Design and Implementation of Partition profile – An experience report

Jordi Cabot, Cristina Gómez

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

Abstract

The definition of UML profiles is an important research area in information systems engineering. In this paper, we aim at contributing to the area in twofold. On the one hand, we study the limitations of CASE tools when a profile definition is considered. On the other hand, we select one of them and show, using the partition profile with the specification of partition evolution operations defined in [GoOl02], how a profile can be implemented with the selected tool.

Keywords

Conceptual schemas, evolution, meta schemas, profiles, CASE tools, UML

Table of Contents

1. Introduction

In recent years, many UML profiles have been developed to support different tasks of software systems development. Usually, the profiles standardized by the OMG are incorporated to UML CASE tools. However, other profiles, which are not standardized, cannot be used because the majority of tools do not incorporate them and, moreover, do not support new profiles definition. In fact, the proliferation of a huge amount of profiles has not been accompanied by the necessary support of existing CASE tools.

The main goal of this report is twofold. On the one hand, we study the limitations of CASE tools when a profile definition is considered. Nowadays, a lot of CASE tools allow designers to model software systems in UML. We select some of these tools that we think more relevant for its use, functionalities that provides, etc, and we discuss the problems relating to profile definition with them.

On the other hand, taking the UML profile for partitions of entity types with the specification of partition evolution operations defined in [GoOl02], we select one of the studied CASE tools (the one which we consider more suitable for a profile definition) and show how a profile can be implemented with the selected tool.

The rest of this paper is organized as follows. Section 2 introduces the profile of entity types partitions and the specification of some partition evolution operations. Some considerations about the design of the profile and the evolution operations are described in Section 3. Section 4 focuses on the limitations of existing CASE tools with respect to the profiles definition. Section 5 details how a fragment of the profile and the evolution operations can be implemented in the selected CASE tool. And finally, last section remarks some conclusions of this report.

2. Profile for entity types partitions

In this section we briefly describe the profile for entity types partitions and a set of partition evolution operations.

2.1 Profile for entity types partitions

In a UML conceptual schema, *partitions* are represented as a set of generalizations with the same parent and discriminator and with two predefined constraints (complete and disjoint). The parent and the discriminator of partitions identify them. In the UML metamodel (version 1.4) there is not a metaclass that represents the partition construct. Partitions are represented in the metamodel by means of its generalizations and constraints. Figure 1 shows a UML conceptual schema with three base types *Person*, *Man* and *Woman* and a partition of *Person* into *Man* and *Woman* by *sex*.

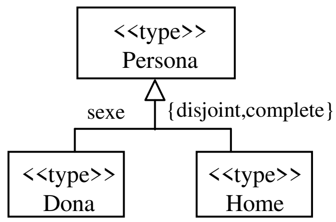


Figure 1: Representation of a partition in UML.

The main goal of the profile for entity type partitions (Figures 2 and 3) is to improve the specification of entity types partitions and taxonomic constraints in UML. This profile allows designers:

1) To define conceptual schemas in UML with the new constructor partition. It will be possible to indicate graphically in the schema this construct. Moreover, the definition of partition construct in UML allows us to define easily the evolution operations related to partitions.
2) To define derived types with different kinds of derivability. Concretely, derived types can be derived by specialization, by union, by exclusion or by other types of derivability.
3) To define graphically taxonomic constraints in the schema indicating if they are enforced or satisfied by the schema. When the constraints are enforced, they are calculated automatically and shown in the schema.

The most important stereotype in the partition profile is <<partition>>. A single instance of a constraint with this stereotype corresponds to a partition in the conceptual schema. 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]. These 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 | g.oclIsTypeOf(Generalization))
```

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

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*. BySchema means that the schema itself satisfies the taxonomic constraint, and thus it is not necessary to define it explicitly. On the other side, Enforced means that the schema does not guarantee the taxonomic constraint, so it must be explicitly defined.

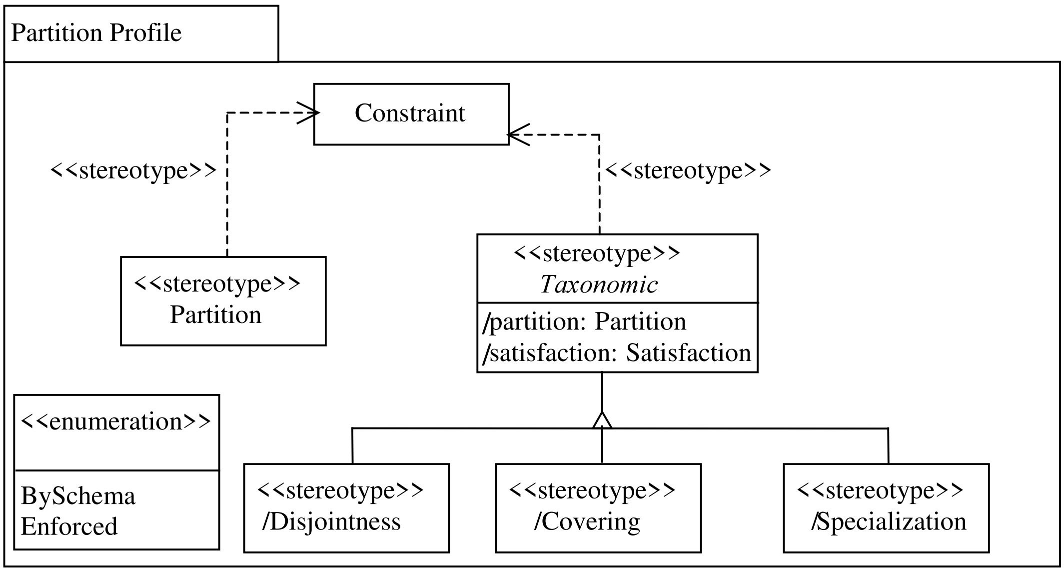


Figure 2: Partition profile (Part 1)

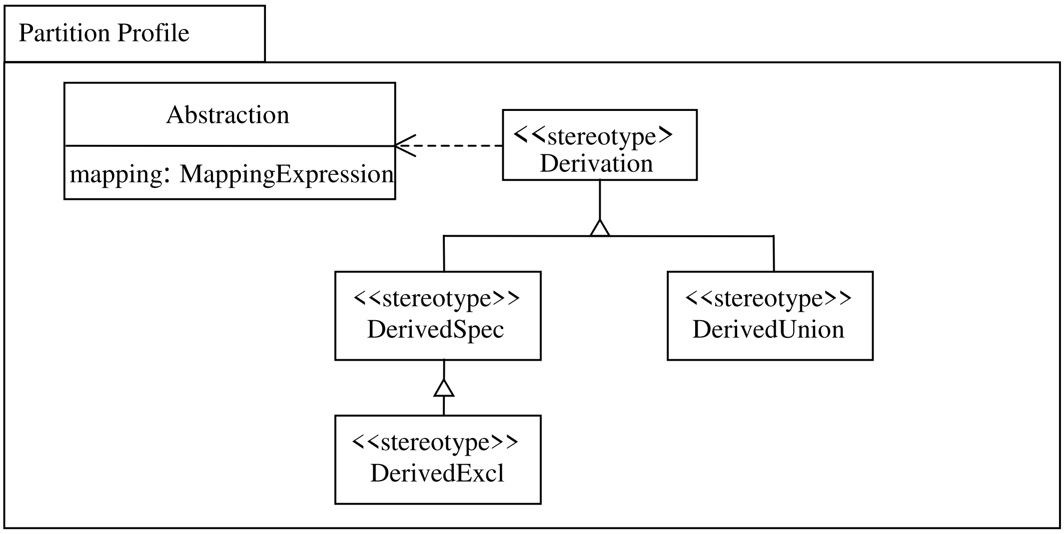


Figure 3: Partition profile (Part 2)

We define the stereotypes *Disjointness*, *Covering* and *Specialization* as derived, because their instances can be obtained automatically from the *Partition* and their *Generalizations*.

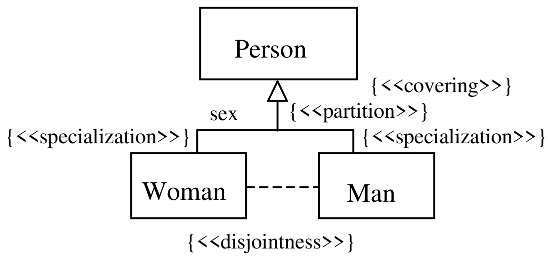
In the UML metamodel, *ModelElements* have a tag called *derived*. A true value indicates that they can be derived from other *ModelElements*. The details of the derivation are given in an *Abstraction* dependency, with the standard stereotype <<derive>>, and name of the stereotype class *Derivation* [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 profile has a set of meta derivation rules that derive new information in the profile from the base information. Concretely, derivation rules that calculate the instances of the stereotypes *Disjointness*, *Covering* and *Specialization* and derivation rules that calculates the values of the *satisfaction* attribute are examples of meta derivation rules. (Meta) derivation rules are represented in UML as constraints. We show an example of a meta derivation rule (constraint) that calculates the values of the *satisfaction* attribute for the *Specialization* constraint. This derivation rule is defined formally in the OCL as follows:

```
context Specialization inv:
self.taggedValue->one(type.name='satisfaction').dataValue =
if self.constrainedElement.parent.SpecSatisfiedBySchema(self.constrainedElement.child) then
  Satisfaction::BySchema
else
  Satisfaction::Enforced
endif
```

The operation *t1.SpecSatisfiedBySchema(t2)* is an operation defined in the partition profile and attached to the *Type* stereotype. This operation calculates if the two types *t1* and *t2* satisfy the specialization constraint by the schema.

We need to distinguish between the three classes of derived entity types, and therefore we define in our profile the three stereotypes of *Abstraction* shown in Figure 3: <<DerivedUnion>>, <<DerivedSpec>> and <<DerivedExcl>>. The first two are subtypes of the standard *Derivation*, 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.




