Generating Alternative Representations for OCL Integrity Constraints

Jordi Cabot[1,2] and Ernest Teniente[2]

[1]Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
jcabot@uoc.edu
[2] Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
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 language. However, due to the high expressiveness of the OCL, the designer has different syntactic alternatives to express each IC. In the context of the MDA, the choice of a particular definition has a direct effect on the efficiency of the automatically generated implementation. The method presented in this paper assists the designer during the definition of ICs by means of generating equivalent alternatives for the initially defined constraints. Our method can also be applied to help in the detection of equivalent (redundant) constraints and as a tool to facilitate the learning of the OCL.

1. Introduction

Integrity constraints are a fundamental part in the definition of conceptual schemas (CS)[5]. 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 [3]. In the UML this is usually done by means of invariants written in the OCL language [9]. Predefined constraints can also be expressed in OCL [4].

Due to the high expressiveness of the OCL, the designer has different syntactic possibilities to define an integrity constraint. For instance, given the following CS:

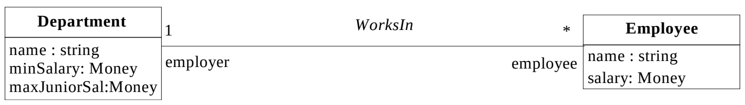


Figure 1.1 – Example Conceptual Schema

the constraint “all employees must earn more than the minimum salary of its department” may be defined as (among many other options):

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

Obviously, the designer may not be aware of all different alternatives, and thus, he may just choose the one he cares about at the moment of defining the constraint. Many times, this implies that the designer does not define the constraint in the *best* way. As we will discuss later, the meaning of *best* varies depending on the specific goal intended by the designer (for instance understandability or efficiency).

In this paper we provide the designer with an automatic method that obtains a set of alternative constraint representations which are semantically equivalent to a given integrity constraint. Moreover, we define how to obtain the best one according to a designer-defined complexity model. These are the two main contributions of the work reported here.

There exist two different ways to generate an alternative representation for a given constraint: 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 context (as it happens with 1 and 2).

Our method addresses the first case by defining a set of equivalence rules between the different elements and constructs that can appear in the OCL expression defining the body of the constraint. Afterwards, the redefinition of the constraint using an alternative context 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.

We can generate the alternative representations for a given constraint by means of generating all redefinitions of the constraint using a different context entity type and then, for each redefinition, to generate the set of equivalent bodies (or vice versa). Our method generates all alternative redefinitions of the constraint when using a different entity type as a context but not all possible equivalent bodies for each of them because of the huge number of equivalences among the OCL constructs.

The method described in this paper is useful in several situations. First, at design time, it can assist the designer in the definition of the integrity constraints. Secondly, in the context of the MDA [11], where the final implementation of the system is derived from the specification, the simplicity of the constraints has a direct effect on the efficiency of the implementation. Therefore, our method can be used to increase the efficiency of the final system by generating equivalent but more efficient constraints than the original ones written by the designer. Additionally, it may be useful in schema validation when comparing a set of constraints in search of redundancies among them. For instance, it could help in the detection that the three previous constraints are equivalent, thus concluding that two of them are redundant. Finally, it may also be used as a tool to facilitate the learning of the OCL language.

To the best of our knowledge, ours is the first method to deal with the automatic generation of alternative syntactic definitions for an integrity constraint. [7] 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. [2] tries to improve the understandability of OCL constraints but without considering the possibility of redefining the constraint using a different context.

The structure of the paper is as follows. Next section defines several equivalences between OCL expressions. Then, we propose techniques to change the context of a constraint to a particular entity type (section 3) and we extend them to any entity type of the CS (section 4). Section 5 discusses how to select the best representation among the alternatives generated in sections 2 and 4. Finally, we give our conclusions and point out future work in Section 6.

2. Equivalences between OCL expressions

As we said, one of the possible ways to generate an alternative representation for a certain constraint is to replace its body with an equivalent one. We achieve it by means of the list of equivalences between OCL expressions presented in this section. Hence, each expression on the one side of the equivalence may be replaced with the expression on the other side. The list is not exhaustive but it contains those equivalences we believe to be the most usual and/or useful ones.

