Automatic Integrity Constraint Evolution due to Model Subtract Operations

Jordi Cabot and Jordi Conesa

Universitat Politècnica de Catalunya
Dept. Llenguatges i Sistemes Informàtics
Jordi Girona 1-3, 08034 Barcelona

[jcabot|jconesa]@lsi.upc.es

Abstract. When evolving Conceptual Schemas (CS) one of the most common operations is the removal of some model elements. This removal affects the set of integrity constraints (IC) defined over the CS. Most times they must be modified to remain consistent with the evolved CS. The aim of this paper is to define an automatic evolutionary method to delete only the minimum set of constraints (or some of their parts) needed to keep the consistency with the CS after subtract operations. We consider that a set of constraints is consistent with an evolved CS when: 1) none of them refer to any removed element of the original CS and 2) the set of constraints is equal or less restrictive than the original one. In this paper we present our method assuming CS defined in UML with ICs specified in OCL, but it can be applied to other languages with similar results.

1. Introduction

Evolution is critical in the life span of an information system (IS) [Sjø92]. Until now, the issues that arise when evolving an IS have been studied from an implementation point of view, specially in the field of schema evolution in relational and object-oriented databases [Ban87, Bre96, Mon93, RCR93]. However, nowadays conceptual schemas (CS) are gaining a lot of relevance in the life cycle of the IS; for instance, in the context of the MDA [OMG03b], they are taken as a basis for the code generation of the IS. In such cases, the evolution must take place at the conceptual schema level.

The evolution of conceptual schemas has been poorly studied. Most of the evolution operations are included in the concepts of refactorings and model transformations [Opd92, Fow99, Por03, OMG02b, GO02].

A complete conceptual schema must include the definition of all relevant integrity constraints [ISO82]. When evolving conceptual schemas one of the most common operations is the removal of some model elements. This removal affects the set of integrity constraints (IC) defined over the CS. Obviously these constraints must be modified to remain consistent with the evolved CS when they reference any of the removed model elements. As far as we know, none of the existing approaches deal with the evolution of integrity constraints in such a case.

In this paper we define a method to automatically evolve ICs assuming CS defined in UML [OMG02a] with ICs specified in OCL [OMG03a]. However, our method could be adapted to other conceptual modeling languages that provide a formal (or semi-formal) language to write ICs, such as EER [GH91] or ORM [Hal01].

A naïve approach to solve this problem would consist of removing all the IC where a subtracted element appears. However, this approach removes constraints that need to be still enforced in the evolved CS. As an example, assume that we have the conceptual schema of Figure 1, devoted to represent the relationship between employees, their departments and the projects where they work, with the following constraints:

```
-- Every employee must be older than 16 and he/she cannot work in the
company for more than 40 years
   context Employee inv IC1: self.age>16 and self.seniority<40

-- Bosses with a seniority >10 must earn more than 30000 euros
   context Employee inv IC2:
   (self.category='boss' and seniority>10) implies salary>30000

-- Bosses or employees with seniority > 5 have a car park
   context Employee inv IC3:
      (self.category='boss' or seniority>5) implies carPark

-- Employees under 18 must be scholarship holders
   context Employee inv IC4:
     self.age>18 or self.category='scholarship holder'

-- The assignment for each employee must be <= 40
   context Employee inv IC5: self.assignment.hours->sum()<=40
```

If we delete the *seniority* attribute this naïve approach will remove the first three constraints. We consider this is an incorrect solution, since then, for instance, we can insert a new *employee* younger than 16 years old or a *boss* without *carPark*, which was not allowed in the original CS.

We would also like to point out that another simple solution, consisting in deleting only the part of the OCL constraint where the deleted elements appear, does not work either. For example, this approach would leave IC2 as *"category='boss' implies salary>30000"*. Note that this constraint is more restrictive than the original one, since before the deletion of *seniority*, a boss could earn a salary lower than 30000 (when his/her seniority was under 10) but now all bosses are forced to grow their salary up to 30000 or more. Thus, employees satisfying the constraint before the removal of the attribute now may violate it, and of course, this is not what we intended. We believe it is not acceptable that deleting an element from a CS results in a more restrictive CS.

