Dealing with Version Pertinence to Design an Efficient Schema Evolution Framework

Boualem BENATALLAH
James Cook University
Department of Computer Science
GPO Box 6811, Q 4870 Australia
boualem@cs.jcu.edu.au

Zahir TARI
Royal Melbourne Institute of Technology
Department of Computer Science
Plenty Rd., VIC 3083 Australia
zahirt@cs.rmit.edu.au

Abstract

This paper addresses the design of a schema evolution framework enabling an efficient management of object versions. This framework is based on the adaptation and extension of two main schema evolution approaches, that is the approaches based on schema modification and those based on schema versioning. The framework provides an integrated environment to support different levels of adaptation (such as, modification and versioning at the schema level, conversion, object versioning, and emulation at the instance level). In addition, we introduce the concept of class/schema version pertinence enabling the database administrator to judge the pertinence of versions with regard the application programs. Finally, we provide operations for immediate refreshing of a database to enable an efficient manipulation of versions by a large number of application programs.

1 Introduction

Motivation and Background

During a database life span, the schema is likely to undergo significant changes in functional requirements (e.g., application domain change) and/or non functional requirements (e.g., performance) [10]. A schema can be updated by adding, deleting, or updating its constituents, such as attribute, method, class or inheritance relationship [12]. When a schema has been changed, objects may be inconsistent and programs may be incompatible. The support of schema evolution is the ability to allow changes of the structure and the behavior of schema's constituents without affecting application programs that are using them. This type of consistency introduces two aspects: *efficiency* and *durability*. The former relates to the fact that the mechanisms introduced, to apply the changes to schema and propagate the effects on the schema and objects, should operate efficiently in the sense that performance degradation is to be avoided. The latter issue relates to the fact that these mechanisms must ensure some degree of durability for stored information, i.e., avoid the loss of information. To the best of our knowledge, these issues are never considered together. The aim of the research presented in this paper, is to provide a solution that takes into account data durability and efficiency when addressing schema update.

A good survey on schema update approaches is presented in [2]. Here, we overview two major approaches that are relevant w.r.t the issues addressed in this paper, namely *modification-based* and *versioning-based* approaches.

- In a *modification-based* approach [4, 6, 8, 9]. the new schema definition replaces the old one. The *conversion* technique is used for object adaptation [4]. All existing objects must be converted to objects fitting class definitions in the new schema. The main drawback of this approach, is the incompatibility of old programs w.r.t the new schema when conversion occurs.

- In a *versioning-based* approach [11, 3, 5, 7, 12], a schema update induces the derivation of new versions of classes or a schema. Old programs can continue to interact with (old or new) objects in the database using the old schema, which has been retained. In this case, two techniques of object adaptation are provided in the literature: *emulation* and *object versioning*.

  1. In the first technique [7, 12], an object is associated to the version of a class under which it is created. The representation of an object is never restructured after its creation. The system is responsible for assuring access to old and new objects by programs. Since

properties are never deleted, a program will always find a reference to the property it requires, even if the property does not exist for objects associated to some versions of the class. This technique supports program compatibility w.r.t the schema. However, it has two major limitations. The first limitation relates to the cost of accessing the objects. The second limitation relates to the loss of information. The value of an added (resp., deleted) attribute in an old (resp., new) object is always a default value (i.e., supplied by the system), because it is impossible to restructure physically the value of an object after its creation.

2. In the second technique, the derivation of a new version of a class, leads to the creation of a new version of each object of the class that has been versioned [11, 3, 5]. This technique resolves the problem of information loss, by supporting many object versions. However, its problem is the proliferation of the number of versions, due to the dynamic nature of object oriented applications. This will result in considerable overhead on the system, because the storage requirement for versions and the cost of maintaining versions relationships, increase with versions number.

This paper addresses the schema update problem by integrating *modification* and *versioning* support functionality, into one single environment. Our objective is to limit the versions number to those which are necessary, so as to reduce versions management costs [1]. The *schema update process* combines schema *modification* and *versioning*. The technique followed for supporting *object adaptation* after a schema update, combines object *conversion*, *versioning* and *emulation*. cases where one solution is more interesting than another. We have also implemented the underlying mechanism related to our schema update approach on the top of O_2 database system [8]. This prototype is called *DBEvolution*.

Our Approach

For the update of schemata, we used the existing taxonomy for schema evolution operations. This paper does not discuss these operations, however details can be found in [1, 4, 6, 9, 12]. Using these operations, a user may operate changes on a schema. Updates are to be accepted when the schema remains in a consistent state. Our approach with regard to the consistency issue can be summarized as follows:

1. When an update is accepted, this is effectively translated either to the modification of the schema or the generation of a new version of the schema. In the absence of the user directive, each *substractive* update (i.e, information deletion) will result in the generation of new version and each *non-substractive* update will result in a modification.

