# Transforming OCL constraints: a context change approach

Jordi Cabot
Estudis d'Informàtica i Multimèdia
Universitat Oberta de Catalunya
Av. Tibidabo, 39-43 E08035 Barcelona
jcabot@uoc.edu

Ernest Teniente
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord, Ed. Omega, 132, E08034 Barcelona
teniente@lsi.upc.edu

## ABSTRACT

Integrity constraints (ICs) play a key role in the definition of conceptual schemas. In the UML, ICs are usually specified as invariants written in the OCL. However, due to the high expressiveness of the OCL, the designer has different syntactic alternatives to express each IC, mainly depending on the type used as a context of the constraint. The method presented in this paper assists the designer during the definition of ICs by means of automatically transforming the initially defined constraints into equivalent alternatives. The method is also useful in the context of the MDA, where the choice of a particular alternative has a direct effect on the efficiency of the automatically generated implementation.

## Categories and Subject Descriptors

D.2.1 [**Software Engineering**] Requirements/Specifications;
D.2.2 [**Software Engineering**]: Design tools and techniques.

## General Terms

Design, Languages, Algorithm

## Keywords

OCL, integrity constraint, context change, transformation

## 1. INTRODUCTION

Integrity constraints (ICs) are a fundamental part in the definition of a conceptual schema (CS) [4]. Many constraints cannot be expressed using only the predefined constructs provided by the conceptual modeling language and require the use of a general-purpose (textual) sublanguage [2]. In the UML, textual ICs are usually written in OCL [10]. An OCL constraint is associated with a type (the context entity type) and must be true for all instances of that type at any time. Predefined constraints, like cardinality constraints, can also be expressed in OCL [3].

Due to the high expressiveness of the OCL, the designer has different syntactic possibilities to define an IC. For instance, given the following CS of Figure 1.1, the IC *MinSalary*: "all employees must earn more than the minimum salary of their department", may be defined as (among many other options):

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SAC'06*, April, 23-27, 2006, Dijon, France.
Copyright 2006 ACM 1-59593-108-2/06/0004…$5.00.

1. context Department inv: self.employee -> forAll (e| e.salary>self.minSalary)
2. context Employee inv: self.salary>self.employer.minSalary
3. context Department inv: self.employee -> select(e| e.salary<=self.minSalary)->size()=0

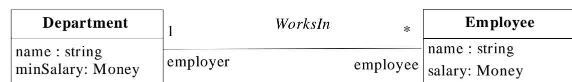



**Figure 1.1 – Example Conceptual Schema**

Therefore, only when designers are aware of all these different alternatives, they can choose the *best* definition for the IC. Obviously, the meaning of *best* varies depending on the specific goal intended by the designer (for instance understandability or efficiency).

There exist two different ways to generate an alternative representation for a given IC: we can either replace the body of the constraint with an equivalent one (as it happens between constraints 1 and 3 of the previous example) or rewrite the constraint by using a different entity type as a context for the constraint (as it happens with 1 and 2 or with 2 and 3).

In this paper we focus in this latter case. In particular, we propose an automatic method that, given an OCL integrity constraint, allows obtaining all alternative definitions of this constraint considering a different context entity type than the original one. The redefinition process is formalized as a path problem over a graph representing the CS. Using the graph we identify which entity types are candidates for acting as new context and obtain all the possible redefinitions for each of them.

The method described in this paper may be useful in several situations. First, at design time, it may assist the designers in the definition of the ICs since then they may be aware of the different alternative representations that exist and choose the one they think the best according to their particular interest.

Secondly, in the context of the MDA [9], our approach can be used in PIM to PIM transformations as well as in PIM to PSM transformations. In PIM-to-PIM transformations, the context change is required, for instance, in the application of refactoring operations at the model level [7].

