Example 6.3: [BL84] Assume that we have two more basic types RICH and SURGERY and Specialty is an attribute of the type DOCTOR. Let D1 correspond to "person each of whose children is a doctor". Let D2 correspond to "person each of whose children is rich, and a male each of whose rich children is a doctor who has a surgery specialty". D1 subsumes D2 (i.e., D1 is more general than D2 and D2 is a subclass of D1). TR-1 representations of D1 and D2 are:

```
D1 = (AND PERSON
        (ALL Child DOCTOR))
D2 = (AND
        (AND PERSON
                (ALL Child RICH))
        (AND MALE
                (ALL (RESTR Child RICH)
                        (AND DOCTOR
                                (SOME (RESTR Specialty SURGERY)))))
```

□

Theorem 6.1: [LB86] Type subsumption in TR-1 is NP-complete.

(Proof) Levesque and Brachman defined a mapping from propositional formulas in conjunctive normal form to languages in TR-1. □

Theorem 6.2: [LB86] Type subsumption in TR-2 is $O(n^2)$ where n is the element number of the longest type.

(Proof) The following is from [LB86]:

Subsumption Algorithm for TR-2: SUBSUME?(exp_1, exp_2)

1. Flatten both exp_1 and exp_2 by removing all nested AND operators. So, for example, (AND x (AND y z) w) becomes (AND x y z w).
2. Collect all arguments to an ALL for a given role. For example, (AND (ALL r (AND a b c)) (ALL r (AND e f))) becomes (AND (ALL r (AND a b c e f))).
3. Assuming exp_1 is now (AND a_1 ... a_n) and exp_2 is (AND b_1 ... b_m), then return true iff for each a_i,
   a. if a_i is an atom, then one of the b_j is subsumed by a_i.
   b. if a_i is a SOME, then one of the b_j is subsumed by a_i.
   c. if a_i is (ALL r x), then one of b_j is (ALL r y), where SUBSUME?(x, y).

Clearly, step 1 is linear time. Step 2 requires traversals on exp_1 and exp_2. Step 3.a and 3.b are a step of comparing atomic types which are represented in a table or a type graph (class hierarchy for atomic types). This step can be solved in two ways: (1) using inference rules and (2) using DAG traversal. Both solutions are linear time.

- Inference Rules: (1.) Given a type A, A $\stackrel{isa}{\Longrightarrow}$ A, (2.) Given three types A,B, and C, if A $\stackrel{isa}{\Longrightarrow}$ B and B $\stackrel{isa}{\Longrightarrow}$ C, then A $\stackrel{isa}{\Longrightarrow}$ C (i.e., transitivity). Subsumption problem between two atoms is just a membership checking problem.
- Graph Traversal: If atoms are nodes of a DAG, subsumption problem between two atoms is just a reachability problem.

Step 3.c requires a traversal of exp_1 for each element of exp_1. Step 2 and 3.c can be done in $O(n^2)$ time where n is the element number of the longest expression. □

Normally, a class definition is composed of a set of superclasses, a set of instance variables, and a set of methods (operations). TR-1 and TR-2 can represent only superclasses and instance variables. Therefore, a mechanism for deciding subsumption between methods is needed. Deciding subsumption between methods can be understood as follows. We follow the definition of method subsumption in [Car83].

Definition 6.1: A method M has a set of input parameters I_1, I_2,..., I_n and a set of output parameters O_1, O_2,..., O_n. i.e., M: domain(I_1) × domain(I_2) ... × domain(I_n) ⇒ domain(O_1) × domain(O_2) ... × domain(O_n). Let M': domain(I'_1) × domain(I'_2) ... × domain(I'_n) ⇒ domain(O'_1) × domain(O'_2) ... × domain(O'_n). Then M subsumes M' iff for all i, domain(I'_i) subsumes domain(I_i) and domain(O_i) subsumes domain(O'_i) respectively.

Figure 49 shows the subsumption between methods. Obviously the computational overhead for method subsumption depends on the representation language for input and output parameters. Hence we can make use of TR-1 or TR-2 for representing methods.

Example 6.4: [Car83] Let M: VEHICLE → VEHICLE and M': CAR → OBJECT. Since VEHICLE subsumes CAR (i.e., CAR ISA VEHICLE) and OBJECT subsumes VEHICLE (i.e., VEHICLE ISA OBJECT), M subsumes M' (i.e., M' ISA M). □

6.1.2 Constraint Membership Problem

We can characterize the constraints membership problem in three different formal systems: inference rule system, first order logic system, and graph theory system.

- **Inference Rule System**

The 4 types of constraints have their own inference rules. The most important issues in an inference rule system is are whether every valid constraint can be generated by the inference rules (completeness), and whether the inference rules generate only valid constraints (soundness). Fortunately, the soundness and completeness of the inference rules for the four constraints (SIC, MIC, CC and DC) were proved in the literature [AM86, AP86, Len87]. Another important issue is how fast a new constraint can be derived from a given set of constraints. The computational complexities of the membership checking algorithms were also investigated [AM86].

Definition 6.2: We call a set of constraints given by the user *CSET*. The definitions of inference rules for each constraint are as follow [AM86, AP86, Len87]:

(1) Inference Rules for SIC:

SIC-R1: Given a class A, A $\stackrel{isa}{\Longrightarrow}$ A

SIC-R2: Given three classes A, B, and C, if A $\stackrel{isa}{\Longrightarrow}$ B and B $\stackrel{isa}{\Longrightarrow}$ C, then A $\stackrel{isa}{\Longrightarrow}$ C (i.e., transitivity)

SIC-R3: Given classes A,B_1,B_2,...,B_n, if A $\stackrel{isa}{\Longrightarrow}$ $B_1 \cap B_2 \cap ... \cap B_n$, then A $\stackrel{isa}{\Longrightarrow}$ B_i for every $i = 1, ..., n$

(2) Inference Rules for MIC:

MIC-R1: Given classes A,B_1,B_2,...,B_n, if A $\stackrel{isa}{\Longrightarrow}$ B_i for every $i = 1, ..., n$. then A $\stackrel{isa}{\Longrightarrow}$ $B_1 \cap B_2 \cap ... \cap B_n$

(3) Inference Rules for CC:

CC-R1: Let G be a set of classes. If A $\in$ G, A $\stackrel{cover}{\Longrightarrow}$ G

CC-R2: Let G1 and G2 be sets of classes, if A $\stackrel{cover}{\Longrightarrow}$ G1 and B $\stackrel{cover}{\Longrightarrow}$ G2 and B $\in$ G1, then A $\stackrel{cover}{\Longrightarrow}$ (G1 - B) $\cup$ G2

(4) Inference Rules for DC:

DC-R1: Given two classes, A and B, if A $\stackrel{disjoint}{\Longleftrightarrow}$ A, then A $\stackrel{disjoint}{\Longleftrightarrow}$ B

DC-R2: Given three classes, A,B, and C, if A $\stackrel{disjoint}{\Longleftrightarrow}$ B and C $\stackrel{isa}{\Longrightarrow}$ A, then C $\stackrel{disjoint}{\Longleftrightarrow}$ B

DC-R3: Given two classes, A and B, if A $\stackrel{disjoint}{\Longleftrightarrow}$ A, then A $\stackrel{isa}{\Longrightarrow}$ B

Theorem 6.3: [AM86] SIC-R1 and SIC-R2 are sound and complete with respect to SICs.

Theorem 6.4: [AM86] SIC-R1,SIC-R2,SIC-R3, and MIC-R1 are sound and complete with respect to SICs and MICs.

Theorem 6.5: [AP86] DC-R1, DC-R2, and DC-R3 are sound and complete with respect to DCs.

Theorem 6.6: [AP86] SIC-R1, SIC-R2, DC-R1, DC-R2, and DC-R3 are sound and complete with respect to SICs and DCs.

Theorem 6.7: [Len87] CC-R1 and CC-R2 are sound and complete with respect to CCs.

Lemma 6.8: SIC-R1, SIC-R2, CC-R1, CC-R2, DC-R1, DC-R2, and DC-R3 are sound and complete with respect to SICs, CCs and DCs.

Proof: CCs and SICs are independent in that CC rules do not depend on any SICs and also SIC rules do not depend on CCs. The same argument is applied to CCs and DCs. By the theorem 2.7, the soundness and completeness of CC rules are guaranteed. DC rules depend on SICs. By the theorem 2.6,

the soundness and completeness of SIC rules and DCs with respect to SICs and DCs are guaranteed. □

Theorem 6.9: SIC-R1, SIC-R2, SIC-R3, MIC-R1, CC-R1, CC-R2, DC-R1, DC-R2, and DC-R3 are sound and complete with respect to SICs, MICs, CCs and DCs.

Proof: Every MIC can be transformed into a set of SICs by the rule SIC-R3. By the lemma 6.8, this theorem holds. □

As we mentioned earlier, an important issue of the constraint membership problem is to determine how fast a member (new constraint) can be derived from the given set of constraint.

Theorem 6.10: [AM86] Testing the membership of SIC's is O(k) where k is the number of SICs in the given set *CSET*.

Theorem 6.11: [AM86] Testing the membership of SIC's and MIC's is O(k) where k is the number of SICs and MICs in the given set *CSET*.

Theorem 6.12: [Len87] Testing the membership of CC's is O(k) where k is the number of CCs in the given set *CSET*.

Theorem 6.13: Testing the membership of DC's is O(nk) where k is the number of SICs and MICs and n is the number of DCs in the given set *CSET*.

Proof:

The following algorithm is for testing the membership of a DC A $\overset{\text{disjoint}}{\Longleftrightarrow}$ B in *CSET*. Owing to Theorem 6.6 and Theorem 6.10, the algorithm becomes simple. The algorithm is correct in accordance with Theorem 6.6 (Completeness and Soundness of SIC and DC inference rules) because the inference rules of SIC and DC are implemented in the algorithm. More precisely, the algorithm implements only SIC-R2 (line 5), SIC-R3 (line 1) and DC-R2 (line 3 and 5) because SIC-R1, DC-R1 and DC-R3 are trivial inference rules.

DISJOINT?(A,B)

/* A,B: Classes */

begin

1 transform all MICs in *CSET* into SICs and assign them to $EXTRA$;

2 $CSET \leftarrow CSET \cup EXTRA$;

3 *foreach* DC X $\overset{\text{disjoint}}{\Longleftrightarrow}$ Y in CSET: *do*

4 *begin*

5 *if* ((A $\overset{\text{isa}}{\Longrightarrow}$ X can be derived from $CSET$) ∧
(B $\overset{\text{isa}}{\Longrightarrow}$ Y can be derived from $CSET$)) ∨
((B $\overset{\text{isa}}{\Longrightarrow}$ X can be derived from $CSET$) ∧
(A $\overset{\text{isa}}{\Longrightarrow}$ Y can be derived from $CSET$))

6 *then* return("YES");

7 *end*

8 return("NO");

end

Since steps in line 1 and 5 takes O(k) where k is the number of SICs and MICs and the foreach loop halts in O(n) where n is the number of DCs in the given set *CSET*, in total, this algorithm takes time of O(nk). □

- **First Order Logic System**

The above 4 types of constraints can be expressed with formulas of the first order theory [Len87]. The formulas are universally quantified and include only unary predicates and no function symbols.

A $\overset{\text{isa}}{\Longrightarrow}$ B	is transformed into	$\forall$ x (A(x) → B(x))
A $\overset{\text{isa}}{\Longrightarrow}$ B ∩ C	is transformed into	$\forall$ x (A(x) → B(x) ∧ C(x))

$A \overset{disjoint}{\Longleftrightarrow} B$ is transformed into $\forall x\ (A(x) \rightarrow \neg\ B(x))$

$A \overset{cover}{\Longrightarrow} B \cup C$ is transformed into $\forall x\ (A(x) \rightarrow B(x) \cup C(x))$

We denote a set of first order theory formulas which are provided by the user as *ASET* (the axiom set). Constraints Membership problems can be reconstructed as follows.