```
context Person inv:
Person.allInstances->includesAll(Man.allInstances)
Person.allInstances->includesAll(Woman.allInstances)
context Man inv:
Man.allInstances->union(Woman.allInstances)->includesAll(Person.allInstances)
context Woman inv:
Woman.allInstances->excludesAll(Man.allInstances)
```

Figure 4: Representation of a UML partition using the partition profile

Figure 4 shows the partition represented in Figure 1 with the partition construct defined in partition profile. As we can see in the figure, the partition is represented by means of the partition construct defined in the profile. Moreover, in this example, all the taxonomic constraints are enforced because *Person*, *Man* and *Woman* are base types. This means that taxonomic constraints are calculated automatically and shown in the schema.

In [GoOl02] we can find the complete definition of the partition profile, with the all meta integrity constraints and meta derivation rules.

2.2 Evolution operations of a partition

In this section, we focus on the operations needed to evolve partitions. A complete definition of these operations can be found in [GoOl02]. In general, these operations have as input the parameters of the operation, a conceptual schema and an information base. The outputs are an updated conceptual schema and, if required, an updated information base.

We adopt the classical framework with the reflective architecture [ISO82, Mant93, PeÖz93, LoOl00]. In our case (Figure 5), the meta conceptual schema is the UML metamodel and the profile presented in the previous section. The meta external events that the meta information processor (MIP) receives are the operations listed in this section. The effects of these operations modify the meta information base (MIB) and as a consequence the conceptual schema expressed in UML with the profile constructs. In general, changes in conceptual schemas can involve violations of UML metamodel and profile constraints. Moreover, the instances in the information base (IB) can violate constraints defined in the schema. In these cases, the MIP rejects the operation and undoes all the changes in the MIB and the IB.

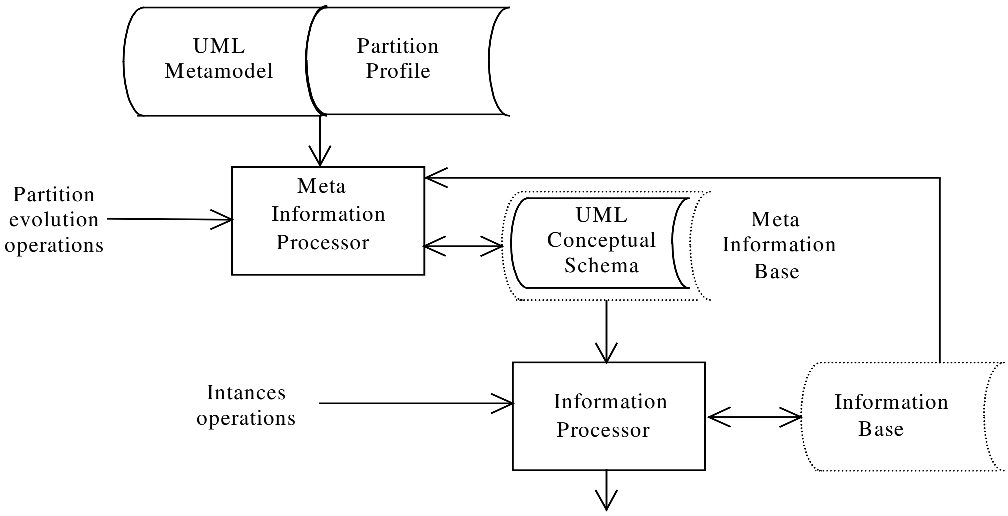


Figure 5: Framework for partition evolution

The list of the evolution operations of partitions is:

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.

For brevity, we only specify the operation creating a partition to a conceptual schema. The specification of the other operations defined for the partition evolution can be found in [GoOl02]. For *AddPartition* operation we specify a formal specification in 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 UML metamodel and the profile. Moreover, the IB instances must satisfy schema constraints
- May induce effects on the schema (MIB) and/or the information base defined by the meta derivation rules attached to the profile stereotypes and schema derivation rules.

2.2.1 Creating Partitions

The operation *AddPartition* allows 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.

This operation must satisfy the UML metamodel constraints, the profile constraints and the schema constraints (the intances of the IB must satisfy conceptual schema constraints). It is clear that some of the postconditions of the *AddPartition* operation may lead to a violation of a certain constraint. In this sense, the preconditions of the operation should check if the constraints are satisfied. If we check these constraints in the operation we have a certain redundancy between the static analysis model (UML metamodel, profile and conceptual schema) and the dinamic one (operation contracts). [CoST02] details the advantages of a non-redundant analysis model and how to achieve this model. Concretely, the authors propose to define the constraints only in the static analysis model and not in the dinamic one. In our case, all the constraints can be checked by the static analysis model. Therefore, to avoid the redundancy between the analisys model (the profile, the UML metamodel and the conceptual schema) and the dynamic model (*AddPartition* operation) these constraints are not verified in the preconditions of the operation.

We consider interesting to show here the redundant preconditions of the *AddPartition* operation because we use them later. Some of preconditions (1,2) are guaranteed by means of the external interface. The preconditions (3,4,5,6) are satisfied by means of the profile constraints and the preconditions (7,8) are satisfied by the UML metamodel constraints. The rest of preconditions (9,10,11) are satisfied by taxonomic constraints defined in the conceptual schema. The redundant preconditions are the following:

1. *SuperMustExist*: the type *super* must exist. This precondition is ensured by the definition of the type parameter *super*. We assume the existence of an external interface for this operation that allows designers to verify if *super* is an existing type in the conceptual schema.
2. *SubsMustExist*: the types defined in the set *subs* must exist. This precondition is ensured by the definition of the type parameter *subs*. We assume the existence of an external interface for this operation that allows designers to verify if each type of *subs* is an existing type in the conceptual schema.

3. *PartitionMustNotExist*: the partition must not exist. This precondition is ensured by a constraint related to *Partition* stereotype defined in the profile.
4. *SubsCanNotBeEmpty*: the set of types *subs* cannot be empty. This precondition is ensured by a constraint related to *Partition* stereotype defined in the profile.
5. *TwoSubsNotEqual:* it cannot exist two equal types in the *subs* set. This precondition is ensured by a constraint related to *Partition* stereotype defined in the profile.
6. *NotGenOrPart:* it cannot exist other generalizations or partitions with the same parent and discriminator. This precondition is ensured by a constraint related to *Partition* stereotype defined in the profile.
7. *SuperNotSub*: the type *super* cannot be one of *subs* types. This precondition is ensured by a constraint defined in the UML metamodel related to *GeneralizableElement* class that does not allow circular inheritance [OMG01, page 2-61].
8. *SubsNotParents*: the types in *subs* set cannot be one of the direct or indirect parents of *super*. This precondition is ensured by a constraint defined in the UML metamodel related to *GeneralizableElement* class that does not allow circular inheritance [OMG01, page 2-61].
9. *SpecAreSatisfied*: The instances of each subtype must be a subset of the instances of the supertype. This constraint should be verified only if the schema does not satisfy it. This precondition is ensured by the specialization constraint defined in the conceptual schema.
10. *DisjAreSatisfied*: The instances of the subtypes must be mutually disjoint. This constraint should be verified only if the schema does not satisfy it. This precondition is ensured by the disjointness constraint defined in the conceptual schema.
11. *CovIsSatisfied*: The union of the instances of the subtypes must cover the instances of the supertype. This constraint should be verified only if the schema does not satisfy it. This precondition is ensured by the covering constraint defined in the conceptual 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 specification of the non-redundant operation is:

```
context Partition::AddPartition (discriminator:Name, super: Class, subs:Set(Class))
post:
p. oclIsNew()
and p.oclIsTypeOf(Constraint)
and p.stereotype->one(name=' Partition')
and  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))
```

The profile meta derivation rules and the schema derivation rules induce some effects in the MIB and in the IB. These are the induced effects for *AddPartition* operation:

- For each generalization, an instance of *Constraint* with the *Specialization* stereotype has been created. The *partition* attribute and the *constrainedElement* role have been initialized.
- For each pair of subtypes, an instance of *Constraint* with the *Disjointness* stereotype has been created. The *partition* attribute and the *constrainedElement* role have been initialized.
- An instance of *Constraint* with the *Covering* stereotype has been created. The *partition* attribute and the *constrainedElement* role have been initialized.
- The *satisfaction* attributes for each constraint instance have been initialized by means of profile meta derivation rules.

After the effects produced by the postconditions and the effects produced by (meta) derivation rules, if some UML metamodel, profile or schema constraint is violated the MIP rejects the operation and undoes the changes in the MIB and in the IB.

3. Design aspects

The design of a software system is the description of a solution of the problem described in the specification of the system with technology considerations. In this section, we present the design of the partition profile and the evolution operations for partitions assuming that this design will be implemented in any CASE tool that allows the definition of conceptual schemas in UML. We think that is better to use an existing CASE tool to design and implement the profile and the operations because we can take advantages of all the functionalities that these tools provide (for instance, round-trip engineering, documentation generation, etc). Note that, in this case, the CASE tool is the MIP that receive the evolution partition operations and modify the MIB (conceptual schema). In the design stage we do not select a specific CASE tool. This selection will be done in the implementation stage. Figure 6 shows the framework for the evolution of partitions with the CASE tool.

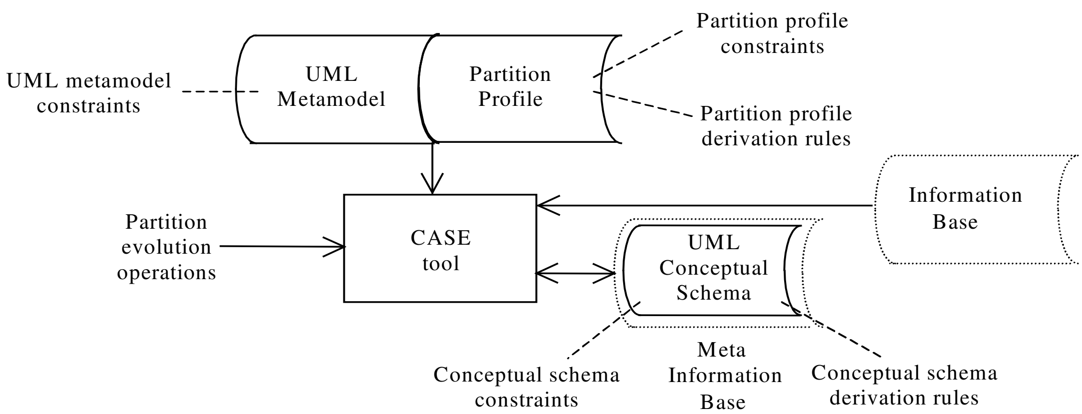


Figure 6: (Meta) integrity constraints and (meta) derivation rules in the framework for partition evolution

There are several aspects to consider in the design of the profile and the evolution operations:
1. Which component of the architecture (Figure 6) is the responsible of verifying the constraints defined in the UML metamodel and the partition profile?

2. Which component is the responsible of managing the derivation rules defined in the partition profile?
3. Which component is the responsible of verifying if the constraints defined in the conceptual schema are satisfied by the instances of the IB?
4. Which component is the responsible of managing the derivation rules defined in the conceptual schema?

In the design process, we are going to assign the above responsibilities to different components of the architecture. The design of the profile with the evolution operations involve two levels of the architecture. The UML metamodel and the profile constraints must be satisfied in the conceptual schema (MIB) whereas the schema constraints must be satisfied in the IB. The derivation rules of the partition profile must be managed in the conceptual schema (MIB) whereas derivation rules of the schema must be managed in the IB.

The use of CASE tools in our architecture provides some advantages and some inconvenients. The main advantage is that CASE tools verify UML metamodel constraints when a conceptual schema is being created. This means that the component responsible of verifying these constraints is the CASE tool. Unfortunately, the use of a CASE tool supposes some technological limitations. We mention here only the ones that affect partition profile and partition evolution operations:

1. The majority of UML CASE tools do not support UML version 1.4. This is the version used for the definition of partition profile. This means that not all the profile can be implemented with the existing CASE tools.
2. None of tools that allow the definition of UML conceptual schema provide an information base to have instances for the schema.
3. The majority of tools do not support verification of UML constraints in OCL. This means that the constraints defined in the profile cannot be verified by the CASE tool and we must assign this responsibility to other component.
4. Similarly to the constraints, tools do not support the management of UML derivation rules. We must assign this responsibility to other component of our architecture.

In next sections we present the design of the profile and the design of *AddPartition* operation taking into account all the technological limitations described in this section. Design of the other operations can be found in the appendix.

3.1 Design of partition profile

In general, from the design and/or implementation point of view, there are two different approaches to deal with derived information. This information can be *computed* if it is calculated by means of an operation or *materialized* if it is explicitly stored in the information base.

In the partition profile, several derived information appears: three stereotypes (*Specialization*, *Disjointness* and *Covering*) and two attributes (*satisfaction* and *partition*). In this design, we decide to materialize all derived information. There is an important reason to do this. When we compute derived information, this information is substituted by an operation that calculates the same information that the derived information provided. If we compute the three stereotypes, they would be substituted by three operations to calculate its instances. These operations cannot be defined in the stereotypes, because they have been removed.

Figure 7 shows a part of the partition profile, related to the partition definition and its taxonomic constraints, with materialized derived information. The part of the partition profile related to derived types has not been changed.

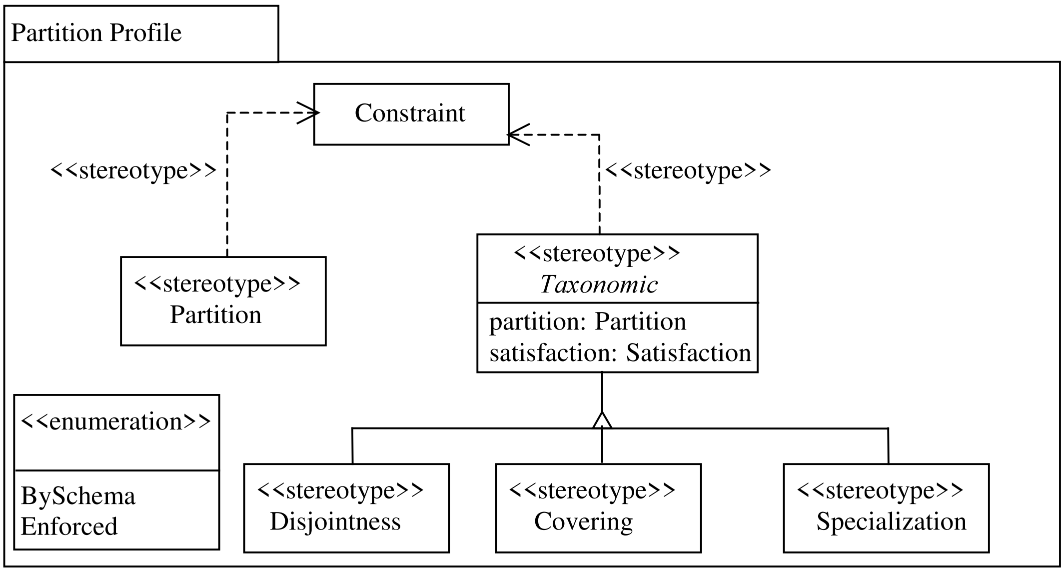


Figure 7: Partition profile (Part 1) with materialized derived information

3.2 Design of the evolution operations of a partition

In the partition profile we have defined a set of meta integrity constraints and meta derivation rules. As we have explained in the introduction of section 3, UML CASE tools neither support verification of UML constraints in OCL nor manage meta derivation rules. However, the evolution operations defined for partitions must satisfy all meta integrity constraints and perform all the changes induced by meta derivation rules. To achieve this, we can assign the responsability of verifying the constraints and managing the derivation rules defined in the profile to the evolution operations or to a component of the CASE tool. In our case, we decide to assign these responsabilities to the evolution operations. As a consequence:

1. The meta integrity constraints defined in the profile must be guaranteed by the preconditions of the operation. If a constraint is violated, then the preconditions of the evolution operation are not satisfied and the CASE tool rejects the operation.
2. The induced effects by meta derivation rules defined in the profile will be postconditions of the evolution operations. Once we have materialized all derived information, the postconditions of evolution operations must guarantee the same changes than those induced by meta derivation rules. If some change (induced or not) violates a constraint, then the CASE tool rejects the operation and undoes the changes in the MIB (conceptual schema) and in the IB.

The meta integrity constraints defined in the UML metamodel are guaranteed by the CASE tool. So this constraints must not be preconditions of the operation.

There are two aspects of the evolution operations that have not been considered yet. If we have a conceptual schema and an information base with schema instances when an evolution operation is performed, it may occur that:

1. As a consequence of base updates in the IB, new updates are induced by means of derivation rules defined in the schema. This happens in the *Resize* operation.
2. The IB instances can violate some constraint defined in the schema. In this case the operation must be rejected. This can happen in the *Resize* operation.

The existing CASE tools cannot manage derivation rules and constraints because they do not provide an information base. Clearly, the designer could integrate a database or a file system with the CASE tool. Then, we could assign the responsability of verifying the schema constraints and managing the schema derivation rules to the evolution operations or to a component of the CASE tool. In this case, the CASE tool would be responsible of generating the induced effects of schema derivation rules and verify if some schema constraint is violated. This means that evolution operations must not deal with schema derivation rules and constraints. The CASE tool would have a component to calculate all the changes that are induced by derivation rules and to calculate if these changes violate schema integrity constraints when an update is performed.

Note that, we follow different strategies to calculate all the changes induced by derivation rules or by meta derivation rules when an update is performed. The changes induced by derivation rules (schema level) are calculated by a component of the CASE tool. In the same way, this component verifies if these changes violate a schema constraint. We do not implement this component, because we can use one of the existing implementations for deductive databases. By constrast, changes induced by meta derivation rules (meta schema level) and verification of UML metamodel and profile constraints are managed by operation post and preconditions. We do not use, in this case, the same component since we would have to adapt the existing implementations to be able to interact with every CASE tool. We think, this effort it is not worthwhile due to the small amount of partition evolution operations

As we have said above, this new component of the CASE tool must calculate the changes induced by derivation rules and verify this changes satisfy the constraints in the shema level. These two aspects have been studied in depth and implemented in the context of deductive databases. The first one is called *change computation* and the second one is called *integrity constraint checking*. Change computation in deductive databases refers to the process of computing the changes on the extension of the derived predicates induced by changes on the base facts specified by a transaction. Integrity constraint checking consists of rejecting the transaction that violates some constraint. This is a particular application of change computation, because integrity constraints can be transformed in derivation rules with a head predicate that indicates the inconsistency. More details about change computation and integrity constraint checking can be found in [Teni00]. There are a lot of methods for change computation and for integrity constraint enforcement. A brief explanation of these methods can be found in [TeUr03]. We use one of these methods, adapted to our case, to calculate all the changes induced by derivation rules and the possibility that the changes violate a schema constraint when an update is performed. Concretely we use the method presented in [Oliv91].

In this method, a deductive database D consists of three finite sets: a set F of facts, a set R of deductive rules, and a set I of integrity constraints. The integrity constraints are defined as derivation rules with a head inconsistency predicate *Ic*. Due to the deductive rules, when an update U is produced the method calculates all the induced updates of the derived predicates and calculates if some integrity constraint is violated. This method can also be applied with complex updates, such as insertion or deletion of deductive rules, insertion or deletion of integrity constraints and transactions with multiple updates. The modifications of integrity constraints and derivation rules are dealt as deletions and subsequent insertions of integrity constraints and derivation rules.

