Evolving Partitions in Conceptual Schemas in the UML (Extended Version)

Cristina Gómez, Antoni Olivé

Universitat Politècnica Catalunya
Departament de Llenguatges i Sistemes Informàtics
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.

Keywords

Conceptual schemas, evolution, meta schemas, profiles
1 INTRODUCTION .................................................................1

2 PARTITIONS .................................................................2

2.1 Taxonomic Constraints and Partitions ................................3
2.2 Derived Types ............................................................3
2.3 Satisfaction of Partition Constraints .................................4

3 UML PROFILE FOR PARTITIONS ........................................5

3.1 Constraints ...............................................................5
3.2 Derived Types ...........................................................8
3.3 Satisfaction of Constraints ............................................9

4 EVOLVING PARTITIONS ................................................10

4.1 Changes to Partitions ..................................................10
4.2 Creating Partitions .....................................................11
4.3 Adding a subtype to a partition ......................................13
4.4 Removing a subtype from a partition ...............................13
4.5 Replacing subtypes .....................................................14
4.6 Resizing a partition .....................................................15
4.7 Removing a partition ....................................................16

5 EVOLVING THE DERIVABILITY OF ENTITY TYPES .......................16

5.1 Changing derivability to base .......................................16
5.2 Changing derivability to derived .................................17

6 CONCLUSIONS ...........................................................18

ACKNOWLEDGMENTS ......................................................19

REFERENCES ..............................................................19

APPENDIX .................................................................21
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 [AnFi01]. Automated support for information system evolution becomes central to satisfy these evolution requirements [MeDH00].

Ideally, the evolution of information systems should follow the strategy called ‘forward information system maintenance’ in [HEH+94]: 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, and need not to be visible outside.

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) [PeÖz97]. Both aspects have been studied extensively for the relational and the object-oriented data models, in the temporal and the non-temporal variants [Rodd95]. The results have been often incorporated into commercial or prototype database systems (e.g., Orion [BCG+87], O₂ [Zica92], Cocoon [TrSc92], F₂ [AlLe98] and Tigukat [GSÖP98]).

More recently, in the software engineering field, the problem of software evolution is being dealt with a refactoring approach [Opdy92]. 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 [SPH+01]. The approaches in the databases and software engineering fields are similar, because database schema evolution transformations have their parallels in refactoring transformations [ToBa01].

In this paper, we aim at contributing to the general field of information systems evolution from conceptual schemas. We extend the work reported in [LoOl00] 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 [SmSm77, dCLF93, WidS95, OICS99].

However, partitions have not been studied in the literature on schema evolution. Since the early works of Orion [BCG+87], 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 [FrGM01], 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, in this way, 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 [LoO100].

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. Finally, Section 6 gives the conclusions and points out future work. The full details of the profile and operations can be found in the appendix.

2. PARTITIONS

In this section, we review briefly the basic concepts and the terminology that will be used throughout the paper, taken mainly from [Oliv01].
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 also be instance of $E$. For example, there is a specialization constraint between $Man$ and $Person$.

A disjointness constraint between entity types $E_1$ and $E_2$ means that the populations of $E_1$ and $E_2$ are disjoint. For example, there is a disjointness constraint between $Man$ and $Woman$.

Finally, a covering constraint between an entity type $E$ and a set of entity types $\{E_i, \ldots, E_n\}$, means that if an entity is instance of $E$, it must be also instance of at least one $E_i$. For example, there is a covering constraint between $Person$ and the set $\{Man, Woman\}$. Notice that a covering constraint between $E_i$ and $E$ does not imply a specialization constraint between $E_i$ and $E$. For example, we could have a covering constraint between $Employee$ and $\{Man, Woman\}$ even if not all men and women are employees.

A generalization corresponds to a set of specialization constraints between $E_i$ and $E$, for $i = 1, \ldots, n$, 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 $E_i, \ldots, E_n$; 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. For example, the partition of $Person$ into $Man$ and $Woman$. A partition of $E$ into $E_1, \ldots, E_n$ is semantically equivalent to:

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

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 $E_1, \ldots, E_n$, 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, \textit{Young} may be defined as a specialization of \textit{Person}, with the condition "age less than 18 years".

- Derived by \textit{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 $E_i, ..., E_n$, with $n \geq 1$. For instance, \textit{Unmarried} may be defined as specialization of \textit{Person}, excluding \textit{Married}.

- Derived by \textit{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, \textit{Person} may be defined as the union of \textit{Man} and \textit{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 \textit{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 \textit{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/postconditions), to be executed whenever the information base is updated.

An analysis of the taxonomic constraints satisfied by a schema is presented in [Oliv01]. 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 the set $\{E_1, ..., E_n\}$ 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 $E_j$. 
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. More examples and explanations of the above relationships can be found in [Oliv01].

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 [RuJB99], 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 the appendix.

3.1 Constraints

In the UML metamodel a Generalization is a taxonomic relationship between two GeneralizableElements: child and parent (Figure 1). In this paper we only deal with GeneralizableElements that are entity types, which we represent as classes with the standard stereotype <<type>> [OMG01, p.2-27]. 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 (Figure 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 Figure 2.

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 Figure 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 [BCG+87]. 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 as follows:

**context** Partition **inv:**

--The constrained elements are Generalizations
   self.constrainedElement -> forAll(g | goclIsTypeOf(Generalization))

The other main constraints attached to Partition are (we only give their description; see the appendix for the formal definition in the OCL):

- 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.

Other constraints are already guaranteed by the well-formedness rules of the UML metamodel. For example, the constraint that a generalizable element cannot be a direct or indirect subtype of itself [OMG01, p. 2-61].

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 Figure 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 the derivation are given in an Abstraction dependency, with the standard stereotype <<derive>>, and name of the stereotype class Derivation [OMG01, p.2-18]. 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 (Figure 1). The expression can be defined formally in the OCL.

The expression corresponding to Covering would be:

\[
\text{Partition.allInstances} \rightarrow \text{forAll}(p:\text{Partition} | \\
\text{Covering.allInstances} \rightarrow \text{one}(\text{cov:\text{Covering} | \\
\text{cov.partition} = p \text{ and} \\
\text{cov.constrainedElement} = \text{Sequence}\{p\}))
\]

The rule defines that for each Partition \( p \) there must be one (and only one) instance \( \text{cov} \) 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 Figure 3: <<DerivedUnion>>, <<DerivedSpec>> and <<DerivedExcl>>. The first two are subtype of the standard Derivation (shown in Figure 1), and the third one subtype of DerivedSpec. In the three cases, the client is the derived entity type.