1. Testing the membership of a SIC $A \overset{isa}{\Longrightarrow} B$ is equivalent to proving $ASET \vdash \forall x\ (A(x) \rightarrow B(x))$.
2. Testing the membership of a DC $A \overset{disjoint}{\Longleftrightarrow} B$ is equivalent to proving $ASET \vdash \forall x\ (A(x) \rightarrow \neg\ B(x))$.
3. Testing the membership of a SIC $A \overset{cover}{\Longrightarrow} B \cup C$ is equivalent to proving $ASET \vdash \forall x\ (A(x) \rightarrow B(x) \cup C(x))$.

- **Graph Theory System**

Constraints membership problems can be reconstructed in the graph theory. We call a class hierarchy with SICs, DCs and CCs G.

1. Testing the membership of a SIC $A \overset{isa}{\Longrightarrow} B$ is equivalent to finding a direct path (reachability problem) between the node A and the node B in G.
2. Testing the membership of a DC $A \overset{disjoint}{\Longleftrightarrow} B$ is equivalent to check whether G includes those graphs in Figure 50 as subgraphs (subgraph matching problem) [AP86].
3. Testing the membership of a CC is involving a closure computing on the graph of given CCs. Consider the graphs in Figure 51: Figure 51.a has three CCs $A \overset{cover}{\Longrightarrow} B \cup C \cup D$, $C \overset{cover}{\Longrightarrow} E \cup F \cup G$, and $E \overset{cover}{\Longrightarrow} H \cup I$, Figure 51.b has two CCs $A \overset{cover}{\Longrightarrow} B \cup E \cup F \cup G \cup D$, and $E \overset{cover}{\Longrightarrow} H \cup I$, Figure 51.c has two CCs $A \overset{cover}{\Longrightarrow} B \cup C \cup D$, and $C \overset{cover}{\Longrightarrow} H \cup I \cup F \cup G$, and finally Figure 51.d has one CC, $A \overset{cover}{\Longrightarrow} B \cup H \cup I \cup F \cup G \cup D$.

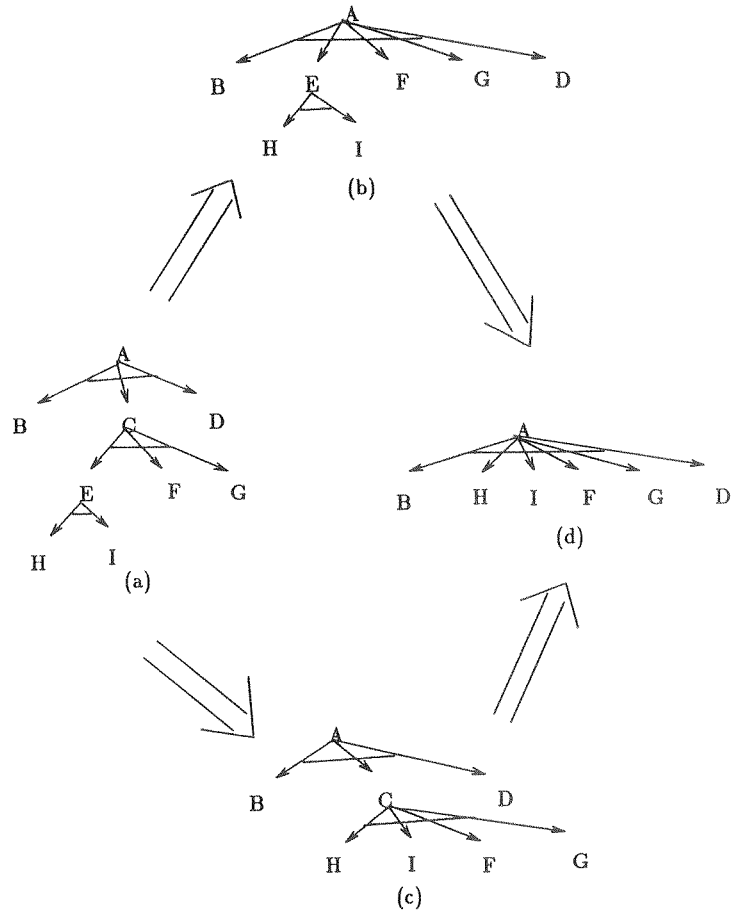


Figure 51: Graphs associating CCs

G ∪ D. One-level trees of Figures 51.a-51.b are the closure of CCs in Figure 51.a. Therefore, testing the membership of a CC is viewed as graph matching problem.

Existing graph algorithms may be used directly or need to be slightly modified for the above problems.

6.1.3 Undesirable Property Detection Problem

As we can solve the constraint membership problem in three ways, inference rules, first order logic, and graph algorithms, we also can solve the undersirable property detection problem in three different formal systems. As we mentioned earlier, we consider three cases of undesirable properties: inconsistent schema, schema with equivalent classes (cyclic structures), and schema with redundant ISA relationships.

- **Inference Rule System**

1. The inconsistency checking problem is equivalent to checking whether for a certain class x, $x \overset{\text{disjoint}}{\Longleftrightarrow} x$ is derived from $CSET$ [AP86].
2. The redundant class problem is equivalent to running the following algorithm:

   begin

   foreach class x in $CSET$: *do*

   if $\{x\}^+$ includes x

   then x is a redundant class

   end

/* where $\{x\}^+$ means the transitive closure of SICs whose left-hand-side are x, but excluding the starting x */

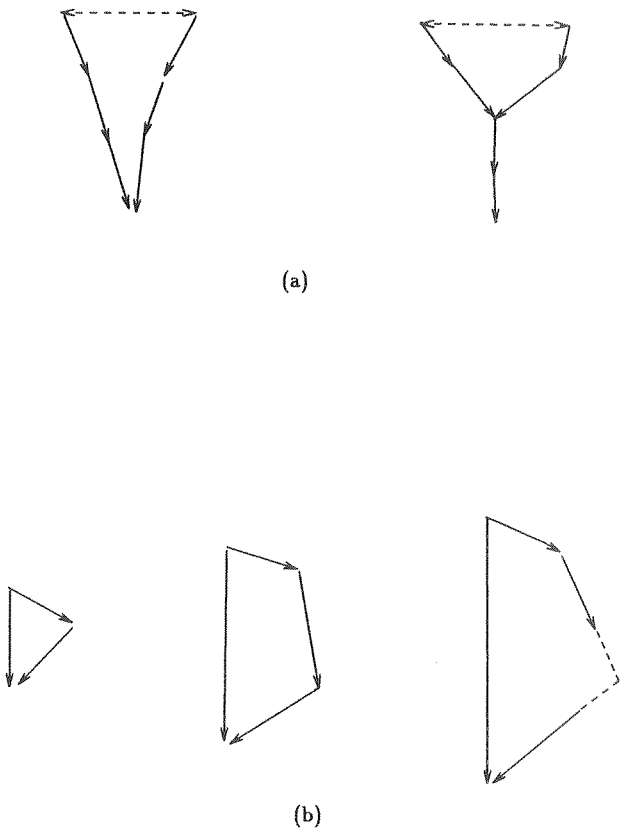


Figure 52: Graphs associating undesirable properties (from [AP86])

3. The redundant ISA problem is equivalent to running the following algorithm:

   begin

   foreach SIC i in $CSET$: *do*

   if i can be derived from $CSET - \{i\}$

   then i is a redundant ISA relationship

   end

- **First Order Logic System**

1. The inconsistency checking problem is equivalent to checking whether formulas in $ASET$ are satisfiable.

Lenzerini [Len87] showed that a set of universally quantified formulas of first order theories with unary predicates and no functions symbols can be mapped (1 to 1) into a propositional formulas in conjunctive normal forms.

It turned out that first order logic formulas of SICs, MICs, and DCs have exactly two unary predicates. Formulas of CCs can have more than two unary predicates. This minor difference causes enormous difference in computational complexity of constraint membership problem.

Theorem 6.14: Satisfiability problem with respect to SICs, MICs, and DCs has polynomial time complexity.

Proof: SICs and DCs are transformed into first order logic formulas having exactly two unary predicates. Every MIC can be transformed into a set of SICs and the SICs in the set, in turn, are transformed into formulas having two unary predicates. Now every transformed formulas has only two unary predicates and the satisfiability problem of such formulas is called the 2-SAT problem. Since 2-SAT is in P [AHU76], the theorem holds. □

Theorem 6.15: Satisfiability problem with respect to SICs, MICs, DCs and CCs is NP-complete.

Proof: CCs are transformed into formulas having more than two unary predicates and all unary predicates are connected with ∪. For example, A $\stackrel{\text{cover}}{\Longrightarrow}$ B ∪ C is transformed into ∀ x ((¬ A(x)) ∪ B(x) ∪ C(x)). Obviously, such formulas cannot be transformed into formulas having only two unary predicates. Now some of transformed formulas have more than two unary predicates and the satisfiability problem of such formulas is called the 3-SAT problem. Since 3-SAT is in NP-complete [AHU76], the theorem holds. □

2. The redundant class problem is equivalent to running the following algorithm:

   begin

   foreach i of the form $(\forall\, x\ \mathrm{A}(x) \Rightarrow \mathrm{B}(x))$ in $ASET$: *do*

   if $ASET \vdash (\forall\, x\ (\mathrm{A}(x) \Rightarrow \mathrm{B}(x) \wedge (\mathrm{B}(x) \Rightarrow \mathrm{A}(x)))$

   then A and B are equivalent classes

   end

3. The redundant ISA problem is equivalent to running the following algorithm:

   begin

   foreach i of the form $(\forall\, x\ \mathrm{A}(x) \Rightarrow \mathrm{B}(x))$ in $ASET$: *do*

   if $(ASET - i) \vdash i$

   then i is a redundant ISA relationship

   end

- **Graph Theory System**

1. The inconsistency checking problem is equivalent to checking whether one of the graphs in Figure 52.a is a subgraph of G.

2. The redundant class problem is equivalent to finding a cycle in G. Cyclicity checking algorithm is in $O(k)$ where k is the number of edges in G. The cyclicity checking algorithm is based on depth-first search. However, finding out all cycles and printing out all classes involving the cycles is computationally hard (exponential time).

3. The redundant ISA problem is equivalent to checking whether one of the graphs in Figure 52.b is a subgraph of G. The following is the graph algorithm for the redundant ISA problem.

   begin

   foreach edge (x,y) in G: *do*

   begin

   delete the edge (x,y) from G;

   add a new edge (y,x) to G;

   run cycle detection algorithm starting from y;

   if a cycle is found *then* the ISA from x to y is redundant;

   end

   end

   This algorithm has $O(k^2)$ where k is the number of edges in G. However, finding out all classes involving redundant edges is computationally hard as is finding out all cycles.

Potential Implementation

For inference rules system based solutions, a PROLOG interpreter seems the best vehicle for implementation. For first order logic system based solutions,

general theorem provers for first order logic (say, resolution theorem prover) can be used. However, we suspect that database environment cannot allow the use of theorem provers because of performance reasons. For graph theory based solutions, any general programming language can be used.

6.2 The Framework and the ORION Data Model

In this section we revisit the ORION data model from the viewpoint of the unified framework we established in the previous Chapter.