Therefore, the aim of this paper is to define an automatic method in order to delete only the minimum set of constraints (or some of their parts) needed to keep the consistency with the CS after the deletion of some model element. We consider that a

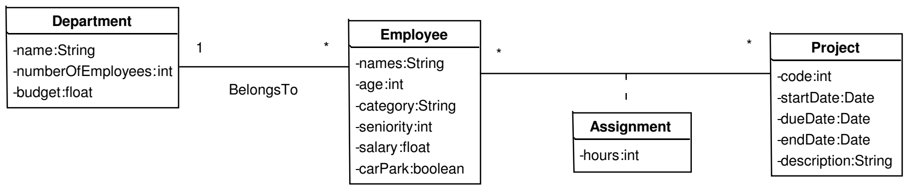


Figure 1 – Example of a conceptual schema

set of constraints is consistent with an evolved CS when none of them refers to a removed element of the original CS and the set of constraints is equal or less restrictive than the original one. Note that this also ensures that if the original CS was satisfiable (i.e. there exists an instantiation of the model for which no IC is violated) the resulting CS will also be satisfiable.

The rationale of our method is to translate each OCL constraint into a sort of conjunctive normal form, and then, apply an algorithm that, taking into account the specific structure of the constraint, decides whether the constraint (or a part of it) is still relevant for the evolved CS.

We will see that applying our approach to previous constraints, only one constraint would be deleted (IC2), another two would not be affected (IC4 and IC5) while the others would be simplified (IC1 and IC3):

```
context Employee inv IC1: self.age>16
context Employee inv IC3: category='boss' implies carPark
```

The rest of the paper is structured as follows. Next section describes the different removal operations that can be executed over a CS. Section 3 describes the transformation of an OCL constraint to its conjunctive normal form. Then, in Section 4, we introduce the OCL metamodel that it is needed in Section 5 to present our method to evolve the constraints. Afterwards, we present some optimizations in Section 6. Finally, we give our conclusions and point out future work in Section 7.

2. Schema Evolution Operations

Conceptual schema changes can be seen as additive, subtractive or both [Mon93]. In additive changes new elements are created without altering the existing ones. On the other hand, a subtractive change involves the removal of one or more model elements.

Several subtractive operations are defined in the field of schema evolution; much of these operations have been created to be applied to software or databases [Ban87, Bre96, Mon93, RCR93], but not to conceptual schemas. Therefore, we extend their set of subtractive operations to allow the user to delete any model element of a conceptual schema. The list of operations that we deal with is the following:

- Removal of an attribute.
- Removal of a specialization/generalization link. We admit CS with multiple inheritance.
- Removal of a class: a deletion of a class implies the deletion of all its attributes, associations and generalizations in which it takes part.
- Removal of an association: the deletion of an association implies the deletion of all its association ends and generalization relationships.
- Removal of an association end: To delete an association end its owner association must be a n-ary association with n>2.
- Removal of an association class: this deletion is treated as an elimination of the association followed by the deletion of its class

3. Transforming OCL constraints to conjunctive normal form

The conversion of an OCL constraint to a conjunctive normal form (CNF) is necessary to reduce the complexity of its treatment in the next sections. We could also have chosen the disjunctive normal form instead. In this section we describe the well-known rules for transforming a logical formula into a conjunctive normal form and we adapt them to the case of OCL constraints.

A logical formula is in conjunctive normal form if it is a conjunction (sequence of ANDs) consisting of one or more clauses, each of which is a disjunction (OR) of one or more literals (or negated literals).

OCL expressions that form the body of OCL constraints can be regarded as a kind of logical formula since they can be evaluated to a boolean value. Therefore, we can define a conjunctive normal form for the OCL expressions exactly in the same way as that of the logical formulas. The only difference is the definition of a literal. We define a literal as any subset of the OCL constraint that can be evaluated to a boolean value and that does not include a boolean operator (*or*, *xor*, *and*, *not* and *implies* [OMG03a, pp.6-9]). We say that an OCL constraint is in conjunctive normal form when the OCL expression that appears in its body is in conjunctive normal form. For instance, constraints IC1, IC4 and IC5 are already in CNF while IC2 and IC3 need to be transformed.

