Maintaining Database Consistency in the Presence of Schema Evolution: An Evolutionary Approach based on Versions of Schema

Ling Liu
University of Alberta
Department of Computing Science
615 GSB, Edmonton, Canada T6G 2H1
Email: lingliu@cs.ualberta.ca

Abstract

With rapid advances in computer network technology and the increasing interest in global information sharing, grows the need for facilities that can effectively maintain the database consistency and program compatibility as the schema evolves. In this paper, we present a general framework based on versions of schema for supporting seamless schema evolution in large-scale object-oriented software systems. We argue that the effectiveness of using the schema version model to materialize schema evolution relies not only on the management of version derivation of schema, but also on the ability to maintain consistency of the database instances with the new schema versions, and the flexibility of sharing instance access scopes among versions of schema. Semantics of schema versioning is studied with the objective to facilitate instance adaptation and program compatibility in the presence of schema updates. A number of options for sharing of instance access scopes among versions of a schema is developed. Applications may derive versions of schema upon requests for schema updates, and define the instance access scope for each new version of schema by either creating their own instance access scope or inheriting the instance access scope of its ancestor schema versions in terms of a selection of options. The significance of our approach is

- the abilities for maintaining database consistency, in the presence of schema modification, without irreversibly changing the objects that exist before the schema modification, and
- the facilities that allow multi-users and applications to create and manipulate different collections of objects under different versions of schema.

As a consequence, many organizational investments of the existing customer set, such as application design and development, can remain operational in the presence of schema changes.

Index Terms: *Change management, data models, database consistency, engineering and design databases, program compatibility, schema evolution, schema versioning, software maintenance, version control.*

1 Introduction

Seamless schema evolution and schema versioning are highly desirable functionalities in a multi-user design environment. A design schema often can be quite complex and large in size, requires multi-authoring, and evolves frequently. The representative applications include financial trading systems, global decision support systems (such as risk assessment in banking and insurance), value-added telecommunication services, hospital and federal health-care information systems, and workflow management systems. In these advanced applications, it is critical to preserve the history of evolution of objects in the presence of schema evolution, and be able to automate the conformance of database instances to the new versions of a schema, such that the users may update a schema and maintain the database consistency without irreversibly changing the objects that exist before the schema modification. As a consequence, the amount of effort required for reprogramming of existing application programs, due to schema changes, can either be avoided or be substantially reduced. The impact of schema changes made by one user or a single application, over the entire system and the existing customer set, can be limited to the minimum. Furthermore, users may derive versions of schema upon requests for schema evolution, and create and manipulate different collections of objects under different versions of schema.

Unfortunately, existing research and development in the area of schema evolution has so far mostly focused on the enhancement of schema evolution functionality in database management systems with no explicit support for versions of schema (cf. [4, 7, 9, 10, 11]). However, if a schema cannot be versioned, objects that existed before a schema change, in general, will be irreversibly changed due to schema modification. For example, if an attribute of a class is dropped, the values of this attribute in the existing object instances of the class become no longer visible to the applications, even when the system uses the filtering or object versioning approach to prevent the values of the deleted attribute from getting lost. The primary reason is because, after the schema change and the instance adaptation, values of the deleted attribute become no longer visible to the existing applications under the updated schema, even though they might still be accessible to the system software.

Furthermore, there has been surprisingly little attention given to the role of the schema versioning in managing schema evolution, and the potential feasibility of maintaining the database consistency with the new version of a schema without irreversibly changing the objects that exist before the schema modification. An exception is found in Orion [3], which presents a comprehensive study on the semantics and implementation of versions of schema. However, Orion's proposal addresses the modeling and the implementation issue of schema versioning more at the schema level. An important issue, which left open, is how the model of schema versioning may enhance the functionality of schema evolution, and, in particular, may facilitate the conformance of database instances to the new schema version after schema updates. Moreover, from our experience with a multi-user design environment, it is highly desirable to provide the users with a multi-level flexibility in sharing instance access scopes among the versions of schema. The two-level access scope sharing (i.e., share everything or nothing) proposed in Orion is too limited. To highlight this point, suppose a version of a schema, say SV_1, is given. First of all, the creator of SV_1 should have authority to decide whether the instance access scope of SV_1 can be shared (i.e., sharable) by its descendant schema versions. Secondly, for any child version, say SV_2, of SV_1, the creator of SV_2 should be able to determine the extent to which SV_2 needs to inherit the instance access scope of its parent schema version SV_1, rather than being forced to share everything or nothing. Several possibilities exist. For instance, SV_2 may inherit only the snapshot of the access scope of SV_1 at the time of schema version derivation. This means that all the subsequent database updates (insertion, deletion, modification) under SV_1 are transparent to SV_2. Alternatively, SV_2 may want to inherit the instance access scope of SV_1 *as well as* the subsequent object modifications and deletions under SV_1. It also means that only the subsequent insertions to the database of SV_1 are transparent to SV_2.

In this paper, we present a general framework, called DB-EVOLVE, for schema evolution and instance adaptation based on versions of schema. Semantics of schema versioning is studied with the objectives to facilitate schema evolution and instance propagation due to schema updates. A selection of options for sharing of instance access scope among versions of schema is developed, offering various levels of flexibility for schema designers, application developers, and end-users to manipulate and maintain available database resources in the presence of schema evolution. As a result, users may derive versions of schema upon a request for schema update, and define the instance access scope for each new version of schema by either creating their own instance scope or inheriting the instance access scope of its parent version(s) in terms of the multiple inheritance options. Based on our general framework, we also develop a collection of rules for triggering the default or user-defined transformation methods, which conform the objects of a schema version to the newly derived schema version. In short, the effectiveness of using the model of schema versions to support schema evolution relies not only on the management of version derivation of a schema but also on the ability to maintain the database consistency with the new schema version after schema updates, and the flexibility of sharing instance access scopes among versions of schema.

The rest of the paper proceeds as follows. In Section 2, we give a brief presentation of our reference object model and basic concepts for schema evolution. In Section 3, we present a general framework for versions of schema and a selection of options for sharing of instance access scope among versions of schema. Section 4 defines a high-level user interface which allows users to work with the proposed framework for realization of schema evolution and for maintaining database consistency as required, due to schema changes. We outline our implementation considerations in Section 5, including the data structure for objects, the algorithms for accessing objects under different versions of schema, and the storage representation of classes and schema versions. We compare our work with the related research in Section 6, and conclude in Section 7 with a summary and some future lines of research.

2 Basic Concepts

2.1 The Reference Object Model

We assume a fairly standard basic object-oriented data model. Objects are either of primitive types (such as Integer, Real, String) or of constructed types. The constructive types are built through recursive application of type constructors like tuple, set, list to the primitive types. Each object is described by a unique identity, the structure description and the set of methods. We use properties to refer to instance variables (attributes) and methods of objects.

Objects are grouped into classes based on a set of common properties, and are only accessible through their property functions defined in classes. The term *class* serves a dual purpose. It imposes a *type description* which consists of a finite set of property functions as a common interface and meanwhile denotes the *set* of objects which conform to its *type*. Thus, each class C is described by a unique class name, a type description and a set membership. Two kinds of relationships are explicitly distinguished between classes: inheritance (or *is-a*) relationships and object reference (called *construction*) relationships.

A class C_1 has a *is-a* relationship with a class C_2 if and only if all properties of objects of C_2 are also properties of objects of C_1. We call the class C_1 *subclass* and C_2 *superclass*, and refer to the property sharing as inheritance. The set of *is-a* relationships in a schema forms the *is-a* class hierarchy. No cycle is allowed in the *is-a* hierarchy. When a class inherits properties from more than one superclass, we call this feature *multiple inheritance*. Name conflicts between a class and its superclasses and among the superclasses of a given class are resolved by giving the precedence to the definition within the class over that in its superclasses, and by using superclass ordering (details see the next subsection). In addition, a class may override an instance variable or method by defining one locally with the same name.

When a class C_2 is a domain of a reference property of C_1, we say that the two classes have a *construction* relationship and refer to C_2 as a *component* class and C_1 as a *composite* class for presentation convenience. We call the set of object *construction* relationships the *construction hierarchy*. Loops and self-loops are allowed in the class construction hierarchy.

2.2 Basic Schema Evolution Invariants

We below list five DB-EVOLVE basic schema evolution invariants. They are to some extent similar to thosed used in Orion [3] and OTGen [7].

1. Unique Name Invariant Each class must have a unique name. Each instance variable and method, defined or inherited by a class, must have a unique name.