Type subsumption in The ORION data model

ORION class definition is simple enough to be represented in TR-2. A ORION class is composed of a set of superclasses, a set of pairs (variable, domain), and a set of methods. Since methods in ORION are defined in CommonLisp, input and output parameters are not strongly typed (i.e., only input parameters are declared without parameter types and output parameters are not declared). Therefore we ignore subsumption among ORION methods. For superclasses and instance variables of ORION class, it is very straightforward to transform to TR-2 expressions. Here is an example of ORION class definition in a LISP-like syntax: SUBMARINE class has two superclasses NUCLEAR-VEHICLE and WATER-VEHICLE and two instance variables Speed and Manufacturer. The domain of Speed is INTEGER and the domain of Manufacturer is COMPANY.

(SUBMARINE :superclasses NUCLEAR-VEHICLE WATER-VEHICLE

:variables (Speed INTEGER) (Manufacturer COMPANY))

The TR-2 representation of SUBMARINE is:

SUBMARINE ≡ (AND NUCLEAR-VEHICLE WATER-VEHICLE

(ALL Speed INTEGER)

(ALL Manufacturer COMPANY))

We note that even SOME clauses in TR-2 are never used in ORION

classes. Obviously the type subsumption problem in the ORION data model has $O(n^2)$ complexity according to Theorem 2.2.

Constraints in The ORION data model

Only SICs and MICs are expressed implicitly in the ORION model. As we mentioned earlier, the constraint membership problem with respect to SIC and MIC has a linear time algorithm.

Undesirable Properties in The ORION data model

- Inconsistent Schema

Since only SICs and MICs are allowed in the ORION model, we find an interesting theorem about consistency.

Theorem 6.16: Any object-oriented database schema with only SIC and MIC constraints is consistent.

Proof: [Len87] The formula, $\forall x\ (\alpha(x) \rightarrow \beta(x))$ for any arbitrary α and β, is always satisfiable. Consider a set of such formulas T. Since there is no disjointness constraint, each formula in T can be satisfied by the same set of objects. Any object-oriented database schema with only SIC and MIC constraints is consistent. □

However, even though the user did not declare disjointness constraints explicitly, the system can check the type disjointness between every pair of classes. If there are not disjoint classes, any schema design with only SICs and MICs is consistent. If there are some disjoint classes, we have to check the consistency of the schema by using 2-SAT algorithm as we mentioned earlier.

- Redundant classes and redundant ISAs

Finding cycles (equivalent classes) and redundant IS-A relationships are same as in the previous section. We may use any of solutions in the previous section.

6.3 The Framework and the ORION Schema Evolution Model

In this section we revisit the ORION schema evolution framework from the view point of the object-oriented database design framework that we establish. We present below necessary *verification tasks* which are entailed after each schema change operation from the viewpoint of the unified framework for object-oriented database schema. The reader may refer to the semantics of schema change operations in section 2.1.4.

- **(1.1.1) Add a new instance variable to a class C:** If V already exists in C an inherited instance variable, it should be verified that the domain of new V is a subclass of the domain of old V before committing this operation.
- **(1.1.2) Drop an instance variable V from a class C:** It should be verified that V is not an inherited instance variable before committing this operation.
- **(1.1.3) Change the name of an instance variable V of a class C:** It should be verified that V is not in C or subclasses of C before executing this operation.
- **(1.1.4) Change the domain of an instance variable V of a class C:** As we discussed earlier in chapter 2, the domain, class D, of an instance variable V of a class C may be changed only to a superclass of D. Suppose C inherited V from C'. It should be verified that the new domain of V of C is a subclass of the domain of V of C'.
- **(1.1.5) Change the inheritance (parent) of an instance variable:** (Inherit another instance variable with the same name) Nothing to verify.
- **(1.1.6) Change the default value of an instance variable V of a class C:** It should be verified that the new default value is an instance of the domain of V.

- **(1.1.7.1) Add the shared value of a variable V of a class C:** It should be verified that the shared value is an instance of the domain of V.
- **(1.1.7.2) Change the shared value of a variable V of a class C:** It should be verified that the shared value is an instance of the domain of V.
- **(1.1.7.3) Drop the shared value of a variable V of a class C:** Nothing to verify.
- **(1.2) Change an instance method** The verification tasks for operations 1.2.1, 1.2.2, 1.2.3, 1.2.4, and 1.2.5 are easily inferred from 1.1.1, 1.1.2, 1.1.3, 1.1.4, and 1.1.5 respectively.
- **(2.1) Make a class S a superclass of a class C:** It should be checked that the class definition of C is subsumed by the class definition of S before committing this operation. The consistency and redundancy of the resulting schema should be checked.
- **(2.2) Remove a class S as a superclass of the class C:** It should be checked that C and S do not have any common instances any more before committing this operation. The consistency and redundancy of the resulting schema need not be checked because the resulting schema can be inconsistent or redundant if the previous schema was consistent and nonredundant.
- **(2.3) Change the order of the superclasses of a class C:** Nothing to verify.
- **(3.1) Define a new class C:** It should be checked that C is subsumed by every superclass before committing this operation. The consistency and redundancy of the resulting schema should be checked.
- **(3.2) Drop a class C:** If C had more than one superclasses, it should be checked that no common instances exist in any pair of C's superclasses

before committing this operation. The consistency and redundancy of the resulting schema need not be checked because the resulting schema can be inconsistent or redundant if the previous schema was consistent and nonredundant.

- **(3.3) Change the name of a class:** It should be verified that the new name is unique among all class names in the class hierarchy before executing this operation.

We note that the above verification tasks are either trivially easy or one of the 2 problems (type subsumption and undesirable property detection) that were addressed in the section 6.2.

Chapter 7
More on Subclassing

Subclassing (creating a new subclass from a class) is the most frequently used schema change operation. A new class needs to be created from a class when a new concept that cannot be accommodated in the existing classes has to be introduced. Most subclassings involve the imposition of restrictions on instance variables of a parent class. The constraints that accompany these subclassings are called *subclassing conditions*, which are predicate expressions on instance variables.

In this chapter, we shall present the results of our research into various issues of subclassing. First, we present a taxonomy of subclassing and the semantics of each case. Second, we address the issue of subclassing condition management because most subclassings are accompanied by subclassing conditions. As a database and schema grow in size and complexity, it is very difficult to maintain consistent class hierarchies without taking advantage of subclassing conditions. We also consider the inverse operation of subclassing, *desubclassing* (dropping an existing class). Third, subclassing conditions are useful in many applications of object-oriented databases. We identify those applications and introduce the techniques of applying subclassing conditions to the applications.

7.1 Taxonomy of Subclassings

In order to achieve a consistent and optimal class hierarchy, we first have to understand semantics of schema change operations in the previous section completely. Among the operations in the taxonomy, we believe that (3.1) "Add a new class" (subclassing) is the most frequently used operation in object-oriented applications. In the remainder of this chapter we examine subclassing in great detail.

A new class needs to be created when a new concept has to be introduced. The new class may be a specialization of an existing class or classes.

These latter classes, then, can be specified as the superclasses of the new class. All instance variables and methods of the superclasses are inherited by the new subclass. Either some changes to the inherited instance variables or methods may be needed, or new instance variables or new methods may need to be added to the new subclass.

We identify three cases in which subclassing is needed.

- Case (1): The user (database designer) needs to create a new subclass from a class C when he wants to partition existing instances of the class C for conceptual convenience. In this case C is the unique superclass of the new subclass. For example, the user wants to classify expensive and cheap automobiles from AUTOMOBILE class. Two refined class definitions of AUTOMOBILE class are needed for expensive and cheap automobiles respectively: if the price of an automobile is more than $10000, then the automobile is an expensive automobile, otherwise a cheap automobile. Instances of AUTOMOBILE class must be moved down to two new subclasses because an instance cannot belong to more than one class. In general, partitioning of instances of a class can be achieved by applying some predicates (we shall call them *subclassing conditions*) to existing instances of the class. If partitioning is determined by predicates on an instance variable, we call such variable *partition-control instance variable*. The price instance variable is a partition-control instance variable.

- Case (2): When the designer wants to insert an instance in a database, if more than one class can accommodate the instance and the classes do not have ISA relationships, a new class must be created due to the restriction that an instance cannot belong to more than one class. For example, if the user wants to insert an instance I to a database, and I can belong to two existing classes C_1 and C_2 in which there is no ISA relationships, then a new subclass having C_1 and C_2 as superclasses needs to be created for storing I. In this case, a test must be performed by either the user or the system as to whether I belongs to more than one class.

• Case (3): When the designer wants to insert an instance in a database, if no existing class can accommodate the instance, a new class must be created for storing the new instance. Again, it can be checked by either the user or the system whether an instance cannot belong to any existing class.

We shall present subclassing algorithms for the above three situations in section 7.2. As we briefly mentioned earlier in this section, a new class definition accompany *restrictions*, such as declaration of new instance variables and modifications on inherited instance variables. The new class definition also accompany declaration of new methods and modifications of inherited methods. However, in this chapter, we concentrate on instance variable oriented restrictions. We discuss below the types of restriction and the semantics of each.

Subclassing Taxonomy

(1) Predicate-based

(1.1) Interval Reduction

(1.2) Value Isolation

(1.3) Value Negation

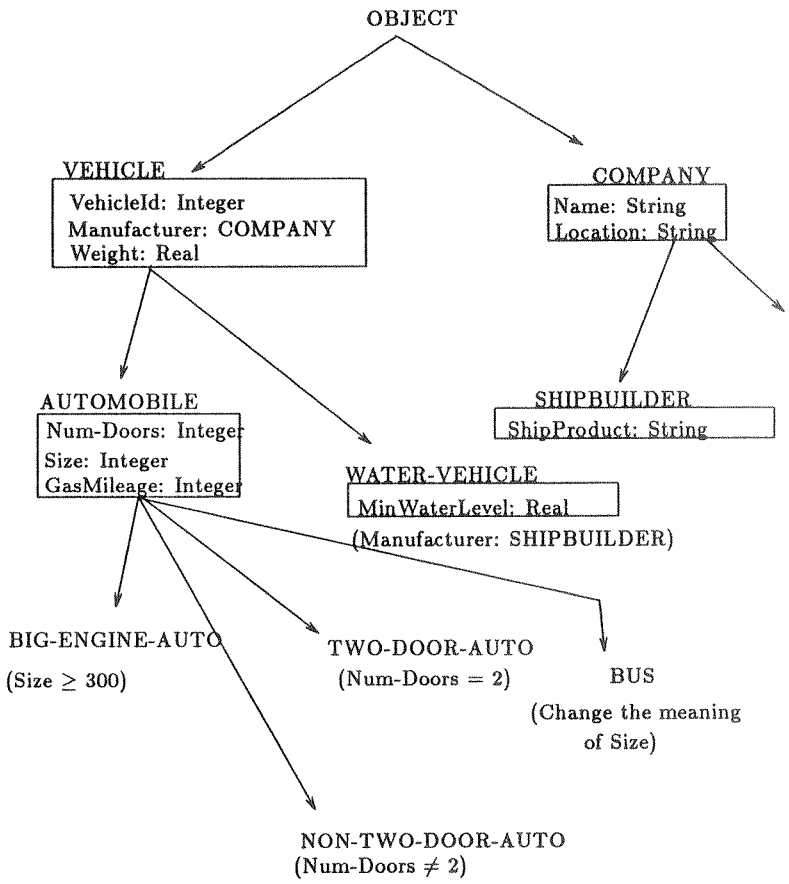
(1.4) Instance Variable Comparison

(2) Domain Reduction

(3) Instance Variable Overriding

(4) Instance Variable Addition

Subclassing involves one or more combination of the above cases. The cases of (1) are the application of predicates to domains of instance variables of a parent class. Interval reduction (1.1) is to reduce the interval domain of a partition-control instance variable of a superclass, whereas value isolation (1.2) and value negation (1.3) is to specify and to exclude a particular value in the



Legend:

A box has a set of locally defined instance variables

A brace has a subclassing criteria.

