Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Journal of Systems Architecture 52 (2006) 563–577
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

A schema version model for complex objects in object-oriented databases ☆

Sang-Won Lee [a,*], Jung-Ho Ahn [b], Hyoung-Joo Kim [c]

[a] *School of Information and Communication Engineering, Sungkyunkwan University, 300 Cheoncheon-dong, Jangan-gu, Suwon, Gyeonggi-do 440-746, Republic of Korea*
[b] *IT4Web, YungJeon Building, Samsung-dong 154-10, Kangnam-gu, Seoul, Republic of Korea*
[c] *Computer Science and Engineering, Seoul National University, San 56-1, Shilim-dong, Gwanak-gu, Seoul, Republic of Korea*

Received 15 June 2004; received in revised form 23 March 2006; accepted 6 April 2006
Available online 22 June 2006

Abstract

In this paper, we propose a schema version model which allows to restructure complex object hierarchy in object-oriented databases. This model extends a schema version model, called *RiBS*, which is based on the concept of *Rich Base Schema*. In the *RiBS* model, each schema version is in the form of updatable class hierarchy view over one base schema, called the *RiBS* layer, which has richer schema information than any existing schema version in the database.

In this paper, we introduce new operations for restructuring composite object hierarchy in schema versions, and explain their semantics. We also touch upon the ways to transform queries posed against a restructured composite object hierarchy into one against the base schema. In addition, we identify several types of conflicts during schema version merging which result from the restructuring operations, and provide a semi-automatic algorithm to resolve the conflicts. The originality of this paper lies in that (1) we introduce several new operations to restructure composite object hierarchy, and (2) this extended *RiBS* model operations raise the concept of data independence in OODBs upto the schema level.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Complex objects; Schema versions; Object-oriented databases; Schema version merging

1. Introduction

Recently, many new database applications such as CAD, CASE and WWW have emerged. One common requirement from these applications is to model complex objects and the relational databases are not adequate for them. In order to support these applications, many object-oriented data models have been developed since the mid 1980s, and the advantages of object-oriented database systems

☆ This research was supported by the Ministry of Information and Communication, Korea under the Information Technology Research Center support program supervised by the Institute of Information Technology Assessment, IITA-2005-(C1090-0501-0019).
* Corresponding author. Tel.: +82 31 290 7988; fax: +82 31 290 7230.
E-mail addresses: swlee@skku.edu (S.-W. Lee), jhahn@oopsla.snu.ac.kr (J.-H. Ahn), hjk@oopsla.snu.ac.kr (H.-J. Kim).

1383-7621/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.04.001

(OODBMSs) have been widely recognized in terms of rich modeling power and performance.

Differently from the traditional RDBMS applications, new OODB applications require the functionality of powerful schema management such as dynamic schema changes and more flexible schema management. Therefore, there has been so much work on schema evolutions in OODBMSs [5,31], and many commercial OODBMSs (e.g. O2 [31], ObjectStore [23], Objectivity [21], and Versant [29]) support basic schema evolution operations.

With 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 valid. This single schema modification mechanism has several drawbacks [13]. First, schema updates may invalidate programs written against old schema. Second, because all the users share a single schema, a schema update by one user may change the views of all other users. These shortcomings can be summarized as the lack of logical data independence in OODBMSs. In order to overcome these drawbacks, schema version mechanism has been introduced [15,16].

Recently, the functionality of schema versions has been re-motivated in the emerging OODBMS applications such as Repositories [7], Portable Common Tool Environment (PCTE) [10], and WWW [3]. Atwood points out that many web sites publish new versions of their applications with new database schema versions without changing the existing versions of the applications and their schema [3]. Data repositories, which are expected to be a killer application for DBMS technology, also should be able to change the structure of information and its metadata without breaking existing applications [7,27]. Another strong requirements for schema versions come from PCTE where the role of OODBMSs is to manage PCTE schema, to support its evolution over time, and to manage the resulting schema versions, as Loomis says [19], Thus, in order for OODBMSs to be adopted in the new applications, the functionality of schema versions is essential. In fact, a commercial OODBMS, POET, supports basic schema version mechanism [22].

There have been several approaches to schema version mechanisms in object-oriented databases (OODBs) [15,20,24], but they have not reached a satisfactory status yet. In particular, these works have mainly focused on schema versions over class inheritance hierarchy. To our best knowledge, there has been no works on schema versions for complex object hierarchy. In fact, many OODBMS applications requires schema versions over complex object hierarchy, rather than over class hierarchy.

In [17], we developed a simple-yet-powerful schema version model, called *RiBS*, which is based mainly on two major concepts of *rich base schema* and *updatable class hierarchy view*. We call this schema version model as the basic *RiBS* model, because it supports schema versions only for class hierarchy. In this paper, we propose an extended schema version model, called the extended *RiBS* model, which can restructure the complex object hierarchy. The main contribution of this paper is to propose a set of schema evolution operations for complex object hierarchy and to describe the semantics of each operation. As far as we know, there has been no work on this issue. In addition, we deal with the issues of query processing and schema version merging.

The remainder of this paper is organized as follows. Section 2 briefly explains the object model and the basic *RiBS* model. Section 3 proposes new schema evolution operations for restructuring complex object hierarchy, and explain their semantics. Then, Sections 4 and 5 deals with query processing and schema version merging, respectively. And Section 6 compares the related works with our extended *RiBS* model, and Section 7 concludes this paper.

2. Object model and basic *RiBS* model

In this section, we describe the object model assumed in this paper, and review the basic *RiBS* model briefly. We suppose that the readers are somewhat familiar with some important concepts of object-oriented data models, such as class hierarchy and inheritance [13,17].

2.1. Object model

A user-defined class, as well as the built-in data type, can be used as the domain of an attribute of a class, and an instance object of this class may have the object identifier of an object of that class as its value of the attribute. The object-oriented data model, in contrast to the relational data model in which the relationships between entities are implicitly represented using data values, explicitly represents the relationships using object identifiers. These reference relationships between classes make a complex object hierarchy. Even though the term 'complex object' is generally used to represent the

part-of relationship as in [13], it is also used, in this paper, to cover simple references between objects.

Fig. 1 shows an example of complex object hierarchy throughout this paper. The complex object hierarchy has the VEHICLE class as its root class and each of the other classes as its part class. In Fig. 1, for the simplicity, we use the same name for a class and an attribute, which refers the class. A dotted arrow in Fig. 1 represents a reference relationship from a starting class to an end class. Each attribute in a class can be classified into either a leaf or a non-leaf attribute depending on the type of its domain: that is, an attribute whose domain is a built-in class, such as integer and string, is called a leaf attribute, while an attribute whose domain is a user defined class is a non-leaf attribute.