In PIM to PSM transformations, aimed at generating a final implementation of the system directly derived from its specification, we may produce a more efficient implementation since our method allows obtaining simpler IC definitions than the original ones. For instance, after updating the salary of an employee *e*, it is more efficient to verify that *e* does not violate

the constraint using the second constraint (where we only need to take employee *e* and the department where he/she works into account) than the other two constraints (where we compare the *minSalary* of his/her department with the salary of all its employees).

Finally, our work may also be helpful in schema validation when searching for redundant constraints. Translating the context of the second constraint from *Employee* to *Department* would facilitate noticing that is equivalent to the other two.

In contrast with the huge amount of research devoted to model transformations, redefinition of OCL expressions has received little attention in the past. In particular, [6] discusses the advantages of changing the context but does not define which are the possible new contexts nor provides a method to generate such redefined constraints. On the other hand [1] tries to improve the understandability of OCL constraints but without considering the possibility of redefining them using a different context. [7] mentions context changes but restricts them to associations with multiplicity 1 on both association ends. Hence, as far as we know, ours is the first method able to deal completely with this kind of ICs' automatic transformations.

The structure of the paper is as follows. The next section proposes techniques to change the context of a constraint to a particular entity type. Then, in section 3, we extend them to any entity type of the CS. Finally, we give our conclusions and point out future work in Section 4.

# 2. CHANGING THE CONTEXT OF AN IC

In general, the designer may choose among several entity types to define the context of a particular IC. As we have argued in the introduction, it is more useful sometimes to use a certain context (with its corresponding IC definition) instead of a different one.

An IC defines a condition that must be satisfied in all states of the Information Base. More precisely, when defined in OCL, each IC must evaluate to true for all instances of the context entity type where it is defined. We can therefore guarantee that two constraints $c_1$ and $c_2$ are semantically equivalent when the sets of instances taken into account by both constraints coincide and the expression to be evaluated over them is also the same.

Given two different context entity types $cet_1$ and $cet_2$ and an IC $c_1$ defined over $cet_1$, we show in this section how to automatically obtain an IC $c_2$ defined over $cet_2$ which is semantically equivalent to $c_1$. Our method always guarantees the previous two conditions for semantic equivalence of ICs. It may happen that several semantically equivalent constraints defined over $cet_2$ exist. Then, our method generates all of them.

Changing the context makes only sense when the constraint is defined by using a single instance of the context entity type (i.e. when the expression that defines it contains the *self* variable). Otherwise, i.e. when the constraint is defined using the *allInstances* operation, it is not worthy since its body will always be the same regardless the context used. Apart from that, all the expressivity of the OCL is allowed in the definition of ICs.

In section 2.1, we assume that $cet_2$ is any entity type of the CS related with $cet_1$ through a sequence of associations. Afterwards, in section 2.2, we allow $cet_2$ to belong to the same taxonomy as $cet_1$. Both alternatives are not exclusive since $cet_2$ may belong to the same taxonomy as $cet_1$ and be also related with it.

## 2.1 Changing the Context between Related Entity Types

This section focuses on the transformation of a constraint $c_1$ with context $cet_1$ to a semantically equivalent constraint $c_2$ with context $cet_2$, where $cet_1$ and $cet_2$ are related through one or more sequence of associations that allows navigating between them.

According to one of the requirements to guarantee the semantic equivalence of $c_2$ and $c_1$, the context change from $cet_1$ to $cet_2$ is only possible when there exists at least one sequence of associations $seq_{as}$ relating both types. Moreover, $seq_{as}$ has to verify that $set_{cet1}$ = $set'_{cet1}$; where $set_{cet1}$ is the population of $cet_1$ (the set of instances that $c_1$ restricts) while $set'_{cet1}$ is the set of instances of $cet_1$ obtained when navigating from the instances of $cet_2$ to $cet_1$ through $seq_{as}$. Otherwise, it is not possible to obtain $c_2$ since it would not be possible to verify it over the same set of instances as $c_1$. In particular, the set of instances $set_{cet1} - set'_{cet1}$ would not be restricted by $c_2$.