Figure 53: VEHICLE and COMPANY class hierarchies

domain of a partition-control instance variable respectively. Instance variable comparison (1.4) is to compare values of two different variables to select out instances. For examples of (1), consider the Figure 53. A new class BIG-ENGINE-AUTO may be created from the class AUTOMOBILE by specifying a predicate (Size $\geq$ 300). A new class TWO-DOOR-AUTO may be created from the AUTOMOBILE class by specifying a predicate (Num-Doors = 2) whereas NON-TWO-DOOR-AUTO by a predicate (Num-Doors $\neq$ 2). These examples represent (1.1),(1.2) and (1.3) respectively. As an example of (1.4), consider PEOPLE class with three instance variables (*name*, *income*, *outgo*). A new class DEBTER is defined as people who spent more money than what he earned (i.e., *income* < *outgo*).

Domain reduction (2) is to change the domain D of an inherited instance variable to a subclass of D. Again in Figure 53, VEHICLE class inherits three instance variables of VEHICLE class, but changes the domain of Manufacturer instance variable from COMPANY to SHIPBUILDER which is a subclass of COMPANY class.

Instance variable overriding (3) is to override the domain of an inherited variable with a different domain (i.e., change the meaning of the instance variable). For example, suppose BUS class is created from AUTOMOBILE class. The meaning of Size in AUTOMOBILE is engine size. The designer can change the meaning of Size to "number of people that can be accommodated in a bus" as shown in Figure 53. Instance variable addition (4) is to add new instance variables when subclassing happens. As shown in Figure 53, AUTOMOBILE class has three new instance variable as well as three inherited instance variables from VEHICLE class.

7.2 Subclassings and Subclassing Conditions

We believe that a large number of subclassings involve the predicate-based restrictions in (1). In this section we shall elaborate on predicate-based restrictions.

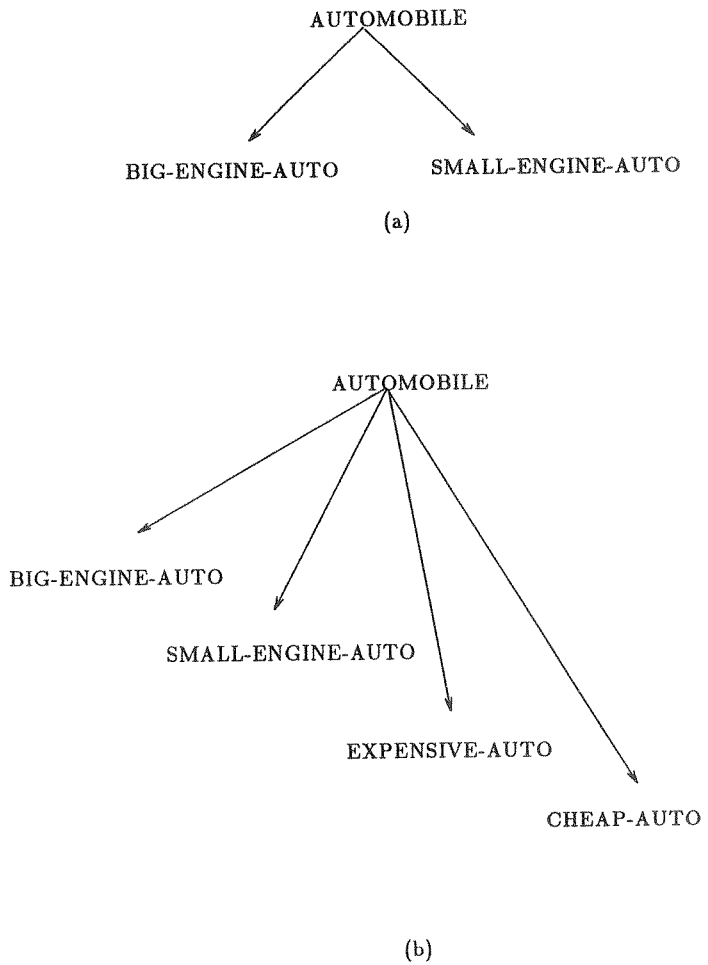


Figure 54: Illustrating subclassing mechanism

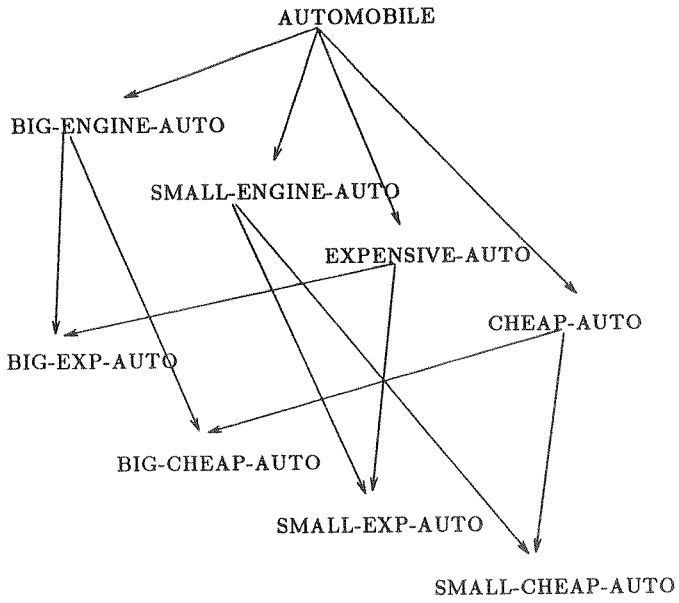
Conceptual Subclassing Condition vs. Actual Subclassing Condition

Suppose AUTOMOBILE class has two subclasses, BIG-ENGINE-AUTO and SMALL-ENGINE-AUTO, as shown in Figure 54.a. AUTOMOBILE class has four instance variables: Id (Positive Integer), Manufacturer (String), Engine-size (Positive Integer), and Price (Real).

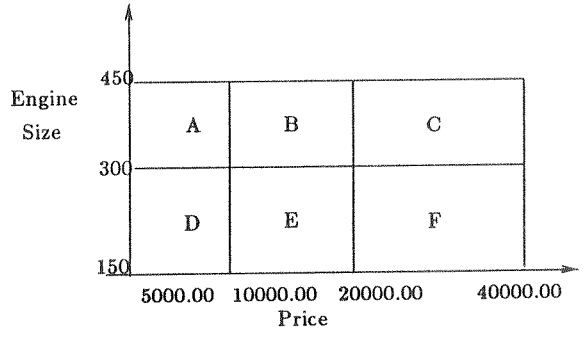
Assume that the subclassing condition of AUTOMOBILE class is (150 $\leq$ Engine-Size $\leq$ 450) $\wedge$ (5000.00 $\leq$ Price $\leq$ 40000.00) and let this predicate be denoted as P. Suppose BIG-ENGINE-AUTO class and SMALL-ENGINE-AUTO are for automobiles whose engine size is bigger than 300 cubic inches and smaller than or equal to 300 cubic inches respectively. Then the subclassing condition of BIG-ENGINE and SMALL-ENGINE are P $\wedge$ (Engine-Size $>$ 300) and P $\wedge$ (Engine-Size $\leq$ 300) respectively. Note than P is inherited by BIG-ENGINE and SMALL-ENGINE. As all properties of a class such as instance variables or methods are inherited into subclasses, subclassing conditions should also be inherited. Subclassing conditions are inherited in a conjunctive form.

Clearly both P $\wedge$ (Engine-Size $>$ 300) and P $\wedge$ (Engine-Size $\leq$ 300) imply P. IS-A relationships between AUTOMOBILE and BIG-ENGINE and between AUTOMOBILE and SMALL-ENGINE-AUTO make sense, i.e., every automobile with engine size bigger than 300 and smaller than or equal to 300 is an automobile. We call P, P $\wedge$ (Engine-Size $>$ 300), and P $\wedge$ (Engine-Size $\leq$ 300) *conceptual subclassing condition predicates* (denoted CSCP) of AUTOMOBILE, BIG-ENGINE-AUTO, and SMALL-ENGINE-AUTO respectively. In summary, conceptual subclassing conditions define membership in a class and its subclasses. Whereas actual subclassing conditions (to be addressed next) define membership in a class but in no subclasses.

As we mentioned in section 1.2.2, an instance must belong to (be physically stored in) one and only one class. Again in Figure 54.a, none of the automobile instances are physically stored in the AUTOMOBILE class. That is because an automobile, which is an instance of both AUTOMOBILE and



(c)



(d)

Figure 54 (Cont'd): Illustrating subclassing mechanism

BIG-ENGINE-AUTO, is stored in BIG-ENGINE-AUTO while an automobile, which is an instance of both AUTOMOBILE and SMALL-ENGINE-AUTO, is stored to SMALL-ENGINE-AUTO. This gives rise to define another concept *actual subclassing condition predicates* (denoted ASCP) which define membership in a class but in no subclasses.

The actual subclassing condition predicates of AUTOMOBILE, SMALL-ENGINE-AUTO and BIG-ENGINE-AUTO are $\mathbf{P} \wedge \neg$ **(Engine-Size** $>$ **300)** $\wedge \neg$ **(Engine-Size** $\leq$ **300)**, $\mathbf{P} \wedge$ **(Engine-Size** $\leq$ **300)**, and $\mathbf{P} \wedge$ **(Engine-Size** $>$ **300)** respectively. The actual subclassing condition of AUTOMOBILE is unsatisfiable (false) and no instance can belong to AUTOMOBILE. Only instances, which are satisfying ASCP of a class C, can belong to the class C. We note that ASCPs of AUTOMOBILE, SMALL-ENGINE-AUTO, and BIG-ENGINE-AUTO are mutually disjoint. In summary, in Figure 54.a,:

- CSCP(BIG-ENGINE-AUTO) implies CSCP(AUTOMOBILE)
- CSCP(SMALL-ENGINE-AUTO) implies CSCP(AUTOMOBILE)
- ASCP(BIG-ENGINE-AUTO) $\wedge$ ASCP(AUTOMOBILE) is unsatisfiable
- ASCP(SMALL-ENGINE-AUTO) $\wedge$ ASCP(AUTOMOBILE) is unsatisfiable
- ASCP(BIG-ENGINE-AUTO) $\wedge$ ASCP(SMALL-ENGINE-AUTO) is unsatisfiable

CSCPs and ASCPs are important in that they are used to keep class hierarchy consistent and nonredundant. Below we present three different subclassing algorithms and show how CSCPs and ASCPs should be managed.