Section 2.1 presents a list of basic equivalences. Section 2.2 defines equivalences to be able to remove the *allInstances* operation. Finally, section 2.3 provides equivalences to transform an OCL expression to conjunctive normal form (CNF). Equivalences in sections 2.1 and 2.3 may be applied to any OCL expression, including derivation rules and operation pre and postconditions. Section 2.2 is specific for integrity constraints.

Assume we define an integrity constraint in the CS of Figure 1.1 to prevent junior employees (those with an age<25) to earn more than the *maxJuniorSal* defined for their department. It could be defined by means of the following OCL expression:

context *Department* ***inv*** *MaxSalary: Department.allInstances->forAll(d| not d.employee->select(e|e.age<25)->exists(e|e.salary>d.maxJuniorSal))*

Applying the set of equivalences we propose, we could transform the expression defining the previous constraint into the equivalent one:

context *Department* ***inv*** *MaxSalary': self.employee->forAll(e| not e.age<25 or not e.salary>self.maxJuniorSal)).*

Note that the meaning of both constraints is exactly the same. However, the second expression is clearly much simpler. We have obtained it by applying first equivalences 31 and 25 in section 2.1; then removing the *allInstances* operation (see section 2.2) and finally transforming the resulting expression to CNF (section 2.3). We will see in section 3 that *MaxSalary'* may even be defined by means of a simpler OCL expression if using another entity type as a context entity type.

2.1 Basic equivalences

We group the equivalences by the type of expressions they affect. The capital letters *X, Y* and *Z* represent arbitrary OCL expressions of the appropriate type. The letter *o* represents an arbitrary object.

Note that, the list is specified in a manner that when applied in the left-right direction, the equivalences reduce the number of different operations that can appear in an OCL expression (for instance, equivalence 14 allows to avoid using the *includes* operation) or generate shorter expressions (see equivalences 28-31).

Table 2.1 List of equivalences

Boolean types	1. <> ↔ not =	2. X = true ↔ X
	3. X = false ↔ not X	4. not false ↔ true
	5. not true ↔ false	6. X and false ↔ false
	7. X and true ↔ X	8. X or false ↔ X
	9. X or true ↔ true	10.not X>Y ↔ X<=Y
	11.not X>=Y ↔ X<Y	12.not X<Y ↔ X>=Y
	13.not X<=Y ↔ X>Y	
Collection types	14. X->includes(o)↔ X->count(o)>0	15. X->excludes(o) ↔ X->count(o)=0
	16. X->includesAll(Y) ↔ Y->forAll(y1| X->count(y1)>0)	17. X->excludesAll(Y) ↔ Y->forAll(y1| X->count (y1)=0)
	18. X-> isEmpty() ↔ X->size()=0	19. X->notEmpty() ↔ X->size()>0
	20. X.attr ↔ X->collect(attr)	21. X->including(o) ↔ X->union(Set{o})
	22. X->excluding(o) ↔ X->- (Set{o})	23. X->union(Y)->forAll(Z) ↔ X->forAll(Z) and Y->forAll(Z)
Predefined iterators	24. X->exists(Y) ↔ X->select(Y)->size()>0	25. not X->exists(Y) ↔ X->forAll(not Y)
	26. X->reject(Y) ↔ X->select(not Y)	27. X->one(Y) ↔ X->select(Y)->size()=1
	28. X->select(Y)->size()=0 ↔ X->forAll(not Y)	29. X->select(Y)->size()=X->size() ↔ X->forAll(Y)
	30. X->select(Y)->forAll(Z) ↔ X->forAll(Y implies Z)	31. X->select(Y)->exists(Z) ↔ X->exists(Y and Z)

2.2 Removing the allInstances operation

AllInstances is a predefined feature on classes that gives as a result the set of all instances of the type that exist at the specific time when the expression is evaluated [9]. For instance, a constraint like "all employees must be older than 16" can be expressed as:

***context** Employee **inv** ValidAge: Employee.allInstances->forAll (e| e.age>16)*

This constraint could also be specified using the variable *self* that represents any instance of the context entity type:

***context** Employee **inv** ValidAge': self.age>16*

Since constraints are assumed to be true for all instances of the context entity type (i.e. for all possible values of the *self* variable), both constraints are equivalent. Moreover, *ValidAge'* is clearly simpler than *ValidAge*.

