Evolving Partitions in Conceptual Schemas in the UML

Cristina Gómez and Antoni Olivé

Departament de Llenguatges i Sistemes Informàtics,
Universitat Politècnica Catalunya,
Jordi Girona 1-3 E08034 Barcelona (Catalonia)
{cristina,olive}@lsi.upc.es

Abstract. The evolution of information systems from their conceptual schemas is an important research area in information systems engineering. In this paper, we aim at contributing to the area by focusing on a particular conceptual modeling construct, the partitions. We analyze the evolution of partitions in conceptual schemas of information systems. We deal with conceptual models with multiple specialization and classification, and consider whether entity types are base or derived. We provide a list of possible schema changes and, for each of them, we give its preconditions, and its effects on the schema, taking into account the state of the information base. In this paper, we deal with conceptual schemas in the UML. However, the results reported here should be applicable to most conceptual modeling languages and also to object-oriented database schemas.

1 Introduction

The evolution of information systems is one of the most important problems in the field of information systems engineering. For several reasons, many organizations need to change very often their activities, and this usually requires an evolution of the information system that supports those activities. The evolution must be done always efficiently, and often quickly and without interrupting critical services [2]. Automated support for information system evolution becomes central to satisfy these evolution requirements [12].

Ideally, the evolution of information systems should follow the strategy called 'forward information system maintenance' in [8]: changes should be applied directly to the conceptual schema, and from here they should propagate automatically down to the database logical schema(s) and application programs. If needed, the database extension(s) should be also converted automatically. This strategy implies that the conceptual schema is the only description to be defined, and the basis for the specification of the evolution. All the others are internal to the system.

Many past and current research efforts aim directly or indirectly at that ideal. Most of them have been done in the database field and, more precisely, in the subfield that deals with the problem of schema evolution. The problem has two aspects: the semantics of changes (i.e. their effects on the schema) and the change propagation (i.e. the propagation of the schema changes to the underlying existing instances) [18]. Both aspects have been studied extensively for the relational and the object-oriented data models, in the temporal and the non-temporal variants [19]. The results have been often incorporated into commercial or prototype database systems (e.g., Orion [3], O_2 [26], Cocoon [24], F2 [1] and Tigukat [7]).

More recently, in the software engineering field, the problem of software evolution is being dealt with a refactoring approach [16]. A refactoring is a parameterized behavior-preserving program transformation that automatically updates an application's design and source code. Design refactoring deals with design constructs rather than code, and therefore it can be applied also to models in the UML [22]. The approaches in the databases and software engineering fields are similar, because database schema evolution transformations have their parallels in refactoring transformations [23].

In this paper, we aim at contributing to the general field of information systems evolution from conceptual schemas. We extend the work reported in [10] by dealing with a particular conceptual modeling construct, partitions, and analyze, the possible changes and their effects at the schema and instance levels.

Partitions are well-known constructs, used in conceptual modeling, object-oriented software design and object-oriented database schemas. A partition of an entity type like, for example, *Person* into entity types *Man* and *Woman* states that *Man* and *Woman* are subtypes of *Person*, that *Man* and *Woman* are disjoint, and that the population of *Person* is exactly the union of that of *Man* and *Woman*. The interest of partitions lies in their simplicity, expressiveness and generality (since specializations and generalizations can be transformed into partitions). Often it is easier to develop, analyze and reason about conceptual schemas, when only partitions are considered [21, 4, 25, 13].

However, partitions have not been studied in the literature on schema evolution. Since the early works of Orion [3], there has been a lot of work related to the evolution of specializations (or generalizations or subclass/superclass relationships) but, as far as we know, there are not published results on the evolution of partitions. The work most similar to ours is [5], which takes into account disjointness and completeness constraints between entity types, but partitions are not considered schema objects, and the context is restricted to object-oriented databases.

The main contribution of our paper is the analysis of the evolution of partitions in conceptual schemas of information systems. We deal with conceptual models with multiple specialization and classification, and consider whether entity types are base or derived (with different kinds of derivability). We show that derivability has an important influence on the evolution of partitions. We provide a list of possible schema changes (related to partitions and derivability) and, for each of them, we give its preconditions, and its effects on the schema, taking into account the state of the information base.

In this paper, we deal with conceptual schemas expressed in the UML. We hope that, we ease the application of our results to industrial projects, and the integration with other ongoing projects. However, the results reported here should be applicable to most conceptual modeling languages and also to object-oriented database schemas. In particular, the results can be adapted to the logic-based language used in [10].

The rest of the paper is structured as follows. Section 2 reviews the concept of taxonomic constraints, partitions, derived types and constraint satisfaction. In Section 3, we propose an UML Profile for Partitions in Conceptual Modeling. This profile is an extension to the UML, using the standard mechanisms provided by the language. We explain that the profile is needed to represent partitions, taxonomic constraints and different kinds of derived types in the UML. In Sections 4 and 5, we present the operations that we propose to evolve partitions and derivability, respectively. For each operation, we give a description and an intuitive explanation of its pre and postconditions. Due to space limitations, we can include the formal specification in the OCL of only one operation. The full details of the profile and operations can be found in [6]. Finally, Section 6 gives the conclusions and points out future work.

2 Partitions

In this section, we review briefly the basic concepts and the terminology that will be used throughout the paper, taken mainly from [14].