In relation to complex object hierarchy, one important concept is the path expression [9,30], which is used to represent a path of attributes along complex object hierarchy. In particular, a path expression is used to express a query condition succinctly. The following query shows a usage of path expression to select all car objects whose engine power is more than 100 horse powers from the VEHICLE class in Fig. 1:

```
select Car
from VEHICLE Car
where Car.DriveTrain.Engine.Power >=
'100hp'.
```

2.2. Basic *RiBS* model

Fig. 2 shows the structural component of the basic *RiBS* model, and it has a three-layer architec-

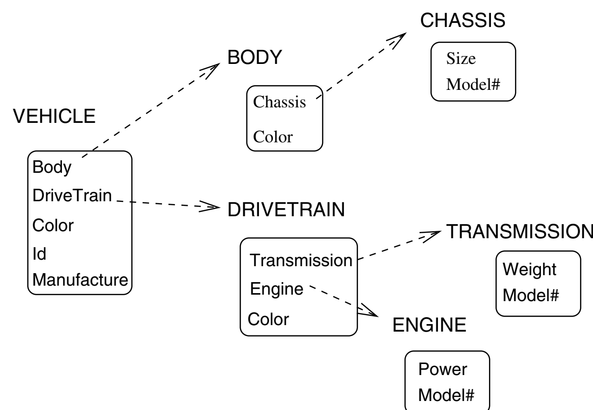

Fig. 1. VEHICLE complex object hierarchy.

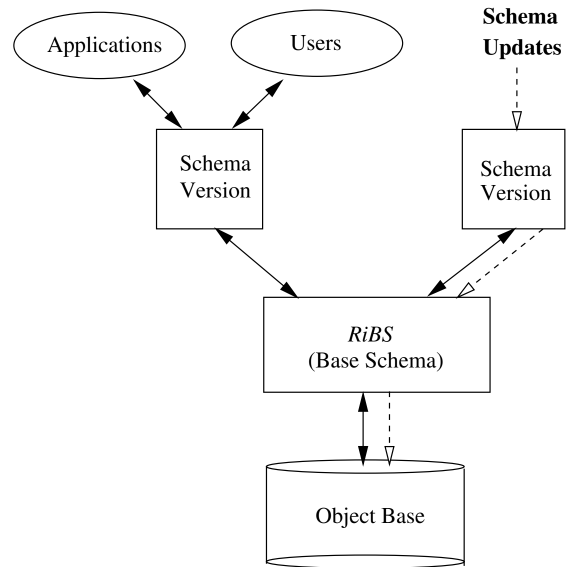

Fig. 2. RiBS model: architecture.

ture: (1) (extensional) object base layer, (2) *RiBS* (Rich Base Schema) layer, and (3) schema version layer [17]. Each application or user is concerned only with schema versions in the schema version layer. In turn, each schema version is in the form of a class hierarchy view over the next *RiBS* layer. The *RiBS* layer accumulates all the necessary schema information ever defined in any schema version, including classes, inheritance, attributes, and relationships. Finally, the object base layer physically stores all the instance objects of the *RiBS* layer.

With the *RiBS* model, user can execute arbitrary schema evolution operations against current schema version. The *RiBS* model supports all the schema change operations in the taxonomy of Orion OODBMS [13]. As shown in Fig. 2 direct schema updates on schema versions are allowed, and their effects are, if necessary, automatically propagated down to the *RiBS* layer and/or to the object base layer. At a certain point of time, either a user or an application can access and manipulate database through a specific schema version, which we call *current schema version* (*CSV*). A schema version is actually a logical view over the *RiBS* layer in the sense that all the objects visible through a schema version are derived from the objects stored in the extensional object base. Thus, an application program or a query over a schema version, before its execution, need to be automatically rewritten into one over the *RiBS* layer.

Meanwhile, it should be noted in Fig. 2 that any schema change within a schema version has no effect on other schema versions in the database. That is, every schema change operation in the basic *RiBS* model has effects only on the schema version itself and the *RiBS* layer.

One important characteristic of the *RiBS* model is to allow direct schema updates over a schema version which is a class hierarchy view over the *RiBS* layer. This allows us to avoid some disadvantages of the existing schema version models, (1) storage overhead due to many replicated copies for an object [15,24], (2) the complexity in schema change process in view approaches [8], and (3) schema management overhead in class version model [20].

In the *RiBS* model, however, the execution of a schema evolution operation does not derive a new schema version implicitly. Instead, the *RiBS* model provides another operation to derive a new schema version from the existing ones: the former is called the "*child schema version*" and the latter "*parent schema version* (*s*)". The *derived-from* relationships between schema versions constitute a Schema Version Derivation Graph (SVDG). This schema derivation operation requires a rather complex process to create a new schema version by merging two or more existing ones, which we call "*schema version merging*". The schema version merging in the basic *RiBS* model might encounter three types of conflicts: (1) homonym, (2) synonym and (3) extent migration conflict. The first two conflicts are a type of name conflict while the last one a kind of structural conflict. The basic *RiBS* model identifies the conflicts during schema version merging and provides a schema version merging algorithm for resolving the conflicts. Please refer to [17] for more details.

The basic *RiBS* model supports the query rewrite: that is, an application program or a query posed against a schema version is automatically translated so as to run against the *RiBS* layer for its execution. In practice, this translation can be handled by an ODL/OML (Object Definition Language/Object Manipulation Language) preprocessor [9], as suggested by ODMG. During the translation, the preprocessor might need to interact with the schema manager 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 runtime overhead.

At this point, some readers might regard our *RiBS* approach as yet another view mechanism for OODBs because you can think that the *RiBS* model has several different logical schema views over one common base schema. In this respect, it is very subtle to differentiate schema versions in the *RiBS* model from other existing view approaches. Even though we will compare our approach with the related works later, here we would like to argue some important differences between them so that the readers can understand the unique aspects of our approach. First, schema versions in the *RiBS* model are updatable while the old views in the existing view approaches should be dropped and redefined for simulating a schema evolution. Next, the existing view approaches do not support the capacity augumenting schema updates which are the basic functionality in the *RiBS* model. Finally, the existing view approaches do not support any type of schema version merging. In summary, our *RiBS* model raises the level of data independence up to the schema level, while the existing view approaches merely support the logical data independence.

In this subsection, we described the basic *RiBS* model which supports schema evolutions against class inheritance hierarchy in a schema version. In this paper, we extend the basic *RiBS* model to support the restructuring of complex object hierarchy, and we call the extended schema version model as the *extended RiBS model*. Hereafter, whenever there is no ambiguity, we will call the extended *RiBS* model simply as *RiBS* model. In other cases, we will explicitly differentiate two models.

3. Restructuring complex object hierarchy

In this section, we give a set of operations for restructuring complex object hierarchy; its taxonomy and a detailed semantics of each operation.

3.1. Operations for restructuring complex object hierarchy

The following operations are introduced for restructuring complex object hierarchy. The first two operations are originally included in the basic *RiBS* model, but we include them in this taxonomy because they can be regarded as to restructure complex object hierarchy. The remaining four operations are newly introduced in the extended *RiBS* model.