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 (the constraints attached to DerivedSpec apply here as well):
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, Figure 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 some involved entity type, or in the composition of the partition. This is one of the advantages of derived attributes of schema objects: The evolution 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. Figure 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\}, if it is not satisfied by the schema, would have for the body attribute 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.

Note that the value of the body attribute may change automatically if there is an evolution in the derivability of some involved entity type, or in the composition of the partition.

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 [ISO82, Mant93, PeÖz93, LoOl00] shown in Figure 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, and also to an environment in which both information bases are integrated.

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 can include the formal specification in the OCL of only one operation. The full details of operations can be found in the appendix.

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.

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. The information system has been in use for some time, and 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 (the covering constraint would be violated). 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 (Figure 5). The idea is to have initially all persons automatically instance of PersonWithUnknownStatus, and ask the users to enter progressively the marital status. Later on, once known the marital status of each person, the entity type PersonWithUnknownStatus will be removed from the partition and the conceptual schema.
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 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 (Figure 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 other constraints (covering of *Person*, specialization of *PersonWithUnknownStatus* and *Person*, and the disjointness of *PersonWithUnknownStatus* and *Single*, *Married* and *Divorced*) are satisfied by the schema.

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.

Note that we have defined partitions in a very general context, which includes derived types (with different kinds of derivability), multiple specialization and multiple classification. Our operations could be adapted to more restrictive contexts, such as conceptual models with single classification or object-oriented database schemas.
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 Figure 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 operation does not change in any case the population of the supertype and, therefore, no indirect constraints can be affected.

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 Figure 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 operation does not change in any case the population of the supertype and, therefore, no indirect constraints can be affected.

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 Figure 6). In this case, we would execute three operations:

1. Define a new entity type Unmarried, derived by union of Single, Divorced and Widower.
2. Create a new partition of Unmarried into Single, Divorced and Widower, with the operation defined in Section 4.2.
3. In the partition of Person by maritalStatus, replace the subtypes Single, Divorced and Widower 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. The operation does not change in any case the population of the supertype and, therefore, no indirect constraints can be affected.

There is a particular case, which happens very often, for which the above preconditions need not to be checked, because they are already guaranteed: when the old set and the new set are the super and the subtypes of an existing partition. This is what happens in the example of Figure 6.

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 (see Figure 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.

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 (or, what is the same, with the union of the populations of the other subtypes). 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). Figure 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.

As for any other operation, the execution of Remove induces other effects on the schema, as defined by the derivation rules included in the Profile. In this case, the induced effects are:
- For each Generalization deleted by the operation, the associated instance of Specialization is deleted as well.
- For each pair of subtypes of the old partition, the associated instance of Disjointness is deleted as well.
- The associated instance of Covering is deleted.

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.
As an example, consider the partition of *Book* into *LoanableBook* and *NonLoanableBook*, shown in Figure 7. *Book* is base, *LoanableBook* is derived by specialization of *Book*, and *NonLoanableBook* is derived by specialization of *Book* with the exclusion of *LoanableBook*. The derivation rule of *LoanableBook* defines a book as loanable if it satisfies some condition. Assume now that the librarians want to have complete control on which books can be loaned. Now they prefer to communicate explicitly which books are loanable; that is, they want *LoanableBook* to be base. The execution of the operation *ChangeDerivabilityToBase* will achieve this objective. The *NonLoanableBooks* will still be those books that are not instance of *LoanableBook*.

The only noticeable precondition of this operation is that the entity type must not be base already.

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, and the one that we have adopted as postcondition, is to assume that the population will not change. In the example, this means that the population of *LoanableBook* will no be affected and, therefore, no taxonomic constraints will be affected neither.

The implementation of this postcondition in our framework (see Figure 4) requires that the meta information processor issue 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’*).

Assume, in the example of Figure 7, that we want to change the derivation rule of *LoanableBook*. We want to maintain it as derived by specialization of *Book*, but with a different condition. We define a new auxiliary entity type, that we call *NewLoanableBook*, with a derivation rule that will serve as model for *LoanableBook*. Then, we invoke the operation *ChangeDerivabilityToDerived* with parameters *LoanableBook* and *NewLoanableBook*. After the execution of the operation, the designer can remove the auxiliary entity type, if she wants to. The example shows that it would be convenient to have a single compound operation that creates the model entity type, invokes the operation *ChangeDerivabilityToDerived* and removes the model entity type. Compound operations are beyond the scope of this paper.
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 [LoO100], and to develop a UML Profile for Temporal Conceptual Modeling.
ACKNOWLEDGMENTS

We would like to thank Juan Ramon López for his useful comments. This work has been partly supported by CICYT program project TIC99-1048-C02-1.

REFERENCES

[SmSm77] Smith, J.M.; Smith, D.C.P. "Database Abstractions: Aggregation and Generalization", ACM TODS, 2,2, pp. 105-133.
APPENDIX: UML Profile and Evolving operations for Partitions in Conceptual Modeling

CONTENTS

This document contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction”</td>
<td>1</td>
</tr>
<tr>
<td>“Summary of Profile”</td>
<td>2</td>
</tr>
<tr>
<td>“Stereotypes and Notation”</td>
<td>3</td>
</tr>
<tr>
<td>“Well-Formedness Rules”</td>
<td>5</td>
</tr>
<tr>
<td>“Meta Derivation Rules”</td>
<td>6</td>
</tr>
<tr>
<td>“Additional Operations”</td>
<td>10</td>
</tr>
<tr>
<td>“Evolving operations”</td>
<td>12</td>
</tr>
</tbody>
</table>

1. INTRODUCTION

This profile is an extension to the UML, using the standard mechanisms provided by the language. This profile is needed to represent partitions, taxonomic constraints and different kinds of derived types in UML. To define it, we use stereotypes, tagged values, constraints and meta derivation rules attached to the stereotypes.
2. SUMMARY OF PROFILE

Figures 1 and 2 describe the stereotypes with its corresponding tags needed for the Partition profile. The semantic of these stereotypes is given in section 3.

Figure 1. Stereotypes of Constraint in the profile.

Figure 2. Stereotypes of Abstraction dependency in the profile.
3. STEREOTYPES AND NOTATION

In this section we define all the stereotypes with a briefly description of its semantics and tagged values.