2.1 Taxonomic Constraints and Partitions

A taxonomy consists of a set of entity types and their specialization relationships. There are also taxonomies of relationship types, but these will not be studied in this paper. We call taxonomic constraints the set of specialization, disjointness and covering constraints defined in a schema.

A *specialization* constraint between entity types E' (the subtype) and E (the supertype) means that if an entity e is instance of E', then it must be instance of E.

A *disjointness* constraint between entity types E_1 and E_2 means that the populations of E_1 and E_2 are disjoint.

Finally, a *covering* constraint between an entity type E and a set of entity types $\{E_1,...,E_n\}$, means that if e is instance of E, it must be also instance of at least one E_i.

A generalization corresponds to a set of specialization constraints between E_i and E, for $i = 1,..,n$, with a common supertype E. A generalization is *disjoint* if their subtypes are mutually disjoint; otherwise, it is *overlapping*. A generalization is *complete* if the supertype E is covered by the subtypes $E_1,...,E_n$; otherwise it is *incomplete*.

A partition is a conceptual modeling construct that allows us to define in a succinct way a set of taxonomic constraints. A *partition* is a generalization that is both disjoint and complete. A partition of E into $E_1,...,E_n$ is semantically equivalent to:

- A set of n specializations constraints between E_i and E, for $i = 1,..,n$
- A covering constraint of E by $\{E_1,...,E_n\}$
- A set of $n(n-1)/2$ disjointness constraints between E_i and E_j, for $i,j = 1,..,n$, $i > j$.

2.2 Derived Types

The entity types involved in a partition can be base or derived. We will see that this aspect has a strong influence on the satisfaction of taxonomic constraints related to a partition. An entity type E is derived when the population of E can be obtained from

the facts in the information base, using a derivation rule. Derived entity types can be classified depending on the form of their derivation rule. We give a special treatment to the following classes:
- Derived by *specialization*. Entity type E is derived by specialization of entity types E_1, ..., E_n ,with $n \geq 1$, if the population of E is the subset of the intersection of the populations of E_1,..., E_n, that satisfy some condition. For example, *Young* may be defined as a specialization of *Person*, with the condition "age less than 18 years".
- Derived by *exclusion*. This is a particular case of specialization. Entity type E is derived by exclusion if its population corresponds to the population of an entity type E', excluding those entities that belong also to some entity types E_1, ..., E_n, with $n \geq 1$. For instance, *Unmarried* may be defined as specialization of *Person*, excluding *Married*.
- Derived by *union*. Entity type E is derived by union if its population is the union of the populations of several entity types E_1, ..., E_n, with $n \geq 1$. For instance, *Person* may be defined as the union of *Man* and *Woman*.

2.3 Satisfaction of Partition Taxonomic Constraints

In general, satisfaction of integrity constraints can be ensured by the schema or by enforcement. A constraint *IC* is satisfied *by the schema* when the schema entails *IC*. That is, the derivation rules and the (other) constraints defined in the schema imply *IC* or, in other words, *IC* is a logical consequence of the schema. In this case no particular action must be taken at runtime to ensure the satisfaction of *IC*.

A constraint *IC* is satisfied *by enforcement* when it is not satisfied by the schema, but it is entailed by the information base. That is, *IC* is a condition true in the information base. In this case, the system has to enforce *IC* by means of checking and corrective actions (database checks, assertions, triggers, or transaction pre/post-conditions), to be executed whenever the information base is updated.

An analysis of the taxonomic constraints satisfied by a schema is presented in [14]. We restructure and summarize the conclusions presented there as follows:

- - A specialization constraint between E_i and E is satisfied by a schema when:
  - E_i is derived by specialization of E.
  - E_i is derived and E is base.
  - E is derived by union of a set of types that includes E_i.
- - A covering constraint between E and $\{E_1,...,E_n\}$ is satisfied by a schema when:
  - E is derived by union of $\{E_1,...,E_n\}$.
  - There is an $E_i \in \{E_1,...,E_n\}$ derived by specialization of E and exclusion of $\{E_1,...,E_n\} - \{E_i\}$.
  - There is a partition P with supertype E and subtypes $\{E_1,...,E_n\}$ and such that all subtypes are derived by specialization of E.
- -A disjointness constraint between E_i and E_j is satisfied by a schema when:
  - There is a partition P with supertype E and subtypes $\{E_1,...,E_n\}$, with E_i, $E_j \in \{E_1,...,E_n\}$ and such that all subtypes are derived by specialization of E.
  - E_i is base and E_j is derived.
  - E_i is derived by specialization of some E and exclusion of a set of entity types that includes E_j.

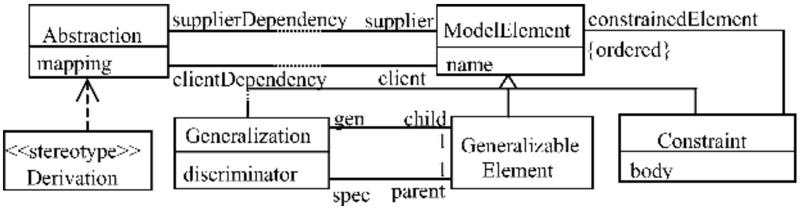

Fig. 1. Fragment of UML metamodel

These relationships allow us to determine which taxonomic constraints are satisfied by the schema and, complementarily, which ones need to be enforced. The distinction is very important when efficiency is a concern, as it is the case in this paper.