2. Subclass and Superclass Invariant The subclass-superclass relationship forms a *is-a* class lattice, with the system-defined class `OBJECT` as the root.

3. Typed Instance Variable Invariant The type of each instance variable must have a corresponding class in the class lattice.

4. Inheritance Invariant A class inherits all properties (instance variables and methods) from its superclasses, unless it redefine a property with the same name. When more than one superclass defines the same name property, the class should only inherit the one defined by the superclass that appears earliest in the superclass list of the class.

5. Type Compatibility Invariant When a class C_i defines an instance variable with the same name as an instance variable it would otherwise inherit from one of its superclasses C_j, the type of C_i's instance variable must be a subclass of the type of C_j's instance variable.

Numerous extensions can be made to this basic model of schema evolution invariants without compromising the capabilities of DB-EVOLVE. For example, one possible extension is the addition of component class invariants to provide "*part-of*" semantics.

2.3 Schema Evolution Primitives and Effect of Schema Changes

Schema evolution may require changes to a single class or a relationship between two classes. In an object-oriented model with inheritance, changes to a single class may affect all subclasses of the changed class. The schema evolution primitives supported by the DB-EVOLVE include

- Adding a property (instance variable and method);
- Deleting a property (instance variable and method);
- Renaming a property (instance variable and method);
- Modifying the domain type of an instance variable or the signature of a method;
- Adding a class to the superclass list of a given class;
- Removing a class from the superclass list of a given class;
- adding a class;
- Deleting a class;
- Renaming a class.

Whenever a schema change is requested, the database administrator (DBA) initiates the change by updating the class definition using DB-EVOLVE. If the change violates any schema evolution invariants, for example, any name conflicts arise during the inheritance recomputation, or the class lattice becomes disconnected, or some type incompatibility is incurred, the DB-EVOLVE will provide the DBA with a warning and an option for committing or aborting the change request. Of course, temporary violations of the schema invariants are allowed. However, all invariants must hold when the database is transformed into a new state.

2.4 Transformation Methods

There are mainly two ways to associate with object transformations when schema changes occur. One is to define and associate transformations with each schema evolution operation. The other way is to define transformations for each modified class. The DB-EVOLVE adopts the second alternative.

To illustrate the transformation methods and their associations to a class, consider the following schema:

```
Class Person                               Class Address
 pname:        Person -> String;            street#:     Integer;
 birthday:     Person -> Integer;           street-name: String;
 home-address: Person -> Address;           zipcode:     String;
 end Person                                 end Address
```

Suppose now a user want to modify the above schema by adding the details of address information into the `home-address` of `Person` objects, instead of via reference to object of class `Address`. The user may simply define the expected schema and the intended transformation method as follows:

```
Class Person
  pname:        Person -> String;
  age:          Person -> Integer;
  home-address: Person -> tuple(street-no:   Integer;
                                street-name: String;
                                zipcode:     String)
  associate class Person with cf1(old, new):
       new.Person.pname <- old.Person.pname,
       new.Person.age <- (year(today)-year(old.Person.birthday)),
       new.Person.home-address.street-no <- old.Address.street#,
       new.Person.home-address.street-name <- old.Address.street-name,
       new.Person.home-address.zipcode <- old.Address.zipcode;
end Person
```

In DB-EVOLVE, both system-supplied default transformations and user-defined transformations are supported. The former is used mostly for converting instance objects among primitive types (such as String, Real, Integer, etc.). The latter is used when the extra information is required for specification of correct transformation or the complex transformations are involved (cf. [1, 7]).

3 The General Framework for Versions of Schema

3.1 Basic Terminology

We distinguish two types of versions of schema: the **released** schema versions which can only be deleted but not updatable, and the **transient** schema versions which can be updated at any time. In order to allow multiple users and applications to work concurrently under different versions of schema, it is important to support check-ins and check-outs of schema versions in an object-oriented database environment. If an application wants to extend or modify an existing schema version, it should first check the schema version out of the library of the released schema versions by either demoting the schema version into a transient one and then modifying it, or by deriving a schema version from it. Once the application generates a new schema version, it should check the transient schema version into the public library as a newly released schema version.

For any two schema versions SV_1 and SV_2, if SV_2 is derived directly from SV_1, we call SV_2 a **child** schema version and SV_1 a **parent** schema version. Similarly, we call all the versions (say SV_i) which are derived directly or indirectly from a schema version SV_k the **descendant** schema versions of SV_k, and SV_k is called the **ancestor** schema version of SV_i. The set of version derivation relationships forms a schema version derivation hierarchy.

Important to note is that, any new schema version may be derived by application of a sequence of schema update primitives. Therefore, it should follow both the schema evolution invariants presented in Section 2.2 and the number of invariants for versions of schema. The first schema version invariant below defines the baseline for schema version derivation.

1. Schema Version Derivation Invariant *Any number of new schema versions may be derived at any time from an existing schema version. A new schema version should be derived from a released schema version and is initially a transient schema version.*

In reality, we may allow a new schema version to be derived from a transient version. However, once a new version is derived from it, this transient schema version should be automatically promoted and checked into the library of the released schema versions.

In addition, for a schema version SV_i, we refer to the set of instance objects that are created under SV_i the **direct instance access scope** of SV_i, denoted by DIAS(SV_i), and refer to the set of objects that are accessible under SV_i as the **instance access scope** of SV_i, denoted by IAS(SV_i). Obviously, we have DIAS(SV_i)$\subseteq$IAS(SV_i). The access scope of SV_i is actually the set of objects which are either created under SV_i or inherited from the instance access scope of the ancestor schema versions of SV_i. Therefore, for any schema version SV_i, all objects in the instance access scope of SV_i are visible to SV_i. It means that they can be read or updated under SV_i. Nothing else is visible to SV_i. For example, if SV_i is the parent schema version of SV_j, and SV_j is the parent version of SV_k, SV_j inherits the access scope of SV_i, and SV_k inherits the access scope of SV_j, then the access scope of SV_k is the set of objects created under SV_i, SV_j, and SV_k (see Figure 1). We have $IAS(SV_k) = IAS(SV_i) \cup DIAS(SV_j) \cup DIAS(SV_k)$ for $0 \leq i < j < k$. The difference of $IAS(SV_k) - DIAS(SV_k)$ represents the *inherited* instance access scope of SV_k.

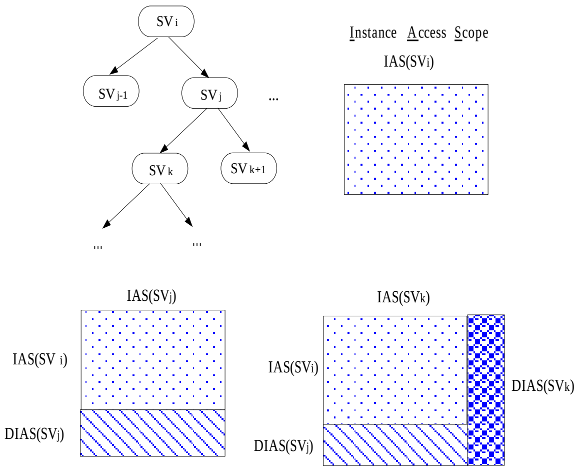


Figure 1: Schema version derivation hierarchy and the concept of instance access scope.

2. Schema Version Deletion Invariant *Once a schema version is derived, it may only be deleted when it has no child schema version, regardless of whether it is a released or transient schema version. When a schema version is deleted, its direct instance access scope is also deleted. But nothing will be deleted from its inherited instance access scope.*

In short, the schema version deletion invariant assures that a schema version may only "*own*" the objects created under it.

Interesting to observe is that, in practice, a creator of a schema version (say SV_i) may want to grant the other users to derive their own schema versions from SV_i, and meanwhile allow the descendant schema versions of SV_i to see only some collections of objects of SV_i, rather than the complete instance access scope of SV_i. To support this requirement, our schema version model allows the creator of a schema version SV_i to have the authority to declare whether the direct access scope of SV_i is sharable by its descendant schema versions, and which classes in the direct instance access scope of SV_i are sharable (or non-sharable) by its descendant schema versions. Hence, we further divide the direct instance access scope of a schema version into two disjoint subscopes: **non-sharable** and **sharable**.