Any logical formula can be translated into a conjunctive normal form by applying a well-known set of rules. We use the same rules in the transformation of OCL expressions with the addition of a new rule to deal with the *if-then-else* construct. The rules are the following:

1. Eliminate non-basic boolean operators: the *if-then-else* construct and the *implies* and *xor* operators, using the rules:
   a. A *implies* B == *not* A *or* B
   b. *if* A *then* B *else* C == (A *implies* B) and (*not* A *implies* C) == (*not* A *or* B) *and* (A *or* C)
   c. A *xor* B == (A *or* B) *and not* (A *and* B) == (A *or* B) *and* (*not* A *or not* B)
2. Move *not* inwards until the negations be immediately before literals by repeatedly use the laws:
   a. *not* (*not* A) == A
   b. DeMorgan's laws: *not* (A *or* B) == *not* A *and not* B
      not (A *and* B) == *not* A *or not* B
3. Repeatedly distribute the operator *or* over *and* by means of:
   a. A *or* (B *and* C) == (A *or* B) *and* (A *or* C)

Once these rules have been applied constraints IC2 and IC3 are transformed into:
IC2: not self.category='boss' or not seniority>10 or salary>30000
IC3: (not self.category='boss' or carPark) and (not seniority>5 or carPark)

It is important to note that in order to apply these transformations we do not need to use any Skolemization process to get rid of existential quantifiers, since all variables that appear in OCL expressions are assumed to be universally quantified.

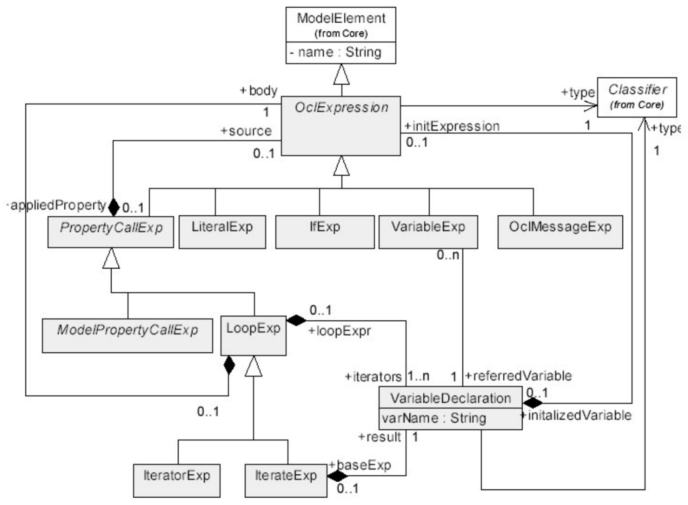


Figure 2 – OCL Metamodel

4. The OCL metamodel

Our method assumes that OCL constraints are represented as instances of the OCL metamodel [OMG03a]. Working at the metamodel level offers us two main advantages. First, we avoid problems due to different syntactic possibilities of the OCL. Second, we can use the taxonomy of the metamodel in order to simplify the evolutionary algorithms.

In this section we briefly describe the OCL metamodel. Its basic structure (Figure 2) consists of the metaclasses *OCLExpression* (abstract superclass of all possible OCL expressions), *VariableExp* (a reference to a variable, as, for example, the variable *self*), *IfExp* (an if-then-else expresion) , *LiteralExp* (constant literals like the integer '1') and *PropertyCallExp* which is an expression that refers to a property and is the superclass of *ModelPropertyCallExp* and *LoopExp*.

ModelPropertyCallExp can be splitted in *AtributteCallExp* (a reference to an attribute), *NavigationCallExp* (a navigation through an association end or an association class) and *OperationCallExp*. This later class is of particular importance in our approach, because its instances are calls to operations defined in any class or metaclass. This includes all the predefined operations of the types defined in the OCL Standard Library [OMG03a, ch.6], such as the add operator ('+') and the 'and' operator. These operations can have a list of arguments if the operation has parameters.