We propose two equivalences to include/remove the *allInstances* operation. They are applicable when the type over which *allInstances* is applied coincides with the context entity type (*cet*) of the constraint. They may not be applied if the constraint already contains any explicit or implicit reference to the *self* variable.

– cet.allInstances->forAll(v|Y) ↔ Y, once replaced all occurrences of v (the iterator variable) in *Y* with *self*. As an example, see the previous *ValidAge'* constraint.
– cet.allInstances->forAll(v_1,v_2,…v_n| Y) ↔ cet.allInstances->forAll(v_2..v_n|Y) once replaced all the occurrences of v_1 in *Y* with *self*.

2.3 Transforming to conjunctive normal form

A logical formula is in conjunctive normal form (CNF) if it is a conjunction (sequence of ANDs) consisting of one or more clauses, each of which is a disjunction (sequence of ORs) of one or more literals (or negated literals). Likewise, we can define a CNF for OCL expressions that evaluate to a boolean value.

OCL expressions can be translated into CNF with the following rules:

1. To eliminate the *if-then-else*, the *implies* and *xor* constructs using:
   a. X *implies* Y ↔ *not* X *or* Y
   b. *if* X *then* Y *else* Z ↔ (*not* X *or* Y) *and* (X *or* Z)
   c. X *xor* Y ↔ (X *or* Y) *and* (*not* X *or not* Y)
2. To move *not* inwards by using:
   a. *not* (*not* X) ↔ X
   b. DeMorgan's laws: *not* (X *or* Y) ↔ *not* X *and not* Y
      not (X *and* Y) ↔ *not* X *or not* Y
3. Repeteadly distributive *or* over *and* by means of:
   a. X *or* (Y *and* Z) ↔ (X *or* Y) *and* (X *or* Z)

3. Changing the context of a constraint

Each OCL constraint is defined in the context of a specific entity type, the context entity type. In general, the designer may choose the context used to define a particular integrity constraint among several entity types. As we have shown in the introduction, it is sometimes useful to use a certain context instead of another one. However, it is not possible to guarantee that the best context will be the one selected by the designer.

We propose in this section a method that, given a constraint c_1 defined over a context entity type cet_1, automatically obtains a semantically equivalent constraint c_2 defined over a different context entity type cet_2, provided by the designer Two constraints are semantically equivalent if they prevent the information base to be in the same set of inconsistent states. Intuitively, we may ensure that c_2 and c_1 are semantically equivalent when the sets of instances verified by both constraints coincide and the condition to be checked is also the same. Our method always guarantees these two conditions.

The change of context makes only sense when the constraint is defined by using a single instance of the context entity type (i.e. when using the *self* variable). Hence, it does not make sense to rewrite an integrity constraint defined with the *allInstances* operation since its body will always be the same. As we have seen, some of the equivalences proposed in the previous section allow reducing the number of times that this happens.

We first address the case where cet_2 is any entity type of the CS related with cet_1 through a sequence of relationship types. After, we deal with the case where cet_2 belongs 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. It may happen that several semantically equivalent constraints defined over cet_2 exist. Then, our method generates all of them.

We generalize those ideas in the next section to be able to rewrite a constraint in terms of all its possible context entity types.

3.1 Changing the context between related entity types

This section focuses on the translation of a constraint c_1 with context cet_1 to a semantically equivalent constraint c_2 with context cet_2. A necessary condition is that cet_1 and cet_2 are related, i.e. that there is a sequence of relationship types that allows navigating between cet_1 and cet_2. 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 (one of the requirements to consider c_1 and c_2 semantically equivalent).

Moreover, we must ensure that there is some sequence of relationship types connecting cet_1 with cet_2 that verifies that $\text{set}_{cet1} = \text{set'}_{cet1}$; where set_{cet1} is the set of instances of cet_1 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 that sequence.