2. Regarding the issue of object adaptation, our approach is based on the *evaluation of the pertinence of an object version availability.* When an object `o` is accessed by either a program or a query under a version `v` of the schema, the system DBEvolution checks if the object `o` has a version under `v`, that is the extension of the class of `o` defined in `v` does not contain a version of `o`. If not, a new version of `o` is generated. This new version is physically *stored* only if its availability is important (e.g., for performance improvement). Otherwise, this version is *calculated*, that is the version available only during the period of use. The question is whether or not to store this new version for future use. To decide if an object version is to be stored or calculated, we have introduced the notion of *class pertinence levels.* A class may be *pertinent* or *obsolete.* It is pertinent if it is associated to the most recent version of the schema or it is used by a larger number of users which, is judged sufficiently high by the database administrator. Otherwise, it is obsolete. Thus, the version of the object `o`, under `v`, is stored only if the class of `o`, which is defined in `v`, is pertinent.

3. Our approach also deals with the problem related to the size of databases, when these record versions. After many updates of a schema, some versions may became "useless", meaning that they are and will never be used by any application program. The presence of a large number of useless versions will result in storage overhead costs and a degradation of the system performance due the maintenance of relationships between versions. Our approach proposes a set of criteria allowing the database administrator to judge the pertinence (or importance) of versions, with regard to the application programs, as well as an operation performing an *immediate refreshing* of the database, by deleting historical versions when these are judged not pertinent.

The paper is organized as follows. Section 2 presents the model of versioning. In Section 3 we address the issue of instance adaptation. Section 4 describes the solution for the problem relating to the proliferation of versions number. Implementation issues are discussed in Section 5. We then make some concluding remarks in Section 6.

2 Versioning Model

This model supports the following concepts: *schema version*, *object version* and *version binding*. This section overviews these concepts and more details can be found in [1].

An *entity* refers to either a schema or an object of a database. At the system level, an entity is stored as a couple `(eid,vers)`, where `eid` is the entity identifier and `vers` is the set of identifiers of its versions. The identifier of a version is also a couple `(eid,num)`, where `eid` is the identifier of the entity and `num` is the number that the system associates to the version, at the time of its creation. The function `VersSet(eid)`, returns the set of versions associated to the entity `eid`.

2.1 Schema - Schema Version

A schema is associated to a set of versions which is organized in a sequence. The first version is the root version. We designate the latest version of a schema as the *current* version. A *historical* version of a schema is any version of the schema which is not the current version. In the sequel, we use "current" or "historical" to denote what we call the *status* of a schema version. A schema version is defined as a triplet `(sid,num,val)`, where `(sid,num)` represents the version identifier and `val` is its value. The value of a schema version is a set of persistent classes related by reference and inheritance relationships.

We introduce two functions that will be used in the sequel of the paper. The function `SchVerClasses(s,n)` returns the set of classes of the version number `n` of the schema `s`. The function `Current(s)` returns the current version of `s`.

Let us now consider one schema at a given time. In this way, a class is identified by its name and the number of the schema version. The class is then represented as a couple `(c,n)`, where `c` is the class name and `n` is the schema version number.

2.2 Object - Object Version

An object is associated to a set of versions, and each one belongs to a different version of the schema.

An object version is defined as a triplet `(oid,num,val)`, where `oid` is the object identifier, `num` is the version number, and `val` is the version value. The object version is identified by the couple `(oid,num)`. Since, an object has at most one version by class, `num` is the one of the corresponding class. The value of an object version is an atomic domain element (e.g., `integer`) or a constructed domain element (e.g., a `tuple value`). It records all information about the version state. An object version is instance of a class. So, the extension of a class refers to the set of its associated object versions.

We introduce two functions that will be used in the sequel of the paper. The function `Objects(c,n)` returns the set of objects associated to the class `(c,n)`. The function `Ext(c,n)` returns the set of instances of the class `(c,n)` (i.e., the extension of `(c,n)`).

For the manipulation of object versions, we introduce three operations: **ObjectVersionCreation**, **ObjectVersionDerivation**, and **ObjectVersionDeletion**.

ObjectVersionCreation `(oid,cid)`**:** This operation creates the first version of the object identified by `oid`. It is triggered by the creation of the object via the user call of the method, **new**(). The identifier of this version is initialized by the couple `(oid,n)`, where `n` is the number of the class under which the object is created. This class is identified by `cid`. The value of this version is initialized by the values supplied as arguments to the **new** method.