3.1 Partition Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition &lt;&lt;partition&gt;&gt;</td>
<td>Constraint</td>
<td>NA</td>
<td>None</td>
<td>Constraints defined in section 4. No derivation rules.</td>
<td>Instances of this stereotype are partitions in the UML schema.</td>
</tr>
</tbody>
</table>

3.2 Taxonomic Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxonomic</td>
<td>Constraint</td>
<td>NA</td>
<td>Partition Satisfied</td>
<td>None</td>
<td>Instances of this stereotype serve only to define the two tags.</td>
</tr>
</tbody>
</table>

3.3 Specialization Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialization</td>
<td>Constraint</td>
<td>Taxonomic</td>
<td>None</td>
<td>No constraints. Derivation rules defined in section 5.</td>
<td>Instances of this stereotype are specialization constraints that must be satisfied in the information base.</td>
</tr>
</tbody>
</table>

3.4 Disjointness Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disjointness</td>
<td>Constraint</td>
<td>Taxonomic</td>
<td>None</td>
<td>No constraints. Derivation rules defined in section 5.</td>
<td>Instances of this stereotype are disjointness constraints that must be satisfied in the information base.</td>
</tr>
</tbody>
</table>
3.5 Covering Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covering</td>
<td>Constraint</td>
<td>Taxonomic</td>
<td>None</td>
<td>No constraints. Derivation rules defined in section 5.</td>
<td>Instances of this stereotype are covering constraints that must be satisfied in the information base.</td>
</tr>
</tbody>
</table>

3.6 DerivedUnion Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivedUnion</td>
<td>Abstraction</td>
<td>Derivation</td>
<td>None</td>
<td>Constraints defined in section 4. Derivation rules defined in section 5.</td>
<td>Instances of this stereotype are derivation dependencies where the client is a type that is derived by union and the suppliers are other types.</td>
</tr>
</tbody>
</table>

3.7 DerivedSpec Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivedSpec</td>
<td>Abstraction</td>
<td>Derivation</td>
<td>None</td>
<td>Constraints defined in section 4. No derivation rules.</td>
<td>Instances of this stereotype are derivation dependencies where the client is a type that is derived by specialization and the suppliers are other types.</td>
</tr>
</tbody>
</table>

3.8 DerivedExcl Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Parent</th>
<th>Tags</th>
<th>Constraints/Derivation Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivedExcl</td>
<td>Abstraction</td>
<td>DerivedSpec</td>
<td>None</td>
<td>Constraints defined in section 4. Derivation rules defined in section 5.</td>
<td>Instances of this stereotype are derivation dependencies where the client is a type that is derived by exclusion and the suppliers are other types.</td>
</tr>
</tbody>
</table>

3.9 Notation

The notation given as part of the UML specification stereotyped constraints can be used in this profile to represent partitions. In the figure we show this notation.

![Diagram](image)

Figure 3. Notation example of partitions.

24
4. WELL-FORMEDNESS RULES

The UML Specification relies on the use of well-formedness rules to express constraints on model elements. This profile uses the same approach.

4.1 Partition Stereotype

The well-formedness rules for this stereotype are the following:

**WFR01**: The constrained elements of a partition are generalizations

**context** Partition **inv**:

self.constrainedElement -> forAll(g | g.oclIsTypeOf(Generalization))

**WFR02**: A partition has one or more generalizations.

**context** Partition **inv**:

self.constrainedElement-> size>0

**WFR03**: All generalizations belonging to the same partition must have the same parent and discriminator.

**context** Partition **inv**:

self.constrainedElement.oclAsType(Generalization) ->forAll(g1,g2:Generalization |

  g1.parent= g2.parent
  and g1.discriminator=g2.discriminator)

**WFR04**: All generalizations with the same parent and discriminator belong to the same partition.

**context** Partition **inv**:

Generalization.allInstances->select(g1,g2:Generalization | g1<>g2 and g1.parent=g2.parent

  and g1.discriminator=g2.discriminator
  and g1.constraint.oclAsType(Partition)<<

  g2.constraint.oclAsType(Partition))-> isEmpty()

**WFR05**: The generalizableElements of generalizations belonging to a partition are Types.

**context** Partition **inv**:

self.constrainedElement.oclAsType(Generalization) ->forAll(g:Generalization |

  g.parent.oclIsTypeOf(Type)
  and g.child.oclIsTypeOf(Type))

**WFR06**: A partition cannot have two generalizations with the same child.

**context** Partition **inv**:

self.constrainedElement.oclAsType(Generalization) ->forAll(g1,g2:Generalization |

  g1<>g2 implies

  g1.child<>g2.child)

**WFR07**: Two partitions with the same parent cannot have a generalization with the same child.

**context** Partition **inv**:

Generalization.allInstances->select(g1,g2:Generalization | g1<>g2

  and g1.child=g2.child
  and g1.parent=g2.parent
  and g1.discriminator<>g2.discriminator) ->isEmpty()
Other constraints are already guaranteed by the well-formedness rules of the UML metamodel. For example, the constraint that a generalizable element cannot be a direct or indirect subtype of itself.

### 4.2 DerivedUnion Stereotype

**WFR08:** A derived by union dependency must have at least one supplier.

```plaintext
context DerivedUnion inv:
self.supplier-> size>0
```

**WFR09:** In a derived by union dependency, the client cannot be one of its direct or indirect suppliers.

```plaintext
context DerivedUnion inv:
self.allSuppliers-> excludes(self)
```

**WFR10:** In a derived by union dependency, the suppliers cannot be direct or indirect suppliers of themselves.

```plaintext
context DerivedUnion inv:
self.allSuppliers-> select(sup1,sup2:ModelElement |
  sup1.allSuppliers->includes(sup2)
  and sup2.allSuppliers->includes(sup1)) ->isEmpty()
```

The operation allSuppliers is an additional operation that we define in section 6.

### 4.3 DerivedSpec Stereotype

**WFR11:** A derived by specialization dependency must have at least one supplier.

```plaintext
context DerivedSpec inv:
self.supplier-> size>0
```

**WFR12:** In a derived by specialization dependency, the client cannot be one of its direct or indirect suppliers.

```plaintext
context DerivedSpec inv:
self.allSuppliers-> excludes(self)
```

**WFR13:** In a derived by specialization dependency, the suppliers cannot be direct or indirect suppliers of themselves.

```plaintext
context DerivedSpec inv:
self.allSuppliers-> select(sup1,sup2:ModelElement |
  sup1.allSuppliers->includes(sup2)
  and sup2.allSuppliers->includes(sup1)) ->isEmpty()
```

The operation allSuppliers is an additional operation that we define in section 6.