1. Add a part class into complex object hierarchy.
2. Remove a part class from complex object hierarchy.
3. Pull an attribute of a part class to root class: **pull**.
4. Unnest a part class on root class: **unnest**.
5. Nest part attributes into a new virtual part class: **nest**.
6. Move a part attribute to a part class: **move**.

Please note that only the first operation will affect the *RiBS* layer, while others have no effect on the *RiBS* layer. The last five operations are intended to customize complex object hierarchy, and thus we need to simply change the mapping information between *RiBS* layer and current schema version.

Before proceeding, we need to mention the cardinality issue of the attributes in the above operations. Informally, the cardinality of an attribute can be either atomic or set. In each operation, we specify the attribute to be moved in a part class as a form of path expression. If the cardinalities of one or more attributes are sets, the type of the new attribute which is created as the result of a schema change operation could be ambiguous. Any valid path expression, in theory, can be used to specify the attribute to be moved, but if we allow an attribute with the cardinality of greater than 1 in the path expression, we might encounter an updatability issue, as like in ambiguous view updatability problem in relational views, which is still very hard to solve [26]. Thus, current *RiBS* model assumes only simple cases where such a subtle issue do not arise: (1) every attribute in the path expression is atomic, and (2) only the last attribute in the path expression is set and all the other attributes are atomic. In both case, the type of a new attribute is same to that of the last attribute in the path expression. For other cases, the *RiBS* model simply rejects the operation.

3.1.1. Add a part class into complex object hierarchy

As we already mentioned, the operation to add a new attribute into a class version, which comes from the basic *RiBS* model, can be regarded as an operation to add a part class into complex object hierarchy. That is, when the domain of the new attribute *pv* of class version CV_1 is another class version CV_2 on the current schema version *CSV*, the complex object hierarchy which contains CV_1 has class version CV_2 as its new part class. In this case, it should be noted that a new base class *bp* corresponding to *pv* is added to the *RiBS* layer.

3.1.2. Remove a part class from complex object hierarchy

A part class can be removed from a complex object hierarchy using the operation *drop an existing attribute version v from a class version*, which also comes from the basic *RiBS* model. Using this operation, we can drop an attribute version *pv* whose domain is the class version to be removed from a complex object hierarchy. In this case, in contrast to adding a part class into a complex object hierarchy, only the information about *pv* is removed from the current schema version, and the corresponding base attribute still remains in the *RiBS* layer.

3.1.3. Pull a property of a part class to root class

In the object-oriented data model, an attribute on a superclass is inherited into a subclass downwards along the class hierarchy via the inheritance mechanism. Similarly, an attribute in a part class on a complex object hierarchy can be conceptually viewed as an attribute of its root class; that is, 'an attribute in a part class *is also an attribute of* its root (or any whole) class upwards along the complex object hierarchy'.

For easy access to an attribute which are deeply nested from its root class, users would like to view the attribute as if it is defined on root class. This is very similar to a relational view which is defined using several complex join conditions and thus can be used as a shorthand for long and complex queries. The following operation allows users to do this kind of restructuring:

pull *path_exp* [**as** *new_attribute_name*] **in class** *class_name*.

The semantics of **pull** operation is that on a complex object hierarchy having the root class *class_name*, the leaf attribute over path expression *path_exp* is pulled to the root class. When the leaf attribute causes a name conflict with an attribute which is locally defined in the root class, this operation is rejected. In order to avoid this kind of name conflict or to assign a more meaningful name to the attribute, the user can use the optional **as** clause. For example, the following operation pulls the attribute of *Model#* in class version *CHASSIS* in Fig. 1 to root class *VEHICLE*, and renames the attribute *ChassisModel* in the root class

pull Body.Chassis.Model# **as** ChassisModel **in class** VEHICLE.

Fig. 3 shows the result of this operation (for the simplicity of presentation, we assume here that before running this operation, the structure of the complex object hierarchy rooting from class version *VEHICLE* is exactly the same as that of the *RiBS*) layer. As the result of the **pull** operation, a new attribute **ChassisModel** is introduced in class version *VEHICLE*, while the attribute version *Model#* is removed from class version *CHASSIS*, the reason for which is given later in this section. Please note that the domain of new attribute *ChassisModel* is same to that of *Body.Chassis.Model#*.

In that a new attribute is added to the root class, the operation **pull** is similar to the operation **add an attribute** from the basic *RiBS* model. However, one important difference between these operations is that the operation **add an attribute** requires a base attribute to be added into the *RiBS* layer while the operation **pull**, without affecting the *RiBS* layer, requires changes only in the current schema version. In case of the **pull** operation, the value of a newly introduced attribute in the root class is derived from that of the original path expression.

The above example shows how to upwardly pull (that is, move) a leaf attribute in a complex object hierarchy. In addition, the **pull** operation can be used to the non-leaf attribute, as shown below

pull DriveTrain.Engine **in class** VEHICLE.

Moreover, the **pull** operation can be applied to the cases where the path expression is cyclic, as shown in Fig. 4. Fig. 5 exemplifies how the **pull** operation is used to introduce a new attribute *GrandFather* in class *PERSON* which is a root class of a complex object hierarchy with a cyclic path.

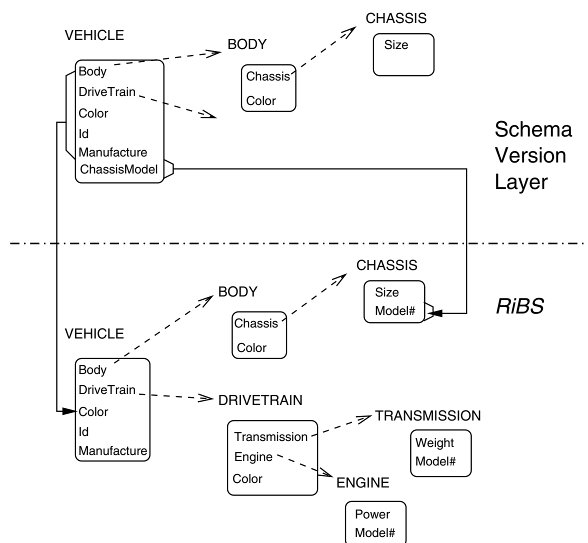

Fig. 3. Restructured complex object hierarchy using the **pull** operation.

The restructured root class using the **pull** operation is similar to the concept of views in relational databases, in the sense that the values of some attributes of its instance objects are derived from those of corresponding base attributes. For example, the restructured class *VEHICLE* in Fig. 3 is similar to a relational view via joining classes *VEHICLE*, *BODY* and *CHASSIS* and then projecting all of the attributes of *VEHICLE* and the attribute *Model#* of *CHASSIS*. However, there exists a big difference between these two concepts, that is, updatability. In relational databases, update operations on a join view defined over two or more base relations are not allowed because of the ambiguity for the updates [26]. In contrast, in case of the restructured root class using the **pull** operation, the concept of the object identifier in the object-oriented data model allows the avoidance of ambiguity in updates, and thus it is possible to update the instance objects of the restructured root class.