In our case, we have an information system with three finite sets: a set of IB instances, a set R of deductive rules, and a set I of integrity constraints. When an update is performed the method calculates if the induced effects of the schema derivation rules and calculates if the schema constraints are violated. In Table 1 we show the updates produced by (meta) derivation rules that can affect to instances in the IB when a partition evolution operation is performed.

Evolution operation	**Insertion**	**Deletion**	**Modification**
Creating a partition	Specialization constraints Disjointness constraints Covering constraint		
Adding a subtype to a partition	Specialization constraint Disjointness constraint		Covering constraint
Removing a subtype from a partition		Specialization constraint Disjointness constraints	Covering constraint
Replacing subtypes	Specialization constraints Disjointness constraints	Specialization constraints Disjointness constraints	Covering constraints
Resizing a partition	Specialization constraint Disjointness constraint IB instances	Specialization constraint Disjointness constraints IB instances	Covering constraint Supertype derivation rule
Removing a partition		Specialization constraints Disjointness constraints Covering constraint	

Table1: Updates produced by (meta) derivation rules that affects to instances in the IB

As we can see in Table 1, the execution of evolution operations implies the insertion, deletion or modification of taxonomic constraints, derivation rules and IB instances. For instance, the operation creating a partition adds a set of taxonomic constraints related with the created partition. The operation resizing a partition that allows designers to add or remove a subtype with instances to (from) a partition, adds, removes and/or modifies taxonomic constraints, modifies the supertype derivation rule and as a consequence adds or deletes IB instances. The modification of these instances can involve violations of other taxonomic constraints defined in the schema. The method described in [Oliv91] detects if a violation is produced.

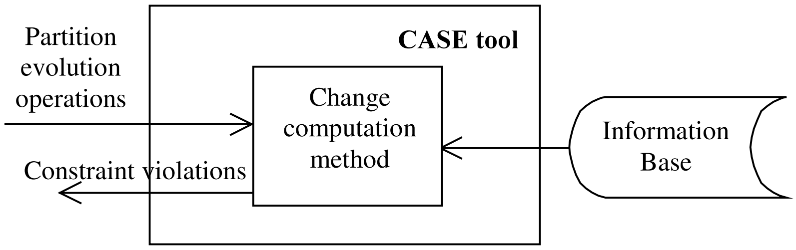


Figure 8: CASE tool component

Figure 8 shows the CASE tool component of our architecture that implements the method. This component receives the partition evolution operations and as a consequence the insertions, deletions and modifications of the derivation rules, integrity constraints and IB instances. The component calculates all the changes produced in the IB for derivation rules. If the changes induce an inconsistency predicate *Ic*, then the CASE tool component rejects the evolution operation.

Once we have assigned all the responsabilities to different components in the architecture, we show the design of the *AddPartition* operation in OCL.

As we can see in the operation design, we have added respect to the *AddPartition* operation specification the new preconditions *PartitionMustNotExist*, *SubsCanNotBeEmpty*, *TwoSubsNotEqual* and *NotGenOrPart.* These preconditions represent the verification of the profile constraints. The preconditions that represent the verification of UML metamodel constraints are omitted because the UML CASE tools already guarantee these constraints. The postconditions *Specialization Induced Effects*, *Disjointness Induced Effects* and *Covering Induced Effects* represent the induced effects produced by meta derivation rules defined in the partition profile.

The preconditions that represent the verification of the constraints defined in the conceptual schema and the postconditions that represent induced effects of the schema derivation rules are omitted since either a new component in UML CASE tools calculates and verifyes these induced effects in the information base or there is not an information base integrated with the CASE tool so they are not necessary.

The design of the *AddPartition* operation in OCL is:

context Partition::AddPartition (discriminator:Name, super: Class, subs:Set(Class))
pre PartitionMustNotExist**:**
Constraint.allInstances->select (c.stereotype->one(name='Partition') and c.hasSuper()=super and c.hasSubtypes()=subs)->isEmpty()
pre SubsCanNotBeEmpty**:**
subs->notEmpty()

pre TwoSubsNotEqual**:**
subs->forAll(sub1,sub2 | sub1<>sub2)
pre NotGenOrPart**:**
Constraint.allInstances->select (c.stereotype->one(name='Partition') and c.hasSuper()=super and c.hasDiscriminator()=discriminator)->isEmpty() and Generalization.allInstances->select (g.parent=super and g.discriminator=discriminator)->isEmpty()
post CreatePartition:
p. oclIsNew()
and p.oclIsTypeOf(Constraint)
and p.stereotype->one(name=' Partition')
and subs ->
forAll(sub:Type | ge.oclIsNew() and ge.oclIsTypeOf(Generalization)
and ge.discriminator = discriminator
and ge.child = sub
and ge.parent = super
and self.constrainedElement -> includes(ge))
post Specialization induced effects:
p.constrainedElement->forAll(ge:Generalization | spec.oclIsNew()
and spec.oclIsTypeOf(Specialization)
and spec.partition=p
if p.constrainedElement.parent.SpecSatisfiedBySchema(p.constrainedElement.child) then
spec.satisfaction=Satisfaction::BySchema and spec.body=''
else
spec.satisfaction=Satisfaction::Enforced
and spec.body=p.constrainedElement.parent.Name.oclAsType(String).concat('.allInstances-> includesAll(').concat(p.constrainedElement.child.Name.oclAsType(String)).concat('.allInstances)')
endif
and spec.constrainedElement= Sequence {ge})
post Disjointness induced effects:
let subsSeq:Sequence(Type) = p.hasSubtypes-> asSequence()
let numberSubtypes:Integer=p.hasSubtypes->size()
in Sequence {1..numberSubtypes}->forAll(i,j:Integer | i<j and disj.oclIsNew()
and disj.oclIsTypeOf(Disjointness)
and disj.partition = p
and disj.constrainedElement = Sequence {subsSeq->at(i), subsSeq-> at(j)}))
and if disj.constrainedElement-> first().DisjSatisfiedBySchema (disj.constrainedElement-> last()) then
disj.satisfaction = Satisfaction::BySchema and disj.body= ''
else
disj.satisfaction = Satisfaction::Enforced
and disj.body=disj.constrainedElement->first().Name.oclAsType(String).concat('.allInstances-> excludesAll(').concat(disj.constrainedElement->last().Name.oclAsType(String)).concat('.allInstances)')
endif
post Covering induced effects:
cov.oclIsNew() and cov.oclIsTypeOf(Covering)
and cov.partition= Sequence {p}
and
if p.constrainedElement.hasSuper().CovSatisfiedBySchema(p.constrainedElement.hasSubtypes()) then
cov.satisfaction =Satisfaction::BySchema and cov.body=''
else
cov.satisfaction =Satisfaction::Enforced
cov.satisfaction= 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.hasSuper().Name.oclAsType(String)).concat('.allInstances)')
endif

4. Comparison of UML Tools for profile implementation

Nowadays, there are plenty of CASE tools (see [ObyD02] or [Jeck02] for a complete list) to model information systems with UML. We must select an UML CASE tool in order to implement the partition profile. In this section we present a brief comparison of different UML tools and select one of them to implement the profile.

To do this comparison we have selected some of the most popular and relevant ones. We think this selection is representative enough and that it could be used as a basis for extending the comparison to other tools.

Given that the majority of CASE tools have the same technology limitations, to select one of them, we compare the extensibility and the profile implementation facilities offered by the tools. We do not take into account other aspects as are code generation capacities, usability, diagrams supported, price, etc., although undoubtedly they could also be considered worthwhile for a user willing to implement his own profile. For a description of these functionalities see, for instance, [jeck02].

This comparison is based on the following criteria:

- Scripting: Can the user create scripts to access (and modify) the user model (i.e. metamodel instances)?
- Metamodel: If the tool supports scripting, it must offer some kind of interface to access the user model. So, the question is: Which kind of metamodel does the interface offered by the tool follow? Note that this does not imply the tool uses the same metamodel internally.
- Profile: Does the tool support the notion of profile as a whole or it only allows the definition of single stereotypes?

Name	Vendor	Scripting	Metamodel	Profile
Rational Rose v2002 [Rose02]	Rational	Yes	Proprietary (Rational Extensibility Interface)	No
Objecteering/UML Enterprise Edition 5.2 [Obje02]	Softeam	Yes	UML Metamodel v.1.3	Yes
Together ControlCenter 6.0 [Toge02]	TogetherSoft	Yes	Proprietary (Together Open API)	No
Microsoft Visio [Visi02]	Microsoft	Yes	Proprietary (Visio Object Model)	No
ArgoUML v0.12 [1] [Argo02]	Open source	No	No	No

Table 2: Comparison of UML CASE tools

1 Since ArgoUML is an open source tool, these features could be implemented on one's own

As can be seen from Table 2, the majority of the tools supports scripting, thus, they would allow the implementation of the profile evolution operations. However, not all the tools offer an interface based on the UML metamodel. Since the evolution operations are specified using the UML metamodel, when the tool offers a proprietaty metamodel interface it would be necessary to adapt them, usually with a significant effort.

The only tool supporting the notion of profile, allowing access to its metamodel and whose metamodel follows the UML Metamodel is Objecteering/UML. Thus, we will implement the partition profile with the Objecteering/UML UML Profile Builder (one of the tools included in Objecteering/UML). Once the new profile is implemented, it can be exported to become available for end-users through the Objecteering/UML UML Modeler (the Objecteering/UML modeling tool for software developers).

5. Implementing the Partition profile with Objecteering/UML Profile Builder

In this section we show the implementation of the partition profile, using the Objecteering/UML UML Profile Builder.

However, we are unable to implement the whole profile as designed in section 3 because this tool does not comply exactly with the UML metamodel (neither v.1.3), although it is the best possible choice. In some aspects, such as dependency relationships, implements a more constrained UML metamodel. For instance, the tool metamodel does not allow dependency relationships having more than one supplier. Note that, as a consequence, we cannot specify entity types derived by union, by specialization, by exclusion or, in general, derived types with more than one supplier.

Due to these tool limitations, this implementation of the partition profile does not take into account the possibility that the entity types involved in the partition be derived. Thus, all constraints will have to be enforced.

To create a new profile we must define a set of new stereotypes with its constraints and operations. In this case there are four stereotypes (<<partition>>, <<specialization>>,<<covering>> and <<disjointness>>) and five operations (AddPartition, AddSubtype, RemoveSubtype, ReplaceSubtypes and RemovePartition) to implement. Since we do not consider derived entity types in the partition, it is not necessary to implement the Resize operation (it would be equal to the AddSubtype one).

The definition of the stereotypes is easily done by tool interaction. The only thing to do is to provide all the stereotype information (base class, tagged values, etc). Figure 9 shows the definition of the <<partition>> stereotype. In this case no tagged values are defined.

Once the new stereotype has been defined, we can specify operations for it. These operations are implemented using J methods. The J methods are methods attached to a specific metaclass or stereotype, and executed using an instance of that metaclass. They are written in the J language. The J language is the proprietary tool language. Nevertheless, its "Java-like" syntax makes it easy to use. It presents some interesting features (see [Soft99] for more details) like model handling capacities. Thanks to these features, the J language is perfectly suitable for implementing the profile operations. In fact, everything the user can achieve through Objecteering/UML UML Modeler can be achieved through programming with the J language.

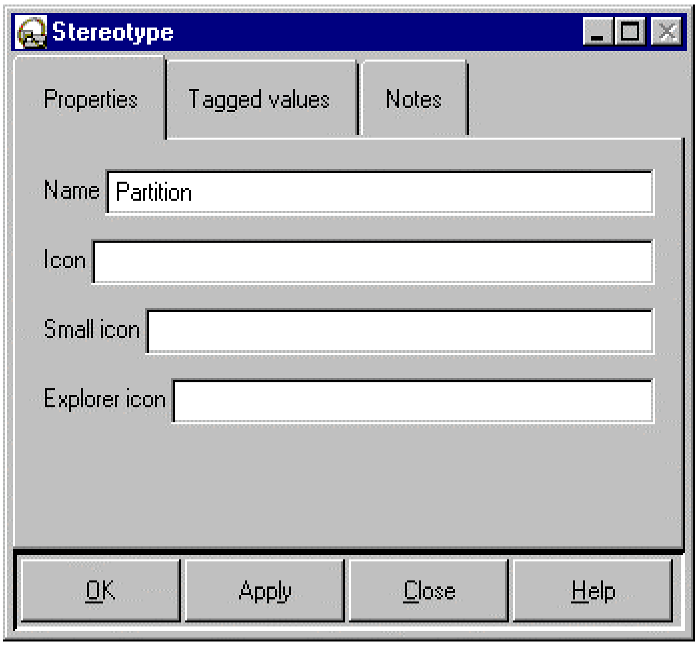


Figure 9: Partition stereotype

The J method *AddPartition* checks the preconditions *PartitionMustNotExist* and *SubsCannotBeEmpty* and ensures the postconditions *CreatePartition, SpecializationInducedEffects, DisjointnessInducedEffects* and *CoveringInducedEffects* defined in the operation design. Other preconditions as *PartitionMustNotExist* and *TwoSubsNotEqual* are already verified by the CASE tool, so there is no need to enforce them in the method code.

The J method corresponding to the *AddPartition* operation, previously specified and designed in OCL (see section 3) is:

```
Class [] CVec;
Class [] CVec2;
Class Supertype;
Class Aux;
NameSpace N;
Generalization G;
Generalization[] GVec;
Generalization[] GVecAux;
Constraint newPartition;
Constraint C;
int i;
String s;
```

Var declaration

```
String [] SVec;
String body;
JBox Form;
boolean errorD;
```

```
//Creation of the Dialog Box
Form=createJBox("id1","Create Partition",JLayoutVertical,false);
Form.addField("f1","Discriminant","",JControlTextField,200);
Form.addText("tx1","Subtypes of the partition","",400,250);

if (Form.show())
{
 sessionBegin("CreatePartition",true);
 N=this.OwnerNameSpace;
 Form.getValue("tx1",s);
 SVec = s.segment(NL);
 //We turn the names of the classes into class references
 SVec
 {
  Aux=N.getClassByName(this); //Searching the class
  if(notVoid (Aux)) CVec.addElement(Aux);
  else infoUser("Warning","The class " + this + " does not exists ");
 }
```

User interaction

```
 errorD=false;
 Form.getValue("f1",s);
 GVecAux=this.SpecializationGeneralization;
 GVecAux
 {
  if (this.Discriminator=s) errorD=true;
 }
```

Pre PartitionMustNotExist

```
 if(!errorD and CVec.size()>0) // There must be at least one subtype
 {
```

Pre SubsCanNotBeEmpty

```
  //Create the new partition
  newPartition=createConstraint("","Partition","");

  //Create the Generalizations
  Supertype=this; //Save a reference of the class
  CVec
  {  // For each subtype
   G=this.createAndAddGeneralization(Supertype);
   G.setDiscriminator(s);
   G.addConstraint(newPartition); //Add the generalization to the Partition
   GVec.addElement(G);
  }
```

Post CreatePartition

```
  //For each generalization we create a Specialization
  GVec
  {
   body=Supertype.Name + ".allInstances->includesAll(" +
this.SubTypeNameSpace.Name+ ".allInstances)";
   this.createAndAddConstraint("","Specialization",body);
  }
```

Post Specialization Induced Effects

```
  //Create a covering constraint for the generalitzacions
  C=createConstraint("","Covering","");
  body="";
  i=1;
  GVec
  {
   if(i==GVec.size()) {body=body + this.SubTypeNameSpace.Name
+".allInstances)->includesAll(";}
   else
    if(i==1) {body=body + this.SubTypeNameSpace.Name+
".allInstances->union(" ;}
    else {body=body + this.SubTypeNameSpace.Name+".allInstances)
         ->union(" ;}
   this.addConstraint(C);
   i=i+1;
  }
  body=body+Supertype.Name + ".allInstances)";
  C.setBody(body);
```

Post Covering Induced Effects

```
  //Create a disjointness restriction for each pair of subtypes
  i=1;
  CVec2=CVec;
  CVec2
  {
   Aux=this;
   CVec2.remove(Aux);
   CVec2
   {
    body=Aux.Name+".allInstances->excludesAll("+ this.Name +
         ".allInstances)";
    C=createConstraint("","Disjointness",body);
    this.addConstraint(C);
    Aux.addConstraint(C);
   }
  }
 }
```

Post Disjointness Induced Effects