### 4.4 DerivedExcl Stereotype

**WFR14:** A derived by exclusion dependency must have at least one supplier.

```plaintext
context DerivedExcl inv:
self.supplier-> size>1
```
5. META DERIVATION RULES

The meta derivation rules are derivation rules needed in the metamodel. In this profile, we define some meta derivation rules attached to some stereotypes (Specialization, Disjointness, Covering, DerivedUnion and DerivedExcl). Meta derivation rules are represented in the metamodel in the same manner than derivation rules. For every meta derivation rule we define a stereotype of abstraction with a mapping Expression that represents the formula that calculates the client from the suppliers. The client is the model Element that is calculated.

5.1 Specialization Stereotype

We show in the following the meta derivation rules for this stereotype:

MDR01: For each Partition p and for each Generalization g there must be one (and only one) instance spec of Specialization such that its partition tag has the value p and its constrainedElements is the sequence consisting in only g.

context Specialization:
Partition.allInstances->forall(p:Partition | p.constrainedElement.oclAsType(Generalization)->
forall(g:Generalization |
Specialization.allInstances-> one(spec:Specialization |
spec.partition = p and
spec.constrainedElement = Sequence {g})))

MDR02: This derivation rule calculates for the body attribute a string that, if the specialization constraint is satisfied by the schema, the body constraint will be empty and if the constraint must be enforced, the body constraint will have an OCL expression corresponding to the constraint that must be satisfied by the information base.

context Specialization:
self.body = not self.constrainedElement.parent.SpecSatisfiedBySchema(self.constrainedElement.child)
implies
self.constrainedElement.parent.Name.oclAsType(String).concat('.allInstances->
includesAll(').concat(self.constrainedElement.child.Name.oclAsType(String)).concat('.allInstances)')

MDR03: This derivation rule calculates the value for the Satisfaction attribute. Satisfaction can be BySchema when the specialization constraint is satisfied by the schema or Enforced when the constraint must be enforced in the information base.

context Specialization:
self.Satisfaction =
if self.constrainedElement.parent.SpecSatisfiedBySchema(self.constrainedElement.child) then
Satisfaction::BySchema
else Satisfaction::Enforced endif

The operation SpecSatisfiedBySchema is an additional operation that we define in section 6.
5.2 Disjointness Stereotype

We show in the following the meta derivation rules for this stereotype:

**MDR04:** For each Partition \( p \) and for each pair of subtypes there must be one (and only one) instance \( \text{disj} \) of Disjointness such that its \( \text{partition} \) tag has the value \( p \) and its \( \text{constrainedElements} \) is the sequence consisting in the two subtypes.

**context** Disjointness:

\[
\text{partition.allInstances} \rightarrow \text{forAll}(p:\text{Partition} | \\
\quad \text{let subsSeq:Sequence(Type) = p.hasSubtypes} \rightarrow \text{asSequence()} \\
\quad \text{let numberSubtypes:Integer = p.hasSubtypes} \rightarrow \text{size()} \\
\quad \text{in Sequence} \{1..\text{numberSubtypes}\} \rightarrow \text{forAll}(i,j:Integer \mid i<j \\
\quad \quad \text{Disjointness.allInstances} \rightarrow \\
\quad \quad \quad \text{one}(\text{disj:Disjointness} \mid \text{disj.partition} = p \\
\quad \quad \quad \quad \text{and disj.constrainedElement} = \text{Sequence} \\
\quad \quad \quad \quad \quad \{ \text{subsSeq} \rightarrow \text{at}(i), \text{subsSeq} \rightarrow \text{at}(j) \})
\]

**MDR05:** This derivation rule calculates for the \( \text{body} \) attribute a string, that if the disjointness constraint is satisfied by the schema, the body constraint will be empty and if the constraint must be enforced, the body constraint will have an OCL expression corresponding to the constraint that must be satisfied by the information base.

**context** Disjointness:

\[
\text{self.body} = \text{not self.constrainedElement} \rightarrow \text{first()}.\text{DisjSatisfiedBySchema} (\text{self.constrainedElement} \rightarrow \text{last()}) \text{ implies} \\
\text{self.constrainedElement} \rightarrow \text{first()}.\text{Name.oclAsType(String)}.\text{concat} (\text{.allInstances} \rightarrow \\
\quad \text{excludesAll(‘’.concat(\text{self.constrainedElement} \rightarrow \text{last()}.\text{Name.oclAsType(String)}.\text{concat} (\text{.allInstances}))))}
\]

**MDR06:** This derivation rule calculates the value for \( \text{Satisfaction} \) attribute. \( \text{Satisfaction} \) can be BySchema when the disjointness constraint is satisfied by the schema or Enforced when the constraint must be enforced in the information base.

**context** Disjointness:

\[
\text{self.Satisfaction} = \\
\quad \text{if self.constrainedElement} \rightarrow \text{first()}.\text{DisjSatisfiedBySchema} (\text{self.constrainedElement} \rightarrow \text{last()}) \text{ then} \\
\quad \text{Satisfaction::BySchema} \\
\quad \text{else Satisfaction::Enforced endif}
\]

The operations DisjSatisfiedBySchema, hasSuper and hasSubtypes are additional operations that we define in section 6.

5.3 Covering Stereotype

We show in the following the meta derivation rules for this stereotype:

**MDR07:** For each Partition \( p \) there must be one (and only one) instance \( \text{cov} \) of Covering such that its \( \text{partition} \) tag has the value \( p \) and its \( \text{constrainedElements} \) is the sequence consisting in only \( p \).

**context** Covering:
MDR08: This derivation rule calculates for the body attribute a string that if the covering constraint is satisfied by the schema, the body constraint will be empty and if the constraint must be enforced, the body constraint will have an OCL expression corresponding to the constraint that must be satisfied by the information base.