We can determine whether $set_{cet1}$ = $set'_{cet1}$ by studying the multiplicity of the associations included in $seq_{as}$.

Intuitively, if two entity types *A* and *B* are related through an association *AB* with the multiplicity *0..*:1..** (see Figure 2.1) it means that each instance of *A* is related at least to an instance of *B*. Thus, if we navigate from the instances of *B* to the related instances of *A* we necessarily obtain all *A* instances. Therefore, it is possible to change the context of an IC defined in *A* from *A* to *B*. However, this is not the case from *B* to *A* because the minimum *0* multiplicity does not guarantee all instances of *B* to be related with instances of *A*. For instance, the IC "*context A inv: self.a1>0*" may be translated to: "*context B inv: self.a->forAll(a1>0)*". On the contrary, the IC "*context B inv:self.b1<5*" when translated to *A* (*context A inv: self.b->forAll(b1<5)*) would not prevent that instances of *B* which are not related to *A* have a value in *b1* lower than 5.

Then, we can state that $set_{cet1}$ = $set'_{cet1}$ if the value of all minimum multiplicities of roles used to navigate from $cet_1$ to $cet_2$ through the associations in $seq_{as}$ is at least one. This guarantees that the navigation from $cet_2$ to $cet_1$ reaches all $cet_1$ instances. Following with the previous example, we can change the context of an IC from *A* to *B*, *A* to *C*, *B* to *C* and *C* to *B*, but not from *B* to *A* or *C* to *A*.

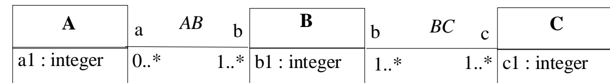



**Figure 2.1 Example of a conceptual schema**

Depending on the specific body of the IC we may be able to relax this multiplicity condition. When the body of $c_1$ permits to deduce that the IC only affects those instances of $cet_1$ related with some instance of $cet_2$ we can use $cet_2$ as context of $c_1$. Roughly, this may happen when each literal appearing in the body of $c_1$ includes a navigation to $cet_2$. As an example consider the first version of the *MinSalary* constraint as defined in the introduction. Even though not all departments have employees assigned, the IC only affects departments with employees (the others always evaluate the constraint to true). Thus, we can use *Employee* as an alternative context for the constraint.

Note that, for a given IC, there may be several different sequences of associations from $cet_1$ to $cet_2$ that verify the previous condition. Each different sequence results in a different alternative representation of $c_1$.

We formalize the problem of changing the context between two related entity types as a path problem over a graph representing the CS. The next subsections explain how to create the graph, to find all different solutions and, for each one, to redefine the IC over the new context.

### *2.1.1 Graph definition*

The basic idea to represent the CS by means of a graph is to consider the entity types in the CS as vertices of the graph and the associations as edges between those vertices. Moreover, for our purposes, we want to obtain a graph that satisfies the following condition: if the graph presents a path from vertex $v_1$ to vertex $v_2$ then ICs defined over $v_1$ can be redefined using $v_2$ as a context entity type.

A path is a sequence of vertices such that each vertex is connected to the next vertex in the sequence (i.e. there exists at least an edge between each pair of consecutive vertices) and where there are no repeated vertices [5].

The graph must be a directed graph (digraph), since being able to change ICs from $cet_1$ to $cet_2$ (i.e. from the vertex representing $cet_1$ to the vertex representing $cet_2$) does not imply that we can also change ICs from $cet_2$ to $cet_1$, the context change is transitive but not symmetric. For instance, consider the graph of Figure 2.2, which represents the CS of Figure 2.1. The graph shows that ICs defined over *A* can also be expressed over *B* or over *C*. ICs defined over *B* can be expressed over *C* but not over *A*. ICs defined over *C* can be expressed over *B*.