Finally, let us examine why the attribute version *Model#* is deleted from class version *CHASSIS* as the result of the **pull** operation, as shown in Fig. 3. This is due to the principle of the *RiBS* model, saying that a concept in the *RiBS* layer, such as a class or an attribute, has only one corresponding concept in a schema version. For instance, unless the attribute *Model#* is not deleted from class version *CHASSIS*, both attributes *Model#* in class *CHASSIS* and *ChassisModel#* in class *VEHICLE* have, as their base attribute, the attribute *Model#* of base class *CHASSIS* in *RiBS* in Fig. 3. However, in a cyclic relationship of complex object hierarchy, a class can become both a root class and a part class (e.g. Fig. 4), and this raises a subtle issue about the attribute list in a class version after a pull operation when the class version is both a root class and a part

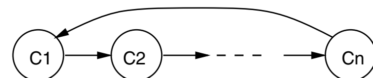

Fig. 4. A complex object hierarchy with a cycle.

pull Father.Father **as** GrandFather **in class** PERSON;

Fig. 5. Restructuring a cyclic complex object hierarchy using the **pull** operation.

class of the pull operation. That is, according to the semantics of the pull operation in non-cyclic relationship, where the pulled attribute is added to a root class, but removed from a part class, what should we do for a class which is a root class as well as a part class in the pull operation? In this case, because a part class itself can become a root class of another complex object hierarchy in the recursive class definition, we simply take the semantics of the perspective of a root class, that is, we do not remove the pulled attribute from the class, and add the pulled attribute (which must be renamed) into the class. This cyclic relationship in a pull operation can be simply detected if the root class in the operation is same to the domain of the pulled attribute, and thus we can automatically enforce this semantic for the pull operation.

3.1.4. Unnest: unnest a part class on root class

In this section, we describe the **unnest** operation, which generalizes the **pull** operation described in the previous section. The **unnest** operation allows information which was modeled as a part class to be represented by a set of attributes of the root class

unnest *path_expression* **in class** *class_name*.

Using the above syntax, all of the attributes of the class corresponding to the leaf node of the path expression *path_exp* are pulled upwards into the root class *class_name*. For this operation to be valid, the path expression should not end with a leaf attribute. The following example shows how this **unnest** operation is used to restructure the information modeled in a part class ENGINE in Fig. 1 as attributes of root class *VEHICLE*

unnest DriveTrain.Transmission **in class** VEHICLE.

This operation changes the complex object hierarchy of *VEHICLE* into the status shown in Fig. 6; while all of the attributes of class *ENGINE* move upwards to class *VEHICLE* preserving their names and domains, the attribute *Engine* is deleted from class *DRIVETRAIN* and class *ENGINE* itself disappears from the complex object hierarchy. The two attributes circled in class *VEHICLE* has come from the unnested class *ENGINE*.

Note that the upwardly pulled attribute *Weight* in Fig. 6 models the weight of the engine of a vehicle, not that of the vehicle itself, and thus the name of the attribute needs to be changed. To do this, we can use the operation **change the name of an attribute version** from the basic *RiBS* model.

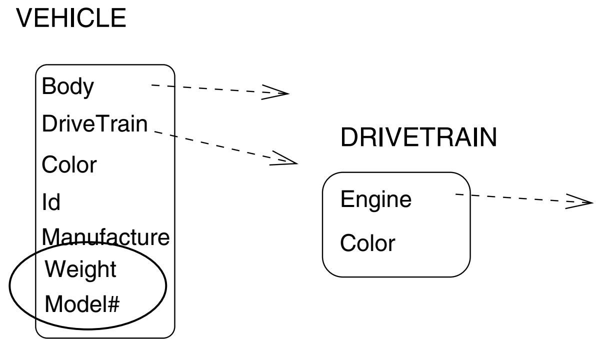

Fig. 6. Restructured class *VEHICLE* using **unnest** operation.

The **unnest** operation is similar to the **pull** operation in that both operations move attributes of part classes to the root class, but there exists a significant semantic difference between them. With the **unnest** operation, a part class is removed from the complex object hierarchy, while, with the **pull** operation, only an attribute is moved to the root class with its origin class surviving. In the latter case, the class version is, even after the operation **pull**, necessary still to be modeled.

3.1.5. Nest: nest part properties into a new virtual part class

This section introduces the **nest** operation which is converse to the **unnest** operation. The **nest** operation allows users to create a new virtual part class by combining several attributes from root class and part classes, and then models the new class as the domain of an attribute of the root class

nest *new_class_name*{*attribute_list*} **as**
new_attribute_name **in class** *class_name*.

In the above syntax, *new_class_name*, *new_attribute_name* and *attribute_list* represents the name of the new virtual class, the name of the new attribute, and the list of the attributes to be included in the virtual class, respectively. *attribute_list* is a list of attributes, whose syntax is as follows:

{*path_exp* [**as** *new_attribute_name*]}*

We call the new class version resulting from the **nest** operation as a virtual class version because the new class version does not have its corresponding base class in the *RiBS* layer. In fact, its attributes may originate from several different class versions and thus the attributes have their corresponding base attributes spreaded over different base classes. In contrast, each class version in the

basic *RiBS* model has its corresponding base class in the *RiBS* layer, and in this respect, all the class version in the basic *RiBS* model are non-virtual.

The following example shows how a virtual class *PARTMODEL* is created, using the **nest** operation: the **nest** operation collects all the model numbers of the parts of class *VEHICLE* into the class *PARTMODEL*. Fig. 7 shows the result of this operation, where a new attribute *PartModel* is added to the class *VEHICLE* and it has the new virtual class *PARTMODEL* as its domain.

nest PARTMODEL {Body.Chassis.Model# **as** ChassisModel#, DriveTrain.Transmission. Model# **as** TransModel#, DriveTrain.Engine.Model # **as** EngineModel#}
as PartModel **in class** VEHICLE.

As we have noted above, the **nest** operation creates a virtual class version in the current schema version. However, the virtual class version raises a subtle issue with regard to schema evolutions. Since a virtual class version is also a class version, all valid schema evolution operations for class versions should also be applicable to this virtual class version. However, unlike the basic *RiBS* model where each class version has its corresponding base class in the *RiBS* layer, each virtual class version in the extended *RiBS* model does not have its base class in the *RiBS* layer, and thus, for example, when the schema evolution operation **add a new attribute version** is imposed on it, it does not have base class to which a corresponding base attribute is added. We take a simple approach to this problem, that is, when the operation *add a new attribute version* is imposed on a virtual class version, we create a corresponding base class in the *RiBS* layer with a base attribute corresponding to the new attribute version, and the class version becomes non-virtual.

3.1.6. Move: move a part property to a part class

In some cases, a user may want to customize his/her own view for a complex object hierarchy in a different way so that an attribute version of a part class is modeled as an attribute version of another part class which is usually in a path from root class different from the original part. In order to support these requirements, we introduce the following **move** operation:

move *src_path_exp* **to** *dest_path_exp*
[**as** *new_attribute_name*] **in class** *class_name*.