Given a sequence of relationship types seq_{RT} connecting cet_1 with cet_2 we can determine whether $\text{set}_{cet1} = \text{set'}_{cet1}$ by studying the multiplicity of the relationship types included in seq_{RT}.

Intuitively, if two entity types *A* and *B* are related through a relationship type *AB* with the multiplicity *0..*:1..** (see Figure 3.1) 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 a constraint 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 constraint "*context A inv: self.a1>0*" may be translated to: "*context B inv: self.a->forAll(a1>0)*". On the contrary, the constraint "*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 $\text{set}_{cet1} = \text{set'}_{cet1}$ if the value of all minimum multiplicities of roles used to navigate from cet_1 to cet_2 through the relationship types in seq_{RT} 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 a constraint from *A* to *B*, *A* to *C*, *B* to *C* and *C* to *B*, but not from B to A or C to A.

Depending on the specific body of the constraint we may be able to relax this multiplicity condition. When the body of c_1 permits to deduce that the constraint 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 happens when each literal appearing in the body of c_1 includes a navigation to cet_2. As an example consider the *MaxSalary* constraint of section 2. Even though not all departments have employees assigned, the constraint 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 constraint, there may be several different sequences of relationship types 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 entity types as a path problem over a graph representing the CS. Next subsections explain how to create the graph, to find all different solutions and, for each one, to redefine the constraint over the new context.

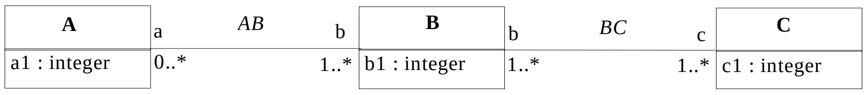


Figure 3.1 - Example of a conceptual schema

3.1.1 Graph definition

The basic idea to represent the CS by means of a graph is to consider the entity types as vertices of the graph and the relationship types as edges between those vertices. Moreover, for our purposes, we want to obtain a graph *G* that satisfies the following condition: if the graph presents a path from vertex v_1 to vertex v_2 then constraints 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 [6].

The graph must be a directed graph (digraph), since being able to change constraints 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 constraints from cet_2 to cet_1, the context change is not symmetric. For instance, consider the graph of Figure 3.2, which represents the CS of Figure 3.1. The graph shows that constraint defined over *A* can also be expressed over *B* or over *C*. Constraints defined over *B* can be expressed over *C* but not over *A*. Constraints defined over *C* can be expressed over *B*.

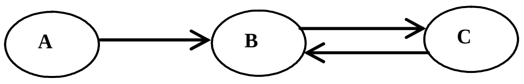


Figure 3.2 – Example graph

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 relationship type.

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 relationship type 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 relationship type *R* 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 *R*, 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 $Ei \rightarrow Ej$ is included in the graph.

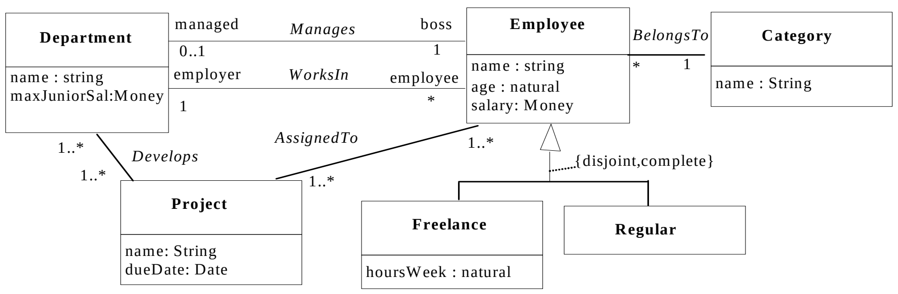
- For each vertex representing a reified entity type *RET*, we add the edges $RET \rightarrow E_1$, $RET \rightarrow E_2, \ldots, RET \rightarrow E_n$ where $E_1..E_n$ are the participants of the relationship type. We add these edges since an instance of the reified type must be always related to an instance of each participant type. We add the inverse edges depending on the multiplicities of the relationship type. If *RET* is the reification of a binary relationship type *R*, we add $E_1 \rightarrow RET$ if $E_1 \rightarrow E_2$ exist (and conversely with E_2). Similarly, If *R* is n-ary, we add $E_j \rightarrow RET$ if exists an E_i that verifies $E_j \rightarrow E_i$.
- Since subtypes inherit all the relationship types 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 A_i.