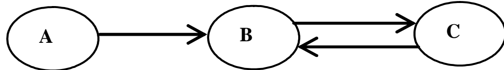



**Figure 2.2 – Example graph**

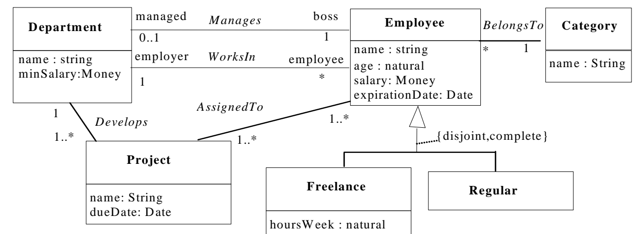
Sometimes the graph may also be a multigraph since it may contain two or more edges with the same direction between a pair of vertices. This happens when the two corresponding entity types are related through more than one association.

According to those ideas, we build the graph *G* by means of the following rules:

- All entity types, including reified ones (i.e. association classes), are vertices of *G*.
- For each binary association between two entity types *A* and *B*, the edge $A \rightarrow B$ is included in *G* if the minimum multiplicity from *A* to *B* is at least one. The edge $B \rightarrow A$ is included when the minimum multiplicity from *B* to *A* is at least one.
- Given a n-ary association *As* among a set of entity types $E_1, \ldots E_n$ we add and edge from $E_i \rightarrow E_j$ if we can deduce, from the multiplicities of roles in *As*, that the minimum multiplicity from $E_i$ to $E_j$ is at least one. In class diagrams, these binary multiplicities remain unspecified. [8] demonstrates that when the multiplicity of the role next to $E_j$ is at least one, all the multiplicities from any $E_i$ to $E_j$ are at least one, and thus, the edge $E_i \rightarrow E_j$ is included in the graph.
- For each vertex representing an association class *AC*, we add the edges $AC \rightarrow E_1$, $AC \rightarrow E_2, \ldots, AC \rightarrow E_n$ where $E_1..E_n$ are the participants of the association. We add these edges since an instance of an association class must be always related to an instance of each participant type. We add the inverse edges depending on the multiplicities of the association. If *AC* is the reification of a binary association, we add $E_1 \rightarrow AC$ if $E_1 \rightarrow E_2$ exists (and conversely with $E_2$). Similarly, if the association is an n-ary association, we add $E_j \rightarrow AC$ if exists an $E_i$ that verifies $E_j \rightarrow E_i$
- Since subtypes inherit all the associations of their supertypes, for each edge $A \rightarrow B$ we add an edge $A_i \rightarrow B$ for each $A_i$ subtype of *A*. Note that for edges of kind $B \rightarrow A$ we do not add $B \rightarrow A_i$ since the fact that each instance of *B* is related with an instance of *A* does not imply that it is also related with an instance of the subtype $A_i$.

The graph obtained with these rules is valid for any IC. Then, if there is a path from $cet_1$ to $cet_2$ all the ICs defined over $cet_1$ can be expressed using $cet_2$. Moreover, as we have seen before, a context change from $cet_1$ to $cet_2$ may also be possible (even though the multiplicity condition is not satisfied) when the body of the IC only affects those instances of $cet_1$ related with instances of $cet_2$. To deal with these special cases, we also add to *G* some edges that are specific for concrete ICs. These edges are labeled with the name of the IC and paths including them are only valid for changing the context of that particular IC.

As a running example, consider the CS of Figure 2.3. It specifies information about the departments of a company, their projects and their employees and it includes five textual ICs. The first it is the previous *MinSalary* constraint. The others ensure that employees are older than 16 years old (*ValidAge*), that departments are not managed by a freelance employee (*NotBossFreelance*), that all projects have at least two project managers (*AtLeastTwoProjectManagers*) and that the number of hours per week that freelances work lies between 5 and 30 (*ValidNHours*).