LoopExp represents a loop construct over a collection of elements. OCL offers some predefined iterators to loop over the elements. The most important ones are *forAll* (defines a condition that must be accomplished by all the instances of the collection), *exists* (defines a condition that at least one element must verify), *one* (the condition must be verified by, exactly, a single element), *select* (returns the subset of

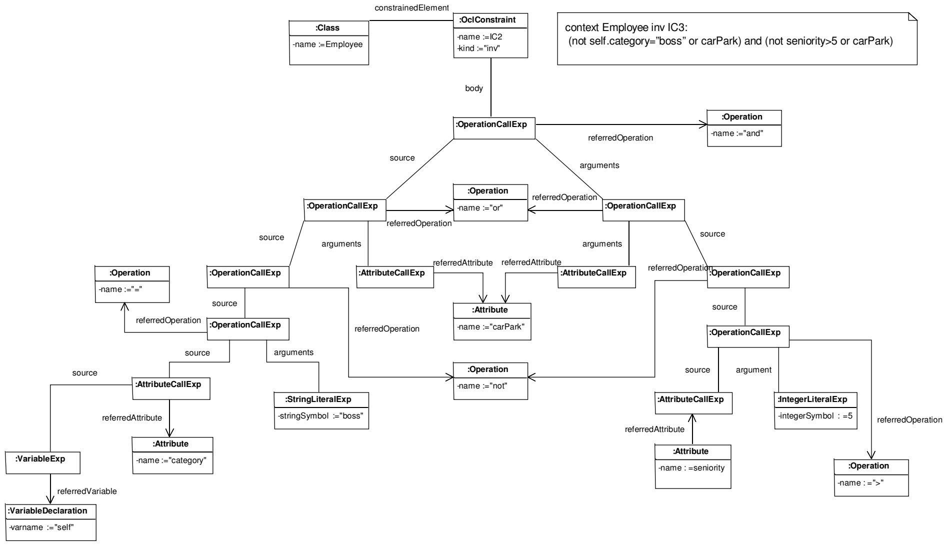


Figure 3 – OCL Constraint expressed as an instance of OCL Metamodel

instances that hold the select condition), *any* (returns one of the instances that holds the condition) and *reject* (returns the subset of instances that do not hold the condition).

As an example, Figure 3 shows the normalized constraint IC3 as an instance of the OCL metamodel.

5. Evolutionary Algorithm

So far, we have transformed all the ICs into a conjunctive normal form (CNF). Therefore, their literals (boolean expressions) can only be connected by *and*, *or* and *not* operators.

Our algorithm begins with marking as undefined every literal that references to an element about to be removed, since then it cannot be evaluated. After that, to compute which parts of the OCL expression need to be deleted we apply the following rules for the *and*, *or* and *not* operators to the set of literals that form the OCL constraint:

1. literal AND undefined → literal
2. literal OR undefined → undefined
3. NOT undefined → undefined

The above rules are deduced from the well-known implication: *A and B* → *A*. To justify rule number one, assume that we have a constraint *c* in the CS with the body "*A and B*" (where *A* and *B* represent two literals, according to our definition). This constraint forces all the instances of the Information Base (IB) to verify both of them. The implication states that no matter what happens with *B* (for example, *B* can become undefined), *A* must still hold in the IB. On the other hand, the rule *A or B* does not imply *A*, so when *B* is undefined we cannot force the IB to verify *A*.

Obviously, if a literal is undefined its negation is undefined as well.

As an example, assume we want to delete the attribute *age* from the CS. In this case, the integrity constraints IC1 (*self.age>16 and self.seniority<40*) and IC4 (*self.age>18 or self.category='scholarship holder'*) will be affected and need to be reviewed. In the former, we keep the second part of the constraint (*self.seniority<40*) since all the instances of the IB that held the constraint (and thus, the value of their seniority attribute was under 40) will continue to do it. In the latter, we mark as undefined the whole constraint since, if we keep the second part of the constraint (*self.category='scholarship holder')*, some instances of the IB would violate the constraint (those instances that are over 18 years old not being scholarship holders).