The above operation moves the (leaf or non-leaf) attribute in path expression *src_path_exp* to the class in path expression *dest_path_exp*. *dest_path_exp* must be a non-leaf node. In this case, the attribute being moved disappears from the original class, according to the principle of the *RiBS* model described above. The **as** clause is optionally used to change the name of the attribute.

The following exemplifies the usage of the **move** operation: the *Chassis* attribute of class *BODY* in Fig. 1 moves to class *DRIVETRAIN*. Fig. 8 shows the result of this operation, where the new attribute *BodyChassis* is added into the class *DRIVETRAIN*

move Body.Chassis **to** DriveTrain **as**
BodyChassis **in class** VEHICLE.

However, with regard to the **move** operation, there is a constraint on the two path expressions *src_path_exp* and *dest_path_exp*, that is, *src_path_exp* should not be a part of *dest_path_exp*. The following example violates this constraint:

move *DriveTrain* **to** *DriveTrain.Transmission*
in class *VEHICLE*.

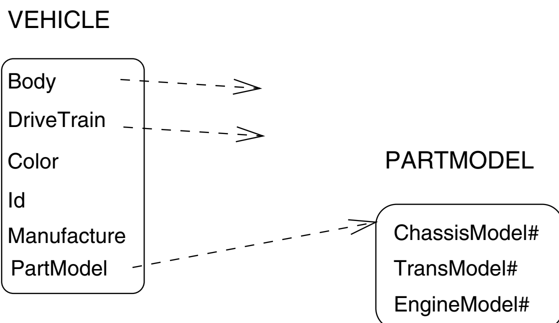

Fig. 7. An example of the **nest** operation.

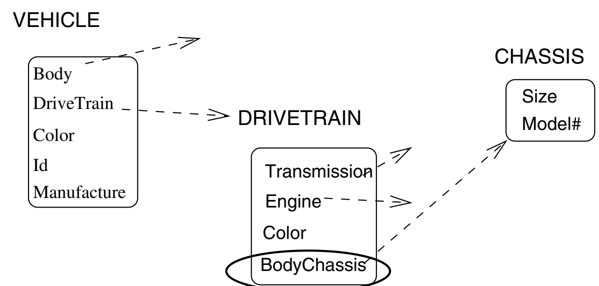

Fig. 8. The **move** operation: an example.

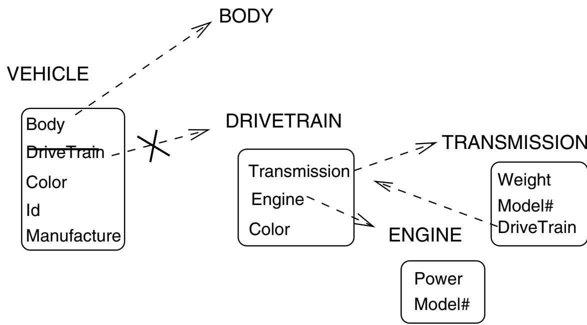

Fig. 9. An illegal usage of the **move** operation.

The above operation, as shown in Fig. 9, causes two problems; (1) class *DRIVETRAIN* is no longer a part class of the complex object hierarchy, and (2) class *DRIVETRAIN* becomes a part class of class *TRANSMISSION*. As a result of the first problem, the information available in the complex object hierarchy with DRIVETRAIN as its root is unreachable from the root class *VEHICLE*. The second problem introduces the relationship *IsPartOf*(*B*, *A*), which is contradictory to the relationship *IsPartOf*(*A*, *B*). Due to these problems, we do not allow the **move** operation which violates the constraint.

4. Query processing for restructured complex object hierarchy

In the basic *RiBS* model, as mentioned in Section 2, a query or an application program written against a schema version, before its execution, is translated into one against the *RiBS* layer. In this section, we will illustrate how the preprocessing step translates a query against a schema version resulting from the operations in Section 3, into the query against the *RiBS* layer.

When a user imposes a query against class *VEHICLE* of Fig. 3, this query, during the preprocessing step, changes into a query against the *RiBS* layer. The attribute *ChassisModel* which has moved to root class *VEHICLE* by the **pull** operation, is replaced with its corresponding path expression *Body.Chassis.Model#*. In case of a complex object hierarchy restructured using the **unnest** operation, this kind of simple transformation is also applicable

```
select Car.Color                                  select Car.Color
from VEHICLE Car                           =>     from VEHICLE Car
where Car.ChassisModel = 'MD001';                 where Car.Body
                                                  .Chasssi.Model# = 'MD001';
```

However, the **move** operation requires a little more complex transformation. For example, let us see an example query written against the schema version of Fig. 8, the query graph of which does not include class version *VEHICLE*

```
select Engine.Model#
from DRIVETRAIN Train
where Train.BodyChassis.Model# =
'MD001'.
```

In case of the above query, the path expression in the `where` clause cannot be replaced with a path expression starting with the base class of class version *DRIVETRAIN*. Instead, this query must be transformed into a query against the *RiBS* layer, the query graph of which starts from class *VEHICLE*, as follows:

```
select Car.DriveTrain.Engine.Model#
from VEHICLE Car
where Car.Body.Chassis.Model# =
'MD001'.
```

A similar transformation should be applied to a query whose query graph starts from the virtual class version which was created using the **nest** operation.

5. Schema version merging

As we have mentioned in Section 2, the basic *RiBS* model supports the schema version merging. In this section, we extend the process of schema version merging so that it can handle the restructured complex object hierarchy also. In particular, we identify several new conflicts during the schema version merging, which arises from the operations in Section 3, and provide an algorithm which can detect and resolve these conflicts. This algorithm is semi-automatic in that it automatically detects all types of conflicts while users are responsible for resolving the conflicts.

5.1. Types of conflicts

As the basic *RiBS* model in Section 2, the operations for restructuring the complex object hierarchy also cause three kinds of new conflicts, as follows. The homonyms and synonyms are not new types of conflicts, but the sources for these conflicts are the restructuring operations.

1. *Homonyms*: Two or more class (attribute) versions, from different schema (class) versions but having the same name, may have different direct base classes (attributes). We call them homonym class (attribute) versions.
2. *Synonyms*: Two or more class (attribute) versions, from different schema (class) versions and having different names, may have the same direct base class (attribute). We call them synonym class (attribute) versions.
3. *Class-attribute conflicts*: In different schema versions, a real-world concept is represented differently using two different constructs of object-oriented data model, that is, the concept of class and attribute. We say that any two schema versions being merged, have class-attribute conflicts if one schema version models a real-world concept using a class version, while the other models the concept as attributes of another class version.

When merging the two schema versions in Fig. 10, we meet all kinds of the above conflicts. Here, we make the following assumption; each schema version, after derived from a common schema version exactly the same as *RiBS*, experienced the following histories of evolutions, respectively. In the following, **drop** and **rename** represents the operation *drop an existing attribute version from a class version* and *change the name of a attribute version*, respectively.