**context** Department **inv** MinSalary: self.employee->forAll(e|e.salary>self.minSalary)
**context** Employee **inv** ValidAge: self.age>16
**context** Department **inv** NotBossFreelance: not self.boss.oclIsTypeOf(Freelance)
**context** Department **inv** AtLeastTwoProjectManagers:
self.project->forAll(p| p.employee->select(e|e.category.name='PM')->size()>=2)
**context** Freelance **inv** ValidNHours: self.hoursWeek>=5 and self.hoursWeek<=30

**Figure 2.3 - Conceptual schema used as a running example**

Figure 2.4 shows the graph corresponding to the previous CS. We can draw from it that ICs over *Project* may be redefined over *Employee, Department* and *Category*; ICs over *Employee* can be redefined over *Project*, *Department* and *Category*; ICs over *Category* can not be changed to any other context; etc.

The edge *WorksIn* from *Department* to *Employee* is labeled with the name of the IC *MinSalary* because this is the unique IC that can be changed from *Department* to *Employee* using the association *WorksIn*.

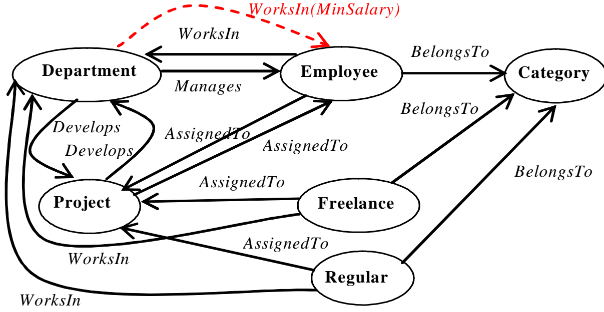



**Figure 2.4 – Graph of the conceptual schema**

### *2.1.2 Computing All Possible Alternative Paths*

Each different path from $cet_1$ to $cet_2$ represents a different way to express the original constraint $c_1$ in terms of the new context $cet_2$. To compute all alternative paths from $cet_1$ to $cet_2$ we have slightly adapted a graph-searching procedure such as the depth-first search one [5], using $cet_1$ as initial vertex and terminating the search only after all different paths reaching $cet_2$ have been generated. Only paths with no repeated vertices are generated.

The next section uses these paths to redefine $c_1$ in terms of the new context $cet_2$.

For instance, the possible paths from *Department* to *Employee* are the following: *Department-Manages-Employee* and *Department-Develops-Project-AssignedTo-Employee*. When looking for alternatives for the IC *MinSalary* we can use the edge *WorksIn* from *Department* to *Employee*, and thus, there is an additional path: *Department-WorksIn-Employee*.

### *2.1.3 Redefining the IC over the new Context*

Given a constraint $c_1$ with a body $X$ defined over $cet_1$ and a path $p=\{e_1,..,e_n\}$ (where $e_1..e_n$ are a set of edges linking the vertices $\{cet_1,v_2,..v_n,cet_2\}$), the semantically equivalent constraint $c_2$ defined over $cet_2$ has the form:

*context* $cet_2$ *inv* $c_2$: $self.r_1.r_2. \ldots r_n$*->notEmpty() implies* $self.r_1.r_2. \ldots r_n$ *->forAll(v|X)*

where all occurrences of *self* in $X$ have been replaced with $v$ and $r_1..r_n$ are the roles to navigate from $cet_2$ to $cet_1$ using the associations appearing in $p$. Therefore, $r_1$ represents the navigation from $cet_2$ to $v_n$ using the association $e_n$, $r_2$ the navigation from $v_n$ to $v_{n-1}$ using $e_{n-1}$, and, finally, $r_n$ represents the navigation to $cet_1$ from $v_2$.

$c_1$ and $c_2$ are equivalent since both apply the same condition to the instances of $cet_1$ (the condition $X$) and apply it over the same set of instances (guaranteed by the graph definition process).