**context** Covering:
let subsSeq:Sequence(Type)= self.constrainedElement.hasSubtypes->asSequence()
let numberSubtypes:Integer=self.constrainedElement.hasSubtypes->size()
in
self.body = not self.constrainedElement-> hasSuper.CovSatisfiedBySchema(self.constrainedElement->
  hasSubtypes) implies
  Sequence {1..numberSubtypes-1}->iterate(i:Integer; acc:String='''
    acc.concat(subsSeq-> at(i).Name.oclAsType(String)).
    concat('.allInstances-> union(')).
    concat(subsSeq-> at(numberSubtypes).Name.oclAsType(String)).
    concat('.allInstances-> includesAll(').
    concat(self.constrainedElement.hasSuper.Name.oclAsType(String)).concat('"allInstances")
  )

MDR09: This derivation rule calculates the value for Satisfaction attribute. Satisfaction can be BySchema when the covering constraint is satisfied by the schema or Enforced when the constraint must be enforced in the information base.

**context** Covering:
self.Satisfaction =
if self.constrainedElement.hasSuper.CovSatisfiedBySchema(self.constrainedElement.hasSubtypes)
then
  Satisfaction::BySchema
else Satisfaction::Enforced endif

The operations CovSatisfiedBySchema, hasSuper and hasSubtypes are additional operations that we define in section 6.

5.4 DerivedUnion Stereotype

MDR10: This derivation rule calculates the value for Mapping attribute. This attribute has the derivation rule that must be satisfied in the information base. In this case the attribute indicates that the instances of the client must be the union of the instances of the suppliers.

**context** DerivedUnion:
let subsSeq:Sequence(Type)= self.supplier->asSequence()
let numberSubtypes:Integer=self.supplier->size()
in
self.Mapping =
self.client.Name.oclAsType(String).concat('"allInstances-> includesAll".concat(
  Sequence {1..numberSubtypes-1}->iterate(i:Integer; acc:String="''
    acc.concat(subsSeq-> at(i).Name.oclAsType(String)).
    concat('"allInstances-> union").  
    concat(subsSeq-> at(numberSubtypes).Name.oclAsType(String)).concat('"allInstances")
  )

29
5.5 DerivedExcl Stereotype

**MDR11:** This derivation rule calculates the value for Mapping attribute. This attribute has the derivation rule that must be satisfied in the information base. In this case the attribute indicates that the instances of the client must be the instances of the first supplier except the instances of the other suppliers.

```plaintext
context DerivedExcl:
let subsSeq:Sequence(Type)= self.supplier->asSequence()
let numberSubtypes:Integer=self.supplier->size()
in
self.Mapping =
self.client.Name.oclAsType(String).concat('.allInstances->includesAll(').
concat(self.supplier->first().Name.oclAsType(String)).
concat('.allInstances->excludesAll(').concat(Sequence {2..numberSubtypes-1}->iterate(i:Integer; acc:String=''
| acc.concat(subsSeq-> at(i).Name.oclAsType(String)).
concat('.allInstances-> union(').
concat(subsSeq-> at(numberSubtypes).Name.oclAsType(String)).concat('.allInstances')
```

6. ADDITIONAL OPERATIONS

In this section we define additional operations that are necessary to define well-formedness rules, meta derivation rules and the operations. For each operation we present its context, semantic, parameters and OCL description.

6.1 Additional operations for partition

The operation **hasSuper** returns the supertype of a partition.
```
context Partition:
hasSuper: GeneralizableElement
hasSuper=self.constrainedElement.oclAsType(Generalization)-> first().parent
```

The operation **hasSubtypes** returns the subtypes of a partition.
```
context Partition:
hasSubtypes: Set(GeneralizableElement)
hasSubtypes= self.constrainedElement.oclAsType(Generalization) ->iterate(g:Generalization,
acc:Set(GeneralizableElement)=Set{} | acc->union(g.child))
```

The operation **hasDiscriminator** returns the discriminator of a partition.
```
context Partition:
hasDiscriminator(): Name
hasDiscriminator=self.constrainedElement.oclAsType(Generalization)-> first().discriminator
```

The operation **Derbyunion** returns the supertype of the partition if it is derived by union of all its subtypes.
```
context Partition:
Derbyunion: GeneralizableElement[0..1]
Derbyunion=self.hasSuper().clientDependency-> exists(du:Dependency |
du.oclIsTypeOf(DerivedUnion)
```
and du.supplier-(self.hasSubtypes)->isEmpty())
implies self.hasSuper

The operation $\text{existsPartition}$ returns true if there exist partitions with the supertype included in $\text{types1}$ and the subtypes included in $\text{types2}$.

**context** Partition:
existsPartition(types1:Set(Type), types2:Set(Type)): Boolean
existsPartition =
types1->forall(sup:Type |
    Partition.allInstances->exists(p:Partition |
        p.hasSuper=sup
        and types2->includesAll(p.hasSubtypes)))
and

types2->forall(sub:Type |
    Partition.allInstances->exists(p:Partition |
        p.hasSubtypes->includes(sub)
        and types1->includes(p.hasSuper)))

The operation $\text{partition}$ returns the partition with the supertype $\text{sup}$ and subtype $\text{sub}$.

**context** Partition:
partition(sup:Type, sub:Type): Partition
partition =
Partition.allInstances->one(p:Partition | p.hasSuper=sup and p.hasSubtypes->includes(sub))

### 6.2 Additional operations for type

The operation $\text{SpecSatisfiedBySchema}$ returns true if the specialization constraint between subtype and self is satisfied by the schema.

**context** Type:
SpecSatisfiedBySchema (subtype:Type): Boolean
SpecSatisfiedBySchema =
self.clientDependency->exists(d:Dependency |
    d.oclIsTypeOf(DerivedUnion)
    and d.supplier-> includes(subtype))
or ---if $E_i$ is derived by union of a set of types that includes $E_i$
    subtype.clientDependency-> exists(d:Dependency |
        d.oclIsTypeOf(DerivedSpec)
        and d.supplier-> includes(self)))
or ---if $E_i$ is derived by specialization of $E_i$
not self.derived and subtype.derived ---if $E_i$ is derived and $E$ is base.

The operation $\text{DisjSatisfiedBySchema}$ returns true if the disjointness constraint between self and type is satisfied by the schema.