Case (1): Subclassing for Partitioning Instances

Now we consider the case of subclassing which stems from a need to partition existing instances. As mentioned earlier, in this case the designer

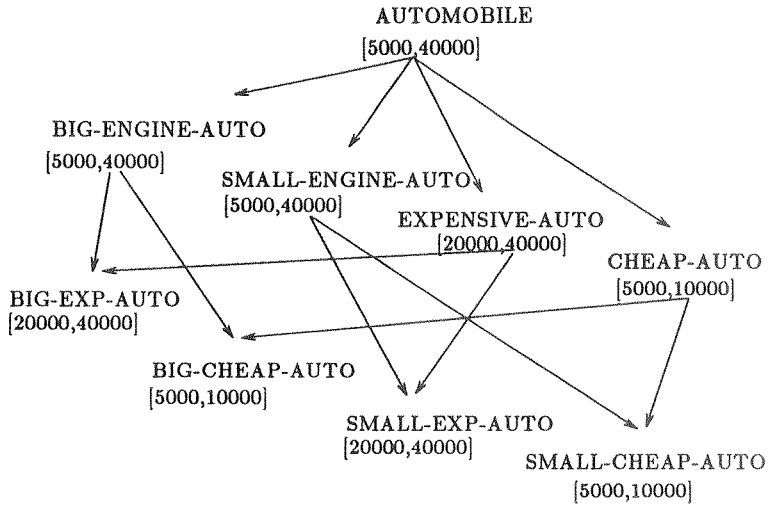


Figure 55.a: Conceptual subclassing condition predicates (CSCPs) on Price

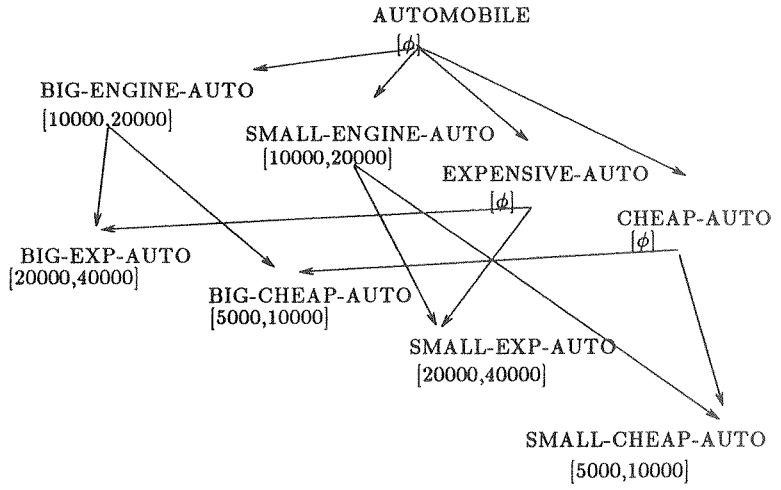


Figure 55.b: Actual subclassing condition pedicates (ASCPs) on Price

specifies a superclass of a new subclass and some restrictions on the superclass, and the superclass is the unique superclass of the new subclass.

Consider the class hierarchy in Figure 54.a. Instances of AUTOMOBILE class are presently partitioned into two classes by the partition-control instance variable Engine-Size, i.e., the instances in the regions A,B, and C of Figure 54.d are positioned in BIG-ENGINE-AUTO whereas the instance of D,E, and F are positioned in SMALL-ENGINE-AUTO. Suppose the designer is trying to create EXPENSIVE-AUTO class and CHEAP-AUTO class for distinguishing automobiles of which price is between $5000.00 and $10000.00 and between $20000.00 and $40000.00 respectively. Then the resulting class hierarchy would be the one in Figure 54.b. Now the designer has to partition instances of AUTOMOBILE class using the partition-control instance variable, Price. An automobile whose engine size is 350 cubic inches and price is $ 34000.00 should appear in two classes, EXPENSIVE-AUTO and BIG-ENGINE-AUTO. Therefore, the class hierarchy in Figure 54.b is not allowed. In this case, either a new subclass is created for accommodating instances belonging to both EXPENSIVE-AUTO and BIG-ENGINE-AUTO, or the request must be rejected. The class hierarchy in Figure 54.c is allowed in that every instances belongs to one and only class.

Whenever a new class S is created as a subclass of C, CSCP(S) should imply CSCP(C) for a meaningful IS-A relationship. Whereas ASCP(S) should be disjoint with ASCPs of existing classes of the database. Otherwise either the creation request of S is rejected or an additional class needs to be created for preserving the property "an instance belongs to one and only one class". We present below the algorithm for Case (1).

SUBCLASSING-CASE-1 (C,S,P)

/* C : parent class of S */

/* S : new subclass */

/* P : conceptual subclassing condition predicate of S */

begin

1 *if* IMPLY(P,CSCP(C)) *then*

begin

2 create C such that CSCP(C) is P and ASCP(C) is P;

3 *foreach* I ∈ existing instances of C: *do*

4 *if* ASCP(S) is satisfied by I *then*

5 move the instance I from C to S;

6 update ASCPs of C's superclasses;

7 *foreach* S' ∈ U - {C and superclasses of C}: *do*

/* U is the set of all classes in the database */

8 *if* (there is an instance satisfying

both ASCP(S) and ASCP(S')) *then*

begin

9 create another new class S* having S and S'

as immediate superclasses and CSCP(S*) and

ASCP(S*) are both (ASCP(S) ∧ ASCP(S'));

11 *foreach* I ∈ instances of S': *do*

12 *if* ASCP(S*) is satisfied by I *then*

13 move the instance I from S' to S*;

14 update ASCPs of superclasses of S* properly;

end;

end;

end;

- In line 1, we used IMPLY(p,q) as a built-in procedure to test if the predicate p implies the predicate q, i.e., to see if $\neg p \vee q$ is true. If a class S is a subclass of a class C, IMPLY(CSCP(C),CSCP(S)) should be true. We shall elaborate on IMPLY in section 7.3.
- Line 6 is to test whether there is an instance belonging to more than one class.
- Lines 3–5 and 11–13 shows repositioning instances from a class to a new subclass. The system may automatically perform the "MOVE" operation according to subclassing conditions. Unless subclassing condition predicates are provided, the system cannot partition instances automatically. In that case we assume that the user is responsible for partitioning instances.
- After creating subclasses, ASCPs of existing classes should be updated. The details of Lines 6 and 14 (updating ASCPs) will be discussed in section 7.3.

Case (2): Subclassing for a New Instance Having More Than One Corresponding Class

Now we consider Case (2): create a new subclass for a new instance which can be accommodated by more than one existing classes. When a new class is created and added to database, the appropriate taxonomic location (superclasses of the new class) should be identified by the system unless the user specifies the superclasses for the new class. As such, we present two algorithms: Case (2A) the one is for the case when the designer specifies superclasses of the class which will accommodate the new instance, Case (2B) the other is for the case when the designer consults with the system to find out the superclasses of the new subclass.

First, consider the case when the user provides the list of superclasses. Here is the algorithm for Case 2A. If CSCPs of any two of superclasses are unsatisfiable, the user's request should be rejected because there will not be any instance belonging to the new class. Also if there is any ISA relationship between any two of classes that are provided by the user as superclasses of the new class, the request should be rejected because the resulting class hierarchy is meaningless. For example, suppose the user declares both BIG-ENGINE-AUTO and AUTOMOBILE in Figure 54.a as immediate superclasses of a new class C. The ISA relationship between C and AUTOMOBILE is redundant in that AUTOMOBILE is already a superclass of C because AUTOMOBILE is a superclass of BIG-ENGINE-AUTO and, in turn, BIG-ENGINE-AUTO is a superclass of C.

```
SUBCLASSING-CASE-2A (I,C,P)
/* I : a given instance */
/* C : a given class */
/* P : user-specified superclasses of C*/
begin
1    foreach Ci,Cj ∈ P: do
2        if IMPLY(TRUE, ¬(CSCP(Ci) ∧ CSCP(Cj))) then
             begin
3                reject the request;
4                return();
             end
5    foreach Ci,Cj ∈ P: do
6        if (Ci is a superclass of Cj) ∨ (Cj is a superclass of Ci) then
```

```
        begin
7           reject the request;
8           return();
        end
9     create the new class C;
10    insert I as an instance of C;
11    update ASCPs of superclasses of C properly;
end;
```

- Line 2-4 tests the satisfiability of CSCPs of any two superclasses which are provided by the user. If any two CSCPs are unsatisfiable, the subclassing request must be rejected.
- Lines 5-8 detect redundant ISA relationships.
- Refer to section 7.3 for the details of line 11.

Second, consider the case when the user does not specify superclasses of the class for the new instance I. The system should find classes which can accommodate I and collect them in the superclass-set set variable. If there is only one class, the instance I should be inserted into the class. Otherwise a new class C having classes in the superclass-set as immediate superclasses should be created and the instance I is inserted into C.

SUBCLASSING-CASE-2B (I)

```
/* I : a given instance */
begin
1     superclass-set ← ∅;
```



```
2     foreach leaf to root path in the class hierarchy: do
3       foreach class C in the path: do
4         if (C can accommodate I)
5         then begin
6           superclass-set ← superclass-set ∪ { C };
7           break the foreach loop in line 3;
          end;
8     if (there is only one class in the superclass-set)
9       then insert I as an instance of the class;
10      else begin
            /* more than one class can accommodate I */
11          create a new class C' having the classes as superclasses;
12          insert I as an instance of C';
13          update ASCPs of superclasses of C' properly;
        end;
end;
```

- In lines 1-7, the system returns a set of classes which can accommodate I. Redundant ISA relationships cannot be made because, once a class C is turned out be a class which can accommodate I, then the superclasses of C are skipped.
- Refer to section 7.3 for the detail of line 12.

Case (3): Subclassing for a New (Exceptional) Instance Having No Corresponding Class

In this case, the procedure, for consulting with the system to find out the corresponding classes for the new instance, is difficult to be automated because the new instance may have instance variables which do not exist in existing classes. We assume that it is the user's responsibility to provide corresponding superclasses in case 3. Then the algorithm becomes to be the same as the previous SUBCLASSING-CASE-2A.

7.3 Subclassing Condition Management

So far, we used two built-in procedures on predicates implicitly. In this section, we discuss the two built-in procedures: 'update ASCPs' and 'IMPLY', in detail.

Update Actual Subclassing Condition Predicates

After a new subclass S of a class C is created, the actual subclassing conditions of C's superclasses should be updated recursively. In the previous section, three subclassing algorithms include a line 'update ASCPs of superclasses of a class C'. The following procedure will do the job.

```
UPDATE-PROPAGATION(S,P)
/* S : a new subclass */
/* P : ASCP of S */
begin;
    foreach S' ∈ immediate superclasses of S: do
        begin;
        ASCP(S') ← (ASCP(S) ∧ ¬ASCP(S'))
        UPDATE-PROPAGATION(S',ASCP(S'));
        end;
end;
```


More on IMPLY

IMPLY is a built-in procedure to test if the predicate p implies the predicate q. The implication problem can be rephrased as the unsatisfiability problem: to check if $p \Rightarrow q$ holds is equivalent to check the unsatisfiability of $\neg (p \Rightarrow q)$. In particular we are interested in testing if a conjunction of two predicates P1 and P2 is satisfiable, that is IMPLY(true,P1 $\wedge$ P2). Similarly, to see if a conjunction of two predicates P1 and P2 is unsatisfiable is to prove IMPLY(true, $\neg$ (P1 $\wedge$ P2)).

As mentioned earlier in the type subsumption problem section of the previous chapter, testing satisfiability or proving arbitrary predicates in first-order predicate calculus is *undecidable*. As such, adopting the first-order predicate calculus for managing subclassing conditions is not desirable. Therefore, the issue is to characterize a subset of first order predicate logic expressions which is powerful enough for expressing subclassing conditions and in which the satisfiability problem can be processed efficiently.

Subclassing conditions can be represented with the "simple predicates" of Eswaran et al. [EGLT76]. The BNF of simple predicates (S-P) is as follows:

⟨ S-P ⟩ ::= ⟨ S-P ⟩ $\wedge$ ⟨ S-P ⟩ | ⟨ S-P ⟩ $\vee$ ⟨ S-P ⟩ | $\neg$ ⟨ S-P ⟩ | ⟨ predicate ⟩

⟨ predicate ⟩ ::= ⟨ variable ⟩ ⟨ comparion-op ⟩ ⟨ right-hand-side ⟩

⟨ comparison-op ⟩ ::= = | $\neq$ | < | $\leq$ | > | $\geq$

⟨ right-hand-side ⟩ ::= ⟨ constant ⟩ | ⟨ variable ⟩ | ⟨ variable ⟩ + ⟨ constant ⟩

Rosencrantz and Hunt [RH80] showed that the satisfiability problem of the set of simple predicates is NP-hard. However, they showed that conjunctive unequalsfree predicates (simple predicates that do not contain $\neq$ and $\vee$) can be processed efficiently (polynomial time). It is interesting to note that a large class of subclassing conditions can be represented with conjunctive unequalsfree predicates.