```
 else
 {
  if(errorD) infoUser("Error",
" There is already a generalization with the same discriminator");
  if(CVec.size()==0) infoUser("Error",
" A partition must have at least one subtype");
 }
 sessionEnd();
}
Form.delete();
```

Operation ending

Once we have implemented the profile and the evolution operations, now we show how the designers can interact with the CASE tool and use the evolution operations. If the designer wants to create a new partition in an UML conceptual schema, s/he must invoke the operation *AddPartition*. Figures 10 and 11 show the process of creating a new partition using this operation. First (Figure 10), we select the supertype of the new partition and execute the operation. Then (Figure 11), The user through a dialog box provides the information about the discriminator and the subtypes involved in the partition. The supertype is the entity type we have executed the operation over. The final result is shown in figure 12.

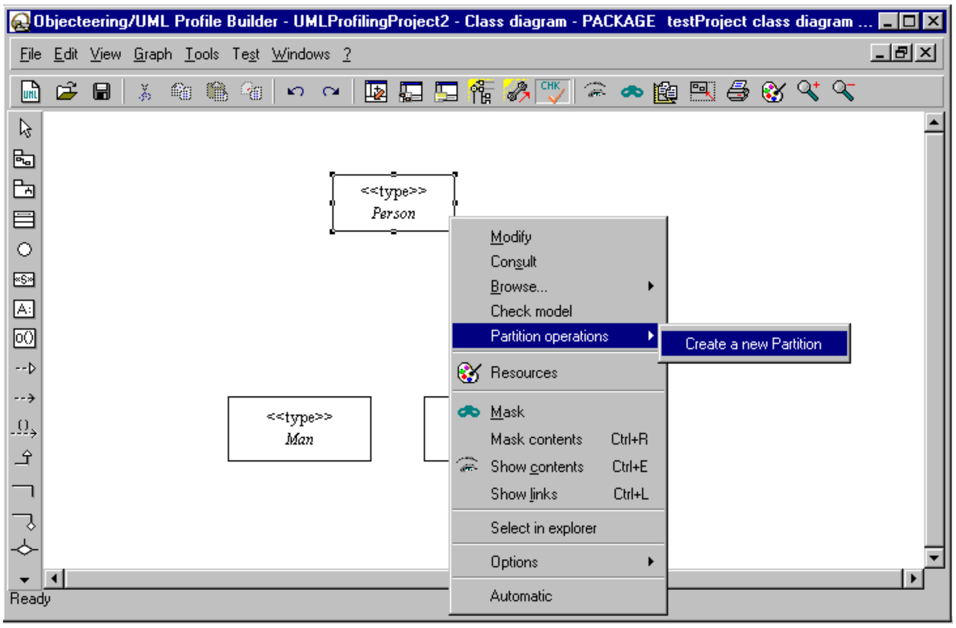


Figure 10: Executing the AddPartition operation

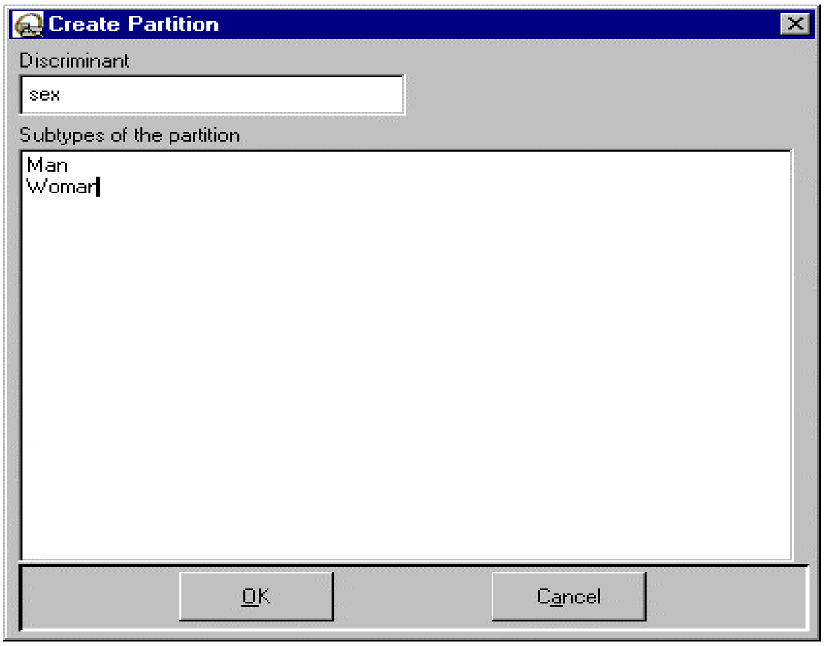


Figure 11: Providing the operation information

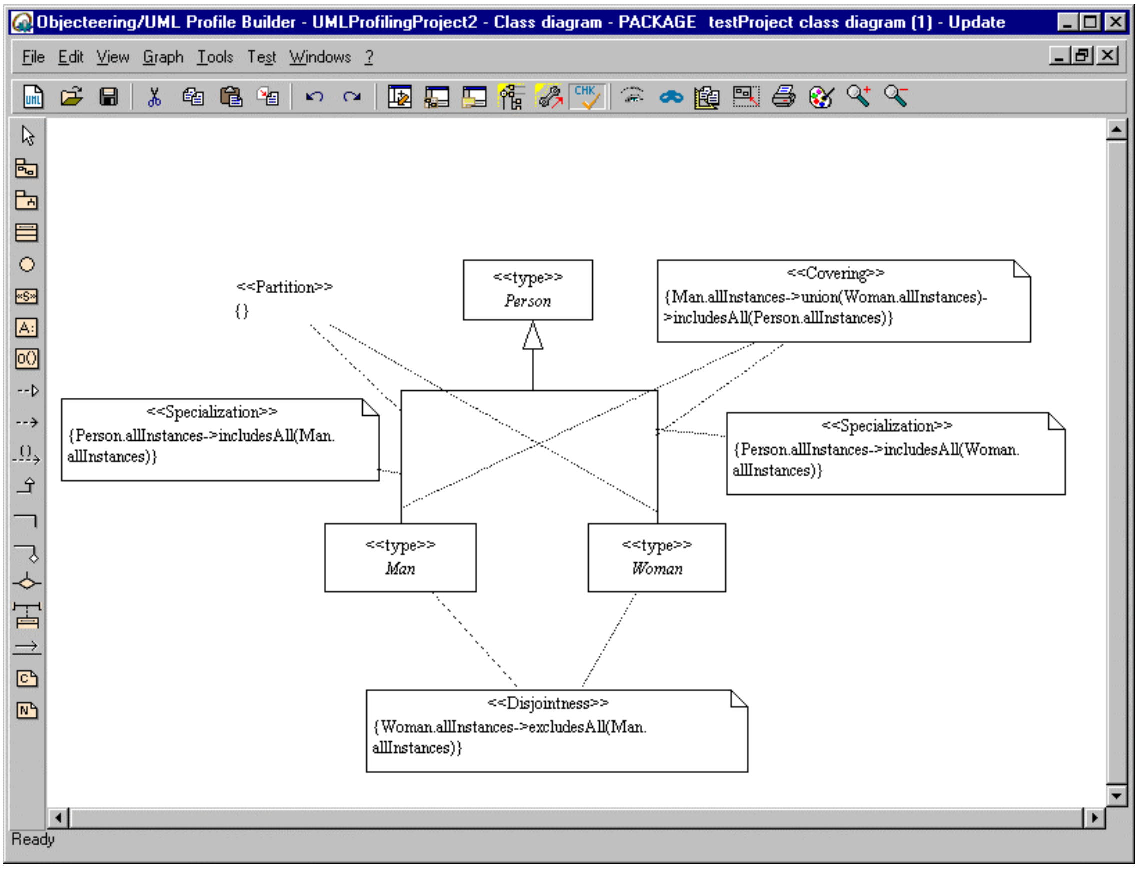


Figure 12: Example of creating a partition

The rest of the operations are implemented in a similar way. Their complete code can be found in the Appendix.

6. Conclusions

In this paper we have described the implementation of a UML profile in an UML CASE tool, using the Partition profile as an example throughout the paper. During the process, we have taken into account the limitations of the present UML CASE tools. Moreover, we have explained some design strategies to avoid them. Finally, we have implemented, partially, the profile within the Objecteering/UML tool.

From this experience we may conclude that, relating to this aspect, UML CASE tools still lack of maturity. In fact, the significant importance of profiles in conceptual modeling has not been accompanied by the necessary support of existing CASE tools. We think it is necessary a big effort from the UML CASE tool vendors in order to allow the software community easily develops and uses them.

Acknowledgments

We would like to thank Ernest Teniente and Antoni Olivé for their useful comments. This work was supported by the Ministerio de Ciencia y Tecnología and Feder under project TIC2002-00744.

References

[Argo02] ArgoUML, http://argouml.tigris.org/, visited November 2002

[BCG+87] Banerjee, J.; Chou, H-T.; Garza, J.F.; Kim, W.; Woelk, D.; Ballou, N. *"Data Model Issues for Object-Oriented Applications"*. ACM TOIS Vol. 5, No. 1, January 1987, pp. 3-26.

[CoST02] Costal, D.; Sancho, M-R; Teniente, E. *"Understanding Redundancy in UML Models for Object-Oriented Analysis"*. CAiSE 02, LNCS 2348, pp. 659-674.

[GoOl02] Gómez, C.; Olivé A. "*Evolving Partitions in Conceptual Schemas in the UML (Extended Version)*", UPC Technical report, LSI-02-15-R.

[Jeck02] Jeckle, M. "UML Tools (Case & Drawing)", http://www.jeckle.de/umltools.htm, visited November 2002

[LoOl00] López, J.R.; Olivé, A. *"A Framework for the Evolution of Temporal Conceptual Schemas of Information Systems*", CAiSE'00, LNCS 1789, pp.369-386.

[Mant93] Manthey, R. *"Beyond Data Dictionaries: Towards a Reflective Architecture of Intelligent Database Systems",* DOOD'93, Springer-Verlag, pp. 328-339.

[Obje02] Objecteering/UML, http://www.objecteering.com/us/produits.php, visited November 2002

[ObyD02] Objects By Design; "UML Modeling Tools", http://www.objectsbydesign.com/tools/umltools_byProduct.html, visited November 2002

[Oliv91] Olivé, A. *"Integrity Constraints Checking In Deductive Databases"*, VLDB 1991, Morgan Kaufmann, pp.513-523

[OMG01] *"Unified modeling Language Specification"*, Version 1.4, September 2001.

[PeÖz93] Peters, R.J.; Özsu, T. *"Reflection in a Uniform Behavioral Object Model"*. Proc. ER'93, Arlington, LNCS 823, Springer-Verlag, pp. 34-45.

[Rose02] Rational Rose Enterprise Edition, http://www.rational.com/products/rose/rsds/renterprise_generic.jsp, visited November 2002

[Soft99] Softeam; "UML Profiles and the J Language";White Paper, 1999, http://www.objecteering.com/pdf/us/uml_profiles.pdf, visited November 2002

[Teni00] Teniente, E. *"Deductive Databases"*, Advanced Database Technology and Design, Eds. Mario Piattini, Oscar Díaz, pp. 91-136.

[TeUr03] Teniente, E.; Urpí, T. *"On the Abductive or Deductive Nature of Database Schema Validation and Update Processing Problems"*, to appear in Theory and Practice of Logic Programming.

[Toge02] Together ControlCenter, http://www.togethersoft.com/products/controlcenter/index.jsp , visited November 2002

[Visi02] Microsoft Visio, http://www.microsoft.com/office/visio/default.asp, visited November 2002

Appendix

In this appendix we show the design and the implementation of the partition evolution operations.

1. AddSubtype operation

1.1 AddSubtype operation design

context Partition::AddSubtype (subtype:Type)
pre PartitionNotIncludesSubtype:
self.hasSubtypes ()-> excludes(subtype)
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)
post Specialization induced effects:
spec.oclIsNew()
and spec.oclIsTypeOf(Specialization)
and spec.partition=self
if ge.parent.SpecSatisfiedBySchema(ge.child) then
  spec.satisfaction=Satisfaction::BySchema
  and spec.body=''
else
  spec.satisfaction=Satisfaction::Enforced
  and spec.body=ge.parent.Name.oclAsType(String).concat('.allInstances->includesAll(').
  concat(ge.child.Name.oclAsType(String)).concat('.allInstances)')
endif
and spec.constrainedElement= Sequence {ge})
post Disjointness induced effects:
let subsSeq:Sequence(Type) = self.hasSubtypes->excludes(subtype)-> asSequence()
let numberSubtypes:Integer=self.hasSubtypes->size() -1
in Sequence {1..numberSubtypes}->forAll(i:Integer | disj.oclIsNew()
and disj.oclIsTypeOf(Disjointness)
and disj.partition = self
and disj.constrainedElement = Sequence {subsSeq->at(i), subtype}))
and if disj.constrainedElement->first().DisjSatisfiedBySchema (disj.constrainedElement->last()) then
  disj.satisfaction = Satisfaction::BySchema
  and disj.body= ''
else
  disj.satisfaction = Satisfaction::Enforced
  and disj.body=disj.constrainedElement->first().Name.oclAsType(String).concat('.allInstances-> excludesAll(').concat(disj.constrainedElement->last().Name.oclAsType(String)).
  concat('.allInstances)')
endif
post Covering induced effects:
let cov:Covering = Constraint.allInstances->select (c.stereotype->one(name='Covering') and c.constrainedElement=self)
if self.constrainedElement.hasSuper().CovSatisfiedBySchema(self.constrainedElement.hasSubtypes()) then
  cov.satisfaction =Satisfaction::BySchema