For example, if *Person* is derived by union of *Man* and *Woman*, then the specialization constraints between *Man* and *Person*, and between *Woman* and *Person* are satisfied by the schema. Similarly, the covering constraint between *Person* and {*Man*, *Woman*} is satisfied by the schema. Note that, in this case, the disjointness constraint between *Man* and *Woman* must be enforced.

3 Uml Profile for Partitions

In this section, we justify the need to extend the UML in order to deal with partitions, derived types and their associated concepts. We use the standard extension mechanisms provided by the UML [20], and define a UML Profile for Partitions in Conceptual Modeling. This profile could be integrated into a larger one for conceptual modeling. We explain below the main elements of the profile. The complete details of the stereotypes, constraints and additional operations (and their formalization in the OCL), tag definitions and tagged values of the profile can be found in [6].

3.1 Constraints

In the UML metamodel a *Generalization* is a taxonomic relationship between two *GeneralizableElements*: *child* and *parent* (Fig 1). In this paper we only deal with *GeneralizableElements* that are entity types, which we represent as classes with the standard stereotype <<type>> [15]. Sets of *Generalizations* sharing a given parent can be distinguished using the *Discriminator*.

A *Constraint* is an assertion (defined in the *body*) on a set of *ModelElements* that must be true in the information base (Fig 1). The UML has only a few predefined constraints. Among them, there are *complete* and *disjoint*.

Therefore, in the UML, a partition is represented by a set of *Generalizations* having the same parent and the same discriminator, and two predefined constraints (complete and disjoint) that have, as *constrainedElement*, those generalizations. However, it is convenient to have a single schema object representing a partition, to which we can attach properties and several rules. On the other hand, we need to have subtypes of *Constraint* corresponding to the taxonomic constraints, to which we can attach also properties and rules. To this end, we define in our profile the five stereotypes of *Constraint* shown in Fig 2.

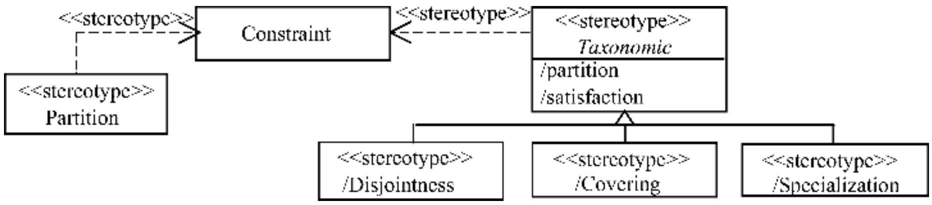

Fig. 2. Stereotypes of Constraint in the profile

The most important stereotype is <<partition>>. A single instance of a constraint with this stereotype will correspond to a partition in the conceptual schema. Graphically, this constraint will appear as shown in the examples of Fig 5.

The *constrainedElements* of a <<partition>> constraint must be a set of *Generalizations*. This is an example of a meta schema integrity constraint, also called "Well-Formedness Rules" in the UML metamodel, or invariants in database schema evolution [3]. The rules are expressed as constraints attached to stereotypes. In this case, we attach the constraint to *Partition*, and define it formally in the OCL:

```
context Partition inv:   --The constrained elements are Generalizations
  self.constrainedElement -> forAll(g | g.oclIsTypeOf(Generalization))
```

The other main constraints attached to *Partition* are (we only give their description):

- A partition has one or more generalizations.
- All generalizations belonging to the same partition must have the same parent and discriminator.
- All generalizations with the same parent and discriminator belong to the same partition.
- The generalizableElements of generalizations belonging to a partition are Types.
- A partition cannot have two generalizations with the same child.
- Two partitions with the same parent cannot have a generalization with the same child.

The *body* of a *Partition* will be empty and, therefore, it is not a real constraint. We will translate automatically a partition into the set of taxonomic constraints semantically equivalent to it. These constraints will be instances of the stereotypes <<disjointness>>, <<covering>> and <<specialization>>, shown in Fig 2. Instances of these stereotypes are constraints that must be satisfied in the information base, like any other instance of *Constraint*. The *body* of these constraints will be derived automatically, as shown in Section 3.3.

The constraint stereotype <<taxonomic>> is abstract, and serves only to define two common derived tags: *partition* and *satisfaction*. Unsurprisingly, *partition* gives the partition corresponding to the constraint; its value is defined when the instances are generated. *Satisfaction* can be *BySchema* or *Enforced*; its value is defined by a derivation rule explained in Section 3.3.

We define the stereotypes *Disjointness*, *Covering* and *Specialization* as derived, because their instances can be obtained automatically from the *Partitions* and their *Generalizations*. In the UML metamodel, *ModelElements* have a tag called *derived*. A true value indicates that it can be derived from other *ModelElements*. The details of derivation are given in an *Abstraction* dependency, with the standard stereotype <<derive>>, and name of the stereotype class Derivation [15]. A derivation dependency specifies that the client can be computed from the supplier. A derivation rule is an instance of *Derivation*. The expression of the rule is defined in the attribute *Mapping* (Fig 1). The expression can be defined formally in the OCL.

The expression corresponding to *Covering* would be:

```
Partition.allInstances -> forAll(p:Partition | Covering.allInstances→one(cov:Covering |
cov.partition = p and cov.constrainedElement = Sequence {p}))
```

The rule defines that for each *Partition p* there must be one (and only one) instance *cov* of *Covering* such that its *partition* tag has the value *p*, and its *constrainedElements* is the sequence consisting in only *p*. The derivation rules for *Disjointness* and *Specialization* are similar.

