A Unified Framework for Supporting Dynamic Schema Evolution in Object Databases

Boualem Benatallah

Queensland University of Technology
2 George Street, Brisbane GPO BOX 2434, QLD 4001, Australia
boualem@icis.qut.edu.au

Abstract. This paper addresses the design of an integrated framework for managing schema evolution. This framework is based on the adaptation and extension of two main schema evolution approaches, namely *schema modification* and *schema versioning*. The proposed framework provides an integrated environment to support different database evolution techniques (such as, modification and versions at the schema level, conversion, object versioning, and screening at the instance level). We introduce the concept of class/schema version pertinence enabling the database administrator to judge the pertinence of versions with regard to database applications. Finally, we propose a declarative language based on *OQL*, the *ODMG* query language, that the user can use to guide objects adaptation process when dealing with complex or application specific schema updates.

1 Introduction

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 with regard to the new schema. The support of schema evolution is the ability to allow changes of the structure and the behavior of schema's constituents without disturbing applications 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.

1.1 Background

In this paper, we briefly overview the major approaches that are closely related to the work presented in this paper, namely *modification-based*, *versioning-based*, and *view-based* approaches [1,11].

J. Akoka et al. (Eds.): ER'99, LNCS 1728, pp. 16–30, 1999.
© Springer-Verlag Berlin Heidelberg 1999

- In a *modification-based* approach [6,9], the new schema definition replaces the old one. The *conversion* technique is used for object adaptation [6]. All existing objects must be converted to objects fitting class definitions in the new schema. The conversion may cause information loss. Another major limitation of this approach is the incompatibility of old programs w.r.t the new schema when conversion occurs.
- In a *version-based* approach [4,7,8,12], a schema update causes the derivation of new versions of classes/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 approach, two techniques of object adaptation are proposed in the literature: *screening* and *object versioning*.

  1. In the first technique [8,12], an object is associated with the class version under which the object is created. The representation of an object is never restructured after its creation. The system is responsible for ensuring access to old and new objects by programs. This technique supports program compatibility w.r.t the schema. However, it has two major limitations. The first limitation is the extra cost involved when objects are accessed. The second limitation is the information loss. The value of an added (respectively, deleted) attribute in an old (respectively, 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 [4,7]. This technique resolves the problem of information loss, by supporting many object versions. However, its main limitation is the proliferation of the number of versions, due to the dynamic nature of object oriented applications. This results in a considerable overhead on the system, because the storage requirement for versions and the cost of maintaining versions relationships, increase with the number of versions.

- Some proposals (e.g., [10,11,3]) suggest the use of the *view-mechanism* to manage schema evolution. In this approach, a schema update is performed by the creation of a view that simulates the semantics of this update. The support that is provided for schema evolution by a view-based approach is not sufficient. More specifically, this approach does not provide a satisfactory solution for additive schema updates (i.e, updates requiring the reorganization of the database by information addition), such as the addition of a new attribute into the definition of a class. The information added by views (e.g, a derived attribute) are derived from other information that exist already in the database; whereas the information added by a schema update is in general new and most of the time independent from existing information.

1.2 Contributions

This paper addresses the schema update problem by integrating *modification* and *versioning* into a unified framework.

- Modification- and version-based mechanisms are complementary. A version-based mechanism can be used to overcome the problem of information loss. A modification-based mechanism can be used to control the proliferation of the number of versions by limiting the number of versions to those that are necessary.
- The existing approaches propose very primitive schema update languages. In general, these languages provide a fixed set of operations with pre-defined semantics. However, in some situations a schema update can be application specific [5]. The support of application specific updates calls for an extensible language that allows the user to specify on the fly, the desired semantic of a schema update.