ObjectVersionDerivation (`oid`, cid_1, cid_2)**:** The classes cid_1 and cid_2 are associated with two adjacent versions of the schema. This operation creates a new version of the object identified by `oid`, as an instance of the class cid_2. It is derived from the object version under the class cid_1. The identifier of this version is initialized by the couple `(oid,n)`, where `n` is the number of the class cid_2. The version value will conform to the structure of the class cid_2. It is deduced from the object version value under the class cid_1, by using the default transformation mechanism [4, 6].

The default transformation mechanism is based on class structures comparison. The value of an attribute which is present only in the class cid_2, is initialized by the attribute type default value (i.e., `0` for an `integer`, `‘‘’’` for a `string`, `nil` for a `class`, etc.). The value of an attribute present in both cid_1 and cid_2, is deduced according to the relationship between attribute types in these classes. If, for example, attribute types are classes, then the attribute value is preserved only if the attribute type in cid_1, is a sub-type of its type in cid_2. Otherwise, the value is initialized by the default value `nil`.

ObjectVersionDeletion `(vid)`**:** This operation deletes the object version identified by `vid`. It triggers the update of versions that reference the deleted version. The update replaces each reference to the (deleted) version by the default value `nil` (or by an object version whose type is a subtype of the deleted version).

2.3 Version Binding

The management of versions is transparent to users. At a user level, which is related to an application program or a query, entities are manipulated, whereas at the system level, we manipulate versions. The system performs such manipulation by using version binding. The version binding allows the system to decide what version of an entity is required by a user when accessing the entity. Thus, version identifiers do not need to be embedded in application programs or queries.

Briefly, the version binding is defined as follows [1]. At the compilation time, a program is associated with a particular version of the schema (by default the current version). At the run time, the program refers to this schema version. The function `LinkPrSch(p)`, which binds a program to a schema version, returns the schema version associated with the program `p`, that is the version under which `p` is compiled.

The reference to an object in a program, is a reference to a version of this object. This represents the object version, under a class of the schema version associated with the program. The function `LinkObjVer(o,p)` returns the version of the object `o` which belongs to `o`'s reference in `p`. This function is defined as follows:

$LinkObjVer(o,p) \in Ext(c,n)$ *such that:*
$(c,n) \in SchVerClasses(LinkPrSch(p))$

3 Combining Modification and Versioning

We assume that an update operation is applied to the current version of a schema. If the operation is accepted, then this is translated into a modification or a generation of a new version of the schema. The decision to use schema modification or versioning, is outside the scope of this paper. Let us assume that the update by modification or versioning is imposed by the user. We note that when a schema update is translated to a schema modification, the system generates a new version which reflects the modification, and for implementation reasons, keeps the old version. The last one is called a *hidden version*, because it is transparent to the applications. We will show how a hidden version can be physically deleted in section 4.

As mentioned in the introduction, our approach with regard to instances adaptation, combines *conversion*, *object versioning* and *emulation*. To this end, the concept of *class pertinence levels* is introduced. The pertinence level of a class is a means to characterize the importance of the availability of an object version, that is an instance of this class.

3.1 Class Pertinence Levels

This notion is based upon what we call a *weight* of a class. The weight of a class measures how much the availability of its instances is important for the users. Its definition takes into account both the number of class clients and the status of a schema version ("historical" or "current") in which the class is defined.

Before formally introducing the concept of a weight of a class, we introduce the definitions of a number of class clients and the current version of a schema.

- *Number of class clients:* the number of application programs referring to a class, constitutes an important quantitative metric of its pertinence. When a class is deleted, the application programs depending on it, should be modified to make them compatible w.r.t the schema version. So, the more the number of programs associated to a class is high, the less is the interest to delete it.

- *Current version of a schema:* Among all schema versions, we consider that the current version of the schema records information which may reflect more faithfully the real world.

Definition 1 (Class Weight.) *A class weight is a real value within the interval [0,1] and is defined as the ratio of the number of programs associated to the class by the number of all programs associated to the schema. The classes associated to the current schema version have maximal weight, that is 1.*

To compute the weight of a class `(c,n)`, we use the function `CWeight(c,n)`. Before giving the formal definition of this function, we introduce the following definitions. We denote by `S` the set of all possible schema names, `C` the set of all possible class names, `O` the set of all possible object identifiers, and `N` the set of all version numbers. We define:

- The context of a program as the set of properties which it uses. The function `Context(p)` returns the set of classes which are used in the body of application program `p`. This set is formed from classes used as attributes, formal parameters, and variables types.