**context** Type:
DisjSatisfiedBySchema (type:Type): Boolean
DisjSatisfiedBySchema =
if Partition.allInstances ->select(p:Partition |
    p.constrainedElement.oclAsType(Generalization)->
    exists(g1, g2:Generalization | g1.child=self and g2.child=type)
    and p.constrainedElement.oclAsType(Generalization)->
    forAll(g:Generalization |
        g.child.clientDependency-> exists(d:Dependency |
            d.oclIsTypeOf(DerivedSpec)
            and d.supplier-> includes(self))
        )
    )
)
and d.supplier-> includes(p.hasSuper))}) -> isEmpty() then false
else true endif

or

---If there is a partition P with supertype E and subtypes \{E₁, Eₙ\} with \(E_i\) and \(E_j\) \(\in\) \{\(E₁, Eₙ\}\) and such that all subtypes are derived by specialization of E.

not self.derived and type.derived or not type.derived and self.derived

or

---if \(E_i\) is base and \(E_j\) is derived

self.clientDependency->exists(d:Dependency |
  d.oclIsTypeOf(DerivedExcl)
  and d.supplier= d.supplier->first()-> includes(type))

or

type.clientDependency->exists(d:Dependency |
  d.oclIsTypeOf(DerivedExcl)
  and d.supplier= d.supplier->first()-> includes(self))

---if \(E_i\) is derived by exclusion of a set of types that includes \(E_j\)


The operation CovSatisfiedBySchema returns true if the covering constraint between subs and self is satisfied by the schema.

context Type:
CovSatisfiedBySchema (subs:Set(Type)):Boolean
CovSatisfiedBySchema="=
self.clientDependency->exists(d:Dependency |
  d.oclIsTypeOf(DerivedUnion) and d.supplier-> includesAll(subs))
or

---if \(E_i\) is derived per unit of \(E_j\),..., \(E_n\)

if Partition.allInstances ->select(p:Partition |
  p.hasSubtypes=subs
  and p.constrainedElement.oclAsType(Generalization)->
  forAll(g:Generalization |
    g.child. clientDependency-> exists(d:Dependency |
      d.oclIsTypeOf(DerivedSpec)
      and d.supplier-> includes(p.hasSuper))))) -> isEmpty() then false
else true endif

or

---If there is a partition P with supertype E and subtypes \{E₁, Eₙ\} with \(E_i\) and \(E_j\) \(\in\) \{\(E₁, Eₙ\}\} and such that all subtypes are derived by specialization of E.

subs->exists(sub:Type |
  sub.clientDependency->exists(d:Dependency |
    d.oclIsTypeOf(DerivedExcl)
    and d.supplier= d.supplier->first()=self
    and d.supplier->includesAll(subs->excludes(sub))

---if \(E_i\) \(\in\) \{\(E₁, Eₙ\}\} derived by specialization of \(E\) and exclusion of \{\(E₁, Eₙ\) \(\setminus\) \{\(E_i\)\}\}.

The operation ChangeAffectsConstraint returns true if a contraction or expansion in the population of the type affects to taxonomic constraints in the schema and false in other case. We need an auxiliary operation OtherSubtypesAffectsConstraint that returns true if taxonomic constraints are affected in subtypes.

context Type:
OtherSubtypesAffectsConstraint(direction:Direction):Boolean
OtherSubtypesAffectsConstraint="=
if direction=Direction::Contracting then
  if Partition.allInstances ->select(p:Partition |
    p.hasSuper=self) -> forAll(p:Partition |
      p.constrainedElement.oclAsType(Generalization) ->
      forAll(g:Generalization |
        p.hasSuper.SpecSatisfiedBySchema(g.child) and
        p.hasSubtypes-> forAll(sub:Type |
          sub.clientDependency-> exists(d:Dependency |
            d.oclIsTypeOf(DerivedExcl)
            and d.supplier= d.supplier->first()=self
            and d.supplier-> includesAll(subs->excludes(sub))
          ))) -> isEmpty() then false
else true endif
then
  OtherSubtypesAffectsConstraint::false
else
  OtherSubtypesAffectsConstraint::true
endif
endif
if direction=Direction::Expanding then
  if Partition.allInstances->select(p:Partition |
    p.hasSuper=self) ->forAll(p:Partition |
    p.hasSuper.CovSatisfiedBySchema(p.hasSubtypes)
    and p.hasSubtypes->forAll(sub:Type |
    not sub.OtherSubtypesAffectsConstraint(direction))))
  then
    OtherSubtypesAffectsConstraint::false
  else
    OtherSubtypesAffectsConstraint::true
  endif
endif

context Type:
ChangeAffectsConstraint(direction:Direction, sub:Type, p:Partition):Boolean
ChangeAffectsConstraint=
let subsSeq=Sequence(Type)=p.hasSubtypes->excludes(sub)-> asSequence();
let numberSubtypes:Integer=p.hasSubtypes->excludes(sub)->size();
in
if direction=Direction::Contracting then
  if self.CovSatisfiedBySchema(p.hasSubtypes) and
    not self.OtherSubtypesAffectsConstraint(direction) and self.parent->forAll(t:Type | not
    t.ChangeAffectsConstraint(direction,self,partition(t,self))) then
    ChangeAffectsConstraint::false
  else
    ChangeAffectsConstraint::true
  endif
endif
if direction=Direction::Expanding then
  if self.SpecSatisfiedBySchema(sub) and
    {1..numberSubtypes}->forAll(i:Integer | subsSeq-> at(i).DisjSatisfiedBySchema(sub)) and
    not self.OtherSubtypesAffectsConstraint(direction) and self.parent->forAll(t:Type |
    not t.ChangeAffectsConstraint(direction,self,partition(t,self)))then
    ChangeAffectsConstraint::false
  else
    ChangeAffectsConstraint::true
  endif
endif

6.3 Additional operations for dependency

The operation allSuppliers returns all the suppliers of a dependency.

class Dependency:
allSuppliers: Set(ModelElement)
allSuppliers=self.supplier->union(self.supplier.allSuppliers)
7. EVOLVING OPERATIONS

7.1 Partitions Evolving Operations

In this section, we present the operations that we need to evolve partitions. In general, these operations have as input the parameters of the operation, a conceptual schema and an information base. The outputs are a changed conceptual schema and, if required, a changed information base.

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 each of the above operations, and give a formal specification in the OCL 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.

7.1.1 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.

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.