- The evolution history of SV_i
  1. **drop** Color **from** VEHICLE;
  2. **pull** Body.Chassis.Color **in class** VEHICLE;
  3. **move** Body.Chassis **to** DriveTrain **as** Body-Chassis **in class** VEHICLE;
  4. **rename** DriveTrain **as** BodyTrain **in class** VEHICLE;
  5. **nest** MODEL{Transmission.Model# **as** Trans-Model#, Engine.Model# **as** EngineModel#} **as** Model **in class** DRIVETRAIN.

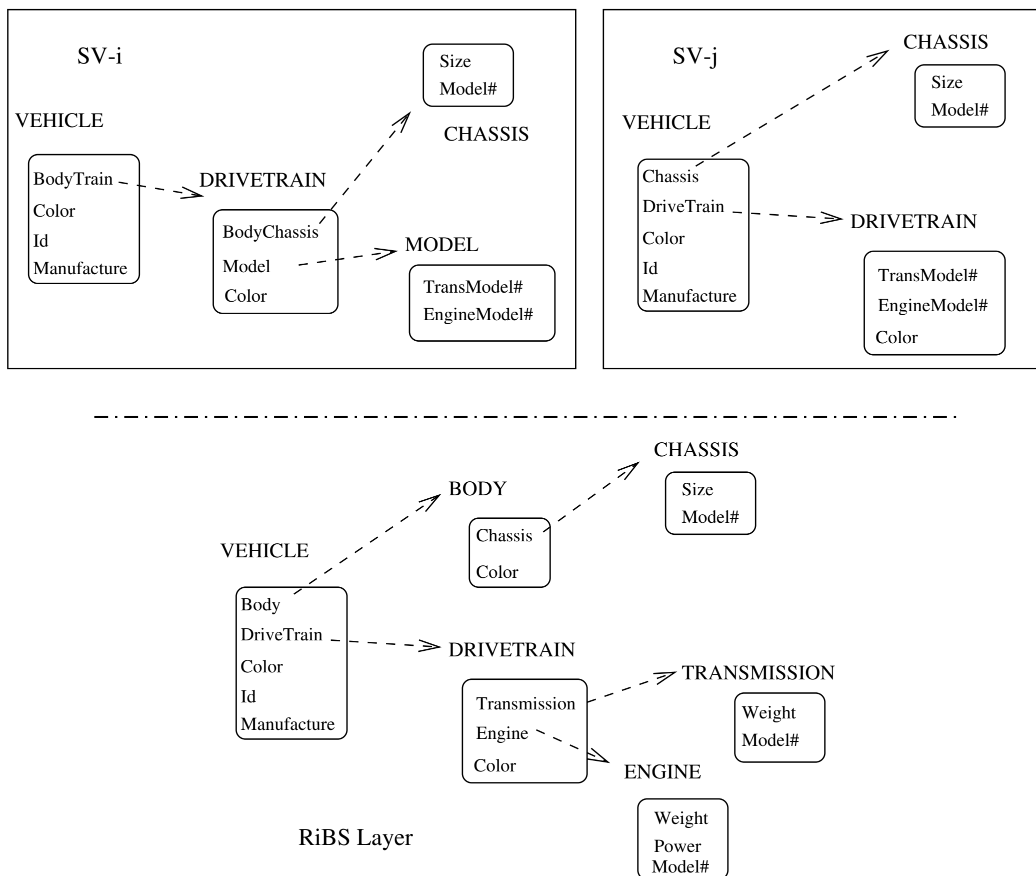

Fig. 10. Schema version merging: an example.

- The evolution history of SV_j
  1. **pull** Body.Chassis **in class** VEHICLE;
  2. **pull** Transmission.Model# **as** TransModel# **in class** DRIVETRAIN;
  3. **pull** Engine.Model# **as** EngineModel# **in class** DRIVETRAIN;
  4. **drop** Transmission **from** DRIVETRAIN;
  5. **drop** Engine **from** DRIVETRAIN.

In Fig. 10, two attribute versions *Color*s in both class version *VEHICLE*s of schema version SV_i and SV_j are an example of homonym; the former has attribute *Color* of class *VEHICLE* in *RiBS* as its base attribute, while the latter has the attribute *Color* of base class *BODY* as its base attribute.

And, there are two examples of synonyms in Fig. 10. First, the attribute *BodyTrain* in class *VEHICLE* of SV_i and the attribute *DriveTrain* in class *VEHICLE* of SV_j are an example of synonyms because both of them, although their names are different from each other, have the attribute *DriveTrain* in class *VEHICLE* in *RiBS* layer as its base attribute. The second example of synonyms in Fig. 10 comes from the attribute version *BodyChassis* of class version *DriveTrain* in SV_i and the attribute version *Chassis* of class version *VEHICLE* in SV_j. Both attribute versions have the attribute *Chassis* of base class *BODY* in *RiBS* as their base attribute. At this point, note that these two attribute versions are from two class versions having different base classes in *RiBS* layer. In the basic *RiBS* model, in contrast, all class versions of synonym attribute versions have the same base class in *RiBS* layer.

Finally, an example of class-attribute conflicts is that class version *MODEL* in SV_i and two attribute versions, *TransModel* and *EngineModel* in class version *DRIVETRAIN* represents the same information.

5.2. Schema version merging

In the basic *RiBS* model, schema version merging largely consists of the following four steps:

1. To identify base classes to be included in the new schema version.
2. To create a corresponding class version for each base class.
3. To calculate local attribute versions for each class version.
4. To make class hierarchy for the new schema version.

Here, we briefly explain the role of each step. Please refer [17] for more detailed explanation. The step 1 identifies the base classes necessary in the new schema version by including all the base class in *RiBS* layer which are used as the direct base class of a class version in any input schema version. The step 2 is responsible for resolving the synonyms, homonyms and extent conflicts among class versions, and the step 3 is for synonyms and homonyms among attribute versions. The final step 4 makes DAG (Direct Acyclic Graph) relationships among all the class versions in SV_{new}. The operations for restructuring the complex object hierarchy requires this process of schema version merging to be extended. The following lists the issues which should be considered:

1. *Virtual class versions*: when we identify base classes to be included in the new schema version, we need to take the virtual class version into considerations, which is created by the **nest** operation.
2. *Class-attribute conflicts:* it is necessary to detect the class-attribute conflicts and resolve them.
3. *Synonyms:* as we stated above, synonym attributes of class versions having different base classes need another extension.

The following shows the extended process of schema version merging taking into accounts all the above issues, where the newly introduced steps are marked using the symbol $*$:

1. To identify base classes to be included in the new schema version.
2. To create class versions
   (a) To create a corresponding class version for each base class.
   (b) To detect and resolve class-attribute conflicts.
   (c) To create virtual class versions ($*$).
3. To calculate local attribute versions
   (a) To detect and resolve the synonym attribute of class versions having different base classes ($*$).
   (b) To calculate local attribute versions for each class version.
4. To make class hierarchy for the new schema version.