Finally, after applying the above rules, if the expression consists only of an undefined value, the whole constraint will be discarded. Otherwise, the constraint body will consist of the remaining literals.

Therefore, given the set of elements that the user wants to delete from the CS, our evolutionary algorithm follows these steps in order to simplify the constraints:

1. Identification of the elements to be deleted (those selected by the user plus all the elements affected by them).
2. Labeling as undefined the literals that refer to the elements to delete
3. Evaluation and simplification of the constraint using the above rules
4. Removal of the elements of step number one from the CS.

We would also like to note that these algorithms are efficient. Both have a polynomial order of complexity. In particular, *O(N),* where *N* represents the number of the nodes of the tree that represents a given constraint, the same cost as the preorder and postorder algorithms over which our algorithms are based as we will see bellow.

Next subsections explain deeply the first three steps. The fourth does not present any further complexity.

5.1 Identification

Given a set S_0 containing the model elements the user wants to remove from the CS, we obtain a set S where $S= S_0 \cup X$, and X is the set of model elements that need to be deleted due to the removal of classes and associations of S_0. It is easy to see the removal of attributes or association ends does not add any other element to the set X. The removal of an association class can be seen as the removal of its class followed by the removal of its association. On the other hand, the consequences of the deletion of a generalization link will be addressed in section 5.2.

More concretely, X = *indClass(*S_0*)* $\cup$ *indAssoc(*S_0*)*. The function *indClass* returns, for each class, the set of associations and generalizations where the class participates and all their features (attributes and association ends). *indAssoc* returns, for each association, the set of generalizations where the association participates and all its association ends.

The formalization in OCL of these operations using the UML 2.0 metamodel is:

```
context System::indClass(S0 : Set(Element)) : Set(Element)
body:let Sclass:Set(Class) = S0-> select(s| s.oclIsKindOf(Class))
      in
      Sclass.feature->union(Sclass.feature.association)->
      union(Sclass.generalization->
      union(Generalization.allInstances()->
      select(g| Sclass->includes(g.general))

context System::indAssoc(S0 : Set(Element)) : Set(Element)
body:let Sassoc:Set(Association) = S0-> select(s|
      s.oclIsKindOf(Association))
      in
      Sassoc.memberEnd->union(Sassoc.generalization)->
      union(Generalization.allInstances()->
      select(g| Sassoc->includes(g.general))
```

In order to calculate the elements of the set X we can use the following algorithm:

```
S0=Elements selected by the user
X0=S0
Repeat
      Xi+1=Xi ∪indClass(Xi) ∪ indAssoc(Xi)
Until Xi+1=Xi
```

As an example, assume we want to delete the attribute *seniority* and the association class *Assignment* from the CS of Figure 1. In this case, S_0 will be composed by *seniority* and *Assignment* elements. In the first iteration of the algorithm, the deletion of *Assignment* causes the selection of the attribute *hours* (*Assignment* regarded as a class), and its two association ends (*Assignment* regarded as an association) by the execution of the *indClass* and *indAssoc* operations. Thus, at the end of the iteration, X_0 will contain *Assignment*, *hours*, *seniority,* and the association ends *Assignment-Employee* and *Assignment-Project*. In the second iteration no more elements will be selected, so the algorithm finishes with the following result set:

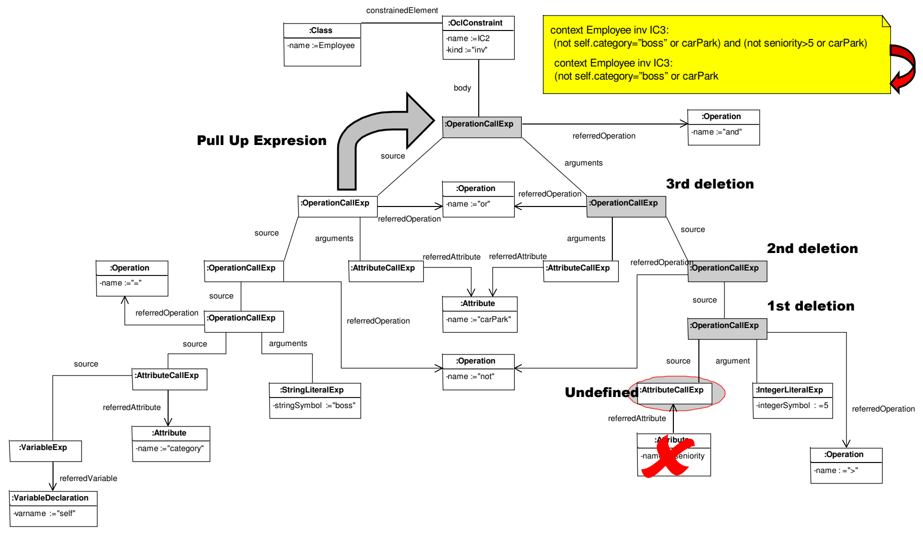
S = {Assignment, *hours*, *Seniority, Assignment-Employee, Assignment-Project}*.

5.2 Marking as undefined

An OCL expression expressed as an instance of the OCL Metamodel can be regarded as a binary tree. Each node represents an atomic subset of the OCL expression (an instance of any metaclass of the OCL metamodel: an operation, an access to an attribute or an association …). The left child is the source of the node (the part of the OCL expression previous to the node). When the node represents an operation the right child is the argument of the operation, if any. When the node represents a loop expression, the right child is the body [1] of the iterator.

To mark as undefined the parts of the constraint that refer to some of the elements of *S* (the set of elements to be deleted), we traverse the tree of the constraint in preorder. For each node we check if its referred element appears in *S*; if it does, we mark the node as undefined. Otherwise, we recursively repeat the process for its children.

[1] the expression that is evaluated for each element in the collection



context Employee inv IC3:
(not self.category="boss" or carPark) and (not seniority>5 or carPark)
context Employee inv IC3:
(not self.category="boss" or carPark
Pull Up Expresion
3rd deletion
2nd deletion
1st deletion
Undefined

Figure 4 – Evolution of the constraint IC3

Besides, we also mark a node as undefined when it refers to a feature (attribute, association end or operation) that now cannot be accessed since its owner is a superclass which cannot be reached because of the deletion of a generalization link.

Figure 4 shows the labeling of constraint IC3 with the set of deleted elements computed in the previous example. In this case, the algorithm marks as undefined the access to the attribute *seniority* (see the grey circle in the Figure).

5.3 Constraint evolution

Once we have marked as undefined all the affected nodes of the OCL expression we are ready to evolve the constraint.

Now, we traverse the tree in postorder. For each node, if the node or any of its children are undefined we pull up the undefined value up to its parents. This applies always except for nodes representing the *boolean* operator *AND*, where we pull up the defined child, if any.

After the execution of this second algorithm, the constraint will consist only of those parts that are consistent with the evolved CS. If the whole constraint is undefined it will be removed from the CS. Figure 4 shows the execution of the algorithm over the constraint IC3 marked in the previous section.

Following our example (*S = {Assignment*, *hours*, *Seniority, Assignment-Employee, Assignment-Project}*), after applying this last step the results will be the deletion of two integrity constraints (IC2 and IC5) and the simplification of another two (IC1 and IC3) while the other (IC4) will remain untouched. The final set of constraints is the following:

IC1: self.age>16
IC3: not self.category='boss' or carPark
IC4: self.age>18 or self.category='scholarship holder'

5.4 Loop expressions

Literals containing loop expressions deserve a special treatment in order to try to keep them in the evolved CS, even when they refer to some of the elements of S (the set of elements to be deleted). We only deal with predefined iterator expressions (section 4).

In an iterator expression, an element of S can appear either in the *source* of the iterator expression or in its *body*. When it appears in the source, the whole iterator expression must be marked as undefined by following the steps explained in the previous sections. On the contrary, if it appears inside the iterator body (but not in the source), depending on the concrete kind of iterator, we will be able to preserve the iterator expression evolving its body.