- `Call(p)` as the set of all programs called in the body of the program `p`;

- `Ref(p)` as the set of all classes directly or indirectly referenced by the classes of `Context(p)`; and

- `ClassesOfP(p)` as the set of all classes whose objects may be accessed by the program `p`. This set is defined as follows:

$$ClassesOfP(p) = Context(p) \cup Ref(p) \bigcup_{p_i \in Call(p)} ClassesOfP(p_i)$$

Using the above notation, we formally introduce the weight of a class. Given a database schema `s`, the function `CWeight` is defined as follows:

$$CWeight : C \times N \longrightarrow [0,1]$$
$$CWeight(c,n) = \begin{cases} 1 & \text{if IsCurrentClass(c,n)} \\ \frac{card\{p_i \in P / (c,n) \in ClassesOfP(p_i)\}}{AllPr(s)} & \text{otherwise} \end{cases}$$

where:

- *IsCurrentClass(c,n)* = $(c,n) \in SchVerClasses(Current(s))$;

- `CWeight(c,n)` returns the weight of the class `(c,n)`; and

- `AllPr(s)` represents the number of all programs which use the database schema `s`. We assume that $AllPr(s) \neq 0$ and `P` is the set of program names.

If the value of `CWeight(c,n)` is `1`, this situation represents the fact that the availability of the instances of `(c,n)` is highly important, because either `(c,n)` is defined as the current version or it used by all the application programs. However the value 0 of `CWeight(c,n)` represents a situation where the availability of the instances of `(c,n)` is useless, that is `(c,n)` is defined in a historical version of the schema and this is not used by any application program. The intermediate values represent situations where the availability of the instances of `(c,n)` is less or more important, depending on the perception of the database administrator (which is responsible to set up the threshold for instances availability importance).

Definition 2 (Pertinence Level of a Class.) *A class pertinence level characterizes the importance of the availability of the instances of the classes with regards to database programs. Two pertinence states of a class are introduced: pertinent or obsolete. A class is said to be pertinent if the availability of its instances is highly important. Otherwise, the class is said to be obsolete.*

A database administrator can fix a *threshold* for the pertinence of a class. In this way, a class is pertinent if its weight is superior to a threshold (*obsolescence threshold*). Otherwise, the class is said to be obsolete. Formally, if we denote by `PL` the function which determines the pertinence level of a class and `OT` ($OT \in R$ *and* $OT \in [0,1]$) the obsolescence threshold, we have:

$$PL : C \times N \longrightarrow \{\text{“}pertinent\text{”}, \text{“}Obsolete''\}$$
$$PL(c,n) = \begin{cases} \text{“}Pertinent'' & CWeight(c,n) > OT \\ \text{“}Obsolete'' & \text{otherwise} \end{cases}$$

3.2 Instance Adaptation

When an object, say `o`, is accessed under a class, say `(c,n)`, the system checks if the object `o` has a version under `(c,n)`. If not, a new version of `o` is generated and this is physically stored only if `(c,n)` is a pertinent class. Otherwise, this version is calculated.

To generate of a new version, stored or calculated, of an object `o` under a class `(c,n)`, a sequence of basic primitives, called *object version derivation* or *deletion*, is applied on versions of the object. The first primitive of the sequence is applied on a stored version of the object, called the *root of generation*.

3.2.1 Primitives

Four primitives are used for the generation of versions: **OriginVersion()**, **ObjectVersionGen()**, **Nextclass()** (resp., **Previousclass()**), and **ObjectVersionDerivation()**.

(P1) Primitive **OriginVersion()**
An object may have stored and calculated versions. During its life span, an object has at least one stored version.

The root version to generate the version of an object `o` under a class `(c,n)` is a stored version of `o`, where the number is the nearest to `n`. This is determined by using the function `OriginVersion()` which is defined as follows:

$$OriginVersion : O \times (C \times N) \longrightarrow O \times N$$
$$OriginVersion(o,c,n) \in \{ObjVers(o) / \forall (o,j) \in ObjVers(o), \mid n-i \mid \leq \mid n-j \mid\}$$

where: `ObjVers(o)` is the set of stored versions of the object `o`.

(P2) Primitive **ObjectVersionGen()**
This operation generates the version of an object when this is accessed under a class whose extension does not contain a version of this object. This primitive is defined as follows:

ObjectVersionGen(oid,[rnum],cid [,status])

where: `oid` is the identifier of the object; `cid` is the identifier of the class; the optional parameter `rnum` is the number of the version from which the new version must be generated; and the optional parameter `status` takes its values from the set {`''stored''`, `''calculated''`}.

The new version generated by the primitive is stored (respectively, calculated) if the value of `status` is `''stored''` (respectively, `''calculated''`). If the value of `status` is not determined, then the generated version depends on the pertinence level of its corresponding class. It is stored if its class is pertinent, otherwise it is calculated. The use of the parameter `status` is very important in some cases, especially when the generated version must be stored whatever the pertinence level of `cid`. These cases are described in the section 4.