As an example, the IC *MinSalary* (*context Department inv: self.employee -> forAll (e| e.salary>self.minSalary)*) may be redefined over *Employee* because of the path $p$={*WorksIn*}. The redefined IC *MinSalary'* is:

> ***context*** *Employee* ***inv*** *MinSalary': self.employer->notEmpty() implies self.employer->forAll(d|d.employee->forAll(e| e.salary>d.minSalary))*

Since OCL does not define the navigation through n-ary associations, when $e_i$ represents an n-ary association between $v_{i+1}$ and $v_i$, we must navigate first from $v_{i+1}$ to the corresponding association class and then from the association class to $v_i$.

Moreover, according to the graph definition process, if an edge $e_i$ links vertices $v_{i+1}$ and $v_i$, the corresponding association must exist between the entity types $E_{i+1}$ (represented by $v_{i+1}$) and $E_i$ (represented by $v_i$) or between $E_{i+1}$ and a supertype of $E_i$. In the latter case when navigating from $E_{i+1}$ to $E_i$ we need to add "*select(oclIsTypeOf(subtype(*$E_i$*))*" to the corresponding $r_i$. For instance, the constraint *ValidNHours* can be translated from *Freelance* to *Category*. The body of the resulting constraint begins with "*self.employee->select(e|e.oclIsTypeOf(Freelance))*" since we are only interested in those employees which are freelances.

We provide some rules to simplify the body of the new constraint $c_2$ (the variable $X$ stands for an arbitrary OCL expression of type boolean).

1. $self.r_1.r_2.\ldots.r_n$*->notEmpty()* → *true*, if the multiplicity of $self.r_1.r_2. \ldots r_n$ is at least one, i.e. if all the minimum multiplicities of $r_1.r_2. \ldots r_n$ are at least one. In such a case, we know for sure that the navigation will return a non-empty set, and thus, the evaluation of the *notEmpty* operation will return a *true* value.
2. $self.r_1.r_2. \ldots r_n$*->forAll(X)* → $X$ (where all the occurrences of $v$ in $X$ are replaced with $self.r_1.r_2. \ldots r_n$), if the multiplicity of $self.r_1.r_2. \ldots r_n$ is at most one, i.e. if all the maximum multiplicities of $r_1.r_2. \ldots r_n$ are at most one. Then, the *forAll* iterator is not necessary.
3. $self.r_1.r_2.\ldots.r_i.r_j\ldots r_n$*->forAll(X)* → $self.r_1.r_2. \ldots.r_{i-1}.r_{j+1}\ldots r_n$*->forAll(X)*, when $r_i$ and $r_j$ are the two roles of the same binary association and the minimum multiplicity at $r_i$ is at least one (see Figure 2.5). When the maximum multiplicity of $r_j$ is also one, the set of objects at $r_j$ are the same than those at $r_{i-1}$, and thus, the navigations $r_i$ and $r_j$ are redundant. Otherwise, we may have more objects at $r_j$, and, in general, this entails that these additional objects are not verified in the right hand expression. However, if the minimum multiplicity of all opposite roles from $r_1$ to $r_{i-1}$ is at least one, those objects must be related with a (different) instance of the context entity type, and thus, will be checked when evaluating that instance.
4. $self.r_1.r_2\ldots r_i$*->forAll(v|* $v.r_j\ldots r_n$*->forAll(*$v_2$*|* $v_2$*.X))* → $self.r_1.r_2 \ldots.r_i.r_j \ldots r_n$*->forAll(*$v_2$*|*$v_2$*.X))*, when $X$ does not contains any reference to $v$. The two expressions are equivalent since in both we apply the condition $X$ to the objects obtained at $r_n$.