We below introduce Rosencrantz and Hunt's algorithm, Satisfiability-Unequalsfree-Conjunctive-Predicates (denoted SUCP) testing in polynomial time the satisfiability of a conjunctive containing no $\neq$ operations.

SUCP(P)

/* P is a conjunctive unequalsfree predicate */

begin

1. Transform P into an equivalent predicate P' containing only $\leq$ operator in the following manner. V1 and V2 stands for variables while C1 stands for a constant.
   - V1 = V2 *is transformed into* (V1 $\leq$ V2 + 0) $\wedge$ (V2 $\leq$ V1 + 0)
   - V1 < V2 *is transformed into* V1 $\leq$ V2 + (−1)
   - V1 $\leq$ V2 *is transformed into* V1 $\leq$ V2 + 0
   - V1 > V2 *is transformed into* V2 $\leq$ V1 + (−1)
   - V1 $\geq$ V2 *is transformed into* V2 $\leq$ V1 + 0
   - V1 = C1 *is transformed into* (V1 $\leq$ 0 + C1) $\wedge$ (0 $\leq$ V1 + (−C1))
   - V1 < C1 *is transformed into* V1 $\leq$ 0 + (C1 −1)
   - V1 $\leq$ C1 *is transformed into* V1 $\leq$ 0 + C1
   - V1 > C1 *is transformed* 0 $\leq$ V1 + (−C1−1)
   - V1 $\geq$ C1 *is transformed* 0 $\leq$ V1 + (−C1)
   - V1 = V1 + C1 *is transformed into* (V1 $\leq$ V2 + C1) $\wedge$ (V2 $\leq$ V1 + (−C1))
   - V1 < V2 + C1 *is transformed into* V1 $\leq$ V2 + (C1 −1)
   - V1 $\leq$ V2 + C1 *is transformed into* V1 $\leq$ V2 + C1

   - V1 > V2 + C1 *is transformed into* V2 $\leq$ V1 + (−C1−1)
   - V1 $\geq$ V2 + C1 *is transformed into* V2 $\leq$ V1 + (−C1)
2. Transform P' into a weighted directed graph. The graph has a node for each variable, plus a node for a constant zero. Transformation is as follows:
   - Each var-1 $\leq$ var-2 + const-1 corresponds to an edge whose weight is const-1 from a node var-1 to a node var-2.
   - Each var-1 $\leq$ 0 + const-1 corresponds to an edge whose weight is const-1 from a zero node to a node var-1.
   - Each 0 $\leq$ var-1 + const-1 corresponds to an edge whose weight is - const-1 from a zero node to a node var-1.
3. If there is more than one edge from one node to another, retain the minimum weight edge, and discard the others.
4. Apply Floyd's all shortest path algorithm to the constructed graph to see if the graph has a negative weight circuit.

end

Rosencrantz and Hunt [RH80] showed that P is satisfiable if and only if its transformed weighted directed graph has no negative weight cycles. In the above algorithm, the step 1, 2, and 3 are processed in a linear time. The step 4 (Floyd's all shortest paths algorithm [AHU76]) takes $O(k^3)$ for a k node graph. As such the above algorithm is in $O(k^3)$ where k is a number of variables in the predicate P.

There are two restrictions in the above algorithm. The one is that each variable should be integer valued. The other is that predicates cannot have $\neq$ operators. Fortunately, many of subclassing conditions involve integer-valued domains such as engine size, price, and number of doors.

For $\neq$ operators, Böttcher, et al. [BJS86] suggested a simple mechanism to include $\neq$ comparisons into the SUCP algorithm. Their algorithm is first to sort the comparisons of each conjunction such that $\neq$ comparisons are located last and to process conjunctions without $\neq$ and then do additional tests for conjunctions with $\neq$. Here is the algorithm.

Satisfiability-Conjunction-Predicate(P)

/* P : conjunctive predicates including $\neq$ */

begin

1. Sort P into P1 $\wedge$ P2 where P1 is conjunctive predicates without $\neq$ and P2 is conjunctive predicates with $\neq$
2. Perform SUCP(P1)
3. *if* (the graph for P1 contains no negative cycles) *then*
   begin
   foreach var-1 $\neq$ var-2 $\in$ P2: *do*
   if (the graph contains zero weight cycle between nodes var-1 and var-2)
   then return("P is unsatisfiable");
   foreach var-1 $\neq$ const-1 $\in$ P2:
   if (the graph contains zero weight cycle between var-1 node and zero node)
   then return("P is unsatisfiable");
   return("P is satisfiable");

end

This algorithm has a time complexity of $O(k^3)$, but only semi-correct in that if P is satisfiable, the algorithm always says P is satisfiable, but if P is unsatisfiable the algorithm may say P is satisfiable. This is acceptable because of the following justifications.

1. Whenever the algorithm says "P is satisfiable", search the database to see if there is really an instance satisfying P. If there is no instance satisfying P, then conclude P is unsatisfiable. In this way, the algorithm can be correct.

2. In fact, subclassing conditions involving $\neq$ are very rare.

7.4 Properties of Subclassing Conditions

Now we summarize the properties of subclassing conditions formally.

Criteria for Consistent Schema Design

The following properties should hold in consistent object-oriented database schemas.

- ASCP(C) = CSCP(C) if C is a leaf class
- CSCP(C) = $(\bigvee_{S \in \text{immediate subclasses of } C} \text{CSCP}(S)) \vee \text{ASCP}(C)$
- CSCP(C) = ASCP(C) $\vee$ $(\bigvee_{S \in \text{superclasses of } C} \text{ASCP}(C))$
- $\nexists$ instance satisfying ASCP(C_i) $\wedge$ ASCP(C_j)) for $C_i \neq C_j$
- IMPLY(ASCP(C),CSCP(C)) is true for any C

Useful Rules for IMPLY processing

The following rules are useful for optimizing algorithms which are introduced so far.

- IMPLY(Pi,Pi) is true for any Pi

- (IMPLY(Pi,Pj) ∧ IMPLY(Pj,Pk)) ⇒ IMPLY(Pi,Pk)
- Pi ∧ Pi is satisfiable for any Pi
- (Pi ∧ Pj is satisfiable) if and only if (Pj ∧ Pi is satisfiable)
- (Pi ∧ Pj is satisfiable) ∧ IMPLY(Pj,Pk)) ⇒ (Pi ∧ Pk is satisfiable)

7.5 Applications of Subclassing Conditions

The major motivation behind keeping track of subclassing conditions was to maintain consistent class hierarchies and partition instances properly. Fortunately, there are many other important applications which can benefit substantially from utilizing subclassing conditions. In this section we enumerate feasible applications of subclassing conditions. We adopt a few observations from Muntz, Shneider, and Steyer [MSS79].

Query Optimization

As we discussed earlier, the query processor needs to visit a target class and its subclasses for processing SELECT-ALL type queries in object-oriented databases. For efficiency reasons it does not make sense to let the query be processed with instances of the target class and all of its subclasses. A central problem in query optimization of object-oriented databases is to find the minimal set of classes sufficient to process a query. Another problem is to simplify a query predicate when it contradict with or is implied by subclassing conditions.

Suppose a SELECT-ALL type query (based on a predicate P) were posed against a class C. Here is an algorithm for finding a minimal set of classes for the query.

```
QUERY-PROCESS(C,P)
/* C: Target Class */
/* P: Query Qualification */
```



```
begin
  if (C is not marked) ∧ (CSCP(C) ∧ P)) then
    begin
    mark C;
    if (ASCP(C) ∧ P) then
      begin
      P ← (ASCP(C) ∧ P);
      select instances of C satisfying the query qualification P;
      end
    foreach S ∈ {immediate subclasses of C}: do
          QUERY-PROCESS(S,P);
    end
  else foreach S* ∈ {subclasses of C}:
          if (S* is not marked) then mark S*;
end
```

Reuse of Access Plan

If a result of a query Q1 is a subset of a previous query Q0 (i.e., query qualification of Q1 implies query qualification of Q0) the access plan of Q0 can be used directly as an access plan of Q1. Or, if we have a priori knowledge such as partial ordering of query qualification of a batch of queries, we can take advantage of the priori knowledge for global query optimization, that is, to reorder the sequence of queries for better performance.

Predicate Locking

Multiple users may compete for instances for a certain class. Suppose predicate locking is a candidate locking mechanism for object-oriented databases. Given a class C, WLPL(C) and RLPL(C) are predicates that describes write-locked instances of C and read-locked instances of C respectively. Suppose a write lock request WR(U,C) to the class C is posed by the user U. If IMPLY(ASCP(C) $\wedge$ $\neg$WLPL(C), WR(U,C)) is true, the use U can get write-lock access to the instances by WR(U,C). Otherwise the request of the user U should wait for until IMPLY(ASCP(C) $\wedge$ $\neg$WLPL(C), WR(U,C)) becomes true.

If the user U gets the write-lock access, WLPL(C) should be updated in the following way.

WLPL(C) $\longleftarrow$ WLPL(C) $\vee$ WR(U,C)

If th user U releases the write-lock access, RLPL(C) should be updated in the following way.

WLPL(C) $\longleftarrow$ WLPL(C) $\wedge$ $\neg$WR(U,C)

A similar (not exactly same) argument is applied to RLPL and read-lock request. The interested reader may refer to Rosencrantz and Hunt [RH80] and Eswaran, et al. [EGLT76] for details of predicate managements in the predicate locking scheme.

Access Control

To control the access of a user to the database is called "Authorization" or "Access Control." Suppose the user U has the access right AR(U,C) against a class C and the user U poses a query with predicates Q(C) to the class. The query is allowed to be processed only if (Q(C) $\wedge$ AR(U,C) $\wedge$ ASCP(C)) is satisfiable. Otherwise the query is rejected. If (Q(C) $\wedge$ AR(U,C) $\wedge$ ASCP(C)) is satisfiable, the next problem is to determine if possible a simpler predicate which is equivalent to (Q(C) $\wedge$ AR(U,C) $\wedge$ ASCP(C)) for efficient processing.

7.6 Desubclassing

In this section, we discuss the inverse operation of subclassing, *desubclassing* (i.e., dropping an existing class). As shown in section 7.2, the designer cannot create an arbitrary class because of the assumption "an instance belongs to one and only one class". By similar reasoning, arbitrary desubclassing is not allowed. For example, in Figure 54.c BIG-EXP-AUTO is not allowed to be dropped as long as instances of BIG-ENGINE-AUTO exist because BIG-EXP-AUTO was created to accommodate common instances belonging to both BIG-ENGINE-AUTO and EXPENSIVE-AUTO. As such, a class with more than one superclass cannot be dropped as long as instances remain in the class.

When a class C is dropped, instances of C are moved up to its superclass and ASCP of superclasses of C should be updated properly. Here is an algorithm for desubclassing.

```
DESUBCLASSING(C,P)
/* C: A Class to be dropped*/
/* P: ASCP of C */
begin
if (C has more than one superclass)
    then if (instances remain in C)
            then reject the request
            else begin
                drop C;
                UPDATE-ASCP-AFTER-DESUBCLASSING(C,P);
                end
    else begin
```

```
        drop C;
        move instances of C to its superclass;
        UPDATE-ASCP-AFTER-DESUBCLASSING(C,P);
        end
end

UPDATE-ASCP-AFTER-DESUBCLASSING(C,P)
/* C: A Class */
/* P: ASCP of C */
begin
        foreach S' ∈ immediate superclasses of S: do
            begin
            ASCP(S') ← ASCP(S) ∨ ASCP(S');
            UPDATE-ASCP-AFTER-DESUBCLASSING(S',ASCP(S');
            end
end
```

Chapter 8
Future Directions

In this chapter we provide directions for future work based on this dissertation.