The primitive **ObjectVersionGen()** triggers a sequence of derivation of versions of the object `oid`. The first primitive of the sequence is the derivation of a version of `oid` from the root version to generate the version of `oid` under the class `cid`, that is `(oid,rnum)`, if the value of the `rnum` is fixed, `OriginVersion(oid,cid)` else. The last primitive of the sequence derives the version of `oid` under `cid`.

In the rest of this section, we denote by (c_1, n_1) the class of the version `OriginVersion(oid,cid)` and (c_2, n_2) the class `cid`. There are two scenarios about the time-based relationships between the two versions:

1. If the class (c_2, n_2) is younger than the class (c_1, n_1) (i.e, $n_2 > n_1$), then in the first step, a version of the object `oid` is derived under the class of `oid`, which is the successor of (c_1, n_1) in the sequence of `oid` classes. At the *ith* step, an object version is derived under the class of `oid`, which is the successor of the version of `oid`, derived at the *i-1th* step.

2. If the class (c_2, n_2) is older than the class (c_1, n_1) (i.e, $n_2 < n_1$), then in the first step, a version of the object `oid` is derived under the class of `oid`, which is the predecessor of (c_1, n_1) in the sequence of `oid` classes. At the *i-th* step, an object version is derived under the class of `oid`, which is the predecessor of the version of `oid`, derived at the *i-1th* step.

(P3) Primitives **NextClass()** and **PreviousClass()**
The successor (resp., the predecessor) of a class in a sequence of an object classes is determined by using the primitive **NextClass()** (resp., **PreviousClass**). These primitives are formally defined as follows:

$$
\begin{aligned}
&NextClass : O \times (C \times N) \longrightarrow C \times N \\
&NextClass(o, c, k) \in \{(o, i) \in Classes(o)/ \\
&\qquad \forall (c, j) \in Classes(o), i < k \wedge i \geq j\}
\end{aligned}
$$

$$
\begin{aligned}
&PreviousClass : O \times (C \times N) \longrightarrow C \times N \\
&NextClass(o, c, k) \in \{(o, i) \in Classes(o)/ \\
&\qquad \forall (c, j) \in Classes(o), i < k \wedge i \geq j\}
\end{aligned}
$$

During the generation process, stored or calculated versions of an object `oid` are derived. If the derivation of a version of `oid` under a class is required, then the pertinence level of this class is to be determined using the function `PL()`. If the class is pertinent, then a stored version of `oid` is derived. Otherwise, a calculated version of `oid` is derived.

(P4) Primitive **ObjectVersionDerivation()**
The new version of `oid` is derived by using the primitive **ObjectVersionDerivation()**. This primitive derives a stored version of an object, whereas here an object version may be stored or calculated. For this reason, this primitive is redefined by adding the `status` parameter, which takes its values in {`''stored''`,`''calculated''`}. Therefore, this primitive derives a stored (resp., calculated) version if the value of `status` is `''stored''` (resp., `''calculated''`):
ObjectVersionDerivation(oid, cid$_1$*, cid*$_2$*, status)*

3.2.2 Algorithm

Below, we summarize the algorithm that implements the instance adaptation technique.

- The system uses this algorithm when an object is accessed under a class in an application. When an object `o` is accessed under a class `(c,n)`, the system checks if the object `o` has a version under the class `(c,n)`. If yes, the object is used by the application without any transformation. If not, a new version of the object must be generated.

- The algorithm uses the `OriginVersion(o,c,n)` expression to determine the root version to generate the version of `o` under `(c,n)`, and uses the primitive `object-version-gen(o,c,n)` to generate the version of `o` under `(c,n)`.

- After generating the version of the object `o` under the class `(c,n)`, the algorithm checks if the weight of the class of the root version is equal to `zero`, and if the object `o` has at least another stored version which is different to the root version. If yes, then the root version is deleted by using the primitive `del-object-version(o,r)` such that *(o,r)=OriginVersion(o,c,n)*. The weight of a class is equal to `zero` in the following cases:

1. The class is only associated to a hidden schema version, that is a schema version on which a schema evolution is translated to a modification.
2. The class is not associated to the current version of the schema and is not used in any application program.

- The deletion of the root version means that the algorithm uses the *conversion* when it is not necessary to keep it. So, the root version is preserved only to satisfy the property requiring that an object must have at least one stored version.
- The algorithm triggers the generation of a stored object version only if the associated class is pertinent, that is:
  - It is defined in the current version of the schema.
  - It is used in a number of programs, judged sufficiently high by the database administrator. Consequently, if we assume that the objects access rate is the same for all programs, then the object versions associated to this class, are frequently accessed. An object version is then generated.

The above algorithm generates a version `(o,n)` of an object `o` from the stored version of `o`, whose number is the nearest to `n`. Therefore, the generation process triggers a minimal set of operations on `o` versions. The redundancy is then avoided and the time of generation is minimized.