3.2 Derived Types

We need to distinguish between the three classes of derived entity types defined in Section 2.2, and therefore we define in our profile the three stereotypes of *Abstraction* shown in Fig 3: <<DerivedUnion>>, <<DerivedSpec>> and <<DerivedExcl>>. The first two are subtype of the standard *Derivation* (Fig 1), and the third one subtype of *DerivedSpec*. In the three cases, the client is the derived entity type.

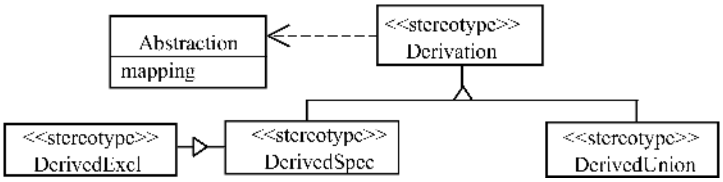

Fig. 3. Stereotypes of Abstraction dependency in the profile

The profile includes several meta schema integrity constraints concerning derived types and partitions. The main constraints attached to *DerivedUnion* are:

- A derived by union dependency must have at least one supplier.
- In a derived by union dependency, the client cannot be one of its direct or indirect suppliers.
- The suppliers of a derived by union dependency cannot be direct or indirect suppliers of themselves.
- The main constraints attached to *DerivedSpec* are:
- A derived by specialization dependency must have at least one supplier.
- In a derived by specialization dependency, the client cannot be one of its direct or indirect suppliers.
- The suppliers of a specialization dependency cannot be direct or indirect suppliers of themselves.
- The main constraint attached to *DerivedExcl* is:
- A derived by exclusion dependency must have at least two suppliers.

3.3 Satisfaction of Constraints

In Section 2.3, we have seen that some constraints are satisfied by the schema. The relationships between the derivability and schema satisfaction, are formalized by three OCL operations on *Type*, that are used in several parts of the profile and in the operations. The names, parameters and (short) description of the operations are:

Type::SpecSatisfiedBySchema (subtype:Type):Boolean
- True if the specialization constraint between subtype and self is satisfied by the schema.

Type::CovSatisfiedBySchema (subs:Set(Type)):Boolean
- True if the covering constraint between self and subs is satisfied by the schema.

Type::DisjSatisfiedBySchema (type:Type):Boolean
- True if the disjointness constraint between self and type is satisfied by the schema.

We have seen (Fig 2), that the instances of *Disjointness*, *Covering* and *Specialization* have a derived tag called *Satisfaction*, with values *BySchema* and *Enforced*. We define a derivation rule for *Satisfaction* in each of the three stereotypes. The rules can be expressed easily using the above operations. As an example, the rule for *Satisfaction* in *Covering* is:

```
context Covering:
  let supertype:Type = .... -- Gives the supertype of the Covering constraint
  let subtypes:Set(Type) = .... -- Gives the set of subtypes of the Covering constraint
  in self.Satisfaction = if supertype.CovSatisfiedBySchema(subtypes) then Satisfaction::BySchema
   else Satisfaction::Enforced endif
```

Note that, for a given constraint, the value of the *Satisfaction* attribute may change automatically if there is an evolution in the derivability of an involved entity type, or in the composition of the partition. This is one of the advantages of derived attributes of schema objects: The operations need not to be concerned with the effect of changes on them. The effects are defined declaratively in a single place of the profile.

We take a similar approach for the definition of the *body*. Fig 2 shows that the instances of *Disjointness*, *Covering* and *Specialization* are *Constraints* and, therefore, have the attribute *body*. The value of this attribute is an OCL expression corresponding to the constraint that must be satisfied by the information base. We define a derivation rule for *body* in each of the three stereotypes. The rules can be defined easily using the above operations.

The generated expression is tailored to each particular constraint, so that its evaluation can be performed efficiently. We distinguish between constraints satisfied by the schema and those to be enforced. The former have an empty *body*, because they need not to be enforced at runtime. The body for the latter is the specific constraint that must be enforced. For example the covering constraint between *Person* and *{Woman, Man}* not satisfied by the schema, would have for the *body* the value:

```
"Woman.allInstances -> union(Man.allInstances) -> includesAll(Person.allInstances)"
```

which means that the population of *Person* must be included in the union of populations of *Woman* and *Man*.

4 Evolving Partitions

In this section, we present the operations that we need to evolve partitions. We adopt the classical framework with the reflective architecture [9, 11, 17, 10] shown in Fig 4. In our case, the meta conceptual schema is the UML metamodel and the Profile presented in the previous section. The meta external events are the operations presented in this section and in the following one. The effects of these operations are a changed meta information base (conceptual schema or UML model) and, if required, a changed information base. The framework is very general and it allows an easy adaptation to an implementation environment in which both processors are integrated or tightly coupled, or which both information bases are integrated.

