ELSEVIER

Information and Software Technology 40 (1998) 157–173

INFORMATION AND SOFTWARE TECHNOLOGY

A model of schema versions for object-oriented databases based on the concept of rich base schema[1]

Sang-Won Lee*, Hyoung-Joo Kim

Department of Computer Engineering, Seoul National University, Shilim-Dong Gwanak-Gu, Seoul 151-742, South Korea

Received 13 October 1997; received in revised form 23 February 1998; accepted 25 February 1998

Abstract

In this paper, we propose a model of schema versions for object-oriented databases called RiBS. At the heart of this model is the concept of the rich base schema called (RiBS). In our model, each schema version is in the form of a class hierarchy view over one base schema, called RiBS, which has richer schema information than any existing schema version in the database. Users are supposed to be concerned only with schema versions. Direct schema updates on schema versions are allowed, and their effects are, if necessary, automatically propagated to RiBS. We first describe the structural part of the model and then introduce a set of invariants that should always be satisfied by structural parts. As the third element of our model, we give a set of schema update operations, the semantics of which are defined, so as to preserve all the invariants.

Another contribution of this paper is the work on schema-version-merging within the RiBS model. We identify several conflicts in schema-version-merging, and then provide a semi-automatic schema-version-merging algorithm to resolve these conflicts. This algorithm is semi-automatic in the sense that it requires minimal user involvement during schema-version-merging. © 1998 Elsevier Science B.V.

Keywords: Schema version; Schema evolution; View; Schema integration

1. Introduction

One of the remarkable differences between object-oriented database management systems (OODBMS) and relational database management systems (RDBMS) is support for schema evolution. Object-oriented data models emerged in the mid 1980s and since then many approaches to schema evolution have been proposed [1–3]. This is mainly because target applications of OODBMSS, such as CAD/CAM, CASE, and multi-media frequently require dynamic schema changes and flexible schema management. Currently, several commercial OODBMSS, such as GemStone [2], O2 [3], ObjectStore [4], and Objectivity [5] support various schema update primitives and provide on-line schema evolution mechanisms. In addition, some products, such as O2, Objectivity and ObjectStore, support user-defined functions for schema updates.

Under these systems, however, only a single schema can exist at any point in time; if a schema evolution operation completes, the previous schema state is no longer maintained. This single schema modification mechanism has several drawbacks [6]. First, schema updates may invalidate programs written against old schema. Second, because all the users share a single schema, schema updates by one user may change the views of all the other. users. Schema version mechanisms were introduced to overcome these problems, and many researchers have stressed their importance since a characteristic of design applications is being able to cope with frequently changing meta-data [7,8].

1.1. Rejuvenation of schema versions

Recently, the necessity for schema versions has been re-motivated in several new OODB application areas including Repositories [9–11], Portable Common Tool Environment (PCTE) [12–14], and the Worldwide Web (WWW) [15,16], all of which may use an OODBMS as an integrator of data.

Data repositories are expected to be one of the important new uses of DBMS technology ([11]). Among many requirements for repository management systems, the ability to change the structure of information and its

* Corresponding author. Fax: 0082 227 16945, e-mail: swlee@oopsla.snu.ac.kr

[1] This research was partially supported by the Ministry of Trade, Industry, and Energy of KOREA under project 943-20-4, "Implementation of Design Tools for Object-Oriented Database".

PII S0950-5849(98)00037-8

meta-data without breaking existing applications (that is, the functionality of schema versions) is mandatory ([10,11]).

Another strong requirement for the functionality of schema version comes from PCTE, where, as Loomis says [13], an important role of the OODBMS is to manage PCTE schema, support their evolution over time, and support the resulting schema versions. In fact, schema evolution and schema version management become a more serious problem in PCTE [12].

Finally, WWW applications, needless to say the most promising areas for OODBMS [15–17], also require schema versions due to their dynamic nature. Atwood points out in Ref. [15] that web sites need to publish new versions of their applications with new database schema versions without changing the existing versions of the applications and their schemas.

Much work has been done to provide schema version mechanisms for objectoriented databases (OODBS) [7,18–20], but they have not reached a satisfactory status yet. Traditional schema version approaches has three outstanding problems: (1) storage overhead for redundant objects [7,20], (2) limited schema update capability [18], and (3) complexity for managing consistent schema versions [19]. Refer to Section 9 for the details. We believe that the lack of flexible schema management and efficient schema version mechanisms in current OODBMSs is one of the major obstacles for their wide acceptance in the market. Consequently, it is essential to fill the gap of flexible schema management between application requirements and current OODBMS functionalities.

1.2. Our perspectives and paper organization

A database schema is a representation of entities and their semantics in the real world, which the database is intended to model. In our view, a schema version in an OODB is another schema, the purpose of which is either to represent a semantically significant snapshot of a schema at a point of time under the ever-evolving real world, or to customize different, but simultaneous user perspectives.

In this paper, we propose a simple, yet powerful, model of schema versions for OODBS, based on the concept of the rich base schema (RiBS). The remainder of this paper is organized as follows. Section 2 gives a brief overview of the RiBS model using an illustrative example. Section 3 describes the object model assumed in this paper. A detailed description of each component of the RiBS model is given in Sections 4–6, respectively. Section 7 touches upon several issues about the implementations of the RiBS model. Section 8 deals with schema-version-merging, another contribution of this paper. Our work is compared to related work in Section 9. Section 10 concludes the paper with a summary and an outline of future work.

2. Basic idea

In this section, we illustrate some basic ideas of the RiBS model with an example. A detailed description of each component of the model will be given in the following sections. For brevity, in this section, we assume the following informal description of a schema in an OODB: A schema in a database consists of classes that are organized into a class hierarchy through 'is-a' (ISA) relationships between them. Each class, in turn, consists of properties including both attributes and methods. To every class is attached a collection of objects, extent. Each instance object belongs to the extent of a single class, and is referred to as a direct instance of the class.

2.1. Rich base schema and schema versions

Before proceeding with the example, we motivate the concept of 'rich base schema', and discuss how it can be exploited in supporting schema versions. We say that schema S1 is richer in schema information than schema S2 if all conditions set out below hold.

(1) For each class in schema S2, there is a corresponding class in S1;
(2) for each property of a class in schema S2, there exists a corresponding property in the corresponding class of S1;
(3) for every direct ISA relationship in schema S2, there is also a corresponding ISA relationship (direct or indirect) in schema S1.

If a schema S1 is richer than another schema S2, it means intuitively that S1 has more schematic information than S2. This, in turn, means that S2 can be specified as a subset view of S1. This concept of rich schema can be re-stated in terms of relative information capacity [21,22]; S1 dominates (or subsumes) S2.

Our model is based on this concept of rich schema. A physical base schema, RiBS (rich base schema), which is richer in schema information than any existing schema version, is maintained, and every schema version is represented as a view over RiBS. In addition, when a schema update is imposed on a schema version, RiBS is, if necessary, automatically augmented so as to be richer than the modified schema version in addition to all other schema versions. In summary, a schema version is an updatable class hierarchical view[2] over RiBS.

In our model, schema versions are strictly separated from RiBS. This separation enables the prevention of problems from occurring when the schema information of schema versions is mingled with that of RiBS. Some previous works on views in OODB [18,23] put normal classes and derived views together in one class hierarchy. However, this

[2] 'Updatable' means that the schema evolution operation can be directly imposed on the view.

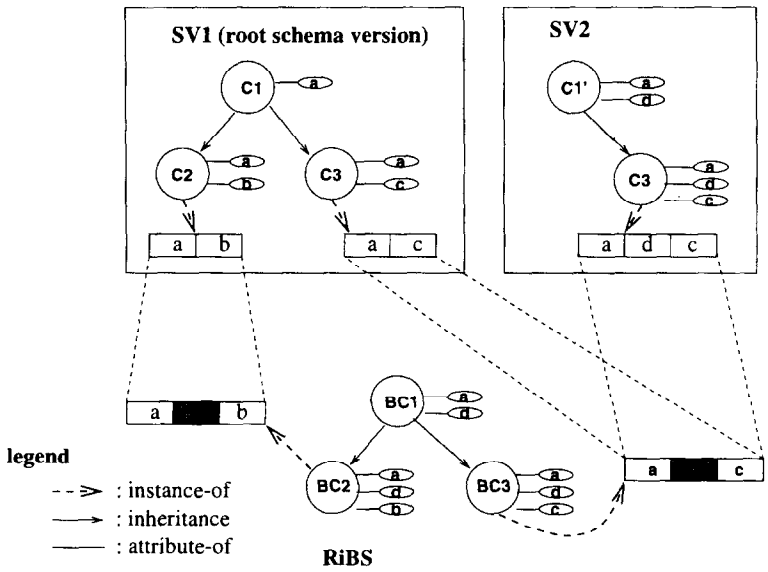

Fig. 1. RiBS and schema versions: an example.

approach has several disadvantages [24]. First, it is difficult for users to understand the complicated class hierarchy. Next, the extents of classes in the hierarchy may overlap. Finally, it is difficult and, in certain cases, impossible to decide where to locate the view class in the class hierarchy.