4 The Database Refreshing Technique for Large Objects

Some of the information contained in the database history may be useless in the present state of a database eventhough this information may have been judged important in the past. Thus, the importance of information retention may vary during the life span of the database. In particular, the use of the algorithm presented in the previous section has the following limitations: (i) since a system needs to keep track of the schema evolution during the life span of the database, then even if some schema versions must be deleted, they are only hidden from applications. Also, (ii) after a creation of a schema version, this one is preserved during all the life span of the database even if it is used by sufficiently a low number of programs. So the database size might increase considerably and a big part of its information become irrelevant or useless for application programs.

As previously pointed out, the pertinence of a historical schema version is the key information to decide whether this can be either kept or deleted from the database. This section addresses the definition of such a concept as well as the operation (called *immediate refreshing*) which is used by the database administrator to improve the performance of a system by deleting those historical schema versions which are judged not pertinent for future uses by application programs. Further, when a schema version is deleted, some of its classes are deleted too. This section also discusses the deletion of a class and a schema version.

4.1 Class - Schema version deletion

Here we provide appropriate conditions for the database administrator to decide about the deletion of a class or a schema version. Also, the primitive allowing such a deletion is provided here.

4.1.1 Class Deletion

Given a class `c`, this may be deleted if it satisfies the following conditions:

(CC1) `c`'s weight is equal to zero.

(CC2) Each of `c`'s sub-classes in all schema versions satisfies deletion conditions.

The condition (CC1) means that the class `c` is not associated to the current version of the schema, and is not used in any application. The condition (CC2) is introduced mainly because of the fact that the deletion of a class, with sub-classes that do not verify deletion conditions, is a costly operation. This operation triggers sub-classes and the reorganization their instances, to preserve the database consistency. Consequently, some programs must be modified.

The deletion operation, denoted as `DeleteClass(cid)`, deletes the class identified by `cid`. The class `cid` must be a leaf in all associated schema versions and must satisfy deletion conditions. At the deletion time, it is possible that the class extension contains stored versions. Some of these versions may be useful to derive object versions under other classes. For this reason, after the deletion of a class, its object versions may be deleted or converted.

Let us consider the class `(c,n)`. Assume that `o` is an object which has a stored version under the class `(c,n)`. The deletion of `(c,n)` triggers the conversion of the version `(o,n)` to another class, only under the following conditions (called *conversion conditions*):

(DC1) The object `o` is a member of other classes which are different from `(c,n)` and do not verify deletion conditions. This means that the object `o` may be accessed under another class after the deletion of `(c,n)`.

(DC2) The object `o` has one stored version, meaning that if `(o,n)` is deleted after the deletion of `(c,n)`, then the root version will generate the version of `o` under another class `(c,m)` when accessing `o` under `(c,m)` does not exist.

If the version `(o,n)` must be converted, that is, it satisfies the conversion conditions, then it is converted to a class of `o` that we call the *reception class* of `o`. The reception class of `o` is the `(c,n)` the nearest pertinent class, if `o` is associated to pertinent classes. Otherwise, this reception class represents nearest class to `(c,n)`. The function which determines the reception class, denoted as `ReceptionClass()`, is formally defined as follows:

$$ReceptionClass : O \times (C \times N) \longrightarrow C \times N$$
$$ReceptionClass(o, c, n) \in$$
$$\begin{cases} \{(x,i) \in PC / \forall (y,j) \in PC, \\ \qquad \mid n-i \mid \leq \mid n-j \mid\} & \text{if } PC \neq \{\} \\ \{(x,i) \in Classes(o) / \forall (y,j) \in Classes(o), \\ \qquad \mid n-i \mid \leq \mid n-j \mid\} & \text{otherwise} \end{cases}$$

The conversion of the object version `(o,n)` to the `ReceptionClass(o,c,n)` class consists of:

1. generating a new stored version of `o` under `ReceptionClass(o,c,n)` by using the primitive `ObjectVersionGen(o,n,ReceptionClass(o, c,n),‘‘stored’’)`; and

2. deleting the version `(o,n)` by using the primitive `DelObjectVersion(o,n)`.

In summary, at the time of a class deletion, for each object version in the extension of this class, the algorithm, which implements the operation **DeleteClass()**, checks if this version satisfies the conversion conditions. If yes, the object version is converted to the object reception class. Otherwise, the version is deleted.

4.1.2 Schema Version Deletion

With regard the deletion of a schema version, we use the notion of *weight* to decide whether or not the schema needs to be deleted. Here we first introduce this notion of (schema) weight and later describe the deletion conditions.