3.2 Inheritance of Instance Access Scope among Versions of a Schema

In a multi-user design environment which supports versions of schema, there are mainly two ways to allow a child schema version (say SV_j) to share (inherit) the instance access scope of its parent schema version (say SV_i). We may either make a physical copy of the instance objects of SV_i into the direct instance access scope of SV_j, or allow automatic inheritance of the instance access scope of SV_i into SV_j. We take the latter choice as a basic premise of our model for versions of schema. The reason is simply because, in the case of using schema versioning to support evolution of schema, the set of objects in the instance access scope of the parent schema version, which need to be visible to the derived schema version, is relatively large. Thus, using the instance access scope inheritance will help to avoid unnecessary copying of those objects of SV_i which are visible to SV_j.

Moreover, it is desirable to provide a schema version with multi-levels of flexibility for inheritance of the (sharable) instance access scope from its parent schema versions, such that the schema designers or end-users may define their inheritance options at will, rather than being forced to inherit either everything or nothing as proposed in Orion [3]. More specifically, for any child schema version of SV_i (say SV_j), besides the two choices of non-inherited and all-inherited, the creator of SV_j may need to define more elaborated semantics for inheritance of the instance access scope of SV_i into SV_j. For example, the creator of SV_j may want to inherit only the snapshot of the access scope of SV_i but not the subsequent database updates under SV_i. We call this option **snapshort-shared**. Alternatively, the creator of SV_j may want to share only the access scope of SV_i and the subsequent object deletions and modifications, which means that all the subsequent object insertions under SV_i are not visible to SV_j. With these requirements in mind, we propose six basic inheritance options and one default option to allow the user to specify their particular inheritance semantics at will. These basic options include **non-inherited**, **all-inherited**, **snapshot-shared**, **insertion-shared**, **deletion-shared**, and **modification-shared**. Semantics of these inheritance options will be presented in the next subsection.

The default option is motivated by the observation that, in certain circumstance, it may be desirable to block the updatability to the database under a schema version SV_i, once a new schema version SV_j is derived and inherits the object instances from SV_i, especially when the creator of the two schema versions are the same user or from the same user group. As a result, the creator of SV_j may guarantee that the objects inherited from SV_i are viewed consistently under SV_j as long as they are not updated under SV_j. More importantly, it will help to restrict the effects of the subsequent database updates under SV_i, on the instance access scope of SV_j, and of the SV_j's descendant versions. To support for this requirement, we provide the creator of SV_j with an opportunity to disallow further database updates under SV_i, after a new schema version is derived and inherits the object instances from SV_i. We define this option as the default rule for the instance access scope inheritance. Certainly, in a system where complex access authorization scheme is applied, a consulation with the authorization model should be carried out before this default option becomes valid.

3. Instance Access Scope Inheritance Invariant *When a schema version SV_j is derived from a schema version SV_i, by default, SV_j inherits the instance access scope of SV_i, and blocks the direct access scope of SV_i to be non-updatable under SV_i. However, the user may optionally use the six basic inheritance options (non-inherited, all-inherited, snapshot-shared, insertion-shared, deletion-shared, and modification-shared) to override the default option at any time.*

With the default inheritance option, to carry out any update to objects of SV_i after SV_j has been derived from it, the creator of SV_i will have to derive a new schema version SV_k from SV_i, which has no difference from SV_i, and then update the objects under SV_k. This shows from another perspective that it is indeed desirable to allow applications or end-users to optionally define the intended semantics for inheritance of instance access scope.

In addition, a user may, on the one hand, define his/her intended inheritance rule by means of any combination of the given basic options, and, on the other hands, be able to dynamically change the inheritance option at will after the initial derivation of a schema version.

3.3 Semantics of Basic Inheritance Options

In contrast to the Orion's two levels of the access scope sharing mechanism, we argue for the need of multi-level sharing mechanisms to automate inheritance of the instance access scope of a parent schema version into the derived schema versions. The following six basic inheritance options have been developed as the baselines to address this issue.

Let SV_1 be a given schema version, SV_2 be a derived schema version from SV_1, and $IAS(SV_1)$ denote the instance access scope of SV_1. Obviously, we may assume that $IAS(SV_1)$ is not empty, because otherwise, it is more reasonable to demote SV_1 to a transient schema version and update it directly, rather than deriving a new schema version SV_2.

- Option 1: **non-inherited**
  With this inheritance option, nothing from the access scope of SV_1 is visible under the derived schema version SV_2. We have $IAS(SV_2) = DIAS(SV_2)$ and $IAS(SV_1) \cap IAS(SV_2) = \emptyset$.
- Option 2: **all-inherited**
  SV_2 inherits the instance access scope of SV_1, including all the subsequent database updates (insertion, deletions and modification) under SV_1. We have $IAS(SV_2) \supseteq IAS(SV_1)$.
- Option 3: **snapshot-shared**
  SV_2 only inherits the snapshot of the instance access scope of SV_1 at the time of schema version derivation. All the subsequent database updates (e.g., insertion, deletion, modification) under its parent schema version SV_1 are transparent to SV_2. We have $DIAS(SV_1) \not\subseteq IAS(SV_2)$, $IAS(SV_1) \cap IAS(SV_2) \neq \emptyset$, and $IAS(SV_2) \not\supseteq IAS(SV_1)$.
- Option 4: **insertion-shared**
  SV_2 inherits the instance access scope of SV_1 at the time of schema update as well as the subsequent insertions to the database of SV_1, which means that by only using this option, all the subsequent deletions and modifications to the database under SV_1 are transparent to SV_2.

- Option 5: **deletion-shared**
  SV_2 inherits the access scope of SV_1 at the time of schema update and the subsequent deletions to the database of SV_1. But with only this option, all the insertions and modifications to the database of SV_1 are transparent to SV_2.
- Option 6: **modification-shared**
  SV_2 inherits the access scope of SV_1 at the time of schema update and only the subsequent modifications to the database of SV_1 are visible under SV_2. Moreover, using this option alone means that all the subsequent insertions and deletions to the database under SV_i are transparent to SV_2.

Obviously, these inheritance options listed above are not mutually exclusive. For example, the option `snapshot-shared` is implied by all the other options except the `non-inherited` one. The option `all-inherited` can be equivalently be expressed by a combination of the options `snapshot-shared`, `deletion-shared`, and `modification-shared`. We describe the semantic relevance of these inheritance options in terms of their logical implications in Figure 2.

$\Longrightarrow$ (logical implication)	non-inherited	all-inherited	snapshot -shared	insertion -shared	deletion -shared	modification -shared	default option
non-inherited	Y						
all-inherited		Y	Y	Y	Y	Y	
snapshot -shared			Y				
insertion -shared			Y	Y			
deletion -shared			Y		Y		
modification -shared			Y			Y	
default option							Y

Figure 2: Logical implications among the inheritance options.

Besides, users may also use any meaningful combination of the given six options to define their need for the instance access scope inheritance. For example, assume the schema version SV_2 is derived from SV_1 through a schema modification. If we want to define the access scope of SV_2 by inheriting the instance access scope of SV_1 and allowing only the subsequent deletions and modifications to be visible under SV_2, we may associate the schema version SV_2 with the options **deletion-shared**, and **modification-shared**.

Questions remain to be addressed includes, for example, whether objects created under a schema version SV_j or inherited from its parent schema version SV_i can be updated under SV_j, what it means to update objects under SV_j, and how to manage the conversion of sharable object instances of SV_i to conform to

SV_j when they are inherited by SV_j. The following invariant addresses the first two questions. We will address the rest of the questions as well as the issues related to the implementation consideration of our general framework in Section 5, including the issues such as how an inherited object is accessed under a derived schema version, and what implementation strategy the system may use to implement the update of an inherited object under a derived schema version.

4. Instance Access Scope Update Invariant *All objects in the instance access scope of SV_j should be able to be updated or deleted under SV_j. However, any update or deletion of the inherited objects under SV_j is only visible to SV_j and to those descendent schema versions of SV_j which inherited the instance access scope from SV_j.*

This schema version invariant assures that any object in the access scope of a schema version SV_i may be updated under SV_i. However, when objects of SV_i are deleted or modified under a derived schema version (say SV_j) of SV_i, or a new object is inserted into SV_i, the effects of such database updates (insertion, deletion, or modification) can only be made visible to SV_j and to the descendant schema versions of SV_j. Furthermore, when viewed from any of the ancestor schema versions of SV_j, it looks as if the updates had never been taken place. Put differently, in the case that an inherited object is deleted under SV_j, this object will no longer be visible under SV_j and any descendant schema version of SV_j, which inherited the instance access scope of SV_j. However, this object will continue to be accessible under the creator schema version SV_i and any of the ancestor schema versions of SV_j which inherit the object directly or indirectly from SV_i. Similarly, when an object inherited from SV_i is modified under SV_j, the resulting object is *persistent* under SV_j, even after this object later is deleted under its creator schema version. Furthermore, this modified object is visible only to SV_j and to the descendant schema versions of SV_j which inherit the instance access scope of SV_j.