Our approach can be summarized as follows:

- When a schema update is accepted, this update is effectively performed by either 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, involves information deletion) will result in the generation of new version and each *non-substractive* update will result in a modification.
- Regarding the issue of object adaptation, our approach is based on the *evaluation of the pertinence of object versions availability*. When an object `o` is accessed under a version `v` of the schema, the system we propose checks if the object `o` has a version under `v`, i.e, the extension of the class of `o` (which is defined in `v`) contains 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*, meaning that the version is available only during the period of use. To decide if an object version is to be stored or calculated, we have introduced the concept of *class pertinence levels*. A class may be *pertinent* or *obsolete*. It is pertinent if it is defined in the most recent version of the schema or it is used in a large number of applications. Otherwise, it is obsolete. Thus, the version of the object `o`, under `v`, is stored only if the class of `o` is pertinent.
- Unlike existing systems, which provide a fixed set of operations with pre-defined semantics, our approach provides an extensible language to support application specific schema updates. This language allows users to describe the relationships between the states of the database before and after the schema update. We show that, a declarative language (OQL) embedded in the primitives of our language, provides a full specification of a desired instance adaptation.

The remainder of this paper is organized as follows. In section 2, we give a brief overview of the version model that provides the basic concepts for our

approach. Section 3 addresses the issue of instance adaptation. Section 4 presents the language for specifying the semantic of a schema update. We then make some concluding remarks in section 5.

2 Basic Concepts and Definitions

The version model used in our approach 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 pair `(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 pair `(eid,num)`, where `eid` is the identifier of the entity and `num` is the number that the system associates with the version, at the time of its creation. The function `VersSet(eid)` returns the set of versions of the entity `eid`.

2.1 Schema - Schema Version

A schema is associated with 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. We use "current" or "historical" to denote what we call the *status* of a schema version. A schema version is defined as a triple `(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 aggregation and inheritance relationships.

We introduce two functions to manipulate schema versions. 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`. We consider one schema at a given time. Thus, a class is identified by its name and the number of the schema version. A class is then represented as a pair `(c,n)`, where `c` is the class name and `n` is the schema version number.

2.2 Object - Object Version

An object is associated with a set of versions. Each version belongs to a different version of the schema. An object version is defined as a triple `(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 pair `(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`). An object version is an instance of a class. So, the extension of a class refers to the set of its associated object versions.

We introduce two functions to manipulate object versions. The function `Objects(c,n)` returns the set of objects associated with 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)`). We also introduce three operations to manipulate object versions: **ObjectVersionCreation**, **ObjectVersionDerivation**, and **ObjectVersionDeletion**.

ObjectVersionCreation `(oid,cid):` This operation creates the first version of the object identified by `oid`. The identifier of this version is initialized by the pair `(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 a default value which is conform to the type of the class `cid`.

ObjectVersionDerivation `(oid,` cid_1`,` cid_2`):` The classes cid_1 and cid_2 are defined in 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 pair `(oid,n)`, where `n` is the number of the class cid_2. The version value conforms to the type of the class cid_2. It is derived from the object version value under the class cid_1, by using the default transformation mechanism [6].

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

2.3 Version Binding

The management of versions is transparent to users. At the compilation time, a program/query 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 under which `p` is compiled.

The reference to an object in a program, is a reference to a version of this object. It represents the object version, under a class of the schema version which is associated with the program. The function `LinkObjVer(o,p)` returns the version of the object `o` which represents `o`'s reference in `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 update is performed by a schema 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. We assume that the update by modification or versioning is imposed by the user.