The graph obtained with these rules is valid for any constraint. Then, if there is a path from cet_1 to cet_2 all the constraints 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 constraint 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 constraints. These edges are labeled with the name of the constraint and paths including them are only valid for changing the context of that particular constraint.

As running example, consider the CS of Figure 3.3. It specifies information about the departments of a company, their projects and their employees and it includes six textual integrity constraints. The first two are the previous *MaxSalary* and *ValidAge* constraints (see section 2). The other ensure that departments with more than five employees are not managed by a freelance employee (*NotBossFreelance*), that all projects have at least two project managers (*AtLeastTwoProjectManagers*), that each employee assigned to a project finishes his contract after the due date of the project (*PossibleEmployee*) and that the number of hours per week that freelances work lies between 5 and 30 (*ValidNHours*).

Figure 3.4 shows the graph corresponding to the previous CS. We can draw from it that constraints over *Project* may be reexpressed over *Employee*, *Department* and *Category*; constraints over *Employee* can be reexpressed over *Project*, *Department* and *Category*; constraints 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 constraint *MaxSalary* because this is the unique constraint that can be changed from *Department* to *Employee*.



context Department **inv** MaxSalary: self.employee->forAll(e| e.age>=25 or e.salary<=self.maxJuniorSal)
context Department **inv** NotBossFreelance:
self.employee->size()>5 implies not self.boss.oclIsTypeOf(Freelance)
context Department **inv** AtLeastTwoProjectManagers:
self.project->forAll(p| p.employee->select(e|e.category.name="PM")->size()>=2
context Project **inv** PossibleEmployeee: self.employee->forAll(e|e.expirationDate<self.dueDate)
context Employee **inv** ValidAge: self.age>16
context Freelance **inv** ValidNHours: self.hoursWeek>=5 and self.hoursWeek<=30

Figure 3.3 - Conceptual schema used as running example

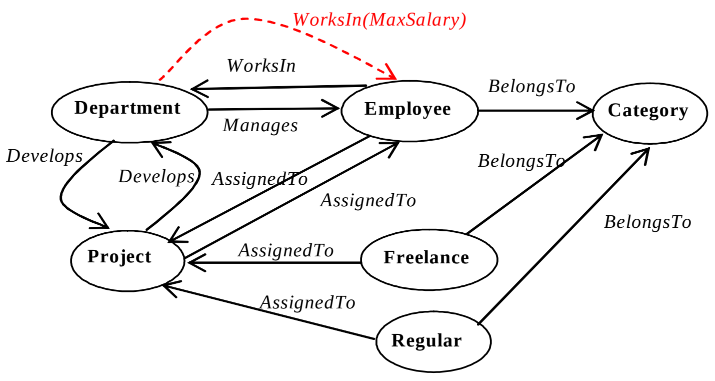


Figure 3.4 – Graph of the conceptual schema

3.1.2 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 can easily adapt (as we have done) a graph-searching procedure such as the depth-first search [6], using cet_1 as initial vertex and terminating the search only after all different paths reaching cet_2 have been generated. Next section uses these paths to redefine c_1 in terms of the 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 constraint *MaxSalary* we can use the edge *WorksIn* from *Department* and *Employee*, and thus, there is an additional path: *Department-WorksIn-Employee*.

3.1.3 Redefining the constraint 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.*$r_1.r_2. \dots r_n$*->notEmpty() implies self.*$r_1.r_2. \dots r_n$ *->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 relationship types appearing in *p*. Therefore, r_1 represents the navigation from cet_2 to v_n using the relationship type 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 constraint *MaxSalary* (*context Department inv:self.employee->forAll(e| e.age>=25 or e.salary<=self.maxJuniorSal)*) *may be* redefined over *Employee* because of the path *p*={*WorksIn*}. The redefined constraint *MaxSalary'* is:

context *Employee* ***inv****: self.employer->notEmpty() implies self.employer->forAll(d|d.employee->forAll(e.age>=25 or e.salary<= d.maxJuniorSal))*

Since OCL does not define the navigation through n-ary relationship types, when e_i represents an n-ary relationship type between v_{i+1} and v_i, we must navigate first from v_{i+1} to the corresponding reified entity type and then from the reified entity type to v_i.