In short, every schema version generated by using DB-EVOLVE should satisfy both the schema evolution invariants and the schema versioning invariants.

3.4 Examples

Let us take a sample schema SV_1 given in Figure 3 as an example to illustrate the instance access scope inheritance invariant and the instance access scope update invariant.

Under the initial schema version SV_1, three object instances e_1, e_2, e_3 are created and they belong to the same class C_1 which has three properties p_1, p_2, and p_3. The schema version SV_2 is derived from SV_1, by adding a new class C_2 and a new property p_4 to the existing class C_1, with the option `snapshot-shared`. According to the semantics of Option 3 (snapshot-shared), all objects and their properties that are visible to SV_1 are now visible to SV_2, along with the new property p_4 in class C_1. The initial instance set of class C_2 is empty, and the default value of p_4 is set to *nil* until the two instances of C_2 and the values of p_4 are inserted (see Figure 3). Further, a new schema SV_3 is derived from SV_2, by deleting a property p_2 from the class C_1 in SV_2, with the inheritance option `insertion-shared`. All objects that are visible to SV_2

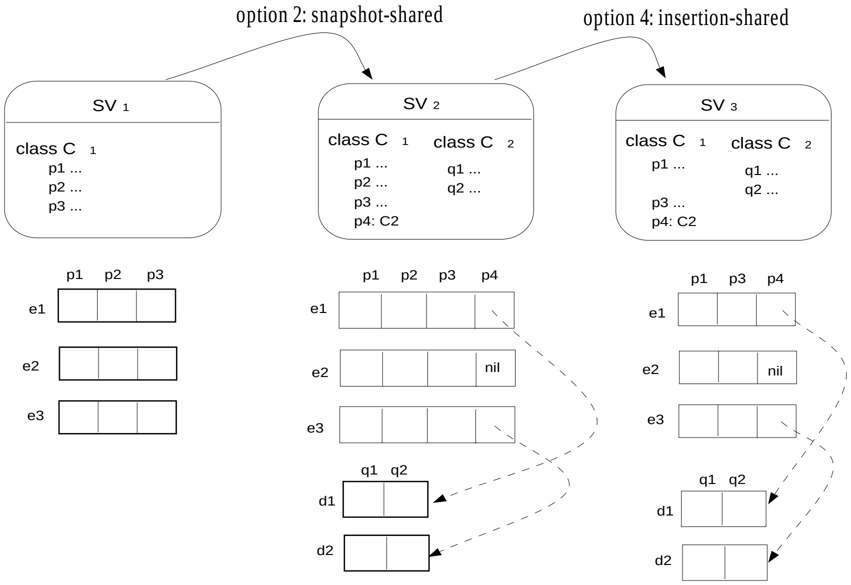


Figure 3: Illustration of the instance access scope inheritance invariant.

before the derivation of SV_3 are now visible to SV_3, without the deleted property p_2 (see Figure 3). So are the subsequent insertions to the database under SV_2. But the subsequent deletions and modifications to the database under SV_2.

Now assume a number of database updates will take place in the following sequence:

```
T1: insert new object e4 under SV2;
T2: modify an existing object e2 under SV2;
T3: insert a new object e5 under SV1;
T4: delete an existing object e2 under SV1.
```

In terms of the instance access scope update invariant, after the execution of transaction T1, SV_2 becomes the creator schema version of e_4. The insertion of e_4 under SV_2 is visible to SV_3, because SV_3 is derived from SV_2 with `insertion-shared`. However, the insertion of e_4 under SV_2 will not be noticed (or visible) to SV_1, the parent schema version of SV_2 (see Figure 4).

The successful completion of T2 under SV_2 updates the inherited object e_2 by replacing the *nil* value (see Figure 3) by a pointer to the object d_1 of class C_2 (Figure 4). This update has no effect on any of the ancestor schema versions of SV_2, which means that e_2 remains unchanged under SV_1. Since SV_3 is derived from SV_2 with the inheritance option **insertion-shared** only, the modification of e_2 under SV_2 is transparent to SV_3.

Now consider T3 and T4, by T3, a new object e_5 is inserted to SV_1, and by T4 an existing object e_2 is

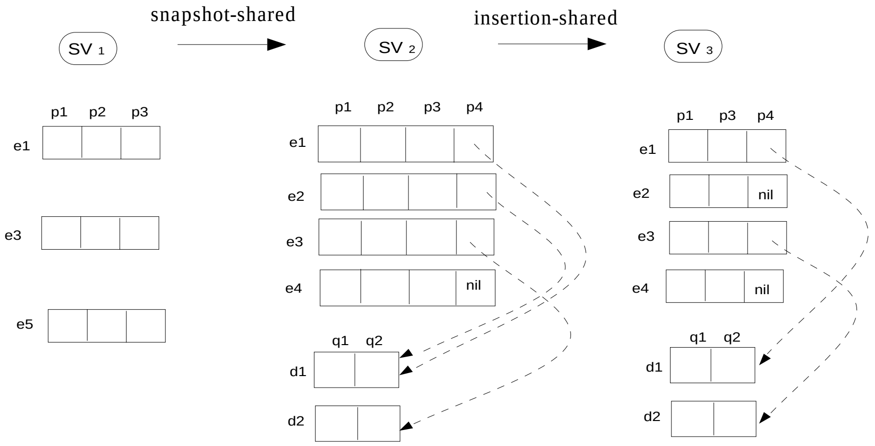


Figure 4: Illustration of the instance access scope update invariant.

deleted from SV_1. Since SV_2 is derived from SV_1 with the inheritance option `snapshot-shared` only, the insertion of e_5 to SV_1 has no effect on the instance access scope of SV_2, which means that the new object e_5 inserted under SV_1 is not visible to SV_2. Similarly, the deletion of object e_2 under SV_1 is transparent to SV_2. Therefore, after the successful execution of T4, e_2 no longer exists in the instance access scope of SV_1 but it is still visible to SV_2 and SV_3 (see Figure 4). We call SV_1 the *terminator* schema version of e_2.

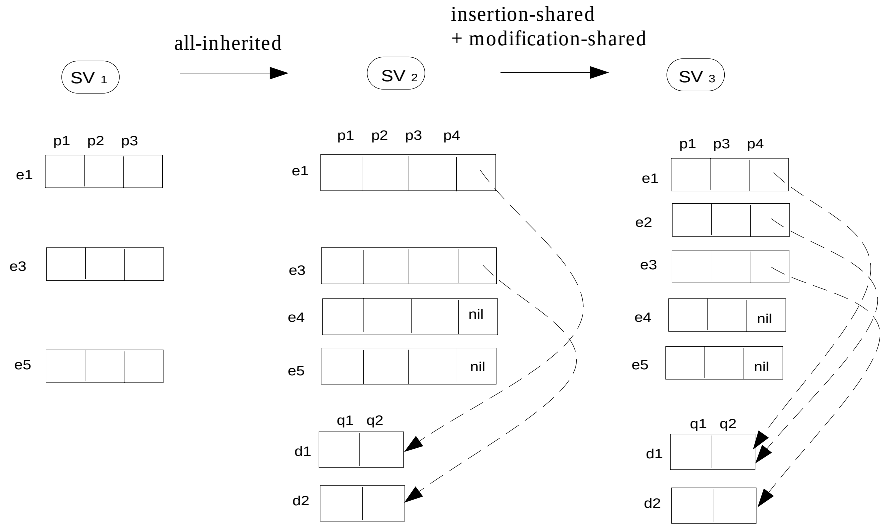
Interesting to note is that, if the inheritance options associated with SV_2 and with SV_3 are different, the effect of execution of the given sequence of database updates may possibly be different too. For example, if we assume that SV_2 is derived from SV_1 with the option `all-inherited`, and SV_3 is derived from SV_2 with the options `insertion-shared` + `modification-shared` as shown in Figure 5, then the executions of T1 and T2 result the same. However, the executions of T3 and T4 will have different effect on SV_2 and SV_3 (comparing Figure 4 with Figure 5). As a consequence, the database state in the example of Figure 4, after the completion of T1, T2, T3 and T4, is changed (see Figure 5), because although the insertion of e_4 and the update of the property p_4 of object e_2, from *nil* to pointing to d_1, under SV_2, are visible to SV_3, the deletion of e_2 under SV_1 will only have the effect on SV_2 but not on SV_3. Besides, the insertion of e_5 under SV_1 is also visible to both SV_2 and SV_3.