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:

```ocl
class Partition
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 derivation rules. In this case, the induced effects are:

- For each `Generalization` created by the operation, an instance of `Specialization` is created as well (MDR01).
- For each pair of subtypes of the new partition, an instance of `Disjointness` is created as well (MDR04).
- A new instance of `Covering` is created (MDR07).

In the three cases, the new instances are associated to the `ModelElements` that they constraint 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 (MDR02, MDR05, MDR08), 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 (MDR03, MDR06, MDR09).

### 7.1.2 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.

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 operation does not change in any case the population of the supertype and, therefore, no indirect constraints can be affected.

The postconditions guarantee that a new generalization will be created, and will be added to the constrained elements of the partition.

The OCL definition of the operation is:

```ocl
class Partition::AddSubtype (subtype:Type) 
  context Partition::AddSubtype (subtype:Type) 
  pre PartitionNotIncludesSubtype: 
    self.hasSubtypes -> excludes(subtype) 
  pre SubtypeHasNoInstances: 
    subtype.allInstances -> isEmpty() 
  post: 
    --Create a new generalization corresponding to subtype, attached to self
    ge.oclIsNew() and ge.oclIsTypeOf(Generalization) 
    and ge.discriminator = self.hasDiscriminator 
    and ge.child = subtype 
    and ge.parent = self.hasSuper 
    and self.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 derivation rules. In this case, the induced effects are:

- For the *Generalization* created by the operation, an instance of *Specialization* is created as well (MDR01).
- For each pair of subtypes one of them subtype, an instance of *Disjointness* is created as well (MDR04).

In the two cases, the new instances are associated to the *ModelElements* that they constraint 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 (MDR02, MDR05), 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 (MDR03, MDR06). The attributes *body* and *satisfaction* for the *Covering* constraint are recalculated (MDR08, MDR09).
7.1.3 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.

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 operation does not change in any case the population of the supertype and, therefore, no indirect constraints can be affected.

The postconditions guarantee that the corresponding generalization will be deleted, and will be removed from the constrained elements of the partition.

The OCL definition of the operation is:

```
class Partition::RemoveSubtype (subtype:Type)

context Partition::RemoveSubtype (subtype:Type)
pre PartitionIncludesSubtype:
  self.hasSubtypes -> includes(subtype)
pre SubtypeHasNoInstances:
  subtype.allInstances -> isEmpty()
post: -- The generalization corresponding to subtype is removed from self
  let generalizationToRemove:Generalization =
  self.contrainedElement@pre.oclAsType(Generalization) ->any(child = subtype) –There is only one
  in
  self.contrainedElement = self.contrainedElement@pre ->excludes(generalizationToRemove)
  Generalization.allInstances->excludes(generalizationToRemove)
```

In addition to the effects defined by the postconditions, the execution of the operation induces other effects on the schema, as defined by the derivation rules. In this case, the induced effects are:
- For the Generalization deleted by the operation, the associated instance of Specialization is deleted as well (MDR01).
- For each pair of subtypes one of them subtype, the associated instance of Disjointness is deleted as well (MDR04).

The Covering attribute body has modifies its value corresponding to the constraint that must be satisfied by the information base (MDR08), 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 (MDR09).

7.1.4 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.

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:
- **NewSubtypesAreMutuallyDisjoint.** 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. The operation does not change in any case the population of the supertype and, therefore, no indirect constraints can be affected. We check this condition in two steps: **OldSubtypesCoveredByNewSubtypes**, which ensures that the old population is a subset of the new one, and **NewSubtypesCoveredByOldSubtypes**, which does the inverse.

There is a particular case, which happens very often, for which the above preconditions need not to be checked, because they are already guaranteed: when the old set and the new set are the super and the subtypes of an existing partition.

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

The OCL definition of the operation then is:

```ocl
context Partition::ReplaceSubtypes (oldSubtypes:Set(Type), newSubtypes:Set(Type))
pre AtLeastOneOldSubtype:
oldSubtypes -> notEmpty()
pre AtLeastOneNewSubtype:
newSubtypes -> notEmpty()
pre PartitionIncludesSubtypesToRemove:
self.hasSubtypes -> includesAll(oldSubtypes)
pre PartitionNotIncludesSubtypesToAdd:
self.hasSubtypes -> excludesAll(newSubtypes)
pre:
not (Partition.existsPartition(oldSubtypes,newSubtypes) or
 Partition.existsPartition(newSubtypes,oldSubtypes)) implies – The sets do not form a partition
-- Check OldSubtypesCoveredByNewSubtypes
oldSubtypes -> forAll (sub:Type | not sub.CovSatisfiedBySchema(newSubtypes) implies
 newSubtypes -> iterate(newSub:Type; acc:Set(Type) = Set{} | acc -> union(newSub.allInstances)) -> includesAll(sub.allInstances))
and
-- Check NewSubtypesCoveredByOldSubtypes:
newSubtypes -> forAll (sub:Type | not sub.CovSatisfiedBySchema(oldSubtypes) implies
 oldSubtypes -> iterate(oldSub:Type; acc:Set(Type) = Set{} | acc -> union(oldSub.allInstances)) -> includesAll(sub.allInstances))
and
-- Check NewSubtypesAreMutuallyDisjoint:
-- Pairs of types in newSubtypes must be mutually disjoint. We avoid duplicate checks.
let subsSeq:Sequence(Type) = newSubtypes -> asSequence()
let numberSubtypes:Integer = newSubtypes -> size()
in Sequence {1..numberSubtypes} ->forall (i, j:Integer | i > j and
```

38
not \( \text{subsSeq} \rightarrow \text{at(i).DisjSatisfiedBySchema(subsSeq \rightarrow \text{at(j)})} \)

implies

\( \text{subsSeq} \rightarrow \text{at(i).allInstances} \rightarrow \text{excludesAll (subsSeq \rightarrow at(j).allInstances)} \)

**post:**

let generalizationsToRemove:Set(Generalization) =

\[
\text{self.contrainedElement@pre.oclAsType(Generalization) ->select(oldSubtypes -> includes(child))}
\]

in

\[
\text{self.constrainedElement} \rightarrow \text{excludesAll(generalizationsToRemove)}
\]

and

\[
\text{Generalization.allInstances} \rightarrow \text{excludesAll(generalizationsToRemove)}
\]

and -- Create a generalization for each new subtype

\[
\text{newSubtypes} \rightarrow \text{forAll(sub:Type | ge.oclIsNew() and ge.oclIsTypeOf(Generalization) and ge.discriminator = self.hasDiscriminator and ge.child = sub and ge.parent = self.hasSuper and self.constrainedElement -> includes(ge))}
\]

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

- For each *Generalization* created by the operation, an instance of *Specialization* is created as well (MDR01).
- For each pair of subtypes one of them included in *newSubtypes*, an instance of *Disjointness* is created as well (MDR04).
- For each *Generalization* deleted by the operation, the associated instance of *Specialization* is deleted as well (MDR01).
- For each pair of subtypes one of them included in *oldSubtype*, the associated instance of *Disjointness* is deleted as well (MDR04).