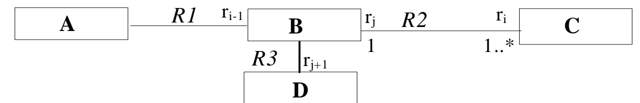



**Figure 2.5 – Abstract example schema for rule 3**

With these equivalences, we can simplify the previously obtained *MinSalary'* constraint. We first apply rule 1 to remove *self.employer->notEmpty()*. Then, rule 2 to remove the first *forAll* (from *self.employer->forAll(d|d.employee->forAll(e|…)* to *self.employer.employee->forAll(e|…)*. Afterwards, we apply rule 3 to remove the redundant navigation *employer.employee* (obtaining *self->forAll(e|…)*). Finally with rule 2 again, we obtain the new IC definition, which is clearly much simpler than the initially obtained one:

***context*** *Employee* ***inv****: self.salary>self.employer.minSalary*

## 2.2 Changing the Context within a Taxonomy

Given a constraint $c_1$ defined over the context entity type $cet_1$ we are interested in redefining $c_1$ using $cet_2$ as a context entity type, where $cet_1$ and $cet_2$ belong to the same taxonomy. This implies that either $cet_1$ is a subtype of $cet_2$, a supertype or both have a common supertype (sibling types).

When $cet_1$ is subtype of $cet_2$, the equivalent constraint $c_2$ defined over $cet_2$ has as a body: *self.oclIsTypeOf(*$cet_1$*) implies X*, where *X* is the body of $c_1$. This way we ensure that $c_2$ is only applied over those instances that are instance of $cet_1$.

As an example, consider the IC *ValidNHours*. If we want to move the IC from *Freelance* to *Employee*, the new IC would be:

***context*** *Employee* ***inv*** *ValidNHours: self.oclIsTypeOf(Freelance) implies self.oclAsType(Freelance).hoursWeek>5 and self.oclAsType(Freelance).hoursWeek<30*

If $cet_1$ is a supertype of $cet_2$, the new constraint $c_2$ is defined in $cet_2$ with exactly the same body as $c_1$. However, $c_2$ cannot replace $c_1$ since in general $cet_1$ may contain instances not appearing in $cet_2$. Thus, both ICs are not semantically equivalent[1]. If the set of generalization relationships between $cet_1$ and its direct subtypes is covering [11] (also called *complete*) $c_1$ can be replaced as long as we add a new IC to each direct subtype of $cet_1$ with the same body as $c_1$. For instance, if we try to change the IC *ValidAge* from *Employee* to *Freelance* we need to add also *ValidAge* to *Regular* to ensure that all employees have a valid age.

When $cet_1$ and $cet_2$ share a common supertype the new constraint $c_2$ can never replace $c_1$ since not all instances of $cet_1$ need to be instances of $cet_2$. As in the subtype case, the body of $c_2$ would be *self.oclIsTypeOf(*$cet_1$*) implies X*.

Before finalizing the context change to a new context entity type *cet* we can apply two simplification rules specially useful for this kind of transformations:

- *self.oclIsTypeOf(cet)* → *true*
- *self.oclAsType(cet).X* → *self.X*

# 3. COMPUTING ALL ALTERNATIVE CONTEXT CHANGES FOR AN IC

Once we know how to change an IC from a context entity type to another (given) context, we generalize the methods of the previous section to obtain all alternative representations of a certain IC assuming that the new context may be any entity type of the CS.

[1] Except for those constraints where the body is already defined to apply only over the instances of the subtype $cet_2$.

Now, to compute all possible representations of a constraint $c_1$, defined over $cet_1$, when redefined over an alternative entity type (or reified type) *E* appearing in the graph we need to consider all possible paths between $cet_1$ and every different *E*.

As an example, we automatically obtain nine different alternative representations of the constraint *MinSalary* defined in Figure 2.3 (one for every path between *Department* and the related types in the graph: *Employee*, *Project* and *Category*). Table 2.1 shows the list of valid paths. The second path would result in the redefinition of *MinSalary* constraint provided in section 2.1.