3.5 Possible Conflicts in Inheritance Options

There are two types of possible conflicts in the inheritance lattice for the instance access scope sharing among schema versions.

- *Conflict with respect to the database updatability*

  Conflicts may occur between the different schema versions (say SV_j and SV_k) that are derived from the same schema version (say SV_i), with respect to the updatability of the access scope of SV_i. We



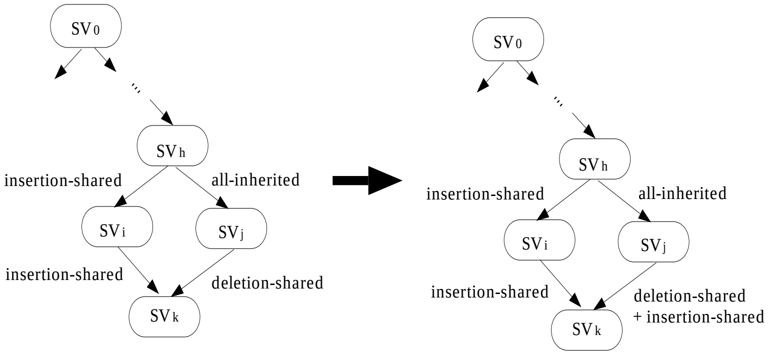
If the inheritance options associated with SV2 and with SV3 are different, for example, all-inherited is now associated with SV2 and deletion-shared + modification-shared with SV3, then the execution of transation T1, T2, T3, T4 will have different effect on the instance access scopes of SV1, SV2 and SV3.

Figure 5: Illustration of the instance access scope update invariant (cont.)

call such a conflict *update conflict*. For example, a schema version SV_j may have been derived from SV_i with the option `insertion-shared`, and a new schema version SV_k may now be derived by using the default inheritance option, which means to inherit the snapshot of SV_i and meanwhile make SV_i *non-updatable*. Now we have SV_i *updatable* in terms of the inheritance option associated with one of its child schema versions SV_j, but *non-updatable* by the inheritance option of the other child schema version SV_k. Conflict occurs. As this kind of conflict may only occur between the default inheritance setting and one of the basic inheritance options (excluding Option 1 and Option 3), we resolve such kind of conflict by allowing the use of the explicit inheritance option to override the default one.

- *Conflicts implied in the schema version derivation lattice*

  When we allow a new schema version SV_k to be derived from more than one existing schema versions (say from SV_i with `insertion-shared` and from SV_j with `deletion-shared`), if SV_i and SV_j have a common ancestor schema version SV_h, and from which they inherits the instance access scope of SV_h by `insertion-shared` and `all-inherited` respectively (see Figure 6(a)), then the *inheritance conflict* can be incurred between the inheritance option of SV_k associated with SV_i and the one associated with SV_j, whenever there is a subsequent database update under SV_h. For example, if a new object e_{20} needs to be inserted to the database of SV_h, in terms of the inheritance option `insertion-shared` associated with SV_i, and the `insertion-shared` with SV_k in connection with SV_i, e_{20} is visible to both SV_i and SV_k. Now consider the other schema derivation path to SV_k from SV_h via SV_j, according to the `all-inherited` option associated with SV_j from SV_h, the new object e_{20} is accessible under SV_j. However, it is not visible under SV_k, because SV_k is derived, and has



(a) Inheritance Conflict

(b) resolution of conflict

Figure 6: Conflicts in the instance access scope inheritance options.

inherited objects, from SV_j by using the inheritance option `deletion-shared`. Conflict occurs. We resolve such kind of inheritance conflicts by using the ordering of the parent schema versions of SV_k in the schema version derivation of SV_k. For instance, if SV_i is derived before SV_j, then we use the inheritance option of SV_k related to SV_i to override the one related to derivation of SV_k from SV_j, by the logical combination of the two specified options associated with SV_k (see Figure 6(b)).

4 User Interface

In this section, we define a set of interface commands which users may use to work with our DB-EVOLVE. In principle, the common steps for a system to implement the derivation of a new schema version SV_j from an existing schema version SV_i is to first get a copy of SV_i, and then make the updates on the copy by using a set of schema evolution primitives. These steps usually should be transparent to the users.

The first user command we introduce here is to derive a new schema version from an existing one, by (i)specifying the preferred options for inheritance of the instance access scope, (ii)by using `include` or `exclude` clause to define which classes of the existing schema version are sharable (or non-sharable) to the derived schema version, and (iii)by presenting what the resulting schema version should look like.

```
derive-schema-version <sch-version-name>
   from <list-of-parent-sch-versions>
   [by <list-of-options>];
   include <list-of-classes>;
   [exclude <list-of-classes>;]
   apply <list-of-sch-evolution-primitives>;
   [with-transformation-method <list-of-user-defined-transformations>];
```

```
   [non-sharable <list-of-class-or-property names>];
end-derivation.
```

Note that the `by` clause for inheritance selection is optional. By default, it means that the derived schema version will inherit the snapshot of the access scope from its parent schema version, and meanwhile block the updatability of the object base under its parent schema version.

Recall the example presented in Figure 3, the schema version SV_2 was derived from schema version SV_1 by simply adding a new class C_2 with two properties q_1 and q_2, and modifying the class C_1 with a new property p_4. To specify this example with our user interface language, the user may simply describe this application by using the schema version derivation command as follows:

```
derive-schema-version SV1
   from SV2
   by all-inherited;
   include all;
   apply
     add-class C2
       q1 ...,
       q2 ...;
     add-property-to C1
       p4: C2;
   with-transformation-method
     associate-with-class C1
       cf1(C1; SV2, SV1):
             SV2.C1.p1 <- SV1.C1.p1;
             SV2.C1.p2 <- SV1.C1.p2;
             SV2.C1.p3 <- SV1.C1.p3;
             SV2.C1.p4 <- nil;
end-derivation.
```

We also provide a number of user commands for additional services. For example, the command

```
delete-schema-version <sch-version-name>
```

is used to delete an existing schema version. Note that, by the schema-version deletion invariant, the schema version to be deleted should have no child schema version. User may also use the command

```
promote-schema-version <sch-version-name>.
```

to promote the status of a schema version to the released mode, if the given schema version is a transient one. Otherwise, no action is taken. This command returns a truth value (`true` or `false`).

When an application wants to change the working schema version to a particular one or to update a particular schema version rather than the current one, the following command can be used:

```
set-current-schema-version <sch-version-name>.
```

This command returns the current schema version identifier.

5 Implementation Issues

5.1 Data Structures for Objects and Object Manipulation

To support object manipulation in the presence of versions of schema, we need to associate with every object three additional system-defined properties:

- a system-defined instance variable, indicating the `creator schema version` of the object.
- a list of `terminator schema versions` under which the object was deleted.
- a data structure, describing a set of copies of the object, each of which is created under a specific schema version. We call it `instance-copy-list`. Conceptually, it is very similar to the concept of *generic instance* of a versioned object introduced in Orion [3] for describing the set of version instances of the object. In the sequel, we sometimes also use generic instance to refer to the instance-copy-list of the object.

Figure 7 presents the sample data structure for each instance object. Note that an object, when first created, exist without a generic instance. In the other words, the instance-copy-list is empty. However, every object will carry the identifier of its creator schema version once it has been created. A generic instance and each of the copies of the object, which the generic instance describes, all share, and are "identified" by, the same identifier of the object in order not to invalidate the existing references to the object. Copies of the object in its generic instance are distinguished from each other in terms of their creator schema version numbers. Whenever the structure of an object is extended (information-augmented), a new copy of the object will be created and added into the instance-copy-list of the object. However, when the structure of the object is information-reduced, no new copy of the object will be created. The visibility (accessibility) of an object may vary under different versions of a schema.

For example, consider the application in Figure 4, class C_1 in SV_1 is augmented in SV_2. Thus, for each object of C_1 in the access scope of SV_1, a copy is created and added into the `instance-copy-list` of the

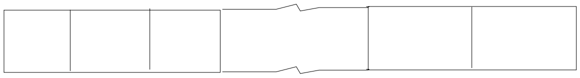


Figure 7: The data structure for instance objects

object (see e_1, e_2 in Figure 8). However, the schema version SV_3 is derived from SV_2 by deleting p_2 from C_1. Thus, SV_3 is information-reduced in comparison with SV_2. We may consider SV_3 as a view schema of SV_2. No copy of the object is created. Any access request to the objects under SV_3 will be processed simply as a view query. The data structure for object e_1, e_2, e_5 and q_1 can be represented as shown in Figure 8.