The candidate iterators for this special treatment are those presenting a boolean expression in their body. We apply the process explained in sections 3 and 5 over that boolean expression. As a result, the original boolean expression may have been left untouched (the iterator did not contain references to elements of S), completely deleted (its whole body expression is undefined) or simplified. When simplified it means that although the body referred to some element of S, by applying the rules defined before, there is still a part of the original boolean expression that could be applied over the new CS. Then, we have two different alternatives depending on the iterator semantics:

- To keep the iterator replacing the original body with the simplified one. This is the case of the iterators *forAll, exists* and *any*
- To mark the whole iterator expression as undefined, which is the case of the rest of the iterators (*select, reject* and *one*). This happens when the iterator with the simplified body may result in a more restrictive OCL constraint, which is not the desired effect.

In the following we justify the possible options for each iterator type.

A ***forAll*** iterator defines a condition that must be accomplished by all the instances of the set over which we apply the iterator. Thus, if every element in the set satisfied the original body it implies that also every element must satisfy a subset of the original body, since it is less restrictive. Therefore, we can keep it on the evolved CS with the defined parts of its original body. As an example, the constraint: *context Department inv: self.employee->forAll(salary>12000 and carPark)* would result in *(self.employee->forAll(carPark)*, after the removal of the attribute *salary*. Note that if all employees assigned to a department earned more than 12000 and had a car park, when we delete seniority it is still true that all of them have a car park.

The ***exists*** iterator defines a condition that must be true for, at least, one element of the set where we apply the iterator. In a similar way as before, if one element satisfied the original boolean expression, it can be followed that at least that element will satisfy a subset of that condition. Again, we can keep expressions containing this iterator since it does not imply a more restrictive constraint over the CS. A similar reasoning can be used for the ***any*** iterator.

The ***one*** iterator defines a condition that must be true for, exactly, a single element in the set. When this kind of iterators refers to an element of S in its body, we must delete the whole iterator, since the resulting expression is more restrictive than the original one. Using the previous example, but replacing *forAll* with *one* we obtain the constraint*: self.employee->one(salary>12000 and carPark)* which states that only one employee per department can earn more than 12000 and have a car park. When we delete *salary* the constraint evolves to *self.employee->one(carPark)*. In the former the IB could contain many employees with a car park as long as they earned less than 12000 to not violate the constraint. Now the constraint is more restrictive, since it only allows a single person with a car park.

A ***select*** iterator selects the subset of the collection where the select is applied that evaluates to true the condition of its body. We can neither keep this iterator when it refers to an element of S, since sometimes it provokes that the constraint where it appears results into a more restrictive one. It depends on the context that surrounds the select expression. In certain cases, as in the constraint: *context Department inv: self.employee->select(salary>12000 and carPark)*->size()>5 when we simplify its body we are relaxing the constraint. If we delete the *salary* attribute we obtain *self.employee->select(carPark)->size()>5* which is easier to fulfill than the original one (it is easier to find five people that satisfies one condition than five that satisfy the same condition plus an additional one).

However in other expressions the simplification of the select condition implies a more restrictive constraint. If we change the previous constraint to *self.employee->select(salary>12000 and carPark)*->size()<5, now, it is not true if we delete the first part of the body (salary>12000) we relax the constraint. With the original constraint the IB could contain more than 5 people with a car park as long as they earned less than 12000, but with the evolved one this is impossible. We can apply a similar reasoning to ***reject*** iterator.

6. Optimizations

We would like to finish sketching some optimizations to our process. These optimizations could reduce the amount of OCL expressions we are forced to remove due to the removal of elements of the conceptual schema.

So far, all the binary OCL operations except for the AND operation, like *union*, *intersection*, *'+'*, *'-'* and so forth, are marked as undefined when any of the two arguments is undefined. However, sometimes we could avoid it if we had enough information about the context where the operation is placed. This is the same problem we have found related to the *select* iterator.