2.2. An intuitive example

Now let us consider the example in Fig. 1, where two schema versions SV1 and SV2 are represented as views over RiBS. As illustrated in Fig. 1, schema information in RiBS is rich enough to contain all the classes and properties for either SV1 or SV2. The base class corresponding to a class version in a schema version is called the direct base class of the class version. For example, BC1 in RiBS is the direct base class of class version C1 in SV1.[3] Every instance object in the direct base class becomes an element of the logical extent of the class version. In Fig. 1, both SV1 and SV2 share instance objects in RiBS. Under a specific schema version, an instance object of a class version is derived from an instance object of the corresponding base class through projection.

In Fig. 1, we assume that after SV1 is created initially (we call it the root schema version) and then SV2 is derived from it, SV2 undergoes three schema updates as follows: (1) rename class version C1 as C1′, (2) drop class version C2, and (3) add attribute version d to C1′. We now describe how each schema update affects SV2 and/or RiBS.

In our model, each schema version maintains names for its own class versions and their property versions, independently of base classes and properties in RiBS. Thus, the first operation renames the class version C1 as C1′ only within SV2, without affecting RiBS or SV1. The second operation just drops the schema information of class version C2 from SV2, without effecting RiBS or SV1. However, this operation, in contrast to the previous operation, raises a subtle semantic issue: that is, the effect of dropping c2 on c2's logical extent. For this, the RiBS model chooses the semantics to migrates all the (logical) direct instances of C2 to the extent of a superclass. Thus, class version C1′ has base classes BC1 and BC2 as its extental base classes. Details will be discussed in Section 6. The last operation, adding attribute d to class version C1′, is different from the above two operations as it requires a change in RiBS as well as changes in SV2: a corresponding attribute should be added to the direct base class of C1′ in RiBS. Attribute d in BC1 is the result of this operation.

In this section, we gave an overview of the RiBS model. The RiBS model has three components: (1) the structural part, which consists of schema versions and a RiBS, (2) a set of invariants to preserve semantic consistency within the structural part, and (3) a set of schema manipulation operations, which are applicable to the schema versions. In Sections 4–6, we will elaborate on these components, respectively.

3. Object model

This section defines the object model assumed in this paper, which is common to RiBS and schema versions. A class C in a database defines the properties of objects. Each class maintains its extent. A property represents either an attribute or a method.[4] Each class may have more than one

[3] In this paper we use the naming convention such as BC1 and BC2 for classes in RiBS. This is only for illustrative purpose. Mechanisms such as object identity (OID), which can uniquely identify class and properties within a system, will be sufficient.

[4] In this paper, we use the term 'property' to represent both attributes and methods. When we need to distinguish them, we will use the specific terminology.

Table 1
Notations for object model

Term	Description
C	Class
S	*RiBS* or schema version
$\mathcal{T}$(S)	Set of all classes of schema S
C.name	Class name
p	Property
Parents(C)	Set of direct parents of C
Parents * (C)	Set of all parents of C
ISA(C_1,C_2)	C_1 is a direct subclass of C_2
ISA * (S)	All direct or indirect ISA relationships within schema S
Interface(C)	Interface of class C (that is, the set of properties)
InheritedPr ops(C)	Inherited properties of C
LocalProps(C)	Locally defined properties of C
Extent(C)	Extent of class C (set of direct instances)
Org(p)	Original property of p

Table 2
Axioms for inheritance

(1) Axiom of closure	$\forall C \epsilon \mathcal{T}$, Parents($C$) $\subset \mathcal{T}$
(2) Axiom of acyclicity	$\forall C \epsilon \mathcal{T}$, $C \notin$ Parents * (C)
(3) Axiom of rootedness	Parents (Object) = {} $\wedge$ $\forall C \epsilon \mathcal{T}$-{Object}, Object$\epsilon$Parents * ($C$)
(4) Axiom of interface	Interface(C) = InheritedProps (C) $\cup$ LocalProps(C)
(5) Axiom of property inheritance	InheritedProps(C) = $\cup_{C' \epsilon \text{Parents}(C)}$Interface($C'$)
(6) Axiom of superclasses	Parents * (C) = $\cup_{C' \epsilon \text{Parents}(C)}$Parents * ($C'$) $\cup$ Parents(C)

superclass, that is, multiple inheritance is supported. The set of direct superclasses of a class C is denoted as Parents(C). All the properties of the superclass(es) are inherited into the subclass. The newly defined local properties of a class C, denoted as LocalProps(C), together with the inherited ones, denoted as InheritedProps(C), constitute the interface of the class, Interface(C). For an inherited property p of a class, there exists an origin property, denoted as Org(p), from which p is inherited. The notations for the object model are summarized in Table 1. The transitive closure of Parents(C), namely the set of all the direct or indirect superclasses of class C, is denoted as Parents * (C).

Table 2 summarizes the inheritance semantics of the object model. As pointed out in Ref. [25], although much research has been focussed on inheritance, researchers rarely agree on its meaning and usage. Even in ODMG-93[5] [26], no clear semantics are given for inheritance, in particular multiple inheritance. Thus, we need to develop these axioms to clarify the inheritance semantics in RiBS model.

Through axioms 1 to 3 in Table 2, we force a schema to be a direct acyclic graph (DAG). Axiom 1 says that all the superclasses of any class in schema S should be also members of $\mathcal{T}$(S). Axiom 2 requires that there be no cycle in the class hierarchy, and axiom 3 does require that a single class Object in $\mathcal{T}$ be the root of the class hierarchy. A schema conforming to these three axioms results in a DAG [1].

[5] An object database standard from ODMG (object database management group).

Axiom 4 means that, as mentioned above, the interface of a class consists of inherited properties and locally defined properties. The inherited properties of a class C, as stated in axiom 5, are the unions of the interfaces of all the superclasses of C. Axiom 6 means that Parents * (C) is the transitive closure of P relationships of class C. According to these axioms, a class, of which two or more superclasses share a common superclass, inherits properties from common superclass only once (like virtual inheritance in C++). Name conflicts between two or more properties of different superclasses, or between inherited and locally defined properties are allowed and users are responsible for designating a specific property (also similar to C++).

4. Structures

The structural component of the RiBS model has a three-level architecture: (1) the (extensional) obiect base, (2) the rich base schema (RiBS), and (3) the schema versiions. Every object physically resides in the extensional object base. RiBS accumulates all the necessary schema information ever defined in any one of the schema versions. Each schema version is in the form of a class hierarchy view over this RiBS. Users are concerned only with the schema versions in the uppermost layer. Direct schema updates on schema versions are allowed, and their effects are, if necessary, automatically propagated down to RiBS. In this section, we give descriptions of all the structural components of the RiBS model.

4.1. Definition 1 (Base schema, RiBS)

In a database, there exists a single (rich) base schema called RiBS, which describes the structures of objects physically stored in the database. RiBS includes a set of base classes, and inheritance relationships between them constitute the class hierarchy of RiBS. The structure of each object stored in an object base conforms to the definition of the base class in RiBS to which the object belongs. To each base class of RiBS an extent is attached, which is the set of all the direct instance objects of the class.

4.2. Definition 2 (Schema version, SV)

A schema version, SV, is a logical class hierarchy view over RiBS, which represents either a snapshot of the ever-revolving database schema at a certain point of time, or a customized view for a particular user. A schema version, SV, is a class hierarchy view because the schema version itself is also a class hierarchy. It is also a logical view in the sense that all the objects visible through a schema version are derived from the objects stored in the extensional object base.

4.3. Definition 3 (Current schema version, CSV)

We call a specific schema version, under which application programs/users access and manipulate the database at a certain point of time, current schema version (CSV). With the RiBS model, a user should designate a schema version as CSV before (s)he accesses the database. A user can do all the normal database operations against the CSV. Moreover, users can change the schema structure of CSV; that is, the schema can evolve. We will give a detailed description of this, mechanism in a later section and describe the semantics of schema changes on schema versions.

In the RiBS model, the execution of a schema evolution operation, however, does not imply derivation of a new schema version. Instead, we provide an operation for users to explicitly derive a new schema version from existing ones: the former is called the 'child schema version' and the latter 'parent schema version(s)'. Derived-from relationships between schema versions constitute the schema version derivation graph, defined as follows:

4.4. Definition 4 (Schema version derivation graph, SVDG)

A schema version derivation graph (SVDG) is a directed acyclic graph, where each node represents a schema version and each directed edge between nodes represents a 'derived-from' relationship. When a database is initialized at its creation time, the 'root schema version' is created, in addition to the initial RiBS. The root schema version is the root of SVDG.