In the rest of this section, we will illustrate the above steps using an example. Because the step 4 is to make class hierarchy, we omit the explanation about it. Fig. 11 shows the steps of merging two schema versions of Fig. 10. We refer to the result schema version as SV_{new}.

First, the step 1 identifies three base classes, *VEHICLE*, *DRIVETRAIN* and *CHASSIS*, each of which is used as a base class of class version(s) in either schema version SV_i or SV_j. Next, using these three base classes, the step 2(a) creates the three corresponding class versions, as shown in Fig. 11(a). Then, through the steps 2(2) and 2(3), new class version *MODEL* is introduced by resolving class-attribute conflicts, as in Fig. 11(b). Let us remind you that, in Fig. 10, class version *MODEL* in SV_i and two attribute versions, *TransModel* and *EngineModel* in class version *DRIVETRAIN* in SV_j cause a class-attribute conflict. We assume that, for this conflict, the user has chosen to model the information as a class version in SV_{new}. Please note that since *MODEL* in SV_i is a virtual class version, the corresponding new class version in SV_{new} must be also virtual. And, during the step 3(a), there are two resolutions for synonym attributes. Here, we assume that for two synonym attributes *BodyChassis* of SV_i and *Chassis* of SV_j in Fig. 10, the user has chosen the latter. We assume also that for two synonym *BodyTrain* of SV_i and *DriveTrain* of SV_j in Fig. 10, the user has chosen the latter. Consequently, as shown in Fig. 11(c), the attribute *Chassis* is included in *VEHICLE* class version and it has the class version *CHASSIS* as its domain, and the attribute *DriveTrain* is included in *VEHICLE* class version and it has the class version *DriveTrain* as its domain. Finally, the step 3(b) calculates the local attribute versions of each class version. In particular, note that local attributes of each class version can be automatically calculated: that is, we create an attribute version for each base attribute in base classes which has its corresponding attribute version in any schema version to be merged. Fig. 11(d) shows the final status of merging two schema versions SV_i and SV_j, where the local attribute versions of each class version in SV_{new} are calculated. Note that two attribute versions *Color* and *BodyColor* of class version *VEHICLE* in SV_{new} was derived from two homonyms attribute versions *Color* of class version *VEHICLE* in both schema version SV_i and SV_j, and the user has resolved this homonym problem so that attribute *Color* from SV_i is renamed as *BodyColor* in SV_{new}.

Before closing this section, we would like to note that any two concepts in new schema version SV_{new} do not have a common corresponding base concept in the *RiBS* layer, and this complies with the principle of the *RiBS* model, saying that a concept in the *RiBS* layer, such as class or attribute, has only one corresponding concept in a schema version. The steps for resolving the synonyms and class-attribute conflicts prevents a concept in the *RiBS* layer to be redundantly modeled by more than one concepts in the new schema version.

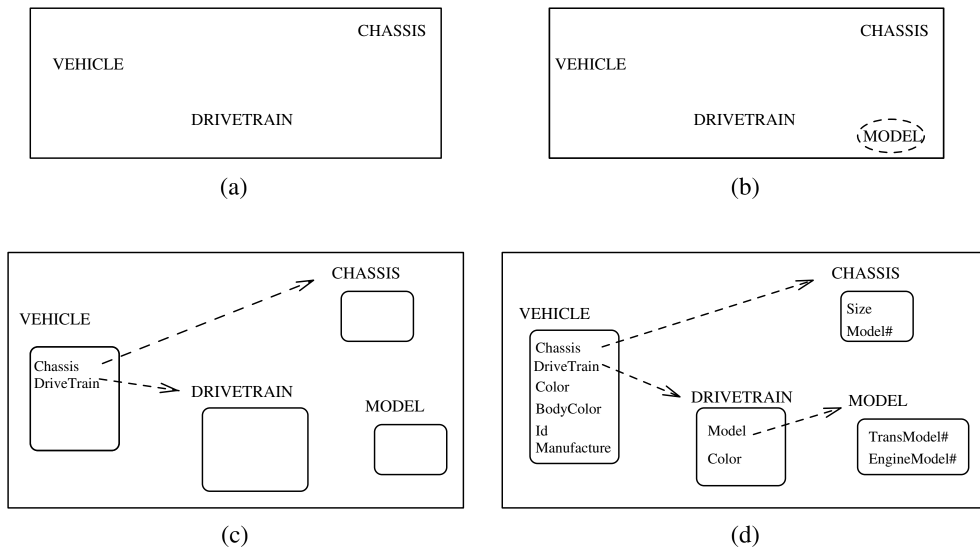

Fig. 11. Schema version merging: an example.

6. Related works

To our best knowledge, there has been no work on schema versions for complex object hierarchy. Therefore, we will review the four categories of researches which are partly related to our schema version model, that is, (1) views in OODBs [1,8,14], (2) schema/class version [15,20], (3) database schema integration [6,14], and (4) web site management system [2,4,12,11]. This section summarizes these approaches and explains their differences from our schema version model.

6.1. View approaches

There have been works to provide OODBs with the advantages of view functionality in relational databases, e.g. logical data independence and content-based security. Ref. [1] describes the view mechanism in O_2 system, which support the restructuring of class hierarch and virtual classes. Kim presented a view semantic within an Object/Relational DBMS, UniSQL, by augmenting semantics of relational views with object-oriented concepts such as inheritance, method, and object identifier [14]. In addition, they extended the use of views to dynamic windows for schema, with which schema evolution in OODBs can be simulated without affecting the database. Rundensteiner proposes the MultiView methodology, where a view schema, according to a user's viewpoint, can be defined over a global schema [25]. These works are along the same line as the approach in [8] simulating schema evolution using views.

In the sense that schema versions of our model provides logical views over base schema, it is in the same vein of these works. However, any of these view mechanisms does not consider the restructuring of complex object hierarchy, which our schema version model supports the restructuring of complex object hierarchy.

Our *RiBS* approach is similar to these view approaches in the sense that each schema version is defined over one global base schema *RiBS*. However, there are several important differences between our schema version model and the previous works on views in OODBs. First, while direct schema updates against a schema version are allowed in our 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 [24]. Finally, the existing view approaches do not even consider the concept of schema version merging which is an essential functionality of OODB applications.

6.2. Schema versioning approaches

The work in [15] is the first substantial research on schema versions in OODBs, based on the object version model of Orion [5]. 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 thus may not be shared among schema versions. Moreover, in contrast to the *RiBS* model, a new schema version can be derived from only one parent schema version and thus the schema version derivation hierarchy results in a tree.

As an alternative to schema versioning, there has been the class versioning approach [20], where the units of versioning are individual classes, instead of the entire class hierarchy. Monk and Somerville proposed a class versioning system CLOSQL [20], 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 the *RiBS* layer 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 [15].