**Table 2.1 – Valid paths for *MinSalary***

| Final context | Path |
|---|---|
| Employee | Department – *Manages* – Employee |
| | Department – *WorksIn* – Employee |
| | Department – *Develops* – Project – *AssignedTo* – Employee |
| Category | Department – *Manages* – Employee – *BelongsTo* – Category |
| | Department – *WorksIn* – Employee – *BelongsTo* – Category |
| | Department – *Develops* – Project – *AssignedTo* - Employee – *BelongsTo* – Category |
| Project | Department – *Develops* – Project |
| | Department – *Manages* – Employee – *AssignedTo* – Project |
| | Department – *WorksIn* – Employee – *AssignedTo* – Project |

Applying the same process to the other ICs of the example we obtain some interesting redefinitions. For instance, constraint *AtLeastTwoProjectManagers* could be redefined over *Project* (after the simplifications, the final body would be: *self.employee ->select(e|e.category.name='PM')->size()>=2*) to reduce the number of navigations. *NotBossFreelance* could be translated from *Department* to *Employee* and then from *Employee* to the *Freelance* subtype (since the body of the IC only affects freelance instances) to finally obtain the body *self.managed->notEmpty() implies false*, and thus, avoiding the *oclIsTypeOf* operation.

# 4. CONCLUSIONS AND FURTHER WORK

We have presented an automatic method that given an OCL constraint transforms it using as a context a different entity type of the conceptual schema. As far as we know, ours is the first method able to generate all alternative representations of a given integrity constraint in terms of possible new context entity types.

Our method is formalized as a path problem over a graph representing the conceptual schema. The graph is created in such a way that every path between two vertices corresponds to a different alternative to represent the set of constraints defined over the first vertex (i.e. over the entity type represented by the vertex) by using the second one as a context. Using this graph we can compute the different alternative representations

Thanks to this generation designers are aware of the different possibilities when defining a constraint and select the one they think the best according to their particular interest (for instance understandability or efficiency).

Our method could be extended to provide this selection automatically. Given the designer's goal, the method could use a set of metrics and complexity models for OCL constraints to decide the best alternative. This is the direction in which we plan to continue our work.

## 5. ACKNOWLEDGMENTS

We would like to thank J. Conesa, D. Costal, X. de Palol, C. Gómez, A. Olivé, A.Queralt, R. Raventós, M. R. Sancho and R. Solana for their many useful comments in the preparation of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIN2005-06053.

## 6. REFERENCES

1. Correa, A. and Werner, C., Applying Refactoring Techniques to UML/OCL Models. in *7th Int. Conf. on the Unified Modeling Language (UML'04)*, (2004), Springer, 173-187.
2. Embley, D.W., Barry, D.K. and Woodfield, S. *Object-Oriented Systems Analysis. A Model-Driven Approach*. Yourdon, 1992.
3. Gogolla, M. and Richters, M. Expressing UML Class Diagrams Properties with OCL. in Clark, A. and Warmer, J. eds. *Object Modeling with the OCL*, Springer-Verlag, 2002, 85-114.
4. ISO/TC97/SC5/WG3. Concepts and Terminology for the Conceptual Schema and Information Base. Griethuysen, J.J.v. ed., ISO, 1982.
5. Jungnickel, D. *Graphs, networks and algorithms*. Springer-Verlag, 1999.
6. Ledru, Y., Dupuy-Chessa, S. and Fadil, H. Towards computer-aided design of OCL constraints. in *CAiSE'04 Workshops Proceedings, Vol. 1*, Riga Technical University, 2004, 329-338.
7. Markovic, S. and Baar, T., Refactoring OCL Annotated UML Class Diagrams. in *8th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'05)*, (2005), LNCS, 280-294.
8. McAllister, A. Complete rules for n-ary relationship cardinality constraints. *Data Knowl. Eng.*, *27* (3). 255-288.
9. OMG. MDA Guide Version 1.0.1. OMG ed., 2003.
10. OMG. UML 2.0 OCL Specification. OMG ed., 2003.
11. OMG. UML 2.0 Superstructure Specification. OMG ed., 2003.