```
  and cov.body=’’
else
  cov.satisfaction =Satisfaction::Enforced
  cov.satisfaction= 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.hasSuper().Name.oclAsType(String)).concat(‘.allInstances)’)
endif
```

1.2 AddSubtype operation implementation

```
Generalization[] GVec;
Class Subtype;
NameSpace N;
String NSub;
JBox Form;
boolean error;
Generalization Aux;
Generalization G;
Constraint C;
String body;
int i;

//Create the form
Form=createJBox("id1","Addition of a new Subtype",JLayoutVertical,false);
Form.addField("f1","Subtype Name","",JControlTextField,200);
if (Form.show())
{
  sessionBegin("AddSubtype",true);
  Form.getValue("f1",NSub);
  //Obtain a class reference
  getItemSet(this.ConstrainedElementModelElement,0,Aux);
  N=Aux.SuperTypeNameSpace.OwnerNameSpace;
  Subtype=N.getClassByName(NSub);
  if(notVoid (Subtype))
  {  //if the class exists
    //Check that the new subtype is not part of the partition yet
    error:=false;
    this.ConstrainedElementModelElement
    {
      if (SubTypeNameSpace.Name==Subtype.Name)
      {  error:=true;
        infoUser("Error","The class " + Subtype.Name + " is already part of the Partition");
      }
    }
    if (!error)
    { //Add the new subtype
      //Add the disjointness constraints
      this.ConstrainedElementModelElement
      {
        SubTypeNameSpace
        {
          body=Subtype.Name+".allInstances->excludesAll("+ this.Name + ".allInstances)";
          C=createConstraint("","Disjointness",body);
          Subtype.addConstraint(C);
          addConstraint(C);
```

```
    }
  }
  //Add the generalization
  getItemSet(this.ConstrainedElementModelElement,0,Aux); //Take one of the existing generalizations
  G=Subtype.createAndAddGeneralization(Aux.SuperTypeNameSpace); //Add the same Supertype and
Discriminant
  G.setDiscriminator(Aux.Discriminator);
  G.addConstraint(this); //Add the generalization to the partition
  //Add the specialization constraint
  body=Aux.SuperTypeNameSpace.Name + ".allInstances->includesAll(" + Subtype.Name+
".allInstances)";
  G.createAndAddConstraint("","Specialization",body);
  //Add the covering constraint
  i=1; body="";
  GVec=this.ConstrainedElementGeneralization;
  GVec
  {
   if (i==GVec.size()) {body=body + this.SubTypeNameSpace.Name+".allInstances)->includesAll(";}
   else
    if(i==1) {body=body + this.SubTypeNameSpace.Name+".allInstances->union(" ;}
    else {body=body + this.SubTypeNameSpace.Name+".allInstances)->union(" ;}
   i=i+1;
  }
  body=body+G.SuperTypeNameSpace.Name + ".allInstances)";
  //Add the new subtype to the covering constraint
  Aux.ConstraintDefinitionConstraint
  {
   if(notVoid(ExtensionStereotype))
   {  if(ExtensionStereotype.Name=="Covering")
     {  G.addConstraint(this);
      this.setBody(body);}
   }
  }

 }
 }else infoUser("Warning","The class " + NSub + " does not exist ");
 sessionEnd();
}
Form.delete();
```

2. RemoveSubtype operation

2.1 RemoveSubtype operation design

context Partition::RemoveSubtype (subtype:Type)
pre PartitionIncludesSubtype:
self.hasSubtypes -> includes(subtype)
pre NotUniqueSubtype:
self.hasSubtypes -> size() >1
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.constrainedElement = self.constrainedElement@pre ->excludes(generalizationToRemove)

Generalization.allInstances->excludes(generalizationToRemove)
post Specialization induced effects:
let spec:Specialization = Constraint.allInstances->select (c.stereotype->one(name='Specialization') and c.constrainedElement=generalizationToRemove)
Constraint.allInstances->select(c.stereotype->one(name='Specialization'))= Constraint.allInstances@pre->select(c.stereotype->one(name='Specialization')) ->excludes (spec)
post Disjointness induced effects:
let disjs:Disjointness = Constraint.allInstances->select (c.stereotype->one(name='Disjointness') and c.constrainedElement->first()=subtype or c.constrainedElement->first()=subtype and c.partition=self)
Constraint.allInstances->select(c.stereotype->one(name='Disjointness'))= Constraint.allInstances@pre->select(c.stereotype->one(name=' Disjointness')) ->excludes (disjs)
post Covering induced effects:
let cov:Covering = Constraint.allInstances->select (c.stereotype->one(name='Covering') and c.constrainedElement=self)
if self.constrainedElement.hasSuper().CovSatisfiedBySchema(self.constrainedElement.hasSubtypes()) then
  cov.satisfaction =Satisfaction::BySchema
  and cov.body=''
else
  cov.satisfaction =Satisfaction::Enforced
  cov.satisfaction= 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.hasSuper().Name.oclAsType(String)).concat('.allInstances)')
endif

2.2 RemoveSubtype operation implementation

```
Class Subtype;
NameSpace N;
String NSub;
JBox Form;
boolean error;
Generalization Aux;
Generalization G;
Generalization[] GVec;
Class[] CVecTotal;
Constraint C;
Partition Part;
int i;
String body;

//Create the form
Form=createJBox("id1","Remove a Subtype",JLayoutVertical,false);
Form.addField("f1","Subtype Name","",JControlTextField,200);
if (Form.show())
{
  sessionBegin("RemoveSubtype",true);
  Form.getValue("f1",NSub);
  //Obtain a class reference
  getItemSet(this.ConstrainedElementModelElement,0,Aux);
  N=Aux.SuperTypeNameSpace.OwnerNameSpace;
  Subtype=N.getClassByName(NSub);
  if(notVoid (Subtype))
```

```
 {  //if the class exists
   error:=true;
   GVec=this.ConstrainedElementGeneralization;
   GVec
   {
     CVecTotal.addElement(this.SubTypeNameSpace);
     if (this.SubTypeNameSpace==Subtype) { G=this; error=false;}
   }
   //Check that the subtype is part of the partition
   if (error) infoUser("Error","The class " + Subtype.Name + " is not part of the Partition");
   //Erase the generalization, specialization and link to the covering constraint
   G.ConstraintDefinitionConstraint
   {
     if(ExtensionStereotype.Name=="Specialization")
       this.delete;
     else
         G.eraseConstraintDefinition(this);
   }
   G.delete;
   //Delete the disjointness restrictions
   Subtype.ConstraintDefinitionConstraint
   {
     if(notVoid(ExtensionStereotype))
     { if(this.ExtensionStereotype.Name=="Disjointness")
         this.delete;
     }
   }
   // Recompute the Covering Constraint
   GVec=this.ConstrainedElementGeneralization;
   i=1; body="";
   GVec
   {
     if (i==GVec.size()) {body=body + this.SubTypeNameSpace.Name+".allInstances)->includesAll(";}
     else
       if(i==1) {body=body + this.SubTypeNameSpace.Name+".allInstances->union(" ;G=this;}
       else {body=body + this.SubTypeNameSpace.Name+".allInstances)->union(" ;}
      i=i+1;
   }
   body=body+G.SuperTypeNameSpace.Name+ ".allInstances)";
   G.ConstraintDefinitionConstraint
   {  //Looking for the Covering Constraint
     if(ExtensionStereotype.Name=="Covering")
       this.setBody(body);
   }
 }else infoUser("Error","The class " + NSub + " does not exist ");
 sessionEnd();
}
Form.delete();
```

3. ReplaceSubtypes operation

3.1 ReplaceSubtypes operation design

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 NotSameSubtypes:
newSubtypes -> forAll(sub1,sub2: Type | sub1<>sub2)
post:
let generalizationsToRemove:Set(Generalization) =
self.contrainedElement@pre.oclAsType(Generalization) ->select(oldSubtypes -> includes(child))
in
self.constrainedElement -> excludesAll(generalizationsToRemove)
and
Generalization.allInstances->excludesAll(generalizationsToRemove)
and **--** Create a generalization for each new subtype
newSubtypes→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))
post Specialization induced effects:
//Old specialization constraints
let specs:Specialization = Constraint.allInstances->select (c.stereotype->one(name='Specialization') and generalizationsToRemove->includes(c.constrainedElement))
Constraint.allInstances->select(c.stereotype->one(name='Specialization'))= Constraint.allInstances@pre->select(c.stereotype->one(name='Specialization'))->excludes (specs)
//New specialization constraints
self.constrainedElement->forAll(ge:Generalization | spec.oclIsNew()
and spec.oclIsTypeOf(Specialization)
and spec.partition=self
if self.constrainedElement.parent.SpecSatisfiedBySchema(self.constrainedElement.child) then
  spec.satisfaction=Satisfaction::BySchema
  and spec.body=''
else
  spec.satisfaction=Satisfaction::Enforced
  and spec.body=self.constrainedElement.parent.Name.oclAsType(String).concat('.allInstances->includesAll(').concat(self.constrainedElement.child.Name.oclAsType(String)).
  concat('.allInstances)')
endif
and spec.constrainedElement= Sequence {ge})
post Disjointness induced effects:
//Old disjointness constraints
let disjs:Disjointness = Constraint.allInstances->select (c.stereotype->one(name='Disjointness') and generalizationsToRemove.child->includes(c.constrainedElement->first() or generalizationsToRemove.child->includes(c.constrainedElement->last()) and c.partition=self)
Constraint.allInstances->select(c.stereotype->one(name='Disjointness'))= Constraint.allInstances@pre->select(c.stereotype->one(name=' Disjointness')) ->excludes (disjs)
//New disjontness constraints
let subsSeq:Sequence(Type) = self.hasSubtypes-> asSequence()
let numberSubtypes:Integer=self.hasSubtypes->size()