As mentioned in the introduction, our approach with regard to instances adaptation, combines *conversion*, *object versioning* and *screening*. To this end, the concept of *class pertinence levels* is introduced [2]. The pertinence level of a class is a means to characterize the importance of the availability of an object version, which 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 database applications. 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:

- *Number of class clients:* the number of application programs referencing a class constitutes an important quantitative metric of its pertinence. When a class is deleted, the application programs which use the class, should be modified to be compatible w.r.t the schema version. So, the greather the number of programs using a class, the less is the interest in deleting 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 with the class by the number of all programs associated with the schema. The classes defined in the current version of the schema have a maximal weight (i.e, 1).*

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

- `Context(p)` as the set of classes which are used in the body of the program `p`. This set is formed from classes used as attributes, formal parameters, and variable types.
- `Call(p)` as the set of programs called in the body of the program `p`.
- `Ref(p)` as the set of classes directly or indirectly referenced in the classes of `Context(p)`.
- `ClassesOfP(p)` as the set of classes whose objects may be accessed by the program `p`. This set is defined as follows:
  $ClassesOfP(p) = Context(p) \cup Ref(p) \bigcup_{p_i \in Call(p)} ClassesOfP(p_i)$

Given a database schema `s`, the function `CWeight` is defined as follows:

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

where:

- *IsCurrentClass(c,n)* = *(c,n)*$\in SchVerClasses(Current(s))$.
- `CWeight(c,n)` is the weight of the class `(c,n)`.
- `AllPr(s)` is the number of all programs which use the database schema `s`. We assume that `AllPr(s)` $\neq$ `0`.

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 the current version or it is used in all application programs. However, the value `0` of `CWeight(c,n)` represents a situation where the availability of the instances of `(c,n)` is useless, i.e, `(c,n)` is defined in a historical version of the schema and is not used in any application. 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 (who 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 class with regards to database applications. Two pertinence values are considered: pertinent or obsolete. A class is pertinent if the availability of its instances is highly important. Otherwise, the class is obsolete.*

A database administrator can fix a *threshold* for the pertinence of classes. Thus, a class is pertinent if its weight is greater than a threshold (*obsolescence threshold*). Otherwise, the class is obsolete. If we denote by `PL` the function which determines the pertinence level of a class and `OT` ($OT \in [0,1]$) the obsolescence threshold, we have:

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

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. This version 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 performed on the object versions. The first primitive of the sequence is applied on a stored version of the object, called the *root of generation*.

Version Generation Primitives Five primitives are used for the generation of object versions: **OriginVersion()**, **ObjectVersionGen()**, **Nextclass()** (respectively, **Previousclass()**), and **ObjectVersionDerivation()**.

(P1) Primitive **OriginVersion()**
An object may have stored and calculated versions. During its life span, an object must have 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`, whose number is the nearest to `n`. It 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 version 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 in the set {`''stored''`,`''calculated''`}.

The new version is stored (respectively, calculated) if the value of `status` is `''stored''` (respectively, `''calculated''`). If the value of `status` is not fixed, then the generated version depends on the pertinence level of its 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 primitives on `oid`'s versions. 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` (i.e, `(oid,rnum)` if the value of `rnum` is fixed, `OriginVersion(oid,cid)` otherwise). The last primitive of the sequence generates the version of `oid` under `cid`.

In the remainder 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`'s 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`'s 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 (respectively, the predecessor) of a class in a sequence of the classes of an object, is determined by using the primitive **NextClass()** (respectively, **PreviousClass**). These primitives are defined as follows:

$NextClass : O \times (C \times N) \longrightarrow C \times N$
$NextClass(o, c, k) \in \{(o, i) \in Classes(o) / \forall (c, j) \in Classes(o), i > k \wedge i \leq j\}$

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

During the generation process, stored or calculated versions of the object `oid` are generated. 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 generated. Otherwise, a calculated version of `oid` is generated.

(P4) Primitive **ObjectVersionDerivation()**
A new version of an object is derived by using the primitive **ObjectVersionDerivation()**. This primitive generates 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 generates a stored (respectively, calculated) version if the value of `status` is `‘‘stored’’` (respectively, `‘‘calculated’’`):
ObjectVersionDerivation(oid, cid_1, cid_2, status)

Algorithm Below, we summarize the algorithm implementing the instance adaptation technique.