4.5. Definition 5 [Class version, CV, and direct base class, B(CV)]

A class version CV of a particular schema version represents a facet of a base class in RiBS, which needs to be modeled within the schema version. We call the base class the direct base class of CV, and formally denote it as B(CV). For each class version CV, there is one and only one direct base class B(CV) in RiBS. However, the converse is not true, that is, a base class in RiBS may not need to be explicitly modeled in a schema version SV, so there might not be a corresponding class version in SV. With respect to schema information capacity, B(CV) is a superset of CV; that is, B(CV) has all the schema information necessary for CV. The purpose of maintaining the direct base class is as follows. When a specific schema update is imposed against CV and its effects need to propagate to RiBS, the schema change in RiBS starts from B(CV).

As mentioned earlier, users are concerned only with schema versions. Hence, each class version in a schema version, like normal classes, is expected to have its own class extent. For this, we maintain extental base classes for each class version.

4.6. Definition 6 [Extental base classes, B^+ (CV)]

The extental base classes of a class version CV, $B^+(CV)$, are a set of base classes in RiBS. The union of the extents of these base classes comprises the logical extent of CV. From these extental base classes, the logical extent of a class version CV is derived as follows.

$$\text{Extent(CV)} = \bigcup_{C' \epsilon B^+(CV)} \text{Extent(C')}$$

As will be discussed later, the set of base classes in $B^+(CV)$ is a connected subgraph of RiBS, rooted at B(CV) (thus, we employ the notation B^+). Therefore, all the objects in $B^+(CV)$ carry values for all the properties necessary in CV.

4.7. Definition 7 [Property version, PV and direct base property, B(PV)]

A property version, PV, of a class version, CV, represents either an attribute or a method of the CV. The direct base property of a PV in a class version CV, denoted as B(PV), is a corresponding property of B(CV). Every PV in a schema version has its direct base property.

The purpose of maintaining a direct base property for each PV is as follows. When a logical object is retrieved through CSV, its values are derived from the corresponding physical object. In this process, the value of each PV is derived from that of the B(PV) of the physical object. The concept of extental base classes, however, complicates this process in that, if the corresponding physical object of an object being accessed under CSV is an instance of a base class which is not a direct base class of any class version of

CSV, how do we derive the value of each PV from the physical object? As stated in the previous section, we assume that in RiBS, a subclass inherits all properties from its superclass(es) and keeps the property information locally, as in Orion [1]. Thus, when deriving the value of each PV, the value of the property which has B(PV) as its origin property is used.

5. Invariants

In this section, as a second component of the RiBS model, we introduce a set of invariants which should always be satisfied by the structural part. Moreover, this set of invariants plays a critical role in defining the semantics of schema evolution operations for schema versions, as discussed in the next section.

Invariants in the RiBS model can be classified generally into three categories: RiBS invariants, schema version invariants, and invariants between RiBS and schema version. In this paper, we describe the last two categories. With respect to RiBS, we assume the well known invariants for schema evolutions, such as DAG invariance, name invariance, origin invariance, full inheritance invariance, and no redundant ISA relationships from [1–3,27].

5.1. Invariants on schema version

For schema versions, in addition to the traditional invariants for schema evolutions, we identify two new invariants, 'no phantom reference' and 'no multiple classification', both of which are related closely to the schema evolution operation class drop. Incorrectly defined semantics for this operation might result in some anomalies. In the next section, we will explain how these two invariants guide the semantics of class drop.

5.1.1. Invariant 1 (no phantom reference)

The value of an attribute of an object may be a reference to a phantom object, which is not a direct instance of any class in the schema version. We refer to this kind of reference as a phantom reference. This is in contrast to a dangling reference, which is a reference to non-existing object. There should be no phantom references within a schema version SV: that is, within SV• for each object O referenced by another object, there should exist a class version CV, where O ϵ Extent(CV).

5.1.2. Invariant 2 (no multiple classification)

This invariant restricts each logical instance object in a schema version to be a direct instance of only one class version. In other words, logical extents of each class version in a schema version should be disjoint to each other, which can be formalized as follows:• for every class version CV_i and CV_j in a schema version SV, where $i \neq j$, Extent(CV_i) $\cap$ Extent(CV_j) $= \phi$.

5.2. Invariants between RiBS and schema version

As mentioned above, each schema version is a logical view over RiBS. The following invariants should hold between each schema version SV and RiBS.

5.2.1. Invariant 3

For each class version CV (and property version PV) in a schema version, there should exist a corresponding B(CV) (and B(PV)) in RiBS.

5.2.2. Invariant 4

Within a schema version, for each base class C in RiBS, there should exist a class version CV, such that C ϵ B^+(CV). The above invariant means that the union of the extental base classes of all class versions in a schema version should be equal to the set of base classes in RiBS, as formalized in the following.

$$\bigcup_{CV_k \epsilon \mathcal{T}(SV)} B^+(CV_k) = \mathcal{T}(RiBS)$$

6. Operations

In this section, we give a set of operations for schema version management, which is the last component of the RiBS model. These operations are classified into two groups: one group is concerned with SVDG manipulation, while the ther includes schema evolution operations against schema versions.

- Operations for SVDG manipulations.

(1) Derive sv-name from parent-list;
(2) Delete sv-name;
(3) Set current schema version to sv-name.

- Operations for schema evolution.

(1) Operations which have no impact on RiBS;

(a) Change the name of a class version *C*,
(b) Drop an existing class version *C*,
(c) Drop an existing property version *v* from a class version *C*,
(d) Drop an edge to remove a class version S as a superclass of another class version *C*,
(e) Change the ordering of superclasses of a class version *C*.

(2) Operations which have impacts on RiBS;

(a) Add an edge to make a class version *S* a superclass of class version *C*,
(b) Add a new property version *v* to a class version *C*.

(3) Operations which have *impacts* both on *RiBS* and on other schema versions;

(a) Create a new class version *C*.

Schema evolution operations available in RiBS model are similar to those from Refs. [1,2]. In this paper, we make a new taxonomy for the eight fundamental schema change operations of Orion, depending on their impacts on RiBS and other schema versions. In the rest of this section, we will describe each of these schema version management operations in greater detail.

6.1. Operations for SVDG manipulations

Derive sv-name from parent-list as mentioned before, this operation is used to derive a new schema version sv-name from existing ones in parent-list. When a schema version is derived from a single parent, this operation can easily be implemented. That is, the schema information of the parent is simply copied into the child. At this point, the schema information of both parent and child, including class versions and their property versions, is exactly the same. In the case of multiple parents, however, the parent schema versions with different structural information should be merged into a new consistent one. We call this process 'schema-version-merging'. In the next section, we will discuss some issues about schema merging and provide our solutions in, detail.

Delete sv-name When a schema verion is no longer needed, this operation is used to remove it from SVDG and delete its schema information from the database; that is, all the class versions and their property versions are deleted. We do not allow to delete root schema version. When a schema version is removed from SVDG, its parent schema versions become new parent schema versions of its child schema version (if any).

Set current schema version to sv-name As mentioned above, every program or query in the RiBS model should be written against a specific schema version, called the current schema version (CSV). This operation is used to designate CSV before applications or query accesses to the database.

6.2. Schema evolution operations

The schema evolution operations of the first group require changes only in the schema information of CSV. In this respect, they are related to earlier works on simulating schema updates using the OODB view [18,24]. However, these approaches have a serious drawback, which is that they do not support operations from our last two groups.

In this subsection, we will explain three representative operations from each group, which raise subtle issues in defining their semantics. In the next subsection, we will give a formal semantics for all the eight operations. For complete descriptions of all the operations, refer to Ref. [28].

6.2.1. Drop an existing class version C

This operation drops a class version C from CSV. C is dropped out from the subclass list of each class version in Parents(C) and from Parents(C_{sub}) of each subclass version C_{sub} of C, if any. If C is the only superclass of any subclass C_{sub}, class versions in Parents(C) become new superclasses of C_{sub} [1]. All the properties that are locally defined within C are also dropped from all its subclasses.

As mentioned before, a (logical) extent in the RiBS model is attached to each class version. Thus, when deleting a class version, we should consider the issue of how to deal with its extent. In relation to this issue, there have been at least two reasonable approaches for class drop [1,3,4] in the area of schema evolution. In the first approach, all the instance objects of a class are deleted from the database [1,4]. However, this semantics, as pointed out in Ref. [1], may introduce the dangling reference problem. A commercial OODBMS, ObjectStore [4], overcomes this problem by nullifying all the references to the deleted instance objects. However, this in turn, makes the operation potentially very time consuming [4]. In the second approach, which is exemplified by the O2 system ([3]), the class drop operation is allowed only if the extent of the class is empty.

In the RiBS model, there could be another possible approach to class drop, where all objects in Extent(C) are filtered out from CSV. According to this approach, objects in Extent(C) cannot be accessed through the extent of any class within CSV when the class drop operation is completed. However, it should be noted that all the physical objects still exist in RiBS. This approach seems to be similar to the view mechanism in relational databases, which provides the functionality of content-based authorization [29], in that it hides some objects from the view of the user. Many other researchers have anticipated that some kind of view mechanism for OODB will also provide the same functionality of content-based authorization [18,24].