Definition 3 (Schema Version Weight) *A schema version weight is a natural integer which measures the degree of availability of the schema version for application programs. This weight represents the number of associated programs, that is the programs depending on it.*

We now introduce a function, denoted as `SWeight()`, which computes the weight of a schema version. This function is formally defined as follows:

$$SWeight : S \times N \longrightarrow N$$
$$SWeight(s, n) =$$
$$\begin{cases} \infty & if\text{(s,n)= Current(s)} \\ card\{p_i \in P/(s,n) = LinkPrSch(p_i)\} & \text{otherwise} \end{cases}$$

where: `LinkPrSch(p)` represents the schema version under which `p` is compiled.

When a schema version has a weight which is equal to `0`, then this schema will be deleted. The corresponding deletion operation, denoted as `DeleteSchemaVersion(svid)`, deletes the schema version identified by `svid`. The version `svid` must satisfy the deletion condition. The deletion of a schema is translated to the deletion of its classes which verify deletion conditions.

To determine the set of classes to be deleted in a schema version, we use the function `ClassesToDel()`. In this function, we use an expression `SSub(c)` to return the set of direct sub-classes of the class `c` in all schema versions.

$$ClassesToDel : S \times N \longrightarrow 2^{C \times N}$$
$$ClassesToDel(s, n) = \{c \in SchVerClasses(s, n)/$$
$$((CWeight(c) = 0) \wedge ((SSub(c) = \{\})$$
$$\vee (\forall x \in SSub(c), x \text{ verifies deletion conditions))}\}$$

The algorithm which implements the operation **DeleteSchemaVersion()** proceeds as follows. For each class `c` in the set of classes which may be deleted in the schema version, the following checking is performed: if `c` is a leaf in all associated schema versions (i.e., schema versions which share `c`), then `c` is deleted by using the primitive `DeleteClass(c)`. Otherwise, the deletion of `c` triggers the deletion of its sub-classes in all schema versions. So, the deletion of class may trigger the deletion of other classes in other version of the schema.

4.2 Refreshing Operation

This section describes the operation allowing the immediate refreshing of a database. This operation provides the database administrator with the possibility to delete historical schema versions if he/she judges that these versions are used in a sufficiently low number of programs and/or to fix the maximal number of historical schema versions.
The primitive **RefreshDatabase()**, used by the administrator to launch the database refreshing, is defined as follows:

RefreshDatabase [np: nump] [nv: numsv[,ord]]

Where all parameters are optional. After the refreshing, the historical schema versions which are associated to a number of programs inferior or equal to `nump`, are to be deleted. The default value of this parameter is `0`. The value of the parameter `numsv` fixes the maximal number of historical schema versions that the administrator decides to preserve. After the refreshing, the number of historical schema versions is inferior or equal to `numsv`. The default value of this parameter is ∞. The parameter `ord` takes its values in the set {`''weight''`,`''age''`}. If the value of `ord` is equal to `''age''`, then the most recent schema versions are preserved. Otherwise, the preserved versions are those whose weight is higher than others. In the last case, if it is necessary to make a choice between versions with same weights, then the most recent are preserved. The default value of this parameter is `''weight''`.

We summarize here the three phases of the algorithm implementing the operation **refresh-database()**:

Phase 1 (searching schema versions which must be deleted): this phase consists of two main steps. In the first step, all schema versions whose weight is inferior or equal to `nump` are identified. These versions will be deleted. In the second step, if the number of schema versions which are not identified in the first step is superior to `numsv` (maximal number of versions that the user decides to preserve), then other versions to be deleted are identified. The identification of these versions is based on the value of the parameter `ord`.

Phase 2 (notification): the programs associated with the schema versions to be deleted, that is those which are identified in the previous step, are notified to the user. They must be recompiled in order to be associated with the current version of the schema.

Phase 3 (schema versions deletion): in this last phase, schema versions identified in the first phase, are finally deleted. A schema version is deleted by using the operation **delete-schema-version()**.

5 The DBEvolution System

The purpose of the DBEvolution system is to be used to change the schema or to refresh the database. The first version of this system is currently operational at University of Grenoble. This is implemented on top of the O_2 object-oriented database system using the O_2C programming language [8]. The physical DBEvolution architecture has five modules as depicted in figure 1:

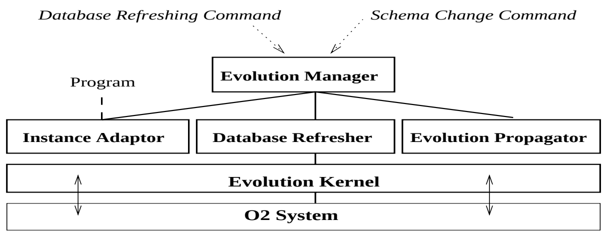

Figure 1: DBEvolution's General Architecture

Evolution Kernel: provides schema evolution services and implements the versions management primitives. The evolution kernel is an application on top of the O_2 system. Versions management primitives are implemented by using methods of the schema of this application. Figure 3 shows a global view of the evolution kernel classes. For example, the class `O-Version` (Object Version) provides methods for the management of object stored versions.