8.1 Schema Evolution

We feel there are four primary issues where future work is desirable for schema evolution: method conversion, grouping schema change operations, concurrency control and authorization. We briefly describe the problems below.

8.1.1 Method Conversion

As schema definitions are modified, existing methods for previous schemas may not be valid. It is important to be able to achieve compatibility of methods in the face of changing schemas. Thus, an obvious question is "Can the affected methods be converted automatically?" The automatic method conversion is a very hard problem because the system should understand the semantics of both methods and schema changes for automatically modifying existing methods in order to comply with schema changes. Therefore, our view is that the system should monitor schema changes and notify the changes to the designer of affected methods when necessary and the method designer should convert the affected methods in accordance with the schema changes. We plan to investigate the notification techniques for method conversion.

Interestingly, the work by Skarra and Zdonik [SZ86, Zdo86] can be viewed from the viewpoint of achieving compatibility of methods in the face of multiple schema versions. They do not try to change existing code, but rather to make its behavior different through the use of error (exception) handlers. In their framework, the users do not have to modify pre-existing programs (i.e., methods) in any way. Existing programs will simply do the right thing because an appropriate exception handler will get control at the right time.

8.1.2 Grouping Schema Change Operations

We have discussed 20 basic schema change operations in our schema evolution framework. The user can accomplish any desired schema change through a combination of these basic operations. We can consider a few useful high level operations that can be implemented by executing a sequence of several basic schema change operations. Furthermore, it is desirable to group schema change operations in order to avoid redundant instance accesses [PS87]. We can view those high level operations as a transaction so that a database may not be corrupted in the middle of executing a set of schema change operations. We plan to identify those high level operations and investigate the efficient implementation of those operations.

8.1.3 Concurrency Control

Schema changes in a shared environment may cause inconsistency of instances: consider two users, one user is modifying a class and another user is updating an instance of the class. A concurrency control scheme, which can handle schema evolution and understand the inherent properties of class hierarchy, is needed. Since schema modification is interactive, transactions will be of long duration. It may be useful to apply long duration transaction protocols such as those of Korth and Speegle [KS88].

8.1.4 Authorization

Authorization is important for effective sharing of data. Schema changes may cause unauthorized reads, writes, and deletes of instances. Schema change operations may have associated rights. For example, some users (with the authorized right of creating classes) can create classes, while others can only look into the instances of classes. The relationship between schema evolution and authorization must be examined.

8.2 PIG: The Formal Model

The PIG model was used in showing the completeness of the schema evolution framework. There is ample opportunity for investigating algebraic properties of the PIG model in developing the theory of object-oriented databases. The followings are interesting problems to pursue:

- Given a PIG P, is there any PIG Q which is equivalent to P in the sense of information (property) carrying?
- Are any two operations of 9 PIG operations commutative?

8.3 Towards an Integrated Graphical Environment

We plan to extend PSYCHO to produce an integrated graphical environment for object-oriented databases. In this section we describe briefly some of the extensions we are considering.

Graphical Query Interface

ORION queries are predicate-based lisp expressions and support relational algebra-like operations [BKK88]. Since the ORION model has the notions of composite objects and multiple inheritance, ORION queries should be able to express complex predicates to navigate the DAG structures of class hierarchies and the tree structures of composite objects. Hence the ORION query language subsumes the power of existing query languages such as SQL or CODASYL DML. Since however ORION queries navigate underlying complex structures such as class hierarchies and composite objects, the user may find it difficult to pose complicated queries. Consider a query asking instances of hundreds of classes or a query to a composite object having thousands of subcomponents. We believe that a graphical query interface and graphical representation of objects will enhance the friendliness of object-oriented query languages.

Graphical Version Controller

There is a general consensus that version control is one of the most important functions in application domains, such as integrated CAD/CAM systems and office information systems with multimedia documents. Users in such environments often need to generate and experiment with multiple versions of an object, before selecting one that satisfies their requirements. So far we have considered two types of graph structures: class hierarchies and composite object hierarchies. The another graph structure to be considered is a "version derivation hierarchy." A node of the version derivation hierarchy may be a CAD object, a software module, or a multimedia document. All of these are considered as composite objects in the ORION model. It is difficult to support version management in a user-friendly manner. We believe that a graphical representation of versions and the relationships among them can assist users with version management.

Composite Object Browser

This tool will be embedded in the previous two tools. Restructuring or querying composite objects will be performed graphically in this environment.

8.4 Schema Versions

In this dissertation, we have investigated the semantics of schema versioning. We hope our methodology for schema versioning will be implemented in the near future. Many issues regarding implementation need to be investigated, including data structures and the overhead during query processing.

8.5 DAG Rearrangement Views

Implementation aspects of DAG rearrangement views have not been addressed in the dissertation. The main issue is to design a data structure to be used for definitions of DAG rearrangement views. It is also worthwhile to investigate the relationship between DAG rearrangement views and authorization.

8.6 Predicate Manager

As shown in chapters 6 and 7, many stages of object-oriented database design and subclassing related tasks need the capability of proving the truth value or the satisfiability of a formula in first order logic. We feel that a proof facility (say, a *predicate manager*) is required that can support the three problems (the type subsumption problem, the constraint membership problem, the undesirable property detection problem) and handle subclassing management. We believe we can benefit significantly from a predicate manager in several aspects of object-oriented database processing. To the best of our knowledge, however, none of the existing object-oriented systems have such a predicate manager. We believe that a predicate manager is needed for next generation of object-oriented database systems. However, conventional AI theorem provers cannot be used as a predicate manager in database environments because of performance reasons. Devising a predicate manager having a reasonable performance is an interesting research topic.

Chapter 9
Summary and Discussion

9.1 Thesis Summary

In this section we summarize the results presented in this dissertation.

In chapter 2, we presented the results of our research on various issues of schema evolution: dynamic changes to a database schema in an object-oriented database environment. These results are presently being incorporated into a prototype object-oriented database system, ORION, at MCC. The schema evolution framework is based on a graph-theoretic model of the class hierarchy. It consists of a set of invariants, those properties of a class hierarchy that must be preserved before and after any schema change operations. Since a number of options are possible for preserving any of the invariants, we also defined a set of rules that will guide the selection of the most meaningful option for any type of schema change operation. Our framework made it possible for us to enumerate possible types of changes to the database schema, and to define the semantics for each of them by applying the sets of invariants and rules. We addressed the issue of completeness of our framework by defining a simple formal model PIG (Property Inheritance Graph) and exploiting the properties of the PIG model.

In Chapter 3, we presented a graphical language PSYCHO which is designed to be a friendly interface for schema design in the ORION object-oriented database system. It is providing us with the opportunity to evaluate the ORION schema evolution framework. We provided a detailed description of PSYCHO using numerous sample sessions. Finally we discussed the implementation of PSYCHO and several other related issues. The technical merits of PSYCHO are:

- Full support of over 34 schema modification operations.
- Easy browsing and navigation on class lattices.

- Representation of complex schemas: Since PSYCHO provides the ability to reorganize class lattices on the screen and to scroll the screen, schemas which are too big to be displayed on the screen can be manipulated.
- Graphical feedback: PSYCHO provides various types of useful visual response during operations.
- Integrity Checking: PSYCHO checks the validity of requested operations by doing associated computations, such as cycle detection and name conflict detection.
- Object-Oriented Implementation: Since PSYCHO was implemented using an object-oriented language, the architecture of PSYCHO is extensible.

In chapter 4 and 5, 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. We believe there are four contributions in chapters 4 and 5:

- 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], it guarantees *minimum storage redundancy* and allows us to get around the problem of *update anomaly*.
- 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. 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. 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 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 is an operational interface for manipulating schema versions and constructing DAG rearrangement views.

In chapter 6, we established a unified framework for the logical design of object-oriented database schema by synthesizing research results of the areas such as AI knowledge representation, database dependency theory, AI theorem proving, and graph algorithms. We identified three problems which are essential in the design steps of object-oriented database design: the type subsumption problem, the constraint membership problem, and the undesirable property detection problem. We borrowed a recent result by Levesque and Brachman [LB86] for the type subsumption problem in object-oriented databases. Further, we characterized the constraint membership problem and the undesirable property detection problem in the three different formal frameworks: an inference rule system, the first order logic system, and the graph theory system. Also in chapter 6, we re-examined the ORION data model and the ORION schema evolution model from the view point of the unified framework of object-oriented database design that we established.

Finally in chapter 7, we presented the results of our result on subclassing in object-oriented databases. We defined the semantics of various types of subclassing. Most subclassings are accompanied by associated constraints, called *subclassing conditions*. We investigated how to use and how to maintain subclassing conditions.

The major application of subclassing conditions is to maintain a class hierarchy in semantically correct states. Managing a huge class hierarchy having hundreds of thousands classes in a consistent way is almost impossible without the aid of subclassing conditions. A similar problem, called *knowledge-base classification problem*, is an important research issue in AI [FS84]. Besides maintaining class hierarchies consistently, subclassing conditions can be used for various applications such as query optimization, reuse of access plans, predicate locking, access control, and so on. However, most of the existing object-oriented systems ignore subclassing conditions.

9.2 Discussion: What Is Really An Object-Oriented Database?

In this section, before closing this thesis out, we would like to discuss some fundamentals of object-oriented databases.

The object-oriented database area is currently a very active research field, but is still quite young in the sense that it does not have any formal common data model yet [Ban88]. There are a number of typical questions which are frequently asked by the people in databases and programming languages: (1) Is there anything we can do with object-oriented databases, but cannot do with relational databases? (2) What are pros and cons of object-oriented databases over relational databases? (3) What are the major differences between object-oriented programming languages and object-oriented database systems? In this section we will briefly discuss those questions.

For the first question, we can think of two aspects: *a computation model for data language* and *system functionality*. Clearly, there is a discrepancy of

computational power between object-oriented databases and relational databases in that the data languages for object-oriented databases are computationally complete (Turing equivalent), whereas the data languages for relational databases are relationally complete. Note that SmallTalk-80 and CommonLisp are the data language for the GEMSTONE and ORION systems respectively. The data languages in relational databases are based on either relational algebra or relational calculus. The notion of relationally completeness is weaker than the notion of computationally completeness in the expressive power of computation. For example, the transitive closure-like operation cannot be expressed in the relational data language. However, one nice thing about the relational data language is that we can get a very efficient optimization scheme for query processing while using simple and declarative queries. Unfortunately, existing object-oriented databases do not provide an ad-hoc and declarative query language. Furthermore, in general, the query optimization issue in object-oriented databases is undecidable because queries in object-oriented database are written in general purpose programming languages such as SmallTalk-80 or CommonLisp.

With respect to system functionality, we can enumerate a number of advanced functions of recent object-oriented database systems which were not available in the conventional relational database systems: schema evolution support, complex object support, version support, unstructured data support, etc.

For the second question, we believe Bancilhon [Ban88] presented a good answer in PODS 88 conference. He characterizes the following features of object-oriented database systems as *pros* over relational database systems: (1) the ability of dealing with complex objects, (2) the notion of object identity, (3) the notion of extensibility by allowing new data types, (4) the ability of encapsulation by storing programs and data at the same time, and (5) the notion of inheritance. He also characterizes the following features of object-oriented database systems as *cons* over relational database systems: (1) the lack of simplicity, (2) the lack of ad-hoc query languages, (3) the lack of declarative

queries, (4) the lack of relational interface, and (5) the lack of speed. As a matter of fact, the answers for the first and second question overlap since the natures of the first and second question are similar.

For the third question, we can think of two major differences: *sharability* and *persistence*. The notion of sharability means 'multiple user support'. Existing object-oriented programming languages are lacking in sharability because they do not support queries, transactions, and concurrency control mechanism. The notion of persistence means 'data outlives programs' which, in general, is not true in programming languages. The another fundamental difference between programming languages and databases is that the amount of data in databases is too big to fit in a main memory whereas, in programming languages, everything is assumed to be in a main memory.