However, navigational object access through object identity (OID) in the object-oriented data model is drastically different from the relational data access paradigm where the only unit of access is either table or view. In the object-oriented data model, a class may be used as the domain of an attribute of another class. Hence, an object may have the OID of another object as its value for an attribute. This characteristic of object traversal through OID introduces the 'phantom reference' problem under our previous semantics of class drop. As shown in Fig. 2, even after a class version CV is dropped, the object $c1$ is still accessible through object $a2$. Under the last semantics, however, object $c1$ cannot be accessed through the extent of any class version in CSV; that is, $c1$ is a phantom object. It should be noted that the phantom reference problem is not confined to the RiBS model. Any view mechanism in OODB should consider and solve this phantom reference problem in order to provide for the functionality of content-based authorization. This phantom reference problem leads us to choose a compromised semantics for class drop. Within CSV, all objects in Extent(C) are migrated (logically) to the extent of a superclass of C. For example,

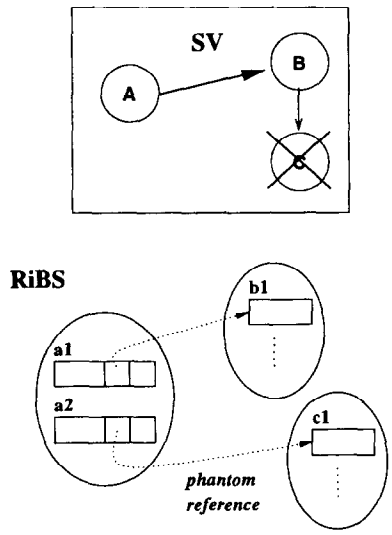

Fig. 2. Phantom reference.

in Fig. 2, all objects in Extent(C) are migrated to Extent(B) after the class version C is dropped.

Under this semantics for class drop, multiple inheritance complicates the situation: to which superclass should the logical extent of the class being dropped migrate? In order to guarantee invariant 2, we require users to explicitly designate a target superclass in the RiBS model.

6.2.2. Add an edge to make class version S a superclass of class version C

This operation adds a class version S to Parents(C). This operation is rejected if it introduces a cycle or a redundant ISA within CSV. C inherits all the properties of S. This operation also affects RiBS, except in the following two cases. The first case is where S is deleted from Parents(C) within CSV before this operation occurs. The second is when another schema update in another schema version has already had the required effect on RiBS. These two cases can be inferred by checking whether [B(S),B(C)] is in ISA * (RiBS). In the case where [B(S),B(C)] is not in ISA * (RiBS), B(S) is added into Parents[B(C)].

In addition, to ensure no redundant ISA relationship in RiBS, the existence of any direct or indirect superclass of B(S) in Parents[B(C)] in RiBS should be checked. If one exists, the inheritance relationship is removed from RiBS. This situation is exemplified in Fig. 3, where we assume that schema version SV_j was derived from SV_i and class version A was made a new superclass of class version C in SV_i. Then, when a schema update which adds class version B to Parents(C) is imposed on SV_j, a new edge from B(B) to B(C) is added and the edge from B(A) to B(C) is deleted. This is required to avoid redundant ISA relations in RiBS. Note that, for ISA(A,C) in SV_i, the corresponding [B(A), B(C)] exists in ISA * (RiBS).

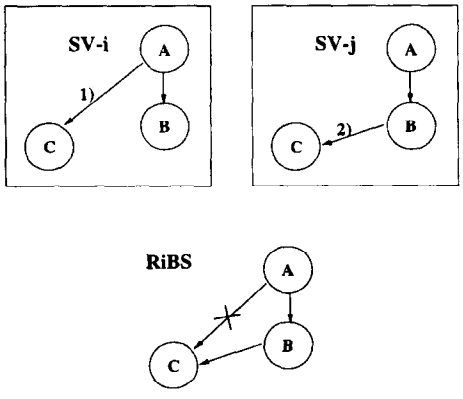

Fig. 3. Addition of superclass to a schema version.

6.2.3. Create a new class version C as a subclasss of S

This operation creates a new class version in CSV. If any class version with same name already exists in CSV, the operation is rejected. To satisfy the invariant 3, a new base class B(C) needs to be created in RiBS. Direct base classes of each superclass of C become the superclasses of B(C). In addition, the direct base class of each domain of an attribute defined in C becomes the domain of the attribute in B(C). The base class is created with the superclass list and property list, and then the new base class is set to B(C).

This operation affects other schema versions, in addition to RiBS. According to invariant 4, B(C) needs to be included into extensional base classes of an appropriate class version in schema versions other than CSV. In order to do this, we choose the following solution: "in schema versions other than CSV, add B(C) to the extental base classes of a class version CV which has B(S) as its extental base class" (refer to formal semantics in the next section).

6.3. Formal semantics of operations

In this subsection, we give the formal semantics of each operation based on the informal semantics from the previous subsection. According to our taxonomy of schema evolution operations, the formal semantics of the three schema evolution operation groups are given in Tables 3–5, respectively. The meanings of all the other operations used in the tables, except for the operation $\alpha(C,p)$, are self-explanatory.

The operation $\alpha(C,p)$ sets a corresponding direct base property for each property version of subclasses of class version C, which has the property version p as its origin property version. To illustrate the semantics of this operation, we use the example in Section 2.2. As shown in Fig. 1, the newly added attribute d to class version C1′ in SV2 is inherited by class version C3. This inherited attribute d of C3 also needs its base property. Thus, base attribute d in base class BC3 in RiBS should be set as base property of d in class version C3. This can be done using the operation $\alpha(C1',d)$. In this paper, we assume that local property versions defined within a class version are inherited into all the direct and indirect subclasses. In addition, according to full inheritance semantics, each subclass keeps the

Table 3
Semantics of operations with no impact on RiBS

Operations	Semantics (only current SV)
Change the name of a class version C	set C.name to new name
Drop an existing property version p from a class version C	**if** $p \notin$ LocalProps(C) **then** reject **else** drop p from LocalProps(C)
Change the ordering of superclasses of a class version C	no action (semantics of unordered inheritance)
Drop an edge to remove class version S as a superclass of class version C	**if** Parents(C) = {S} **then** **if** S = Object **then** reject **else** Parents(C) = Parents(S) **else** remove S from Parents(C)
Drop an existing class version C	**if** C = Object **then** reject **else** $B^+(C_{sup}) = B^+(C_{sup}) \cup B^+(C)$; **for each** subclass C_{sub} of C **if** Parents(C_{sub}) = {C} **then** Parents(C_{sub}) = Parents(C) **else** remove C from Parents(C_{sub});

Table 4
Semantics of operations with impact only on RiBS

Operations	Semantics	
Add a new property version p to a class version C	RiBS	create a base property bp add bp to LocalProps(B(C))
	CSV	create a property version p set B(p) as bp add p to LocalProps(C) $\alpha(C,p)$
Add an edge to make classversion S as a superclass of C	RiBS	**if** B(S) ϵ Parents * [B(C)] **then** no action **else** add B(S) to Parents[B(C)] **if** $\exists bc$ {ϵ Parents[B(C)]} ϵ Parents * [B(S)] **then** remove bc from Parents[B(C)]
	CSV	add S to Parents(C) **for all** new properties p inherited from S, $\alpha(C,p)$

Table 5
Semantics of operations with impact on RiBS and other schema versions

Operations	Semantics	
Create a new class version C as a subclass of S	RiBS	create a base-class bc add B(S) to Parents(bc)
	CSV	set bc as B(C) add S to Parents(C) initialize $B^+(C)$ to {bc} **for all** $p \epsilon$ LocalProps(C), $\alpha(C,p)$
	other SV s	**if** $\exists$ C_j in SV, where B(S) ϵ $B^+(C_j)$ **then** add bc to $B^+(C_j)$

inherited property versions independently from the original class version. For these inherited property versions, direct base properties are also necessary. All the subclass versions of class version C have their own direct base class. All these base classes are also subclasses of B(C) in RiBS. Moreover, for each base class there exists an inherited base property having B(p) as its origin property. $\alpha(C,p)$ traverses all the subclass versions of C, finds direct base property of each inherited property version ip from p, and then sets the base property as B(ip).