```
in Sequence {1..numberSubtypes}->forAll(i,j:Integer | i<j and disj.oclIsNew()
and disj.oclIsTypeOf(Disjointness)
and disj.partition = p
and disj.constrainedElement = Sequence {subsSeq->at(i), subsSeq-> at(j)}))
and if disj.constrainedElement-> first().DisjSatisfiedBySchema (disj.constrainedElement-> last()) then
  disj.satisfaction = Satisfaction::BySchema
  and disj.body= ‘’
else
  disj.satisfaction = Satisfaction::Enforced
  and disj.body=disj.constrainedElement->first().Name.oclAsType(String).concat(‘.allInstances->
  excludesAll(’).concat(disj.constrainedElement->
  last().Name.oclAsType(String)).concat(‘.allInstances)’)
endif
```

post Covering induced effects:

```
//Modified covering constraint
let cov:Covering = Constraint.allInstances->select (c.stereotype->one(name=’Covering’) and
c.constrainedElement=self)
if self.constrainedElement.hasSuper().CovSatisfiedBySchema(self.constrainedElement.hasSubtypes())
then
  cov.satisfaction =Satisfaction::BySchema
  and cov.body=’’
else
  cov.satisfaction =Satisfaction::Enforced
  cov.satisfaction= 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.hasSuper().Name.oclAsType(String)).concat(‘.allInstances)’)
endif
```

3.2 ReplaceSubtypes operation implementation

```
Class [] CVecTotal;
Class [] CVecOld;
Class [] CVecNew;
Class Aux;
Generalization GenAux;
Generalization [] GVec;
Generalization [] GVecOld;
NameSpace N;
Generalization G;
Generalization GNew;
Constraint Part;
Constraint C;
int i;
String s;
String [] SVec;
String body;
JBox Form;

//Crete the form
Form=createJBox("id1","Replace subtypes of the partition",JLayoutVertical,false);
Form.addText("txNou","New subtypes","",400,250);
Form.addText("txOld","Old subtypes","",400,250);
if (Form.show())
{
  sessionBegin("Replace subtypes",true);
```

```
Part=this;
GVec=Part.ConstrainedElementGeneralization;
GVec
{
   CVecTotal.addElement(this.SubTypeNameSpace);
}
getItemSet(this.ConstrainedElementModelElement,0,GenAux);
N=GenAux.SuperTypeNameSpace.OwnerNameSpace;
//Processing the old subtypes
Form.getValue("txOld",s);
SVec = s.segment(NL);
SVec
{
  Aux=N.getClassByName(this);
  if(notVoid (Aux))
  {
     if(CVecTotal.contains(Aux))
     {  CVecOld.addElement(Aux);
        getItemSet(Aux.ParentGeneralization,0,G);GVecOld.addElement(G);
     }
     else infoUser("Warning","The class" + this + " is not in the partition ");
  }
  else infoUser("Warning","The class" + this + " does not exist ");
}
//Processing the new subtypes
Form.getValue("txNou",s);
SVec = s.segment(NL);
SVec
{
  Aux=N.getClassByName(this);
  if(notVoid (Aux))
  {  if(!CVecTotal.contains(Aux))
        CVecNew.addElement(Aux);
     else infoUser("Warning","The class " + this + " is already in the partition ");
  }
  else infoUser("Warning","The class " + this + " does not exist ");
}
//If both sets are not empty, we can proceed
if (CVecNew.size()>0 && CVecOld.size()>0)
{
  //Delete the old subtypes
  //Delete their generalizations and specializations
  GVecOld
  {
    G=this;
    ConstraintDefinitionConstraint
    {
      if(ExtensionStereotype.Name=="Specialization")
        this.delete;
      else
          G.eraseConstraintDefinition(this);
    }
    this.delete; //Erase the generalization
  }
  //Delete the disjointness restrictions
```

```
CVecOld
{
 ConstraintDefinitionConstraint
 {
  if(notVoid(ExtensionStereotype))
  { if(this.ExtensionStereotype.Name=="Disjointness")
     this.delete;
  }
 }
}
//Add the new subtypes
getItemSet(Part.ConstrainedElementModelElement,0,G);
CVecNew
{
 //Add the generalization
 GNew=this.createAndAddGeneralization(G.SuperTypeNameSpace); //Add the same Supertype and discriminant
 GNew.setDiscriminator(G.Discriminator);
 GNew.addConstraint(Part); //Add the generalization to the partition
 //Add the Specialization constraint
 body=G.SuperTypeNameSpace.Name + ".allInstances->includesAll(" + this.Name+ ".allInstances)";
 GNew.createAndAddConstraint("","Specialization",body);
 //Add the covering constraint
 G.ConstraintDefinitionConstraint
 {
  if(notVoid(ExtensionStereotype))
  {  if(ExtensionStereotype.Name=="Covering")
      GNew.addConstraint(this);
  }
 }
}
//Compute and modify the body of the covering constraint
GVec=Part.ConstrainedElementGeneralization;
i=1; body="";
GVec
{
 if (i==GVec.size()) {body=body + this.SubTypeNameSpace.Name+".allInstances)->includesAll(";}
 else
  if(i==1) {body=body + this.SubTypeNameSpace.Name+".allInstances->union(" ;}
  else {body=body + this.SubTypeNameSpace.Name+".allInstances)->union(" ;}
 i=i+1;
}
body=body+G.SuperTypeNameSpace.Name + ".allInstances)";
G.ConstraintDefinitionConstraint
{
 if(notVoid(ExtensionStereotype))
 {  if(ExtensionStereotype.Name=="Covering")
     this.setBody(body);
 }
}
//Add the disjointness constraints
CVecOld
{ CVecTotal.remove(this);}
CVecNew
{ CVecTotal.addElement(this);}
```

```
  //Create the disjointness constraints
  CVecNew
  {
   Aux=this;
   CVecTotal
   {
    body=Aux.Name+".allInstances->excludesAll("+ this.Name + ".allInstances)";
    CVecTotal.remove(Aux); //avoid repetitions
    C=createConstraint("","Disjointness",body);
    Aux.addConstraint(C);
    addConstraint(C);
   }
  }
 } else infoUser("Error", "At least, It must be one subytpe to add and one to replace");
 sessionEnd();
}
Form.delete();
```

4. RemovePartition operation

4.1 RemovePartition operation design

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)
post Specialization induced effects:
let specs:Specialization = Constraint.allInstances->select (c.stereotype->one(name='Specialization') and c.partition=self)
Constraint.allInstances->select(c.stereotype->one(name='Specialization'))= Constraint.allInstances@pre->select(c.stereotype->one(name='Specialization')) ->excludes (specs)
post Disjointness induced effects:
let disjs:Disjointness = Constraint.allInstances->select (c.stereotype->one(name='Disjointness') and c.partition=self)
Constraint.allInstances->select(c.stereotype->one(name='Disjointness'))= Constraint.allInstances@pre->select(c.stereotype->one(name=' Disjointness')) ->excludes (disjs)
let subsSeq:Sequence(Type) = self.hasSubtypes->excludes(subtype)-> asSequence()
post Covering induced effects:
let cov:Covering = Constraint.allInstances->select (c.stereotype->one(name='Covering') and c.constrainedElement=self)
Constraint.allInstances->select(c.stereotype->one(name='Covering'))= Constraint.allInstances@pre->select(c.stereotype->one(name=' Covering')) ->excludes (cov)

4.2 RemovePartitionoperation implementation

```
Generalization[] GVec;
Class[] CVec;

sessionBegin("Remove",true);
GVec=this.ConstrainedElementGeneralization;
GVec
```

```
{
  CVec.addElement(this.SubTypeNameSpace);
}
//Delete their generalizations and specializations and covering constraints
GVec
{
 ConstraintDefinitionConstraint
  { this.delete; }
 this.delete;
}
//Delete the disjointness restrictions
CVec
{
 ConstraintDefinitionConstraint
  {
   if(notVoid(ExtensionStereotype))
   { if(this.ExtensionStereotype.Name=="Disjointness")
       this.delete;
   }
  }
}
this.delete;
sessionEnd();
```