In the two firsts cases, the new instances are associated to the *ModelElements* that they constraint 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 (MDR02, MDR05), 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 (MDR03, MDR06).

The Covering attribute *body* has modifies its value corresponding to the constraint that must be satisfied by the information base (MDR08), 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 (MDR09).

### 7.1.5 Resizing a partition

The operation *Resize* can be applied only to partitions such that its supertype is derived by union of its subtypes. The operation allows the designer to add (to remove) a subtype, and to change simultaneously the derivation rule of the supertype. The overall effect is that the partition has been expanded (contracted). The operation has two parameters: the partition and the non-empty subtype involved.
There 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 subtypes. The specialization and covering constraints are guaranteed by the derivation rule (union) of the supertype.

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.

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

The OCL definition of the operation then is:

```ocl
class Partition::Resize (subtype:Type)
let direction:Direction =
if self.hasSubtypes-> excludes(subtype) implies Direction::Expanding
else Direction::Contracting
endif
pre SuperDerivedByUnionOfSubtypes:
self.hasSuper=self.Derbyunion
pre: -- The change does not affect other constraints
subtype.parent->forAll(t:Type | not ChangeAffectsConstraint(direction,subtype,partition(t,subtype))
pre ExpandingRequiresNewSubtypeDisjoint:
direction = Direction::Expanding implies
self.hasSuper.allInstances -> excludesAll(subtype.allInstances)
post:
let generalization:Generalization=
self.contrainedElement@pre.oclAsType(Generalization) ->select(g | g.child=subtype))
in
if direction=Direction::Contracting then
self.constrainedElement->excludes(generalization) and
Generalization.allInstances->excludes(generalization) and
self.hasSuper.clientDependency->one(d.Dependency | d.oclIsTypeOf(DerivedUnion)).supplier
->excludes(subtype)
endif
if direction=Direction::Expanding then
ge.oclIsNew() and ge.oclIsTypeOf(Generalization)
and ge.discriminator = self.hasDiscriminator
and ge.child = subtype
and ge.parent = self.hasSuper
and self.constrainedElement -> includes(ge)
and self.hasSuper.clientDependency->one(d.Dependency | d.oclIsTypeOf(DerivedUnion)).supplier
->includes(subtype)
endif
```

In addition to the effects defined by the postconditions, the execution of the operation induces other effects on the schema, as defined by the derivation rules. In this case, the induced effects are:
- If the direction is Expanding, for the Generalization created by the operation, an instance of Specialization is created as well (MDR01).
- If the direction is Expanding, for each pair of subtypes one of them subtype, an instance of Disjointness is created as well (MDR04).
- If the direction is Contracting, for the Generalization deleted by the operation, the associated instance of Specialization is removed as well (MDR01).
- If the direction is Contracting, for each pair of subtypes one of them subtype, the associated instance of Disjointness is deleted as well (MDR04).

In the two firsts cases, the new instances are associated to the ModelElements that they constraint 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 (MDR02, MDR05), 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 (MDR03, MDR06).

The Covering attribute body has modified its value corresponding to the constraint that must be satisfied by the information base (MDR08), 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 (MDR09).

The attribute mapping has modified its value corresponding to the new derivation rule related to the supertype (MDR10).

7.1.6 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.

The OCL definition of the operation is:

```oclmjml
context Partition::Remove ()
  -- There are no preconditions
post:
  -- The generalizations corresponding to self are deleted
  Generalization.allInstances->excludesAll(self.constrainedElement@pre)
  and -- The partition itself is deleted
  Partition.allInstances -> excludes(self)
```

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

- For each the Generalization deleted by the operation, the associated instance of Specialization is deleted as well (MDR01).
- For each pair of subtypes of the partition, the associated instance of Disjointness is deleted as well (MDR04).
- An instance of Covering is deleted (MDR07).

7.2 Derived Types Evolving Operations

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.

7.2.1 Change the derivability of an entity type 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 noticeable precondition of this operation is that the entity type must not be base
already.

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, and the one that we
have adopted as postcondition, is to assume that the population will not change. In the
example, this means that the population of LoanableBook will no be affected and, therefore,
no taxonomic constraints will be affected neither.

The OCL definition of the operation is:

```oclmultiline
context Type::ChangeDerivabilitytoBase ()
pre: self.derived=true -- The entity type is derived.
post:
  Dependency.allInstances->excludes(self.clientDependency->one(d: Dependency |
    d.oclIsTypeOf(Derived) and d.client=self)) and -- Removed of the derivation rule
  self.allInstances->includesAll(self.allInstances@pre) and -- The instances are the same
  self.derived=false
```

7.2.2 Change the derivability of an entity type 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).

The OCL definition of the operation is:

```oclmultiline
context Type::ChangeDerivabilitytoDerived (auxtype:Type)
pre AuxTypesIsDerived:
  auxtype.derived=true
```
**pre** BothPopulationsAretheSame:

\[
\text{type.allInstances->includesAll(auxtype.allInstances) and}
\text{auxtype.allInstances->includesAll(type.allInstances)}
\]

**post**:

let auxSuppliers=auxtype.clientDependency->one(d: Dependency | d.oclIsTypeOf(Derived)).supplier
let auxMapping= auxtype.clientDependency->one(d: Dependency | d.oclIsTypeOf(Derived)).Mapping
in
self.derived=true
and auxtype.clientDependency->exists(d: Dependency | d.oclIsTypeOf(DerivedExcl)) implies
d.oclIsNew() and d.oclIsTypeOf(DerivedExcl)
and auxtype.clientDependency->exists(d: Dependency | d.oclIsTypeOf(DerivedSpec)) implies
d.oclIsNew() and d.oclIsTypeOf(DerivedSpec) and d.mapping=auxMapping
and auxtype.clientDependency->exists(d: Dependency | d.oclIsTypeOf(DerivedUnion)) implies
d.oclIsNew() and d.oclIsTypeOf(DerivedUnion)
and auxtype.clientDependency->exists(d: Dependency | d.oclIsTypeOf(Derived)) implies d.oclIsNew()
and d.oclIsTypeOf(Derived) and d.mapping=auxMapping
and d.client=self and d.suppliers=auxSuppliers