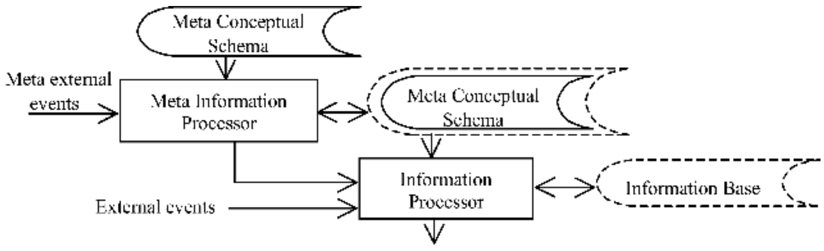

Fig. 4. Framework for the evolution

4.1 Changes to Partitions

The list of evolution operations of partitions is as follows:

1. Creating a partition: allows the designer to define a new partition in the UML schema with one supertype, a set of subtypes and a discriminator.
2. Adding a subtype to a partition: allows the designer to add an empty entity type as a subtype of an existing partition.
3. Removing a subtype from a partition: allows the designer to remove an empty entity type as a subtype of an existing partition.
4. Replacing subtypes: allows the designer to replace a set of subtypes of a given partition by another one.
5. Resizing a partition: allows the designer to add (to remove) a non empty subtype to (from) a partition where the supertype is derived by union of its subtypes.
6. Removing a partition: allows the designer to remove an existing partition.

In the next subsections we describe and justify each of the above operations, and give an intuitive explanation of their pre and postconditions. Preconditions are conditions that must be satisfied when an invocation of the operation occurs. Postconditions define the conditions that are satisfied when the operation finishes. Additionally, and implicitly, the execution of the operations:

- must maintain the meta schema integrity constraints defined in the stereotypes, and
- may induce effects on the schema and/or the information base defined by the derivation rules attached to the stereotypes.

Due to space limitations, we include the formal specification of only one operation.

4.2 Creating Partitions

The operation *AddPartition* allows the designer to define a new partition of existing entity types in a conceptual schema. The parameters are the supertype, a set of one or more subtypes and a discriminator.

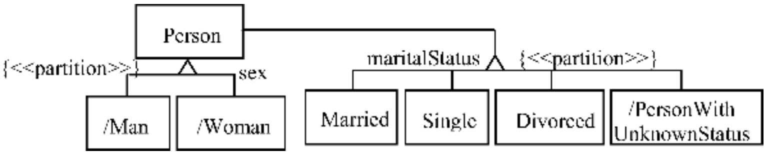

Fig. 5. Two examples of partitions

There are many situations in which it is necessary to add new partitions. For example, assume that our conceptual schema has already a partition of *Person* into *Man* and *Woman*, where *Person* is base, and *Man* and *Woman* are derived by specialization of *Person*. Assume that now we need to define a new partition of *Person* into the set of base entity types {*Single*, *Married, Divorced*}. The information base contains already some instances of *Person*, but it does not know yet their marital status. Initially, then, the population of *Single*, *Married* and *Divorced* will be empty, which implies that the partition is not possible. We decide then to include a fourth, temporary entity type in the partition, that we call *PersonWithUnknownStatus*, and that we define as derived by specialization of *Person* with the exclusion of *Single, Married* and *Divorced* (Fig. 5). The idea is to have initially all persons automatically instance of *PersonWithUnknownStatus*, and ask the users to enter progressively the marital status. The preconditions must ensure that the taxonomic constraints equivalent to the partition will be satisfied in the information base after the operation is executed. Otherwise, the information base would enter in an inconsistent state. The main preconditions are:

- *SpecAreSatisfied*: The instances of each subtype must be a subset of the instances of the supertype.
- *DisjAreSatisfied*: The instances of the subtypes must be mutually disjoint.
- *CovIsSatisfied*: The instances of the supertype must be covered by the union of the instances of the subtypes.

In fact, it is not necessary to test all the taxonomic constraints, but only those that are not satisfied by the schema.

In the example, we will only need to check that *Single, Married* and *Divorced* are subsets of *Person*, and that they are mutually disjoint. These checks are performed by querying the information base. In our framework (Fig. 4) this means that the meta information processor issues a query to the information processor, which is the only one that can access the information base. In the example, the checks will be very easy because the population of *Single*, *Married* and *Divorced* is initially empty.

The postconditions guarantee that a new partition will be created, a generalization will be created for each subtype, and the constrained elements of the partition will be the set of generalizations just created. The OCL definition of the operation is:

```
context Partition::AddPartition (discriminator:Name, super:Type, subs:Set(Type))
  pre: subs -> notEmpty()  -- There must be at least one subtype
  pre SpecAreSatisfied:
  subs -> forAll(sub:Type | not super.SpecSatisfiedBySchema(sub) implies
  super.allInstances -> includesAll (sub.allInstances))
  pre DisjAreSatisfied: -- Pairs of types in subs must be mutually disjoint. We avoid duplicate checks.
    let subsSeq:Sequence(Type) = subs -> asSequence()
    let numberSubtypes:Integer = subs -> size()
    in Sequence {1..numberSubtypes} ->
    forAll (i, j:Integer |  i > j and not subsSeq->at(i).DisjSatisfiedBySchema(subsSeq->at(j))
    implies
    subsSeq -> at(i).allInstances -> excludesAll (subsSeq -> at(j).allInstances))

  pre CovIsSatisfied:
    not super.CovSatisfiedBySchema(subs) implies
   subs -> iterate(sub:Type; acc:Set(Type) = Set{} | acc->union(sub.allInstances))->
   includesAll(super.allInstances)
  post:
p. oclIsNew() and p.oclIsTypeOf(Partition) –A new partition is created
and  -- Create a generalization for each subtype
subs -> forAll(sub:Type | ge.oclIsNew() and ge.oclIsTypeOf(Generalization)
and ge.discriminator = discriminator and ge.child = sub
and ge.parent = super  and p.constrainedElement -> includes(ge))
```

In addition to the effects defined by the postconditions, the execution of the operation induces other effects on the schema, as defined by the Profile derivation rules. In this operation, the induced effects are:

- For each *Generalization* created by the operation, an instance of *Specialization* is created as well.
- For each pair of subtypes of the new partition, an instance of *Disjointness* is created as well.
- A new instance of *Covering* is created.
- In the three cases, the new instances are associated to the *ModelElements* that they constrain and to the partition that originates them. The attribute *body* has as value an OCL expression corresponding to the constraint that must be satisfied by the information base, and the tag *satisfaction* has as value *BySchema* or *Enforced*, depending on whether the constraint is already satisfied by the schema, or needs to be enforced.