– 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 `ObjectVersionGen(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 differ from the root version. If yes, then the root version is deleted by using the primitive `ObjectVersionDeletion(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 defined only in a hidden schema version. As pointed out before, a hidden schema version is a schema version on which a schema update is performed by a modification.
  2. The class is not defined in the current version of the schema and is not used in any application.
- The deletion of the root version means that the algorithm uses the *conversion* when it is not necessary to keep this version. So, the root version is kept 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, i.e:
  - It is defined in the current version of the schema or,
  - It is used in a number of programs, judged sufficiently high by the database administrator. Consequently, if we make the simplifying assumption that objects access rate is the same for all programs, then the object versions associated with this class, are frequently accessed.

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`'s versions. Redundancy is then avoided and the time of generation is minimized.

4 Language support for Customized Schema Updates

As mentioned in the introduction, the existing approaches provide a fixed set of schema update operations with pre-defined semantics. However, such a predefined taxonomy of operations is not sufficient for many advanced applications where schema updates are application specific. The support of such schema updates calls for an extensible language that allows the user to specify the desired semantic of a schema update. In this way, the language provides means for the customization of schema updates to the requirement of specific applications.

We propose a language for the specification of relationships between the states of the database before and after a schema update. We show that, a declarative language (OQL) embedded in the primitives of our language, provides a full specification of a desired instance adaptation .

[0] We note that the reader is not required to be familiar with OQL to understand the material presented in this paper, as the used OQL primitives are self-descriptive.

The aim of the proposed language is to provide constructs for the description of a relationship between two adjacent schema versions v_i and v_j (j= i-1 or i+1). This relationship specifies the desired semantic of the schema update that generates the version v_j from v_i. The information provided by these relationships is used during instances adaptation. A relationship expresses a mapping between the respective instances of v_i and v_j. The source of a mapping is a set of classes defined in v_i and the target of this mapping is a class defined in v_j. In the remainder of this paper, we call this relationship an *Instance Mapping Descriptor (IMD).*

Intuitively, to describe an IMD, we must localize *source objects* that are related to a given *target object* and specify how these objects are related. Thus, in our approach an IMD consists of two basic expressions: the *localization expression* and the *dependency expression.* The next subsection presents the instance mapping language through an example. Subsection 4.2 defines an instance mapping descriptor.

4.1 An Example

To illustrate the process of an IMD specification, we consider the database schema that contains the class `CProgram`, whose objects are programs written in the `C` language and the class `JavaProgram`, whose objects are programs written in the `Java` language. Let us consider the schema update that merges the classes `Cprogram` and `JavaProgram` into one single class, called `Program`. In this example, we want that the definition of the new class `Program` will contain an attribute, called `Language`, whose value is a `string` indicating the language used to write the corresponding program. The definition of the other attributes of the class `Program` is ignored here for clarity reasons.

We will now describe intuitively our instance mapping language. When updating the schema, the user can use this language to specify a mapping between the instances of the database before and after the schema update. In this example, we want to specify the following facts:

- All objects of `CProgram` and `JavaProgram` are also objects of `Program`.
- The value of the attribute `Language` in a version of an object of `Program` which is also an object of `CProgram` (respectively, `JavaProgram`) is ‘‘`CProgram`’’ (respectively, ‘‘`JavaProgram`’’).

As pointed out before, our language features two expressions for specifying an instance mapping: the localization expression and the dependency expression. The first expression localizes source objects that have a link with a given target object. The second expression describes the relationship between the value of the version of an object under the target class and the values of versions of its related objects under the source classes.

In this example, the instance mapping can be specified by the following descriptor:

```
source {CProgram,JavaProgram} target Program
localization
   query Objects(CProgram) union Objects(JavaProgram)
   link same-as-source
dependency
   Language derived with
       element(select *
       from {``CProgram'',``JavaProgram''} as name
       where name in ClassNames(this))
```

Where:

– `ClassNames(o)` is a set that contains the names of the classes of the object `o`.