7. Implementation considerations

In this section, we discuss several issues that should be considered when implementing the RiBS model. In addition,

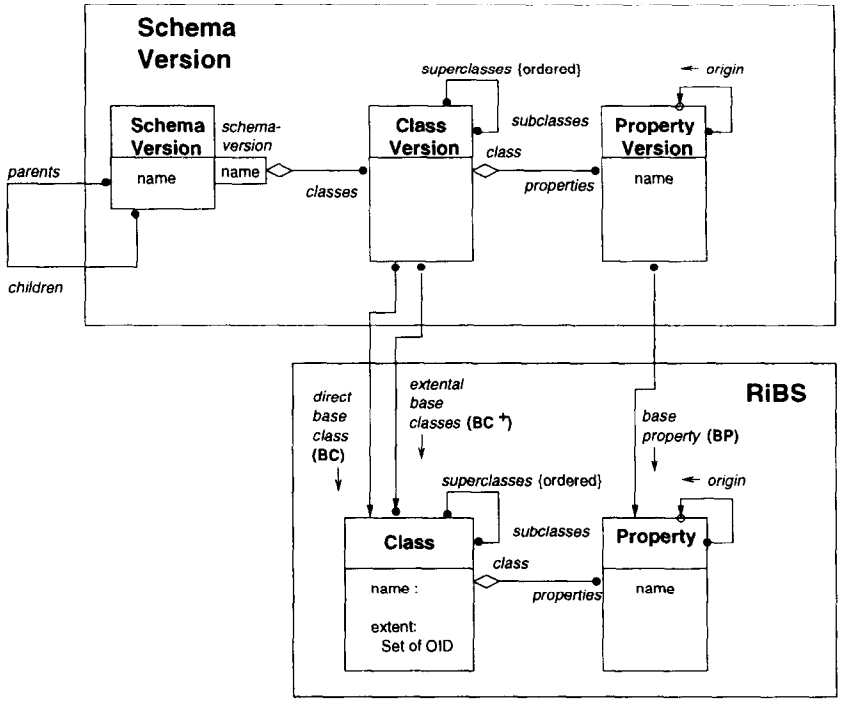

Fig. 4. Systems classes for the RiBS model.

we show that the RiBS model could be supported by current OODBMSs with some extensions, and argue that the perfomance overhead to support the RiBS model is small.

7.1. Data structures

Fig. 4 shows a generic data structure for the implementation of the RiBS model, using the OMT (object modeling technique) notation [30]. The data structures consist of five system classes and their relationships to each other. These classes and their relationships implement the structural components of the RiBS model. The various modeling constructs of the OMT object model, such as 'qualified association', 'aggregation', and 'ordering',[6] are used to describe the data structures concisely and precisely.

In current OODBMSS, a module called SM (scheme manager), maintains the schema information corresponding to system classes class and property [3]. For the implementation of the RiBS model, this SM module needs some extensions to incorporate the system classes for the schema version layer, SchemaVersion, ClassVersion, and PropertyVersion.

7.2. Preprocessing

In the RiBS model, a program or query is written against a schema version, and translated so as to run against RiBS for its execution. This translation can be handled by an ODL/OML (object definition language/object manipulation language) preprocessor [26], as suggested by ODMG. During the translation, the preprocessor might need to interact with the SM module to get information about the schema mapping between RiBS and current schema version. The final program or query against RiBS can be executed without extra run-time overhead.

7.3. Object adaptation

In general, there have been two approaches to the adaptation of objects [1–3], changing the representation of affected objects to a state consistent with the new schema. The first approach is deferred update, where the format of each object is changed only when it is accessed after schema updates. The second approach is immediate update, in which attected objects are updated instantly upon schema updates.

This paper is mainly concerned with the semantics of schema version evolutions for both schema versions and RiBS; thus, object adaptation is not within the scope of this paper. However, either approach can be applied to bring physical objects residing in an extensional base up to a consistent state in the RiBS model.

7.4. Object identity

Two OID schemes, physical OID and logical OID, have been commonly adopted by OODBMSS. A physical OID

[6] We assume that readers are familiar with OMT notation. Refer to [29] for more detailed descriptions regarding the OMT.

encodes the permanent address of the object referred to by itself. This approach provides efficient access to disk-resident objects, but lacks location interdependence. In contrast, a logical OID is generated by the object storage system independently of the physical address of an object. Thus, this representation allows flexible object movement and replication, but with some performance degradation due to the mapping overhead between logical OIDs and their physical addresses. As mentioned above, because a program or query in the RiBS model runs on the RiBS layer after translation, the RiBS model can be supported by any OODBMS, regardless of the OID scheme used.

7.5. Space optimization

With the RiBS model, there might be opportunities for space optimization. For example, consider a base property for which no corresponding property version exists in any schema version. Physical objects in the extensional base reserve space for the obsolete base property, but the space is no longer necessary because the information kept there is not accessible through any schema version. This fact can be exploited by the database administrator by dropping unnecessary base properties from RiBS periodically.

7.6. Implementations using SOP ODMG-compliant OODBMS

SOP (SNU OODBMS Platform) is an ODMG-compliant OODBMS developed from scratch at Seoul National University [31]. SOP consists of several modules, including an object storage system (Soprano) [32], an SM module, an ODMG ODL/OML C_{++} preprocessor, and a cost-based query processor. Soprano supports a physical OID scheme. The SM module maintains the class and property information and supports basic schema evolution primitives from Orion. The current ODMG ODL/OML C_{++} preprocessor was developed to provide a seamless integration of C_{++} programming with SOP by enabling the persistence to be orthogonal to the type. We plan to implement the RiBS model on SOP by extending the SM module, the pre-processor, and the query processor to understand the schema version layer.

8. Schema-version-merging

As noted in Section 6, 'schema-version-merging' is the process of merging two or more existing schema versions. This operation seems to be very useful in several phases of managing OODBS. First, let us consider the initial phase of OODB schema design. Usually at this time, a group of schema designers is involved, and each person is assigned to a different part of the database. The partial schema of each designer is then merged into one global schema. Second, a user might want to customize his/her own schema version from two or more existing ones. In this section, after considering two types of conflicts during schema-version-merging in the RiBS model, we give solutions to overcome those conflicts.

8.1. Conflicts in schema-version-merging

Merging two or more parents with different schema information into a new schema version may introduce two kinds of name conflict:

8.1.1. Homonym problems

Two or more class (property) versions, from different schema (class) versions, but having the same name, may have different direct base classes (properties). We call them homonym class (property) versions.

8.1.2. Synonym problems

Two or more class (property) versions, from different schema (class) versions and having different names, may have the same direct base class (property). We call them synonym class (property) versions. Schema updates such as class renaming and creation cause these name conflicts. Fig. 5 shows examples of synonym and homonym class versions. Assume that (1) schema version SV_j is derived from SV_i, (2) a class version in SV_j which has student in RiBS as its direct base class, is dropped out, and (3) class under was renamed student. Thus, class version Person from SV_i and class version Univ_Person from SV_j are synonym class versions. Other synonym class versions are class version under from SV_i and class version student from SV_j. Class versions students from both schema versions are examples of homonym class versions.

Besides name conflicts, schema-version-merging in the RiBS model introduces another type of conflict, extent migration conflict. We say, that for any two schema versions being merged, a base class in RiBS is said to have extent migration conflict (1) if it has no corresponding class

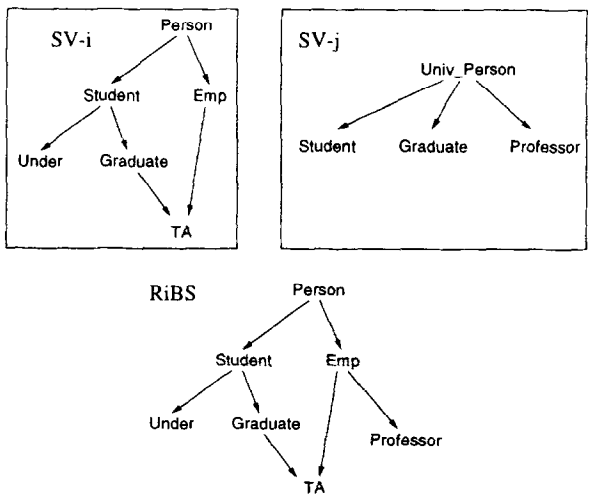

Fig. 5. Schema-version-merging.

version in either schema version, and instead (2) it is included in extental base classes of two class versions from those schema versions, which have different direct base classes. For example, base class C in Fig. 6 has extent migration conflict while merging SV_i and SV_j. Base class C has no corresponding class version in either SV_i or SV_j. C is contained in $B^+(CV_m)$ in SV_i, while $B^+(CV_n)$ is in SV_j. However, CV_m and CV_n have different direct base classes, A and B, respectively. In the new schema version, each corresponding class version is created for CV_m and CV_n. At this point, it is impossible to automatically decide which class version should take base class C as its extental base class. We refer to this situation as 'extent migration conflict'. This is mainly due to multiple inheritance in the object-oriented data model.

In the next subsection, we provide a semi-automatic algorithm considering all these issues. The algorithm is semi-automatic in the sense that it requires some user involvement, as in Ref. [29].

1. For synonym classes, the user chooses the most meaningful name among the names of each synonym class, or assigns a new name in the new schema version.
2. For homonym classes, the user assigns a new name for each new class version.
3. For each base class that has extent migration conflict, the user should designate a logical migration class in the new schema version.

8.2. Schema-version-merging algorithm

The algorithms given in Appendix A describe a way to generate a new schema version (output) from parents (input). Algorithm 1 lists the five main steps of the schema-version-merging process. Algorithm 2 to algorithm 6 correspond to those steps, respectively. The algorithms are quite complex, and thus we give a detailed explanation with an illustrative example in the next subsection.