4.3 Adding a Subtype to a Partition

The operation *AddSubtype* allows the designer to add an empty entity type as a subtype of an existing partition. The parameters are the partition and the subtype.

There are many situations in which it is necessary to add a subtype to an existing partition. In the example of the partition of *Person* by *maritalStatus*, shown in Fig 5, we may be interested now in other marital status such as, for instance, *Widower*. We define then a new entity type and add it to the partition.

The main precondition of the operation is that the new subtype has no instances and, therefore, the new taxonomic constraints equivalent to the changed partition will be necessarily satisfied in the information base after the operation has been executed. The postconditions guarantee that a new generalization will be created, and that it will be added to the constrained elements of the partition.

4.4 Removing a Subtype from a Partition

The operation *RemoveSubtype* allows the designer to remove an empty entity type as a subtype of an existing partition. The parameters are the partition and the subtype.

This operation is the inverse of the previous one. In the example of the partition of *Person* by *maritalStatus*, shown in Fig 5, we may already know the marital status of each person and, therefore, *PersonWithUnknownStatus* is automatically empty. We can then remove it from the partition.

The main precondition of the operation is that the subtype to be removed has no instances and, therefore, the new taxonomic constraints equivalent to the changed partition will be necessarily satisfied in the information base after the operation has been executed. The postconditions guarantee that the corresponding generalization will be deleted, and that it will be removed from the constrained elements of the partition.

4.5 Replacing Subtypes

The operation *ReplaceSubtypes* allows the designer to replace a set of subtypes of a given partition by another one. The parameters are the partition, the old set, and the new set. There are several situations in which the designer may need to evolve a partition using this operation. We explain one of them in our example. Assume that we have in the schema the partition of *Person* by *maritalStatus*, but now we need to group the subtypes *Single*, *Divorced* and *Widower* into a new entity type *Unmarried*, and want also to change the original partition to one with only two subtypes: *Married* and *Unmarried* (see Fig 6).

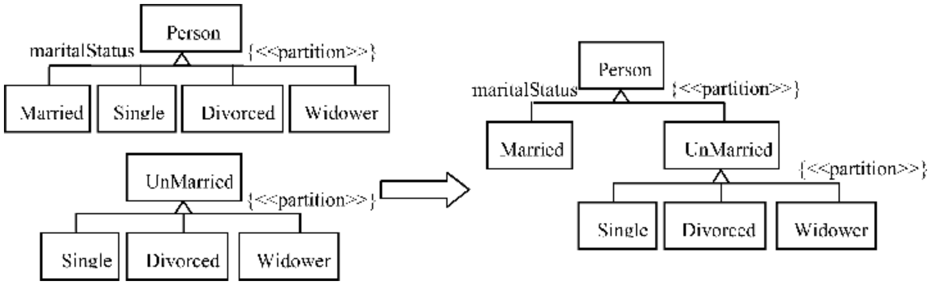

Fig. 6. Example of the Replacing operation: In the partition of *Person* by *maritalStatus*, the set {*Single*, *Divorced*, *Widower*} is replaced by {*UnMarried*}

This operation must satisfy two main preconditions in order to ensure that, after the operation has been executed, the taxonomic constraints equivalent to the partition remain satisfied:

- The instances of the set of new subtypes must be mutually disjoint.
- The union of the populations of the set of old subtypes must be the same as the union of the new subtypes. This condition preserves the covering constraint, as well as the disjointness with the unaffected subtypes.

When the old set and the new set are the super and the subtypes of an existing partition preconditions need not to be checked, because they are already guaranteed.

The postconditions guarantee that the old generalizations are removed, the new ones created, and the constrained elements of the partition have the new value.

4.6 Resizing a Partition

The operations of *Adding*, *Removing* and *Replacing* subtypes allow us to restructure a partition provided that the population of the supertype remains unchanged. However, there are cases in which we need to restructure a partition with the effect of increasing or decreasing the population of the supertype.

Assume, for example, a library that loans books and journals. The library has also audio CD, but they cannot be loaned. The corresponding conceptual schema (Fig 7) includes the base entity types *Book*, *Journal*, *AudioCD* and *ItemOnLoan*. Entity type *Book* is partitioned into *LoanableBook* and *NonLoanableBook*, both of which are derived. Entity type *LoanableItem* is defined as the union of *LoanableBook* and *Journal*. There is also a partition of *LoanableItem* into *ItemOnLoan* and *AvailableItem*. This latter is defined as derived by specialization of LoanableItem with the exclusion of *ItemOnLoan.*

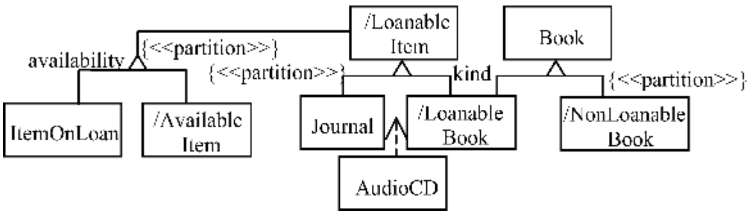

Fig. 7. Example of Resizing partition

Now, the policy of the library changes, and it allows loans of *AudioCD*. Therefore, we have to evolve the existing partition, but this cannot be done with the *Adding* operation, because the population of *AudioCD* is not empty. The operation *Resize* allows us to do that in a controlled manner.