Obviously, the DB-EVOLVE schema version model is more general comparing with the schema version model of Orion [3] where either everything or nothing is shared. Recall the schema version derivation given in Figure 4, by using DB-EVOLVE, only if the schema version SV_2 is derived from SV_1 with the inheritance option `deletion-shared` (or `modification-shared`), a delete (or modify) of an object under its creator schema version SV_1 will physically delete (or modify) the object. However, if SV_2 is derived from SV_1 with inheritance option such as `snapshot-shared`, `insertion-shared`, or `modification-shared`, then, even when an object is deleted under its creator schema version SV_1 (e.g., deleting e_2 under SV_1 in Figure 4), the object will still physically exist in the database. A copy of e_2 will remain accessible under SV_2 and the corresponding descendant schema versions of SV_2.

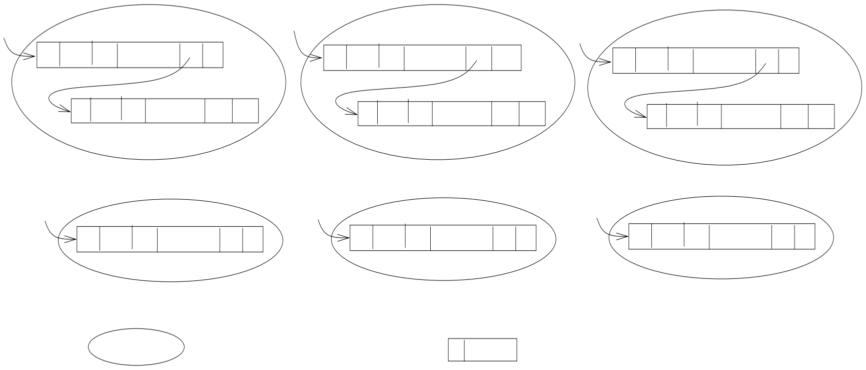


Figure 8: An example data structure for instance objects based on the schema versions in Figure 4.

To support modification of an inherited object under a schema version SV_i, the system will create a new copy of the object under SV_i. To support delete of an object under SV_i, if SV_i is not the creator schema version of the object, then the object is an inherited one under SV_i, and the system will simply add SV_i into the terminator list of the object. However, when SV_i is the creator schema version of the object, the system will check the list of child schema versions of SV_i. If all child schema versions are derived from SV_i with the `deletion-shared` option, then delete of the object under SV_i will physically delete the object.

Otherwise, SV_i will simply be added into the list of terminators of the object. Of course, update of an object under a schema version SV_i is validated only if SV_i is not frozen by the derivation of any of its descendant schema versions (i.e., if the default inheritance option is not valid).

5.2 Algorithms for Object Manipulation

Based on the DB-EVOLVE data structure for instance objects and the concept of generic instance, we below outline the algorithms for object fetch, insert, delete, and modification. Readers may skip this section without loss of continuity.

In design of the algorithms, we use four system-supplied boolean functions: (1)`AncestorSV-of(sv1, sv2)`, which returns `true` if and only if the first argument `sv1` is an ancestor schema version whose access scope is inherited by the the schema version specified in the second argument `sv2`. Otherwise, it returns `false`. (2)`update-blocked(sv)`, which returns `true` when the argument `sv` specifies a schema version whose direct instance access scope is non-updatable under `sv`. (3)`deletion-shared*(sv1, sv2)`, which returns `true` when all the schema versions along the derivation path from sv1 to sv2 are derived with deletion-shared inheritance option. (4)`modification-shared*(sv1, sv2)`, which returns `true` when all the schema versions along the derivation path from `sv1` to `sv2` are derived with modification-shared inheritance option. For presentation convenience, we define `AncestorSV-of*(sv1, sv2)` = `AncestorSV-of(sv1, sv2) .OR. sv1 = sv2`.

The algorithm for fetching an object identified by `obj-id` under `sv` is designed in three steps:

- First, we locate the object in terms of `obj-id`.
- Then we check if the object is the generic instance. If yes, we search for the closest creator schema version of `sv` in the instance-copy-list of this object. If there exists one (say `o.creatorSV`, and this object have never been terminated by any schema version in the derivation path from `o.creatorSV` to `sv`, the algorithm ends by returning the object found.
- If no generic instance exists for the object to be fetched (say `f-object`), and the creator schema version of `f-object` is `sv` or the ancestor schema version of `sv`, then the algorithm returns the object.

```
Algorithm SV_Obj_Fetch(obj-id, sv)
  f-object <- locate_object(obj-id);
  if (f-object.instance-copy-list <> nil)
            /* f-object is a generic instance  */
   do
     g <- f-object;
     f-object <- Find_closest_creator_copy(g.instance-copy-list, sv)
```

```
       if (f-object = nil) .OR.
          No_copy_visible(g.list-of-terminators, sv, f-object.creatorSV)
          /* no copy is visible under sv */
        error;
     end-do;
    else
     if (AncestorSV-of*(f-object.creatorSV, sv) = false)
        /* the only existing copy is not visible */
      error;
    return f-object;
  end SV_Obj_Fetch;
```

The following are the two subroutines that have been used in the algorithm for object fetching. They will also be used in the algorithm for object deletion object update.

```
  Find_closest_creator_copy(o-copy-list, sv)
        /* this is a routine for finding the closest creatorSV copy to sv */
    for each cp in o-copy-list
      if AncestorSV-of*(cp.creatorSV, sv)
         .AND. (cp.instance-copy-list = nil)
                /* a closest creatorSV copy is found  */
        return cp(sv);    /* return the sv view of the object copy cp */
    endfor;
    return nil;    /* no available copy is found */
  end Find_closest_creator_copy;


  No_copy_visible(list-of-terminators, sv, sv')
          /* routine for checking if there is a copy visible under sv  */
  if (sv <> sv')
    for each t-sv in list-of-terminators
      if AncestorSV-of(t-sv, sv) .AND. AncestorSV-of(sv', t-sv)
         .AND. deletion-shared*(t-sv, sv)
      return true;      /* no copy is visible under sv */
    endfor;
  else
    if (sv is in list-of-terminators)
      return true;   /* the only existing copy is terminated under sv */
    else
      return false;     /* there exists a copy visible to sv */
  end No_copy_visible;
```

According to the algorithm `SV-Obj-Fetch()` for object fetching, `SV-Obj-Fetch`(e_1, SV_2) returns the copy of e_1 with SV_2 as the creator (see Figure 4 and Figure 8). However, Fetching object e_2 or e_4 from SV_1 both return error, because e_2 is terminated by SV_1, and e_4 can never visible to SV_1 since it is created initially by SV_2, a descendant schema version of SV_1. Also `SV-Obj-Fetch(`e_5, SV_2`)` returns error, because SV_2 is derived with **snapshot-shared** option before e_5 is inserted under SV_1.