The target is `Program` and the source is the set {`CProgram,JavaProgram`}. The *localization clause* introduces the localization of `CProgram` and `JavaProgram` objects. The *dependency clause* introduces the relationship between the version of an object under `Program` and the version of this object under `Cprogram` or `JavaProgram`.

Let us now show how an IMD is used to customize the semantics of a schema update. In our approach an IMD is used to:

1. *Initialize the extension of the target class.* After merging `CProgram` and `JavaProgram` into `Program`, the system uses the above descriptor to initialize the extension of `Program` as follows :
   – For each object in `Objects(CProgram) union Objects(JavaProgram)`, a new version is generated under the class `Program`.
   – The value of the attribute `Language` in the new version (generated in the previous step), is initialized to be the `name of the class` of the source object. A new version of an object `o` of the class `c` (i.e, `CProgram` or `JavaProgram`), under the class `Program`, is derived by using the following primitive:

   ObjectVersionDerivation(c,Program, o)
   `Language` **derived with** `element(select *`
   `from {``CProgram'',``JavaProgram''} as name`
   `where name in ClassNames(o))`

2. *Propagate the update of the extension of a source class to the extension of a target class.* When a new version of an object `o` is created under the class `CProgram` or the class `JavaProgram`, another version of `o` is created under the class `Program`. The last version is derived from the first one using the primitive described above.

In the remainder of this paper, we focus on the first issue. Details about the second issue can be found in [1].

4.2 Instance mapping descriptor

The syntax we use to specify an IMD is the following:

source {*class-name+*} **target** *class-name*
localization **query** *localization-expression* **link** *target-object-id*
dependency *(attribute-name attribute-specification)+* *attribute-specification* ::= **derived** *attribute** **with** *function-body* *target-object-id* ::= *same-as-source* or *new* *attribute* ::= *attribute-name class-name* *attribute-name, class-name, function-body* ::= string

The *source* and *target* clauses are self-descriptive. The following subsections describe the *localization* and *dependency* clauses.

Localization clause. The localization expression consists of two elements. The first element is a declarative query that selects the identifiers of source objects that will be related within an identifier of a target object. This query is introduced by the keyword **query**. The scope of this query contains objects of source classes. The second element is the expression that defines how a target objet is linked with a set of source objects. This expression is introduced by the keyword **link**. Without loss of generality, we consider two types of links:

1. **The identifier of the target object is new.** In this case, the keyword **link** is followed by the keyword *new*. For example, assume that the schema `s` has two versions `(s,0)` and `(s,1)`. The version `(s,0)` contains the classes `Student` and `Employee`. The version `(s,1)` contains the class `Employee-Student`. Consider the schema update that *merges* the classes `Employee` and `Student` into the class `Employee-Student`. Assume that we want to implement this schema update as follows: the objects of `Employee-Student` are constructed by joining pairs of objects from `Employee` and `Student` on the attribute `Name` (we assume that `Name` is a key attribute for `Employee` and `Student`). To this end, we specify the following descriptor:

source {`Employee`, `Student`} **target** `Employee-Student`
localization **query** `select struct(id1 : x, id2 : y)` `from Objects(Employee) as x, Objects(Students) as y` `where x.Name=y.Name` **link** *new*
...

When the class `Employee-Student` is created, its extension is initialized by using the information provided in the previous IMD. First, the query of the localization expression is evaluated. This query returns a set of pairs of objects formed by joining `Objects(Employee)` and `Objects(Student)`. Two objects form a pair if they have the same name. Second, the link expression of IMD informs the system that for each pair `(x,y)` of the set returned by the query, a new object `o` of the class `Employee-Student` is to be created. The link between `(x,y)` and `o` is materialized for future use. We use the expression `TargetLink(o)` to determine the set of source objects that are related with the target object `o`.