Moreover, if an edge e_i links vertices v_{i+1} and v_i, the corresponding relationship type *R* 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 subtype 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* when translated from *Freelance* to *Category* results in: *self.employee->select(e|e.oclIsTypeOf(Freelance)). oclAsType(Freelance).hoursWeek >= 5 and …*

We provide some equivalences to simplify the new constraint c_2.

1. self.$r_1.r_2. \dots r_n$->notEmpty() $\leftrightarrow$ true if the multiplicity of *self.*$r_1.r_2. \dots r_n$ is at least one, i.e. if all the minimum multiplicities of $r_1.r_2. \dots r_n$ are at least one.
2. X->forAll(v|v.Y) $\leftrightarrow$ X.Y, if X is a collection of a single element. All the occurrences of *v* in *Y* are replaced with *X*.
3. self.$r_1.r_2. \dots .r_i.r_j \dots r_n$->forAll(X) $\leftrightarrow$ self.$r_1.r_2. \dots .r_{i-1}.r_{j+1} \dots r_n$->forAll(X), when r_i and r_j are the two roles of the same binary relationship type and the minimum multiplicity of r_i is at least one. When the maximum multiplicity of r_j is also one, the objects obtained at r_n are the same in both expressions. Otherwise, the sequence of navigations on the left hand side may return more objects at r_n. However, when the minimum multiplicity of all opposite roles from r_1 to r_{i-1} is at least one, those additional objects will be checked in the right hand side expression when evaluating another instance of the context entity type.
4. X.r_i->forAll(v| v.r_j.Y) $\leftrightarrow$ X->forAll(v_2| v_2.Y), when r_i and r_j are the two roles of the same binary relationship type. Each additional occurrence of *v* in *Y* must be

replaced by $v_2.r_i$ or by $v_2.r_i$*->forAll* over the expression where *v* appeared when the multiplicity of r_i is greater than 1. The rule can also be applied when the body of the *forAll* is a conjunction or a disjunction of various literals When it is a disjunction the multiplicity of $v_2.r_i$ must be at most one.

5. Given a reified entity type RET (see Figure 3.5): X.ret.b.Y ↔ X.b.Y
6. Given a reified entity type RET: context RET inv: self.a.b.r_1..r_n->forAll(X) ↔ context RET inv: self.b.r_1..r_n->forAll(X)

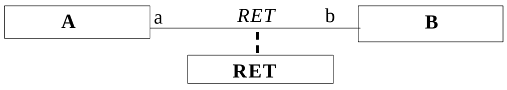


Figure 3.5 – A reified entity type

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

context *Employee* ***inv****: self.age>=25 or self.salary<=self.employer.maxJuniorSal*

3.2 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 (they are 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 constraint *ValidNHours*. If we want to move the constraint from *Freelance* to *Employee*, the new constraint would be:

context *Employee* ***inv*** *ValidNHours:*
self.oclIsTypeOf(Freelance) implies self.hoursWeek>5 and self.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 constraints are not semantically equivalent[1]. If the set of generalization relationships between cet_1 and its direct subtypes is covering [10] (also called *complete*) c_1 can be replaced as long as we add a new constraint to each direct subtype of cet_1 with the same body as c_1. For instance, if we try to change the constraint *ValidAge* from *Employee* to *Freelance* we need to add also *ValidAge* to *Regular* to ensure that all employees have a valid age.

[1] Except for those constraints where the body is already defined to apply only over the instances of the subtype cet_2. In such a case we c_2 is equivalent to c_1

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(cet₁) 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) $\leftrightarrow$ true
– self.oclAsType(cet).Y $\leftrightarrow$ self.Y

4. Computing all alternative context changes for a constraint

Once we know how to change from a context entity type to another (given) context, we are going to show how to obtain all alternative representations of a certain integrity constraint assuming that the new context may be any entity type of the CS.