Below, we provide the algorithm for inserting a new object into the access scope of a given schema version `sv`. An object can only be inserted into a schema version `sv`, if its direct instance scope is updatable under `sv`. Once an object is inserted, the system automatically generates an object identifier for it. A successful execution of the algorithm `SV-Obj-Insert()` returns an object identifier for each inserted object.

```
Algorithm SV_Obj_Insert}(i-object, sv)
    if update-blocked(sv)
      error;
    i-object.creatorSV <- sv;
    obj-id <- assign-oid(i-object);
    return obj-id;
end SV_Obj_Insert;
```

Similarly, the algorithms for deleting and modifying objects under a given schema version `sv` are described below. An object can only be deleted or modified under `sv`, if the updatability of `sv` is not blocked by any descendant schema versions of `sv`.

```
Algorithm SV_Obj_Delete(obj-id, sv)
  if update-blocked(sv) error;
  d-obj <- locate_object(obj-id);
  if (d-obj.instance-copy-list <> nil)
     do     /** d-obj is a generic instance  **/
        g <- d-obj;
        d-obj <- find_closest_creator_copy(g.instance-copy-list, sv)
        if (d-obj = nil) .OR.
           no_copy_visible(g.list-of-terminators, sv, d-obj.creatorSV)
           error;      /* no copy is visible under sv */
        else   /** the object copy to be deleted is found  **/
            if (d-obj.creator = sv)
               do   /** delete by its creator **/
                   if (there exists no such sv' that AncestorSV-of(sv,sv')
                       .AND. not (deletion-shared*(sv, sv')
                      do
                         remove d-obj from d-obj.instance-copy-list;
                         if (no copy exists in d-obj.instance-copy-list)
                            d-obj.instance-copy-list <- nil;
                      end-do;
                   else  add sv to d-obj.terminators-list;
               end-do;
            else   /** delete by non-creator  **/
```

```
                add sv to d-obj.terminators-list;
     end-do;
  else      /** there is no generic instance of d-obj  **/
      if (d-obj.creator = sv)
         do
           if (there exists no such sv' that AncestorSV-of(sv,sv')
              .AND. not (deletion-shared*(sv, sv')
              remove d-obj;
           else
              add sv to d-obj.terminators-list;
         end-do;
      else  if (ancestorSV-of(d-obj.creatorSV, sv)
               add sv to d-obj.terminators-list;
            else  /* the only existing copy of d-obj is not visible under sv */
               error;
end SV_Obj_Delete;


Algorithm SV_Obj_Modify}(obj-id, new-obj, sv)
  if update-blocked(sv) error;
  m-obj <- locate_object(obj-id);
  if (m-obj.instance-copy-list <> nil)
     do  /** m-obj is a generic instance  **/
        g <- m-obj;
        m-obj <- find_closest_creator_copy(g.instance-copy-list, sv)
        if (m-obj = nil) .OR.
          no_copy_visible(g.list-of-terminators, sv, m-obj.creatorSV)
          error;     /* no copy is visible under sv */
        else  /** the object copy to be modified is found **/
           if (m-obj.creator = sv)
               /** modified by its creator schema version **/
              do
                if (there exists no such sv' that AncestorSV-of(sv,sv')
                    .AND. not (modification-shared*(sv, sv')
                   m-obj.data <- new-obj.data;
                      /* using new-obj data to replace m-obj data */
              end-do;
           else do      /** modified by non-creator schema version **/
                  new-obj.creator <- sv;
                  add new-obj to m-obj.instance-copy-list;
                end-do;
     end-do;
  else  /** there is no generic instance **/
      if (m-obj.creator = sv)
             /** modified by its creator schema version **/
```

```
            do
               if (there exists no such sv' that AncestorSV-of(sv,sv')
                  .AND. not (modification-shared*(sv, sv')
                  m-obj.data <- new-obj.data;
               else
                  if (ancestorSV-of(m-obj.creatorSV, sv)
                     do  new-obj.creator <- sv;
                         create generic instance for m-obj by
                            adding new-obj to m-obj.instance-copy-list;
                     end-do;
                  else error;
       end SV_Obj_Modify;
```

We have so far discussed the storage representation for instance objects and the algorithms for object manipulation. These algorithms are also used to retrieve and update of class objects. In what follows, we will present the storage structure for representing versions of schema, class objects, and the schema version derivation hierarchy respectively.

5.3 Storage Representation for Versions of Schema

In DB-EVOLVE, we represent a schema as a set of meta-class objects. The typical meta-classes are `Class`, `InstanceVariables`, and `Methods`. They are actually analogous to system catalogues in conventional database management systems (see Figure 9). For each user-defined class, the meta-class `Class` contains an instance object describing the class name, list of superclasses, list of subclasses, as well as instance variables and methods. The property `superclasses` and `subclasses` describe sets of superclasses and subclasses respectively. The property `instance-variables` (or `methods`) presents the set of all instance variables (or methods) defined for, or inherited into, the class, and has the meta-class `InstanceVariables` (or `Methods`) as ite domain class/type. For every instance variable (or method) defined for or inherited into each class, the meta-class `InstanceVariables` has a corresponding instance object, which is described by properties such as class name, creator schema version, domain type, inherited-from indicating the superclasses from which it is inherited, and plus the list of terminator schema versions, pointer to the next instance variable (or method), and a list of copies (see Figure 9).

When a schema is quite complex, and large in size, one full copy of the schema can require significant storage space. Besides, if the new schema version has a large number of components that are in common with the previous schema version, maintaining the duplicate parts of the schema can become rather costly too. Our objective, therefore, is not to maintain a physically separate copy of the entire schema for each version of the schema. Instead, when a change to the schema occurs, we continue to maintain instances of the meta-classes `Class, InstanceVariables, Methods` as non-versioned objects, but use the generic instance structure to support the updates to a class object under different schema versions. The reason

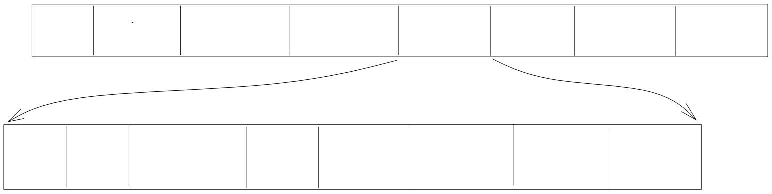


Figure 9: The storage representation for versions of schema.

is simply because of the fact that all changes to a schema are either to the definition of a class or to the relationships between classes, and that in object-oriented data models, relationships between classes are encoded in the class objects. Therefore, when the definition of a class is changed by adding or deleting a superclass/subclass link, or when a non-leaf class is deleted from the *is-a* class lattice, a copy of the class object will be created for each of the classes involved or affected in such update. However, we need no change to the basic representation of the schema. Similarly, when the definition of a class is changed by updating instance variables (or methods), we need no change to the original definition of the instance variables (or methods), but create a copy for each of the instance variables (or methods) updated.

We illustrate our storage representation using the example in Figure 10, where the class lattice is constructed and modified under five schema versions SV_i ($i = 1, 2, ..., 5$). The schema version derivation hierarchy is shown in Figure 10(a). SV_1 has only one class C_1 which has three properties p_1, p_2, p_3. SV_2 is derived from SV_1 by adding a new class C_2 as a subclass of C_1 and modifying the domain type of p_3. Similarly, SV_3 is derived from SV_1 by modifying classes C_1, C_2. SV_4 is derived from SV_2 by modifying classes C_1, C_2 and adding a new class C_3 as a subclass of C_1. Finally, SV_5 is derived from SV_2 and SV_3 by deleting class C_2 and modifying class C_1. (see Figure 10(b)).

Consider the evolution history of class C_1. First, C_1 is created under SV_1 with three properties. Then, C_1 is evolved along two directions: (1)C_1 is changed under SV_2, by modifying the domain type of p_3. When C_1 is modified in SV_2, we continue to maintain the class object C_1 in the meta-class `Class`, and create a copy in the generic instance of C_1 (see Figure 10(c)). (2)C_1 is modified under SV_3 by deleting p_2 and adding p_4. This makes p_2 inaccessible under SV_3. Further, C_1 in SV_2 is updated under SV_4, and C_1 in SV_3 is modified under SV_5. Thus, the generic instance of C_1 includes copies of C_1, each of which is visible in one of these schema versions (see Figure 10(c)).

Similarly, when the property p_1, created under SV_1, is modified in SV_3 by changing the domain type of `Integer` to `String`, we simply create a copy in the generic instance of p_1. Figure 10(c) shows the evolution history of the properties p_1, p_2, p_3, p_4 and p_5.

Interesting to note is that, when a schema version is derived, a user can choose not to inherit any instance objects from their parent (ancestor) schema versions. However, at meta data level, meta-class objects are always inherited by the new schema version at the time of schema evolution.