For instance, the expression *self.a + self.b < self.c*, where a, b and c are assumed to be three attributes with a natural value, is marked as undefined if the designer deletes *a*, *b* or *c* from the CS. However, when *a* or *b* are deleted, we could still preserve the expression by transforming it into *b < c* (or *a<c*, depending of the attribute that is deleted). The reason is that if *a* plus *b* returns a value lower than *c*, it implies that *a* or *b* alone must be also lower than *c*.

To implement this optimization mechanism we would need to propagate through the tree some kind of information to decide in each case which are the operations or iterators (for *select* and *reject* iterators) that must be really marked as undefined. Note that it is not enough to examine the immediate nodes around the operation since, in some cases, the information needed is placed several levels up or down in the tree.

Another possible optimization is to find and remove those constraints that have become tautologies after the process due to the removal of some of their parts. For instance, constraints of the form '*(A and B) or (not A)*' become a tautology when the attribute B is deleted (after converting the constraint into a CNF, it results in: *(A or not A) and (B or not A)*, and once the attribute is deleted: it turns into *(A or not A)*).

7. Conclusions and Further Work

In this article we present a method to automatically evolve integrity constraints after the removal of some model elements of a conceptual schema (CS) in order to keep them consistent with the evolved schema. To the best of our knowledge, ours is the first approach to address this issue at the conceptual level. Moreover, we have developed a tool[2] that implements our approach to prove the feasibility of our method.

We focus our explanation in the evolution of OCL expressions appearing in OCL constraints, but we could generalize the method to evolve OCL expressions appearing in other model elements such as derived elements or operation pre and postconditions.

We apply our method to the particular case of conceptual schemas specified in UML with integrity constraints defined using OCL. However, we could adapt our method to other conceptual modeling languages.

We plan to continue our work in (at least) two directions. First, we would like to extend the method with the optimizations sketched in section 6 and , second, we plan to study the evolution of OCL expressions after any kind of schema evolution operations over a conceptual schema.

References

[Ban87] J. Banerjee. Data Model Issues for Object-Oriented Applications. ACM Transactions on Office Information Systems, 5(1):3-26, January 1987.

[Bre96] P. Brèche. Advanced Primitives for Changing Schemas of Object Databases. Conference on Advanced Information Systems Engineering; 476-495. 1996

[Fow99] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[GH91] M. Gogolla, U. Hohenstein, Towards a Semantic View of an Extended Entity-Relationship Model. ACM Transactions on Database Systems, 1991. 16(3): p. 369-416.

[GO02] C. Gomez, A. Olivé, Evolving Partitions in Conceptual Schemas in the UML. CAiSE 2002. p. 467-483.

[Hal01] T.A. Halpin, Information Modeling and Relational Databases, Morgan Kaufmann, 2001.

[ISO82] ISO/TC97/SC5/WG3. Concepts and Terminology for the Conceptual Schema and Information Base, J.J. van Griethuysen (ed.), 1982.

[2] It can be downloaded from http://www.lsi.upc.es/~gmc/Downloads/ICEvolution.html

[Mon93] S. Monk. A Model for Schema Evolution in ObjectOriented Database Systems. PhD thesis, Lancaster University, 1993.

[OMG02a] OMG, OMG Adopted Specification. "UML 2.0 Superstructure Specification", August 2002.

[OMG02b] OMG. Request for proposal: MOF 2.0 Query/Views/Transformations. OMG, 2002

[OMG03a] OMG, OMG Revised Submission, UML 2.0 OCL, 2003.

[OMG03b] OMG. Model Driven Architecture (MDA). OMG, 2003.

[Opd92] W. F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois, 1992.

[Por03] Ivan Porres. Model refactorings as rule-based update transformations. In UML 2003. Springer,2003.

[RCR93] John F. Roddick, Noel G. Craske, Thomas J. Richards. A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models. International Conference on Conceptual Modeling / the Entity Relationship Approach. 137-148. 1993

[Sjø92] Dag Sjø, Quantifying Schema Evolution. Information and Software Technology, Vol. 35, No. 1, pp. 35-44, Jan. 1993.