The operation *Resize* can be applied only to partitions whose supertype is derived by union of its subtypes. The operation allows the designer to add or to remove a subtype, and to change simultaneously the derivation rule of the supertype. The overall effect is that the population of the supertype has been expanded or contracted. The operation has two parameters: the partition and the non-empty subtype to be added to or removed from it.

The are two non-trivial preconditions for this operation. The first is that if the partition is being expanded, the new subtype must be disjoint with the population of the existing supertype. The specialization and covering constraints are already guaranteed by the derivation rule (union) of the supertype. In the example, this means checking that *AudioCD* is disjoint with the existing population of *LoanableItem*.

The execution of the operation changes (adds or reduces) the population of the supertype. In general, such change could affect other constraints, which can be or not taxonomic. We, therefore, only allow resizing a partition if the resulting change cannot affect any other constraint. We check this in the second precondition. In the example, this could happen, for instance, if *LoanableItem* is a subtype in another partition (the disjointness or the specialization constraints could be violated) or if it is a supertype of other partitions (the covering constraint could be violated). Fig 7 shows that *LoanableItem* is the supertype of another partition into *ItemOnLoan* and *AvailableItem*, but no constraint is affected in this case because *AvailableItem* is derived by exclusion. The existing audios CD become initially, and automatically, available items.

The postconditions ensure that a generalization has been created (deleted), the constrained elements of the partitions have the new value, and that the derivation rule of the supertype has been changed.

4.7 Removing a Partition

The operation *Remove* allows the designer to remove an existing partition. The parameter is the partition. The operation has no preconditions. The postconditions ensure that the generalizations corresponding to a partition are deleted, as well as the partition itself.

5 Evolving the Derivability of Entity Types

We have seen, in the previous sections, that the derivability of the supertype and the subtypes involved in a partition has a strong impact on the satisfaction (by the schema, enforced) of the equivalent taxonomic constraints. This means that a complete account of the evolution of partitions needs to consider the operations for the evolution of derivability. In this respect, we define two operations: one that changes the derivability to base, and one that changes it to derived.

5.1 Changing Derivability to Base

The operation *ChangeDerivabilityToBase* allows the designer to change the derivability of a derived entity type to base. The only parameter of the operation is the entity type.

The only precondition of this operation is that the entity type must not be base. The main problem with this operation lies in its postconditions; more precisely, in what happens to the population of the changed entity type. Several strategies are possible in this respect. One strategy that seems appropriate in the context of partitions is to assume that the population will not change.

The implementation of this postcondition in our framework (Fig 4) requires that the meta information processor issues an event to the information processor, with the intended effect of materializing the population existing at the moment the operation is executed.

5.2 Changing Derivability to Derived

The operation *ChangeDerivabilityToDerived* allows the designer to define a base entity type as derived, or to change the derivation rule of a derived entity type. We are in the context of partitions, and we need to ensure that initially the population remains unchanged. To check this with our preconditions, we require two entity types: the one that has to be changed, *E*, and another one, *E'*, that we call the model, with the derivation rule that we want to assign to *E*. The parameters of the operation are the affected entity type (*E*) and the model (*E'*).

The preconditions must ensure that the population of both entity types are initially the same. The postconditions guarantee that the affected entity type will have the desired derivation rule (that is, the one that the model has).

6 Conclusions

The evolution of information systems from their conceptual schemas is one of the important research areas in information systems engineering. In this paper, we aim at contributing to the area by focusing on a particular conceptual modeling construct, the partitions. We have determined the possible evolutions of partitions in a conceptual schema, and we have defined, for each of them, the preconditions that must be satisfied, and the resulting postconditions. We take into account the state of, and the impact on, both the conceptual schema and the information base.

We have dealt with conceptual schemas in the UML. The choice of the language has been based on its industrial diffusion and the current (and future) availability of CASE tools. However, our results could be adapted easily to other conceptual modeling languages.

We have needed to extend the UML to deal with partitions and their evolution. We have done so using the standard extension mechanisms provided by the language itself. We have adhered strictly to the standard, and we have defined a UML Profile for Partitions in Conceptual Modeling. The profile allows us to define partitions in conceptual schemas, the taxonomic constraints equivalent to them, and the way how they can be satisfied. We hope that the approach we have taken to define particular classes of constraints, and the automatic derivation of schema objects and their attributes may be useful in future developments of UML Profiles.

We have dealt with partitions in conceptual models that include derived types (with different kinds of derivability), multiple specialization and multiple classification. We have taken into account all these elements in our evolution operations. However, the operations could be adapted (and, hopefully, be useful) to more restrictive contexts, such as those of object-oriented database schemas.