The first step, Identify-BCs, identifies the base classes necessary in the new schema version. If a base class in RiBS is used as the direct base class of a class version in any input schema version, it is included in the base class list, BCList, of the new schema version. For each base class BC_i in BCList, we maintain a class version list $CVList_i$, each element of which has BC_i as its direct base class and comes from a different input schema version.

The next step, Create-CVs, creates a class version object CV_{new_i} for each base class BC_i in BCList. After creating CV_{new_i}, this step sets $B(CV_{new_i})$ to BC_i. If the names of all class versions in $CVList_i$ are equal (that is, no synonym problems), then a new class version inherits its name from these class versions. Otherwise, users are requested to resolve synonyms.

After creating all class versions of the new schema version, the algorithm calculates extental base classes of each new class version, via Calculate-B^+. In the first loop, this procedure derives initial extental base classes B^+ for each new class version CV_{new_i} in SV_{new}. For each CV_{new_i} all the B^+s of class versions in $CVList_i$ are intersected into $B^+(CV_{new_i})$. At the end of the first loop, some base classes may not be either in BCList or any $B^+(CV_{new_i})$; that is, these base classes have 'extental migration conflicts'. For each of these base classes, users are requested to resolve the conflicts in the second loop of Calculate-B^+; that is, the user should designate a logical migration class version in SV_{new}.

The next step, Calculate-LPs, creates local properties of each new class version in SV_{new}. For a class version CV_{new_i}, this procedure creates a local property version object PV_{new_m} for each base property BLP_m in $B(CV_{new_i})$ which is used as a base property of a local property version of any class version in $CVList_i$. BLP_m is then set to $B(PV_{new_m})$. After completing BLPList, this algorithm names each PV_{new_m}. As in the case of synonym class versions, users need to be involved in resolving synonym property versions in $LPVList_m$, if any.

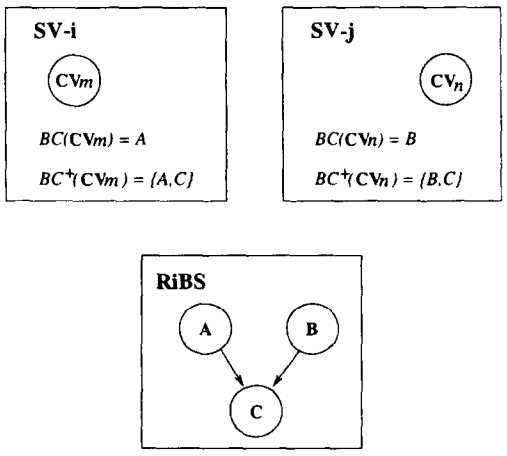

Fig. 6. Extental migration conflict.

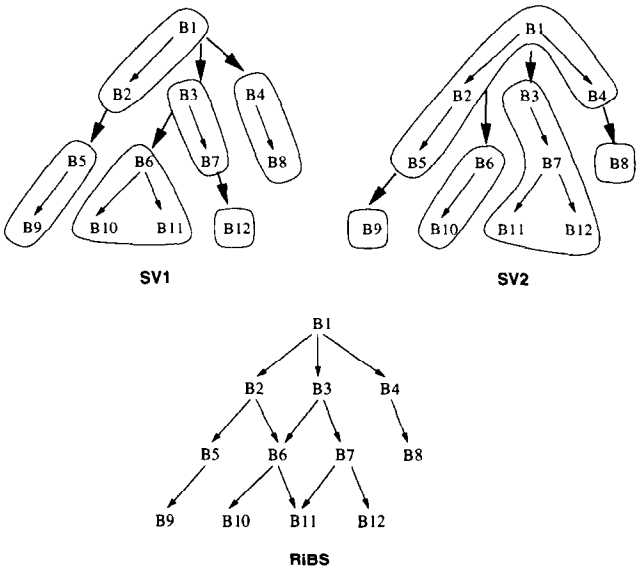

Fig. 7. Schema-version-merging example (1).

The final step, Make-Class-Hierarchy, makes DAG (direct acyclic graph) relationshps for SV_{new}. After blindly deriving all direct ISA relationships between new class versions from relationships of parent schema versions, this procedure removes redundant ISA relationships.

8.3. An illustrative example

Fig. 7 shows a RiBS and two schema versions SV1 and SV2 to be used in exemplifying the schema-version-merging. In the figure, we assume that a class version CV in either schema version is represented by an area surrounded by a solid line. In each area, the root of the corresponding subgraph is its direct base class, B(CV). A bold arrow between class versions represents an ISA relationship in the schema version. The base classes contained in each area comprise the extental base classes of CV, $B^+(CV)$. For simplicity, we do not specify the name of each class version explicitly.

Fig. 8 illustrates how our schema-version-merging algorithm works on the above two schema versions. Fig. 8 (a) shows eight base classes resulting from Identify-BCs. Fig. 8 (b) represents the status at the time of completion of the first loop of Calculate-B^+, with eight new class versions. A temporary set of extental base classes is attached to the name of each new class version. Note that base class B11 is not included in any extental base class of new class versions, an example of 'extental migration conflict'. We assume that the user decides to include B11 into $B^+(CV6)$, which thus results in {B6, B10, B11}. Fig. 8 (c) shows the status after the first loop of make-class-hierarchy. Basically, the ISA relationships between new class versions inherit from the ISA relationships of parent schema versions. For instance, see ISA relationships between CV1 and CV5 or between CV3 and CV12. However, it should be noted that in the case of the ISA between CV5 and CV9, there is no corresponding ISA relationship in any parent schema version instead, in SV2, the class version containing B9 has, as its superclass, the class version corresponding to CV1. The second-level if statement inside the first loop of make-class-hierarchy is concerned with this; when new ISA relationships are built up. the relationship of the extental base classes takes precedence over the explicit ISA relationship in the parent schema versions. For example, ISA (CV9, CV5) is derived from the fact that B(CV9), that is B9, is contained in the B^+ of a class version in SV1 rather than from the ISA relationship between the two class versions containing B1 and B9, respectively, in SV2. The second loop of make-class-hierarchy removes the redundant ISA between CV1 and CV6, thus, Fig. 8(d) shows the final new schema version.

9. Related work

In the field of OODBS, there have been several research activities closely related to the RiBS model, including papers on views, schema versions, and schema evolutions. In addition, our work on schema-version-merging shares

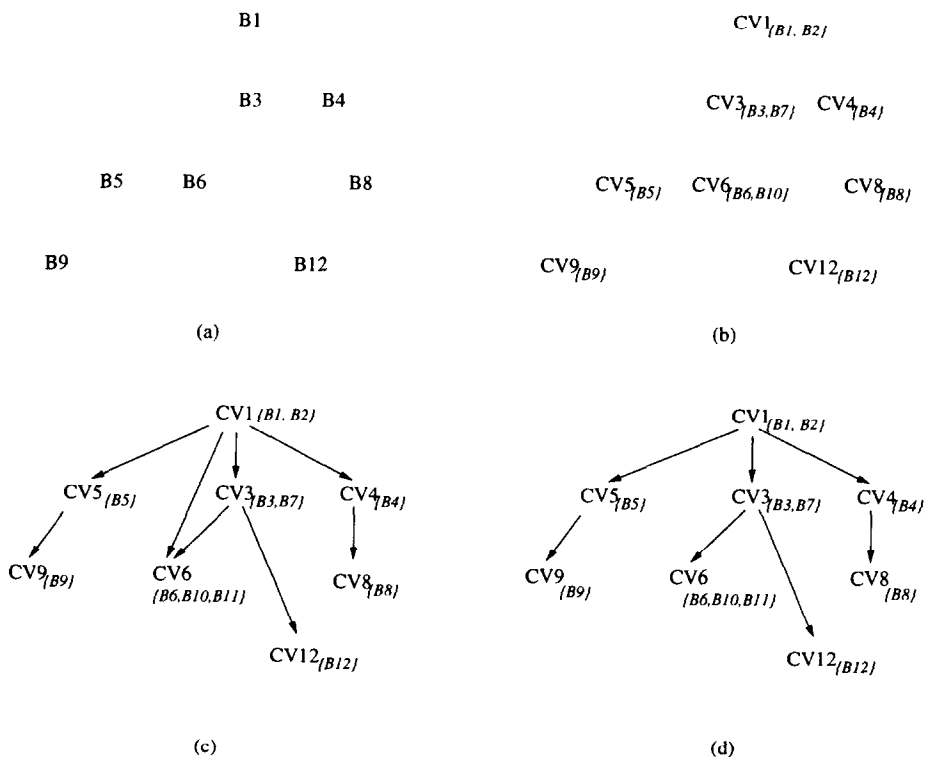

Fig. 8. Schema-version-merging example (2).

some concerns with methodologies for database schema integration. In this section, we summarize these articles and outline their differences from the RiBS model.

9.1. Views and the RiBS model

There have been several attempts to support views in OODB [23,24,27]. In [23], in the context of the O_2 data model, a view mechanism which allows the restructuring of the class hierarchy and supports virtual classes is described with a number of examples. In Ref. [27], the authors proposed a MultiView methodology, where a view schema from a global schema can be defined according to need. [24] presents a view semantic within an object/relational DBMS, UniSQL, by augmenting semantics of relational views with object-oriented concepts such as inheritance, method and OID. In addition, they extend the use of views to dynamic windows for schema, with which schema evolution in OODB can be simulated without affecting the database. This is along the same line as the approach in Ref. [18] simulating schema evolution using views.