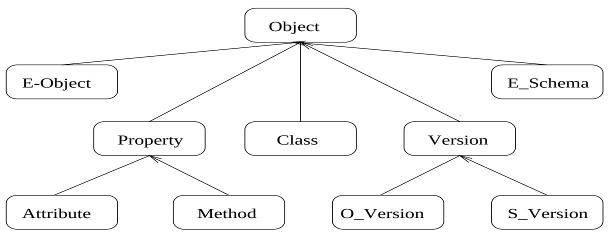

Figure 2: Classes of the evolution kernel

Evolution Manager: provides a database evolution language through which users can specify a schema evolution or a database refreshing command. It calls the appropriate component to handle an evolution. Also, this module calls the evolution propagator, if the command is a schema evolution and calls the database refresher, if the command is a database transformation.

Evolution Propagator: interacts with the evolution manager to handle a schema evolution command. It checks if a schema evolution satisfies the schema consistency. If yes, it propagates evolution effects at different levels: schema and instance levels. At the schema level, the propagation consists to generate a new schema version or to modify the most recent schema version, according to the evolution mode. In the case of modification, programs associated to the most recent schema version must be recompiled. At the instance level, the propagation consists to associate existing object identifiers to other classes.

Instance Adaptor: is used by the system to initialize or to update the value of an object version when accessing it in a program. It checks if the object has a stored version under the associated class (i.e., the class used by the program). If not, a new (stored or calculated) version is generated

Database Refresher: interacts with the evolution manager to handle a database refreshing command. It refreshes the database by deleting schema versions which are not pertinent for applications.

6 Conclusion

In this paper we have proposed a schema evolution framework which combines modification and versioning approaches. A schema update induces a modification of the schema, only if it is not substractive (i.e, involves information removal) or the user imposes "modification" mode. The proposed technique for supporting instances adaptation after a schema update, is based upon the evaluation of the pertinence of the availability of object versions w.r.t application programs. The number of versions is limited to those which are frequently accessed. We have proposed an operation which allows the immediate refreshing of databases. The database administrator can launch this operation when he judges necessary (e.g, space optimization). The database refreshing delete historical schema/object versions, which are not pertinent (or judged not pertinent by the database administrator) for the applications.

References

[1] Benatallah, B. *Evolution du schema d'une base de donnees a objets: une approche par compromis.* PhD dissertation, University of Joseph Fourrier, Grenoble, March 1996.

[2] Benatallah, B. and Fauvet, M-C. *Main Issues on Evolution of Object Database Schema.* In OBJET: logiciels, bases de donnees et reseaux, Editions Hermes (Hermes Publishers), Vol.3(3), pp. 277-298, Sept. 1997.

[3] Clamen, S. *Schema Evolution and Integration.* Distributed and Parallel Databases Journal, Vol. 2(1), Jan. 1993.

[4] Ferrandina, F., Meyer, T., and Zicari, R. *Schema and Database Evolution in the O2 system.* Proc. of the 21th VLDB Int. Conf., Zurich, Sept. 1995.

[5] Fontana, E. Dennebouy, and Y. *Schema Evolution by using Timestamped Versions and Lazy Strategy.* Proc. of the French Database Conf. (BDA), Nancy, Aug. 1994.

[6] Lerner, B. and Habermann, A. *Beyond Schema Evolution to Database Re-Organization.* Proc. of the ECOOP/OOPSLA'90, Ottawa, Oct. 1990.

[7] Monk, S. and Sommerville, I. *Schema Evolution in OODBs Using Class Versioning.* SIGMOD RECORD, 22(3), Sept. 1993.

[8] O2 Technology. *The O2 User Manual.* Version 4.6, Sep. 1995.

[9] Penny, D. and Stein, J. *Class Modification in the GemStone Object-Oriented DBMS.* Proc. of the ACM OOPSLA Int. Conf., Sept. 1987.

[10] Ra, Y. and Rundensteiner, E. *A transparent Object-Oriented Schema Change Approach Using View Evolution.* Technical Report CSE-TR-211-94, Dept. of EECS, Univ. of Michigan, Feb. 1994.

[11] Li, X. and Tari, Z. *Class Versioning for the Schema Evolution.* Proc. of the Australian Database Conference (ADC), Perth, Februrary 1998.

[12] Zdonik, S. *Object-Oriented type evolution.* Advances in Database Programming Languages. ACM Press Nework, (Bancilhon F., Bunema P. editors), 1990, pp. 277-288.