The work reported here can be continued in several directions. We mention three of them here. The first could be to define operations to evolve taxonomies in general. In this paper, we have focused on partitions only, due to their special characteristics that have not been studied before. The inclusion of other well known operations related to taxonomies (add/remove an entity type, add/remove a generalization, etc.) should not be difficult. Compound operations could be defined also. The second continuation could be to extend the profile with other known conceptual modeling constructs, with the aim of developing a complete UML Profile for Conceptual Modeling. The corresponding evolution operations could be defined as well, in the line of the ones described here. The third continuation could be to take into account the temporal aspects of conceptual schemas [10], and to develop a UML Profile for Temporal Conceptual Modeling.

Acknoledgments

We would like to thank Juan Ramon López, Dolors Costal, Maria Ribera Sancho and Ernest Teniente for his useful comments. This work has been partly supported by CICYT program project TIC99-1048-C02-1.

References

1. Al-Jadir, L.; Léonard, M. "Multiobjects to Ease Schema Evolution in an OODBMS", Proc. ER'98, Singapore, LNCS 1507, Springer, pp. 316-333.
2. Andrade, L.F.; Fiadeiro, J.L. “Coordination Technologies for Managing Information System Evolution”, CAiSE 2001, LNCS 2068, pp. 374-387.
3. Banerjee, J.; Chou, H-T.; Garza, J.F.; Kim, W.; Woelk, D.; Ballou, N. “Data Model Issues for Object-Oriented Applications”. ACM TOIS Vol. 5, No. 1, January, pp. 3-26.
4. de Champeaux, D.; Lea, D.; Faure, P. “Object-Oriented System Development”, Addison-Wesley Pub. Co.
5. Franconi, E.; Grandi, F.; Mandreoli, F. “Schema Evolution and Versioning: A Logical and Computational Characterisation”, In Balsters, H.; de Brock, B.; Conrad, S. (eds.) “Database Schema Evolution and Meta-Modeling”, LNCS 2065, pp. 85-99.
6. Gómez, C., Olivé A; "Evolving Partitions in Conceptual Schemas in the UML (Extended Version)",Technical Report UPC, LSI-02-15-R.
7. Goralwalla, I.; Szafron, D.; Özsu, T.; Peters, R. “A Temporal Approach to Managing Schema Evolution in Object Database Systems”. Data&Knowledge Eng. 28(1), October, pp. 73-105.
8. Hainaut, J-L.; Englebert, V.; Henrard, J.; Hick, J-M.; Roland, D. "Database Evolution: the DB-MAIN Approach". 13th. Intl. Conf. on the Entity-Relationship Approach - ER'94, LNCS 881, Springer-Verlag, pp. 112-131.
9. ISO/TC97/SC5/WG3. "Concepts and Terminology for the Conceptual Schema and Information Base", J.J. van Griethuysen (ed.), March.
10. López, J-R.; Olivé, A. “A Framework for the Evolution of Temporal Conceptual Schemas of Information Systems”, CAiSE 2000, LNCS 1789, pp. 369-386.
11. Manthey, R. "Beyond Data Dictionaries: Towards a Reflective Architecture of Intelligent Database Systems", DOOD'93, Springer-Verlag, pp. 328-339.
12. Mens, T.; D’Hondt, T. “Automating Support for Software Evolution in UML”, Automated Software Engineering, 7, pp. 39-59.
13. Olivé, A.; Costal, D.; Sancho, M-R. "Entity Evolution in ISA Hierarchies", ER'99, LNCS 1728, pp. 62-80.
14. Olivé, A. "Taxonomies and Derivation Rules in Conceptual Modelling", CAiSE 2001, LNCS 2068, pp. 417-432.
15. OMG. "Unified Modeling Language Specification", Version 1.4, September 2001.
16. Opdyke, W.F. “Refactoring object-oriented frameworks”, PhD thesis, University of Illinois.

17. Peters, R.J.; Özsu, T. "Reflection in a Uniform Behavioral Object Model". Proc. ER'93, Arlington, LNCS 823, Springer-Verlag, pp. 34-45.
18. Peters, R.J., Özsu, M.T. "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems", ACM TODS, 22(1), pp. 75-114.
19. Roddick, J.F. "A Survey of Schema Versioning Issues for Database Systems", Inf. Softw. Technol, 37(7), pp. 383-393.
20. Rumbaugh, J.; Jacobson, I.; Booch, G. "The Unified Modeling Language Reference Manual", Addison-Wesley, 550 p.
21. Smith, J.M.; Smith, D.C.P. "Database Abstractions: Aggregation and Generalization". ACM TODS, 2,2, pp. 105-133.
22. Sunyé, G.; Pennaneac'h, F.; Ho, W-M.; Le Guennec, Al.; Jézéquel, J-M. "Using UML Action Semantics for Executable Modeling and Beyond", CAiSE 2001, LNCS 2068, pp. 433-447.
23. Tokuda, L.; Batory, D. "Evolving Object-Oriented Designs with Refactorings", Automated Software Engineering, 8, pp. 89-120.339
24. Tresch, M.; Scholl, M.H. "Meta Object Management and its Application to Database Evolution", 11th. Intl. Conf. on the Entity-Relationship Approach - ER'92, LNCS 645, Springer-Verlag, pp. 299-321.
25. Wieringa, R.; de Jonge, W.; Spruit, P. "Using Dynamic Classes and Role Classes to Model Object Migration", TPOS, Vol 1(1), pp. 61-83.
26. Zicari, R. "A Framework for Schema Updates in Object-Oriented Database System", in Bancilhon,F.; Delobel,C.; Kanellakis, P. (ed.) "Building an Object-Oriented Database System - The Story of O2", Morgan Kaufmann Pub., pp. 146-182.