Our RiBS approach is similar to these articles in the sense that each schema version is defined over one global base schema RiBS. However, there is a big difference between the RiBS model and the work on views in OODB. While direct schema updates against a schema version are allowed in the RiBS model, in earlier works a view schema can be changed only by redefining a new view from scratch after deleting the old one. Furthermore, capacity-augmenting schema updates cannot be simulated by earlier view approaches [20].

9.2. Schema versions/evolutions and the RiBS model

The work in Ref. [7] is the first substantial research on schema versions in OODB, based on the object version model of ORION [1]. In this work, the schema version model is expressed as several rules about schema version management and access scope. According to the access scope rules, each schema version has a different set of objects visible to it, that is, the access scope of the version. An instance object may thus not be shared among schema versions. In contrast to the RiBS model, a new schema version can be derived from only one parent nylon and thus the schema version derivation hierarchy results in a tree.

Another approach to schema versions is found in Ref. [20]. This work is most similar to ours in that it also supports schema evolution through views, sharing of instance objects among all the schema versions and schema merging. However, the consider such issues as phantom references and conflicts in schema merging, including homonyms/synonyms and extental migration conflicts. In addition, their automatic classification algorithm introduces a new class in the global schema for every capacity-augmenting schema update, which makes the global schema complicated.

As an alternative to schema versions, there has been the class versioning approach [19,33], where the units of versioning are individual classes, instead of the entire class hierarchy. [19] proposes a class versioning system CLOSQL, based on dynamic instance conversion, which enables an instance object to be seen from the outside by a number of class version interfaces, and determines the type of an instance object by the context of concern (that is, dynamic instance objects). In this respect, we can argue that in the RiBS model a physical object residing in extensional bases is also a dynamic object since it changes its type dynamically depending on the current schema version (CSV) accessing the object. However, with class versioning approach, the burden to construct consistent 'virtual' schema versions from various class versions is left to users [7].

During the past decade there has been much research on the subject of schema evolutions in OODB [1–3]. These articles consider two important issues in schema evolution: semantics of schema change operations and adaptation of objects. The second issue was touched upon in Section 7. A basic solution to the first problem is to define a set of invariants that should be satisfied by the schema, and then to define rules and/or procedures for each schema change operation to guarantee the invariants. In this respect, the RiBS model can be taken as another extension of this framework toward support of schema version functionality, with substantial add-ons. First, we identify several new invariants for schema versions and RiBS, in addition to traditional invariants for schema evolution. Second, we extend the semantics of primitive schema change operations to guarantee all these invariants.

9.3. Database schema integration and schema-version-merging

In the database literature, many methodologies for integrating database schema are found in the form of view integration, database schema integration, or multi-database. At the heart of those methodologies is the detection of conflicts and their resolution. Our work on schema-version-merging shares many concerns with these methodologies.

In Ref. [34], a unifying framework for the problem of view and database schema integration is provided, and several earlier papers are reviewed and compared. The process of integration is divided into four steps: pre-integration, conflict detection, conflict resolution and merging/restructuring. With regard to conflict detection, the authors distlinguish two types of conflict: name conflicts and structural conflicts. Name conflicts are further classified into homonyms and synonyms, as in the RiBS model. However, extental migration conflict in RiBS has no corresponding conflict in their taxonomy, although we classify it as a structural conflict in this paper. It is a unique phenomenon in our

RiBS model. As for conflict resolutions [34] states that automatic resolution is generally not feasible. Our schema-version-merging algorithm also leaves the burden to users. In the final phase of merging/restructuring, several criteria are tested to achieve a desirable global schema. Among the criteria, most methodologies are geared toward minimality, and in particular a removal of redundancy. Our schema-version-merging algorithm also includes a step for removing redundant ISAS. A similar framework for classifying schema and data conflicts in federating multi-database systems can be found in Refs. [29,35].

However, there is one important difference between these articles and our framework for schema-version-merging. Schema versions being merged within the RiBS model share some semantic knowledge (for example, the direct base class for each class version), whereas, for general database schema integration problems, we cannot expect these kinds of knowledge. This semantic knowledge enables the integration of schema versions with less intervention from the user.

10. Conclusion

We strongly believe that the functionality of the schema version will be a pre-requisite for OODBMSs to be widely accepted by newly emerging database applications, including repositories and the WWW. In this paper, we proposed a schema version model for OODBs based on the concept of RiBS. Each schema version is in the form of a class hierarchy view over one global schema, RiBS. Users are supposed to be concerned only with schema versions. Direct schema updates on schema versions are allowed, which are, if necessary, automatically propagated to RiBS. To avoid anomalies such as phantom reference and multiple classification, we introduced several invariants. In addition, we gave the taxonomy of schema update operations over schema versions and defined their semantics. Finally, we identified several types of conflicts during schema-version-merging in the RiBS model, and devised an algorithm for schema-version-merging.

We plan two future projects. With the current RiBS model, customization of the class hierarchy is somewhat restricted. Hence, we intend to incorporate more operations into our schema update taxonomy, such as class partitioning, class merging, and dynamic class [6,23,36,37], for increased flexibility. We expect that this will substantially enhance the modeling capability of the RiBS model. Next,we plan to extond all three elemets of our model, that is, structures, invariants and operations, to support the reorganization of nested complex objects. After identifying a set of basic operations useful to restructure complex objects, we will augment the mapping data structure between RiBS and schema versions in order to model the complex object view, and will define the semantics of those operations, as well as new invariants. For this, we are now considering some role defining operations from ORM [36] and some view definition operations from Chimera ([38]). This extension enables OODBMSs to effectively model dynamic views over complex WWW structures, which are very useful in such applications as health-care systems and CASE [16].

11. Further reading

For further reading see Refs. 39–47.

Acknowledgements

The authors thank all the OOPSLA members involved in developing the SOP OODBMS. We also wish to thank the referees for their valuable comments and suggestions, especially for referee A, which led to a considerable improvement of the presentation.

Appendix A A schema version merging algorithm

Appendix A.1 Algorithm 1 schema-version-merging algorithm

Input: RiBS, schema versions SV_1, SV_2, $\cdots$, SV_n to be merged
Output: newly merged schema version SV_{new}
Data Structure:
BCList: the set of pairs (BC_i, $CVList_i$), where $CVList_i$ is a set of class versions having BC_i as their direct base class;
$ExtBC_{new}$: a temporary set of all extental base classes of SV_{new};
initialize BCList = empty; initialize $ExtBC_{new}$ = empty;
Identify-BCs(); / * identify base classes * /
Create-CVs(); / * create a new class version for each base class * /
Calculate-B$^+$(); / * calculate B + for new class versions * /
Calculate-LPs(); / * identify local properties for each class version * /
Make-Class-Hierarchy(); / * make a class hierarchy for SVnew * /

Appendix A.2 Algorithm 2 Identify-BCs()

for each SV_i **do**

for each class C_j in SV_i **do**

if ($B(C_j) \notin$ BCList **then**

add [$B(C_j)$, $\{C_j\}$] to BCList;

else

add C_j to CV $List_k$ where $BC_k = B(C_j)$;

end if

end for

end for

Appendix A.3 Algorithm 3 Create-CVs()

for each BC_i in BCList **do**

create class version CV_{new_i} in SV_{new};
set $B(CV_{new_i})$ to BC_i;
if name conflict among CVs in $CVList_i$ **then**

let the user resolve the synonym; / * resolve synonym * /

else

inherit CV_{new_i}.name from CVs in $CVList_i$;

end if
if name conflict between CV_{new_i} and other CV_{new_j} **then**

let the user resolve the homonym; / * resolve homonym * /

end if

end for

Appendix A.4 Algorithm 4 Calculate-B^+()

/ * initialize B^+ for each CV_{new_i} * /
for each CV_{new_i} in SV_{new} **do**

$B^+(CV_{new_i}) = \bigcap_{C_j \epsilon CVList_i} B^+(C_j)$;
add $B^+(CV_{new_i})$ to $ExtBC_{new}$;

end for
/ * process extental migration conflicts * /
for each $BC_k \epsilon$ RiBS - $ExtBC_{new}$ **do**

let the user select a logical migration class version CV_{new_m};
add BC_k to $B^+(CV_{new_m})$;

end for

Appendix A.5 Algorithm 5 Calculate-LPs()