2. **The identifier of the target object is a source object.** In this case, the keyword **link** is followed by the keyword *same-as-source*. Consider the schema update that *merges* the classes `Cprogram` and `JavaProgram` into the class `Program`. When the class `Program` is created, its extension is initialized as follows. The query `Objects(CProgram) union Objects(JavaProgram)` is evaluated. After that, each object of the returned set is added to the extension of the class `Program`. This means that for each of these objects, a new version is to be created.

Dependency clause. The dependency expression of an IMD specifies the relationships between the attributes of the target class and those of the source classes. It is introduced by the *dependency clause*. For a given attribute, this expression defines how to compute the value of this attribute. Thus, it provides a means to initialize or modify the value of the version of an object under the target class.

In an IMD, the specification of the dependency expression of an attribute contains two elements. The first element is the name of the attribute of the target class. The second element is the function that computes the value of this attribute. This function is introduced by the keyword **derived with** and is an *OQL* query. Consider the schema update that *merges* the classes `CProgram` and `JavaProgram` into the class `Program`. The dependency expression of the attribute `Language` is described as follows:

```
Language derived with
   element(select *
   from {``CProgram'',``JavaProgram''} as name
   where name in ClassNames(this))
```

When a new version of an object in `Objects(CProgram) union Objects(JavaProgram)` is created under the class `Program`, the value of the attribute `Language` in this version is initialized using the information provided by the above dependency expression. That is, the `name` of the class of the root version, is set to the value of the attribute `Language`.

5 Conclusion

In this paper, we proposed a unified framework for supporting dynamic schema evolution in object oriented databases. A schema update is performed by a modification of the schema, only if it is not substractive (i.e, does not involve information deletion) or the user imposes the "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 applications. The number of versions is limited to those which are frequently accessed. We proposed an extensible language that can be used to specify a desired semantics of a schema update. This language provides means for the customization of schema updates to the requirement of specific applications. We have also implemented the proposed schema update mechanism on top of an object-oriented DBMS. Due to space reasons, the description of the prototype is outside the scope of this paper.

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. 16, 19, 27
2. Benatallah, B. and Tari, Z. *Dealing with Version Pertinence to Design an Efficient Object Database Schema Evolution Mechanism*. The IEEE Int. Database Engineering and Applications Symposium - IDEAS '98, July 1998, Cardiff, Wales, UK. 21
3. Breche, P. and Ferrandina, F. and Kuklok, M. *Simulation of schema and database modification using views*. Proc. of DEXA'95, London, UK, 1995. 17
4. Clamen, S. *Schema Evolution and Integration*. Distributed and Parallel Databases Journal, Vol. 2(1), Jan. 1993. 17, 17
5. Claypool, K. and Rundensteiner, E. *Flexible Database Transformations: The SERF Approach*. IEEE Data Engineering Bulletin 22(1), 1999. 18
6. 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. 17, 17, 20
7. Fontana, E. Dennebouy, Y. *Schema Evolution by using Timestamped Versions and Lazy Strategy*. Proc. of the French Database Conf. (BDA), Nancy, Aug. 1994. 17, 17
8. Monk, S. and Sommerville, I. *Schema Evolution in OODBs Using Class Versioning*. SIGMOD RECORD, 22(3), Sept. 1993. 17, 17
9. Penny, D. and Stein, J. *Class Modification in the GemStone Object-Oriented DBMS*. Proc. of the ACM OOPSLA Int. Conf., Sept. 1987. 17
10. Ra, Y. and Rundensteiner, E. *A Transparent Schema-Evolution System Based on Object-Oriented View Technology*. TKDE, 1997. 16, 17
11. Rodick, J. *A survey of schema versioning issues for database systems*. Information and Software Technology, 1995, 37(7) 383-393, Elsevier Science B.V. 16, 17
12. Zdonik, S. *Object-Oriented type evolution*. Advances in Database Programming Languages. ACM Press, (Bancilhon F., Bunema P. editors), 1990, pp. 277-288. 16, 17, 17