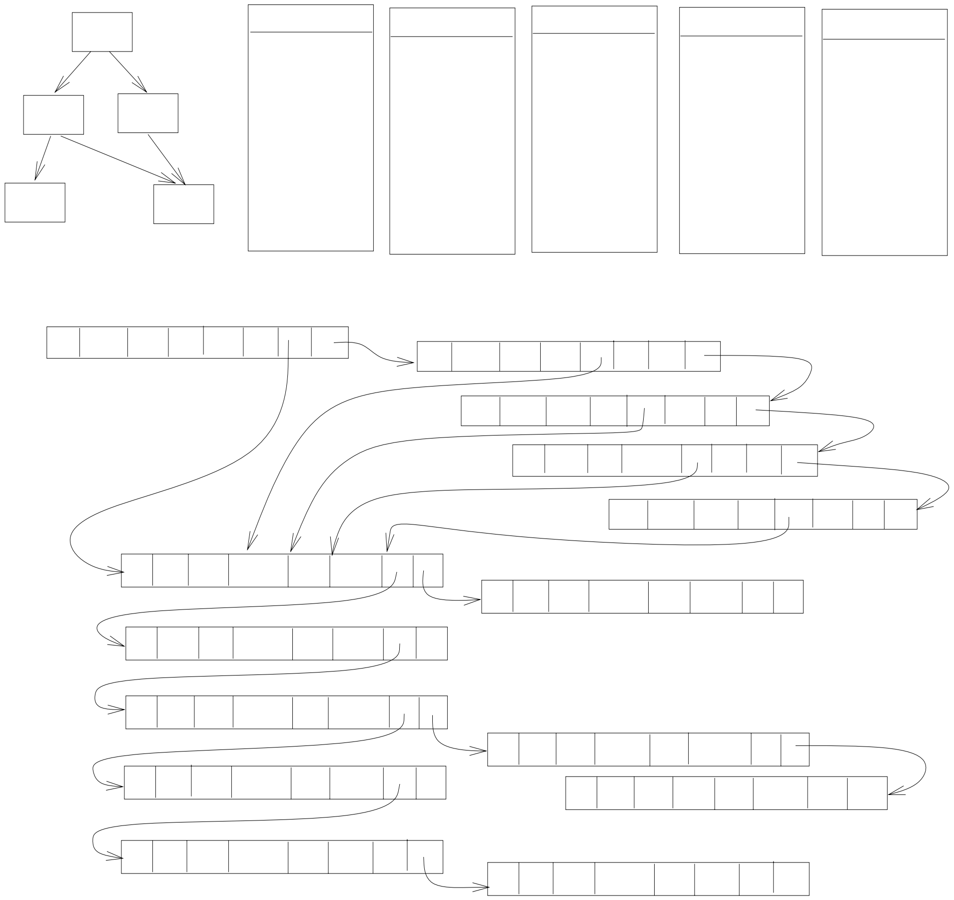


Figure 10: An example for the storage representation of versions of schema.

5.4 Storage Representation for the Schema Version Derivation Hierarchy

In DB-EVOLVE, we take a simple and reflective way to represent the schema version derivation hierarchy, which is to create a system-defined meta-class `Schema` for schema versions. For each given schema, this meta-class has only one instance object. Each schema version is included as a version instance in the generic instance of this versioned object. Thus, the schema version derivation hierarchy can be maintained in the generic instance of this versioned object.

6 Related Research

Schema evolution is a commonly required facility in most persistent object-oriented systems. Generally speaking, a schema describes the interface between a set of application programs and the persistent repository of objects. When a schema changes, so does the interface, which possibly incurs incompatible elements on both sides. Therefore, in an environment where the database schema is expected to evolve, in order to account for additional specifications imposed by new applications, the users face two alternatives: (1)to update his/her current application programs and migrate the exist data resources to match the new schema, or (2)to adopt an automatic transformation mechanism which achieves the compatibility of data instances between versions of the schema. Obviously, if the schema evolves frequently, the first alternative will be very expensive and impractical, and one would prefer the second.

In the existing literature, class modification [4, 9, 11], class versioning [10, 8], and schema versioning [5, 6] are the most common approaches that have been considered to support schema evolution in several available database management systems. However, implementation of these approaches has limitations either in the supported schema evolution operations or in the mechanisms for instance adaptation and program compatibility. For example, the class modification in GemStone [9] only provides mechanisms for maintaining the consistency invariants of the schema after a class modification. No consideration is given on the issue of instance adaptation to maintain the database consistency and the issue of program compatibility to allow existing application programs remain operational. The schema evolution approach proposed in ENCORE [10] restricts the breath of class evolution in order to implement emulation via user-defined exception handling routines. Several schema changes cannot be adequately supported under this scheme, because of the difficulty, if not an impossibility, to define the exception handling routines. The basic model of schema evolution invariants [3, 2] is first proposed in Orion and has been used widely in many operational database systems. The Orion's approach to versions of schema [6] presents a comprehensive study on the semantics and implementation of schema versions. However, Orion's proposal addresses the modeling and the implementation issue of schema versioning mainly at the schema level. No discussion was given on how the model of schema versioning may enhance the functionality of schema evolution, and, in particular, may facilitate the conformance of database instances to the new schema version after schema updates. Although OTGen [7] presents a set of facilities for automatic transformation of instance objects from a class to its updated version, it can only provide partial compatibility of data between a class and its updated version.

7 Concluding Remarks

The DB-EVOLVE development is mainly motivated by the critical requirements for managing schema evolution in an evolving multidatabase computing environment, because, in such an environment, both local and global schemas are expected to evolve, thus, the ability to minimizing the impact of schema changes on the existing database organization and the compatibility of existing application programs becomes critical in supporting up-to-date global information gathering, while preserving the autonomy of local databases.

In this paper we have presented a general framework based on versions of schema, called DB-EVOLVE, for maintaining database consistency in the presence of schema updates. It provides users with powerful facilities for obtaining seamless schema evolution through the support of different levels of object sharing among versions of schema. The salient features of our approach are the following. First, we demonstrate that the effectiveness of using the schema version model to materialize schema evolution relies not only on the management of version derivation of schema, but also on the ability to maintain the consistency between database instances and the new schema versions, as well as the flexibility to share instance access scopes among versions of schema. Second, in our general framework for versions of schema, semantics of schema versioning has been studied with the objectives to facilitate schema evolution and instance propagation due to schema updates. A selection of options for sharing instance access scope among versions of schema is developed. They offer various levels of flexibility for schema designers, application developers, and end-users to manipulate and maintain available database resources in the progress of schema evolution. Thirdly, using our approach, users may derive versions of schema upon a request for schema update, and define the instance access scope for each new version of schema, either by creating their own instance scope or by inheriting the instance access scope of its parent version(s) in terms of multiple inheritance options. Most importantly, our approach allows to preserve the history of evolution of objects in the progress of schema evolution, and is able to automate the conformance of database instances to the new version of schema. As a result, users may update the schema and maintain the database consistency without having to irreversibly change the objects that existed before a schema modification. With our approach, the amount of effort and cost required for database reorganization, and for reprogramming of existing application programs, due to schema changes, can either be avoided or substantially reduced. The impact of schema changes, made by one user or in a single application, over the entire system and the existing customer set can also be minimized. Further, users may derive versions of schema upon requests for schema evolution, and create and manipulate different collections of objects under different versions of schema.

Much work appears promising along with this line of research. For instance, theoretically, we are investigating the possible development of a formal and reflective model for versions of schema and of objects. We are currently also working on developing a collection of rules for triggering the default or user-defined transformation methods to conform the objects of a schema version to the newly derived schema version. On the practical side, we are currently implementing the basic model for versions of schema and the set of inheritance options for sharing instance access scopes among versions of a schema using O_2 database management system. We also plan to build a prototype of the DB-EVOLVE on top of the ObjectStore.

Acknowledgement

The work reported here was initiated when I was working at University of Frankfurt. I am indebted to Roberto Zicari for the discussion and the encouragement and to Sevn E. Lautemann for his comments on an earlier version of this paper.

References

[1] *Objectstore User Guide, Chapter 9.* Object Design Inc., 1993.

[2] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3 – 26, January, 1987.

[3] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented data bases, San Francisco, California. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, May 1987.

[4] E. Bertino. A view mechnism for object-oriented databases. In *International Conference on Extending Data Base Technology*, Vienna, Austria, 1992.

[5] S. M. Calmen. Schema evoluton and integration. *Journal of Distributed and Parallel Databases*, 2(2), 1994.

[6] W. Kim and H.-T. Chou. Versions of Schema for Object-oriented Databases. In *International Conference on Very Large Data Bases*, pages 148–159, 1988.

[7] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, 25(10):67–76, October 1990.

[8] S. Monk and I. Sommerville. Schema evoluton in oodbs using class versionning. *ACM SIGMOD RECORD on Management of Data*, 22(3), 1993.

[9] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 111–125, Orlando, Florida, 1987. ACM Press.

[10] A. Skarra and S. Zdonik. Type evolution in an object-oriented data base. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–416. The MIT Press, 1987.

[11] R. Zicari. A framework of schema updates in an object-oriented database system. In *Building an object-oriented database system: The story of O2 (F.Bancilhon, C. Delobel, P. Kanellakis, editors), Morgan Kaufmann, (the extended version of [Zic91])*, 1992.