Bibliography

[ABH85] Ahlsen, M., A. Bjornerstedt and C. Hulten, "OPAL: An Object-Based System for Application Development," *IEEE Database Engineering Bulletin*, Vol. 8, No. 4, 1985.

[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.

[AHU76] Aho, A., J. Hoftcraft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1976.

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

[AM86] Arisawa, H. and T. Miura, "On the Properties of Extended Inclusion Dependencies," *IEEE Transactions on Software Engineering*, Vol SE-12, No. 11, November, 1986.

[AP86] Atzeni, P. and D.S. Parker, "Formal Properties of Net-Based Knowledge Representation Schemes," *Proceedings of International Conference on Data Engineering*, 1986.

[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,

[Astr76] Astrahan, M. M., et al., "System R: a Relational Approach to Data Management," *ACM Transactions on Database Systems*, Vol. 1, No. 2, 1976.

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

[BJS86] Böttcher, S., M. Jarke, and W. Schmidt, "Adaptive Predicate Management in Database Systems," *Proceedings on International Conference on Very Large Databases*, August, 1986.

[BK85a] Batory, D. and W. Kim, "Modeling Concepts for VLSI CAD Objects," *ACM Transactions on Database Systems*, Vol. 10, No. 3, September, 1985.

[BK85b] Batory, D. and W. Kim, "Support for Versions for VLSI CAD Objects," *to appear in IEEE Transactions on Software Engineering*, Technical Memo, The University of Texas at Austin, 1985.

[BKK88] Banerjee, J., W. Kim, and K.C. Kim, "Queries in Object-Oriented Databases," *Proceedings of International Conference on Data Engineering*, 1988.

[BKKK86] Banerjee, J., H.J. Kim, W. Kim, and H.F. Korth, "Schema Evolution in Object-Oriented Persistent Databases," *Proceedings 6th Advanced Database Symposium*, Tokyo, Japan, 1987.

[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 Management of Data*, San Francisco, CA, May, 1987.

[BL84] Brachman, R. and H. Levesque, "The Tractability of Subsumption in Frame-Based Description Languages," *Proceedings of AAAI Conference*, 1984.

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

[Ban87] Banerjee, J., H. Chou, H. Garza, W. Kim, D. Woelk, N. Ballou and H.J. Kim, "Data Model Issues in Object-Oriented Applica-

tions," *ACM Transactions on Office Information Systems*, March, 1987.

[Ban88] Bancilhon, F., "Object-Oriented Database Systems," *Principles of Database Systems*, March, 1988.

[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.

[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.

[Car83] Cardelli, L., "A Semantics of Multiple Inheritance," in *Semantics of Data Type*, Springer-Verlag, Computer Science Lecture Note, 1983.

[DF75] Delong, S. and J.P. Fry, "An Approach to the Migration of DBMS Applications," DSRG Working Paper 900, University of Michigan, June, 1975.

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

[EGLT76] Eswaran, K.P., J.N. Gray, R. A. Lorie and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November, 1976.

[FS76] Fry, J.P. and S. Shindler, "Towards the Migration of Database Applications," DSRG Technical Report 76-STI, University of Michigan, April, 1976.

[FS84] Finin, T. and D. Silverman, "Interactive Classification," *Proceedings of IEEE Workshop on Principles of Knowledge-based Systems*, 1984.

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

[GR83] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA., 1983.

[Gold81] Goldberg, A., "Introducing the Smalltalk-80 System," *Byte*, Vol. 6, No. 8, August, 1981.

[Hou75] Housel, B.C., D.P. Smith, N.C. Shu, and V.Y. Lum, "DEFINE: A Nonprocedural Data Description Language for Defining Information Easily," *Proceedings of ACM Pacific Conference*, San Francisco, CA., April, 1975.

[IB84] Israel, D. and R. Brachman, "Some Remarks on the Semantics of Representation Languages," in Brodie, M., J. Mylopoulos and J.W. Schmidt (eds.), *On Conceptual Modeling*, Springer-Verlag, New York Inc., 1984.

[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.

[INTE84] *The Knowledge Engineering Environment*, KEE manuals, IntelliCorp, 1984.

[IRA83] *EDBMS: Concepts and Facilities*, Information Research Associate, 1983.

[KCB85] Katz, R.H., E. Chang and R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," *UCB/CSD 86/270*, University of California, Berkeley, 1985.

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

[KK87a] Kim, H.J. and H.F. Korth, "DOD: Non-First-Normal-Fosl Relation-based Object-Oriented Data Model," working paper (in preparation), The University of Texas at Austin, 1987.

[KK88a] Kim, H.J. and H.F. Korth, "Logical Design of Object-Oriented Database Schema (A Unified Framework)," unpublished memo, The University of Texas at Austin, January, 1988.

[KK88b] Kim, H.J. and H.F. Korth, "PSYCHO: A Graphical Language for Supporting Schema Evolution in Object-Oriented Databases (Extended Abstract)," *Proceedings of Third User System Interface Conference (USICON88)*, Austin, February, 1988 (also The University of Texas at Austin, TR-87-43, December, 1987).

[KK88c] Kim, H.J. and H.F. Korth, "Schema Versions and DAG Rearrangement Views in Object-Oriented Databases", The University of Texas at Austin, TR-88-12, February, 1988.

[KKS85] Kim, H.J., H.F. Korth and A. Silberschatz, "PICASSO: A Graphical Query Language", TR-85-30, University of Texas at Austin, (also to appear in *Software Practice and Experience*, 1988), 1985.

[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.

[KLW87] Kemper, A., P. Lockemann and M. Wallrath, "An Object-Oriented Database System for Engineering Applications," *Proceedings of ACM-SIGMOD Conference on Management of Data*, 1987.

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

[KS88] Korth, H. and G. Speegle, "Formal Model of Correctness without Serializability," *Proceedings of ACM-SIGMOD Conference on Management of Data*, 1988.

[KTKB88] Kim, H.J., B. Twichell, H.F. Korth and D.S. Batory, "Contemporary Non-traditional Database Systems," unpublished memo, The University of Texas at Austin, March, 1988.

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

[Kim85a] Kim, H.J., "Graphical Environments for Database Systems," MSc thesis, The University of Texas at Austin, August, 1985.

[Kim85b] Kim, W., "CAD Database Requirements," *MCC Technical Report*, July, 1985.

[Kim86] Kim, H.J., "Graphical Interfaces for Database Systems: A Survey," *Proceedings of The 1986 Mountain Regional ACM Conference*, Santa Fe, New Mexico, April, 1986.

[Kim87] Kim, W. et al., "Composite Object Support in an Object-Oriented Database System," *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1987.

[Kor84] Korth, H., et al., "System/U: A Database System Based on the Universal Relation Assumption," *ACM Transactions on Database*

Systems, Vol. 9, No. 3, 1984.

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

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

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

[Lee86] Lee, W., "private communication", 1986.

[Len87] Lenzerini, M., "Covering and Disjointness Constraints in Type Networks," *Proceedings of International Conference on Data Engineering*, 1987.

[Lor84] Lorie, R. et al., "User Interface and Access Techniques for Engineering Databases," *IBM Research Report: RJ4155*, IBM San Jose, 1984.

[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," *Proceedings of International Conference on Data Engineering*, Vol. 8, No. 4, 1985.

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

[MSOP86] Maier, D., J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1986.

[MSS79] Munz, R., H.-J. Schneider and F. Steyer, "Application of Sub-Predicate Tests in Database Systems," *Proceedings of International Conference on Very Large Databases*, 1979.

[Nov82] Novak, G. "The GEV Display Inspector/Editor," *Heuristic Programming Project*, HPP-82-32, Computer Science Department, Stanford University, 1982.

[Nov83] Novak, G. "GLISP: A Lisp-based Programming System with Data Abstraction," *The AI Magazine*, Fall, 1983.

[PS87] Penny, D.J. and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS," *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1987.

[PSM87] Purdy, A., B. Schuchardt and D. Maier, "Integrating an Object-Server with Other Worlds," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, 1987.

[RH80] Rosenkrantz, D. and Harry B. Hunt, III, "Processing Conjunctive Predicates and Queries," *Proceedings of International Conference on Very Large Databases*, 1980.

[RRP74] Ramirez, J.A., N.A. Rin and N.S. Prywes, "Automatic Generation of Data Conversion Programs using a Data Description Language," *Proceedings of ACM SIGMOD Workshop on Data Description, Access and Control*, May, 1974.

[Rowe86a] Rowe, L., "A Shared Object Hierarchy," *Proceedings of International Workshop on Object-Oriented Database Systems*, 1986.

[Rowe86b] Rowe, L. et al., "A Browser for Directed Graphs," *UCB/CSD 86/292*, University of California, Berkeley, April, 1986.

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

[SDF77] Swartwout, D.E., M.E. Deppe, and J.P. Fry, "Operational Software for Restructuring Network Databases," *Proceedings of National Computer Conference*, 1977.

[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.

[SHL75] Shu, N.C., B.C. Housel and V.Y. Lum, "CONVERT: a High Level Translation Definition Language For Data Conversion," *Communications of the ACM*, Vol. 18, No. 10, October, 1975.

[SL77] Su, S.Y.W. and B.J. Liu, "A Methodology of Application Program Analysis and Conversion Based on Database Semantics," *Proceedings of International Conference on Very Large Databases*, ACM, N.Y., 1977.

[ST82] Shneiderman, B. and G. Thomas, "Automatic database system conversion: Schema Revision, Data Translation, and Source-to-source Program Translation," *Proceedings of National Computer Conference*, 1982.

[SZ86] Skarra, A.H. and S.B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications*, Portland, OR., September, 1986.

[Serv86] "Programming in OPAL," Servio Logic Development Corporation, Oregon, 1986.

[Shn78] Shneiderman, B., "A Framework for Automatic Conversion of Network Database Programs under Schema Transformations," *Third Jerusalem Conference on Information Technology*, J. Moneta, ed., North-Holland, Amsterdam, 1978.

[Shu77] Shu, N.C. et al., "EXPRESS: a Data Extraction, Processing and Restructuring System," *ACM Transactions on Database Systems*, Vol. 2, No. 2, 1977.

[Smit71] Smith, D.C.P., "An Approach to Data Description and Conversion," PhD Dissertation, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, PA., 1971.

[Stef83] Stefik, M. et al., "Knowledge Programming in LOOPS," *The AI Magazine*, Fall, 1983.

[Su76] Su, S.Y.W., "Application Program Conversion Due to Database Changes," *Proceedings of International Conference on Very Large Databases*, Brusels, Belgium, September, 1976.

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

[Symb85] "Symbolics Manuals," Symbolics Inc., Cambridge, MA., 1984.

[Wied86] Wiederhold, G., "Views, Objects, and Databases," *IEEE Computer*, December, 1986.

[WK87a] Woelk, D. and W. Kim, "An Extensible Framework for Multimedia Information Management," *IEEE Database Engineering Bulletin*, Vol. 10, No. 2, 1987.

[WK87b] Woelk, D. and W. Kim, "Multimedia Information Management in an Object-Oriented Database System," *Proceedings of International Conference on Very Large Databases*, 1987.

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

[ZH85] Zara, R.V. and D.R. Henke, "Building a Layered Database for Design Automation," *22nd Design Automation Conference*, 1985.

[Zdo85] Zdonik, S., "Object Management Systems for Design Environments," *IEEE Database Engineering Bulletin*, Vol. 8, No. 4, 1985.

[Zdo86] Zdonik, S., "Maintaining Consistency in a Data with Changing Types," *ACM SIGPLAN Notices*, September, 1986.