However, all these works deals only with changes in 'IS-A' inheritance hierarch, but does not consider complex object issues. Though, the **moving information** operation [18] and **capacity reducing transformation** operation [28] have similar semantics with our **pull** and **unnest** operations, respectively.

6.3. Database schema integration

In the database literature, many methodologies for integrating database schema are found in the form of view integration, database schema integration, or multi-database. Our work on schema version merging for the *RiBS* model shares some concerns with methodologies for database schema

integration; the detection of several types of conflicts and their resolution.

In [6], a unifying framework for the problem of view and database schema integration is provided, and several earlier works 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 distinguish two types of conflict: *name conflicts* and *structural conflicts*. Name conflicts are further classified into homonyms and synonyms. However, *class-attribute conflict* of the extended *RiBS* model have no corresponding concept in the taxonomy presented in [6], although we classify them as structural conflicts. As for conflict resolutions, Batini et al. state that automatic resolution is generally not feasible [6]. 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. A similar framework for classifying schema and data conflicts in federating multi-database systems can be found in [14]. This work also deals with conflict issues such as homonym and synonym.

However, there is one important difference between these works on database schema integration and our work on 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 much less intervention from the user.

6.4. Web site management systems

With the rapid spread of the Internet, enormous information are being provided in the form of Web pages. By the way, one problem during Web site construction is how to separate the logical view of each web page from the physical organization of the website information [12]. Thus, it is very difficult to restructure the content provides in each webpage, that is, the lack of data independence.

To solve this problem, much work has been done on Web site management system, which mainly supports the restructuring of user's logical views [2,4,12,11]. In that both our schema version model and these systems customize user's logical views over complex graph data structure, they have common functionality. The main difference between our model and web site management systems is that our model supports restructuring of structured schema while web site management systems supports semi-structured schema.

7. Conclusions

Recently, the necessity for schema versions has been rejuvenated from several new OODBMS applications, but the schema evolution functionality of commercial OODBMSs, under which only a single schema exists at any point in time, cannot properly cope with the requirements of these applications. Therefore, we strongly believe that the functionality of the schema version will be a pre-requisite for an OODBMS to be widely accepted in the market.

In this paper, we proposed a schema version model for complex objects. The model includes a set of new operations for restructuring complex object hierarchy, with which a user can customize database schema according to his/her own view. We also identified some issues to be considered when preprocessing restructured complex object hierarchy, and proposed ways to transform queries against restructured complex object hierarchy into one against the base schema. Because our model has taken into account the concept of complex object, it is more comprehensive than any other schema version model ever developed. Also, these schema evolution operations for complex objects can be used as an alternative mechanism for view definitions in OODBMSs. Finally we developed a schema version merging algorithm for schema versions of complex objects. We identified several types of conflicts, and took into account these conflicts when devising our merging algorithm. This schema version merging algorithm first resolves several conflicts among schema versions being merged, and then generates class hierarchy for the new schema version. This algorithm is unique, compared to traditional database schema integration algorithms, because it exploits the semantics knowledge shared by the schema versions being merged, and thus can integrate schema versions with minimal user involvement during the merging process.

References

[1] S. Abiteboul, A. Bonner, Objects and views, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1991.

[2] G.O. Arocena, A.O. Mendelzon, WebOQL: restructuring documents, databases and webs, Theory and Practice of Object Systems 5 (3) (1999) 127–141.

[3] T. Atwood, Object databases come of age, Object Magazine (July) (1996).

[4] P. Atzeni, G. Mecca, P. Merialdo, To weave the Web, in: Proceeding of International Conference on Very Large Data Bases, 1997.

[5] J. Banerjee, W. Kim, H.-J. Kim, H. Korth. Semantics and implementation of schema evolution in object-oriented databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Francisco, CA, 1987, pp. 311–322.

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

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

[8] E. Bertino, A view mechanism for object-oriented databases, in: Third International Conference on Extending Database Technology, LNCS 580, Vienna, Austria, 1992, pp. 136–151.

[9] R. Cattell, D.K. Barry (Eds.), The Object Database Standard: ODMG 3.0, Morgan Kaufmann, 2000.

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

[11] G. Falquet, J. Guyot, L. Nerima, Languages and tools to specify hypertext views on databases, in: International Workshops on the Web and Databases, 1998.

[12] M. Fernandez, D. Florescu, J. Kang, A. Levy, Catching the boar with strudel: experiences with a web-site management system, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1998.

[13] W. Kim, Introduction to Object-Oriented Databases, MIT Press, 1990.

[14] W. Kim (Ed.), Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, 1995.

[15] W. Kim, H. Chou, Versions of schema for object-oriented databases, in: Proceeding of International Conference on Very Large Data Bases, 1988.

[16] S.-E. Lautemann, A propagation mechanism for populated schema versions, in: Proceedings of International Conference on Data Engineering, 1997.

[17] S.-W. Lee, H.-J. Kim, Rich base schema (RiBS): a unified framework for OODB schema version, Journal of Database Management (January) (2000) 33–41.

[18] B.S. Lerner, A.N. Habermann, Beyond schema evolution to database reorganization, in: Proceedings of International Conference in Object-Oriented Programming: Systems, Languages, and Applications, 1990.

[19] M.E. Loomis, Object database – integrator for PCTE, Journal of Object-Oriented Programming (May) (1992) 53.

[20] S. Monk, I. Sommerville, Schema evolution in OODB using class versioning, SIGMOD Record 22 (3) (1993).

[21] Objectivity, Inc. "Schema evolution in objectivity/DB". White paper available from: <http://www.objectivity.com/DevCentral/Products/TechDocs/pdfs/SchemaEvolutionWP.pdf>, 2004.

[22] POET Software. "POET object server suite". White paper available from: <http://www.it-analysis.com/research-archivepdf.php?id=309>, 2004.

[23] Progress Software. "Objectstore data sheet, version 6.1". White paper available from: <http://www.objectstore.net/products/docs/objectstore_datasheet.pdf>, 2004.

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

[25] E.A. Rundensteiner, Incremental maintenance of materialized object-oriented views in multiview: strategies and performance evaluation, IEEE Transaction on Knowledge and Data Engineering 10 (1998) 768–792.

[26] A. Silberschartz, H.F. Korth, S. Sudarshan, Database System Concepts, fifth ed., McGraw Hill, 2005.

[27] A. Silberschartz, M. Stonebraker, J. Ullman, Database research: achievements and opportunities into the 21st century, SIGMOD Record 25 (1) (1996) 52–63.

[28] M. Tresch, M.H. Scholl, Schema transformation without database reorganization, SIGMOD Record 22 (1) (1993).

[29] Versant, Versant developer suite, White paper available from: <http://www.versant.net/eu_en/products/vds>, 2004.

[30] C. Zaniolo, The database language GEM, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1983, pp. 207–218.

[31] R. Zicari, F. Ferrandina, Schema and database evolution in object database systems, in: Part 6, Advanced Database Systems, 1997.