BLPList: the set of pairs (BLP_i, $LPVList_i$), where $LPVList_i$ is a set of property versions, the direct base property of which is BLP_i
for each CV_{new_i} in SV_{new} **do**

set BLPList to empty:
for each $CV_j \epsilon CVList_i$ **do**

for each LP_k in LocalProps(CV_j) **do**

if $B(LP_k) \notin$ BLPList **then**

add [$B(LP_k)$,{LP_k}] to BLPList;

else

add LP_k to $BLPList_l$, where $BLP_l = B(LP_k)$;

end if

end for

end for
for each $BLP_m \epsilon$ BLPList **do**

create property version PV_{new_m};
set BLP_m to $B(PV_{new_m})$;
if name conflicts among LPs in $LPVList_m$ **then**

let the user resolve the synonym;

else

inherit PV_{new_m}.name from LPs in $LPVList_m$;

end if
add PV_{new_m} to LocalProps(CV_{new_i});

end for

end for

Appendix A.6 Algorithm 6 make-class-hierarchy()

/ * derive ISA relationships blindly * /
for each CV_{new_i} in SV_{new} **do**

for each CV_j in $CVList_i$ **do**

for each CV_k in Parents(CV_j) **do**

if $CV_{new_k} \epsilon$ Parents(CV_{new_i}) **then**

do nothing;

else

if $\exists$ CV_{new_l} where $B(CV_{new_l}) \epsilon B^+(CV_k)$ **then**

for each CV_{new_l} **do**

for each $CV_m \epsilon$ CV $List_l$ **do**

if $B(CV_i) \epsilon B^+(CV_m)$ **then**
add CV_{new_l} to Parents(CV_{new_i});
end if

end for

end for

else

add CV_{new_k} to Parents(CV_{new_i})

end if

end if

end for

end for

end for
/ * remove redundant ISA * /
for each CV_{new_i} in SV_{new} **do**

for each CV_{par} in Parents(CV_{new_i}) **do**

if $Cv_{par} \epsilon$ Parents * (CV), ($\exists$ CV ϵ Parents(CV_{new_i}) - {CV_{par}})
then

U delete CV_{par} from Parents(CV_{new_i});

end if

end for

end for

References

[1] J. Banerjee, Won Kim, Hyoung-Joo Kim, Hank Korth, Semantics and implementation of schema evolution in object-oriented databases, Proc. ACM SIGMOD, May 1987, pp. 311–322.

[2] D. Jason Penney, Jacon Stein, Class modifications in the GemStone objectoriented DBMS, Proc. OOPSLA, Oct. 1987, pp. 111–117.

[3] Roberto Zicari, Fabrizio Ferrandina, Schema and database evolution in object database systems, Advanced Database Systems, part 6, Morgan Kaufmann, 1997, pp. 412–495.

[4] Object Design, Inc., ObjectStore technical overview, Release 3.0, Object Design Inc., 1994.

[5] Objectivity, Inc., Schema evolution in Objectivity/DB, White paper available from http://www.objy.com/ObjectDatabase/WP/Schema/schema. html.

[6] Won Kim, Introduction to Object Oriented Databases, MIT press, Cambridge, MA, 1991.

[7] Won Kim, H.T. Chou, Versions of schema for object-oriented databases, Proc. VLDB, Sep. 1988, pp. 148–159.

[8] Sven-Eric Lautemann, An introduction to schema versionning in OODBMS, DEXA Workshdexa, Sep. 1996, pp. 132–139.

[9] P.A. Bernstein, Repositories and object oriented databases, Proceedings of BTW '97, 1997.

[10] P.A. Bernstein, B. Harry, P. Sanders, The Microsoft repository, Proc. VLDB, Aug. 1997, pp. 3–12.

[11] A. Silberschartz, M. Stonebraker, J. Ullman, Database research: achievements and opportunities into the 21st century, Report of an NSF Workshop on the Future of Database Systems Research, 1995.

[12] F. Charoy, An object-oriented layer on PCTE, Technical paper available from http://gille.loria.fr:7000/oopcte/oopcte.html, 1994.

[13] Mary E.S. Loomis, Object database-integrator for PCTE, Journal of Object Oriented Programming 5(2) May 1992, pp. 53–57.

[14] L. Wakeman, J. Jowett, PCTE: the standard for open repositories, Prentice-Hall, New York, 1993.

[15] T. Atwood, Object databases come of age, Object Magazine, July 1996.

[16] J. Jingshuang Yang, G.E. Kaiser, An architecture for integrating OODBs with WWW, Columbia University Tech-Report CUCS-004-96, 1996.

[17] A. Bapat, J. Waesch, K. Aberer, J.M. Haake, HyperStorM: an extensible object-oriented hypremedia engine, The Seventh ACM Conference on Hypertext, 1996.

[18] E. Bertino, A view mechanism for object-oriented databases, Proceedings of the Third International Conference on Extending Database Technology, 1992, pp. 136–151.

[19] S. Monk, I. Sommerville, Schema evolution in OODB using class versioning, SIGMOD Records 22 (3) (1993) 16–22.

[20] Y.G. Ra, E.A. Rundensteiner, A transparent object-oriented schema change approach using view evolution, Proceedings of the International Conference on Data Engineering, 1995.

[21] R. Hull, Relative information capacity of simple relational database schemata, Proceedings of the ACM PODS, Apr. 1984, pp. 97–109.

[22] R.J. Miller, Y.E. Ioannidis, R. Ramakrishnam, The use of information capacity in schema integration and translation, Proceedings of the VLDB, Aug. 1993, pp. 120-133.

[23] S. Abiteboul, A. Bonner, Objects and views, Proceedings of the ACM SIGMOD, May 1991, pp. 238–247.

[24] Won Kim, Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, 1995.

[25] A. Taivalsaari, On the notion of inheritance, ACM Computing Survey 28 (3) (1996) 438–479.

[26] R.G.G. Cattell, The Object Database Standard: ODMG-93, Morgan Kaufmann, 1996.

[27] E.A. Rundensteiner, MultiView: a methodology for supporting multiple views in objectoriented databases, Proceedings fo the VLDB, Aug. 1992, pp. 187–198.

[28] Sang-Won Lee, Hyoung-Joo Kim, A model of schema versions for objectoriented databases, based on the concept of rich base schema, Technical Report, SNU OOPSLA Laboratory, 1997.

[29] W. Kim, J. Seo, Classifying schematic and data heterogeneity in multidatabase systems, IEEE Computer 24 (12) (1991) 12–18.

[30] James Rumbaugh, OMT: the object model, Journal of Object Oriented Programming 7(9) Jan. 1995, pp. 21–27.

[31] J.H. Ahn, K.W. Lee, H.J. Song, H.J. Kim, Soprano: design and implementation of an object storage system, Journal of Korea Information Science Society(C) 2 (3) (1997) 10.

[32] Jung-Ho Ahn, Hyoung-Joo Kim, Seof: An adaptable object prefetch policy fro object-oriented database systems, Proceedings of the International Conference on Data Engineering, April 1997, pp. 4–13.

[33] Hyoung-Joo Kim, Issues in object oriented database schema, Ph.D. dissertation at University of Texas, Austin, TX, 1988.

[34] C. Batini, M. Lenzerini, S.B. Navathe, A comparative analysis of methodologies for database schema integration, ACM Computing Survey 18 (4) (1986) 323–364.

[35] E. Pitoura, O. Bukhres, A. Elmagramid, Object orientation in multidatabase systems, ACM Computing Survey 27 (2) (1995) 141–195.

[36] M.P. Papazoglou, B.J. Krämer, A database model for object dynamics, VLDB Journal 6 (2) (1997) 73–96.

[37] R. Wieringa, W. de Jonge, P. Spruit, Using dynamic classes and role classes to model object migration, Theory and Practice of Object Systems 1 (1) (1995) 61–83.

[38] G. Guerrini, E. Bertino, Barbara Catania, Jesus Garcia-Molina, A formal model of views for object-oriented database systems, Theory and Practice of Object Systems 3 (3) (1995) 157–183.

[39] F. Bancilhon, Connecting an object database system to the outside world, Talks in Stanford Weekly Database Seminar, 1996.

[40] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, J. Madec, Schema and database evolution in the O_2 object database system, Proceedings of the VLDB, Sep. 1995, pp. 170–181.

[41] D. Konopnicki, O. Shmueli, W3QS: a query system for the World-Wide Web, Proceedings of the VLDB, Sep. 1995, pp. 66–77.

[42] Sven-Eric Lautemann, A propagation mechanism for populated schema versions, Proceedings of the International Conference on Data Engineering, Apr. 1997, pp. 67–78.

[43] B. Staudt Lerner, A. Nico Habermann, Beyond schema evolution to database reorganization, Proceedings of the OOPSLA, Oct. 1990, pp. 67–76.

[44] R.J. Peters, M.T. Özsu, An axiomatic model of dynamic schema evolution in objectbase systems, ACM TODS 22 (1) (1997) 75–114.

[45] B. Stroustrup, The C++ Programming Language, 2nd edn, Addison-Wesley, New York, 1991.

[46] Katsumi Tanaka, Masatoshi Yoshikawa, Kozo Ishihara, Schema virtualization in object-oriented databases, Proceedings of the International Conference on Data Engineering, Feb. 1988, pp. 23–30.

[47] G. Wiederhold, Views, objects, and databases, IEEE Computers 19 (12) (1986) 37–44.