To compute all possible alternatives we generalize the methods described in section 3. We build the graph as we have defined in section 3.1.1 with the only difference that all relationship types with multiplicities **:** are assumed to be reified. In this way, all reified entity types become vertices of the graph and turn out to be candidate context entity types.

In our example, the previous rule implies including the vertex *AssignedTo* into the graph. Figure 4.1 shows the updated part of the graph where new edges have been added according to the rules described in section 3.1.1.

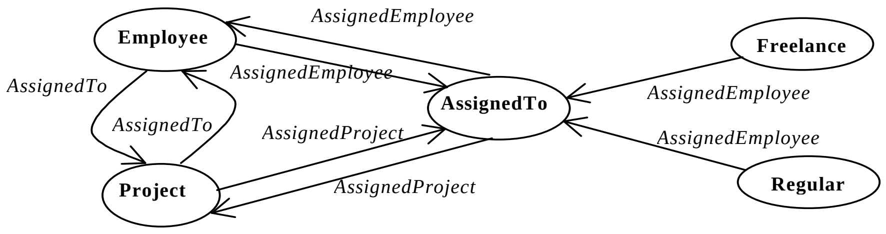


Figure 4.1 – Updated graph

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

As an example, we obtain sixteen different alternative representations of the constraint *MaxSalary* defined in Figure 3.3 (one for every path between *Department* and the related types in the graph: *Employee*, *Project*, *Category* and *AssignedTo*). Table 4.1 shows the list of valid paths.

Table 4.1 – Valid paths for *MaxSalary*

Final context	**Path**
Employee	Department – *Manages* - Employee
	Department - *WorksIn* -Employee
	Department – *Develops* – Project – *AssignedTo* - Employee
	Department – *Develops* – Project – *AssignedProject* – AssignedTo – *AssignedEmployee* - Employee

Category	Department – *Manages* - Employee - *BelongsTo* - Category
	Department - *WorksIn* -Employee -*BelongsTo* – Category
	Department – *Develops* – Project – *AssignedTo* - Employee -*BelongsTo* – Category
	Department – *Develops* – Project – *AssignedProject* – AssignedTo – *AssignedEmployee* - Employee *BelongsTo* - Category
Project	Department – *Develops* – Project
	Department – *Manages* – Employee – *AssignedTo* – Project
	Department – *Manages* – Employee – *AssignedEmployee* – AssignedTo – *AssignedProject* – Project
	Department – *WorksIn* – Employee – *AssignedTo* – Project
	Department – *WorksIn* – Employee – *AssignedEmployee* – AssignedTo – *AssignedProject* – Project
AssignedTo	Department – *Manages* – Employee – *AssignedEmployee* - AssignedTo
	Department - *WorksIn* –Employee– *AssignedEmployee* – AssignedTo
	Department – *Develops* – Project - *AssignedProject*– AssignedTo

When looking for a simpler representation of c_1, we can reduce the search space just by considering the paths including only edges representing relationship types referred in the body of the original constraint.

We can discard the other paths since alternatives obtained with them are more complex than the original one. Recall that any alternative constraint representation c_2 for a constraint c_1 obtained using the graph *G* initially presents a body consisting in a navigation (obtained from the path) from the context cet_2 of c_2 to the context cet_1 of c_1 followed by the same body as c_1. Therefore, if no simplifications can be applied, c_2 is more complex than c_1 since its complexity may be regarded as that of c_1 plus that of the navigation from cet_2 to cet_1. Note that simplifications over c_2 can only be applied when the edges that form the path from cet_2 to cet_1 are also included in the body of c_1.

Therefore, to obtain the relevant alternative representations for a constraint c_1 it is enough to apply the previous algorithm over the graph *G'*, subgraph of *G*, that contains the edges of *G* representing relationship types referenced in the body of c_1 along with their vertices and the vertices corresponding to the reified entity types of those edges (plus the edges between the reified type and the other entity types in *G'*).

The subgraph *G'* corresponding to the constraint *MaxSalary* is shown in Figure 4.2. We reduce the number of alternative representations from sixteen to only one.

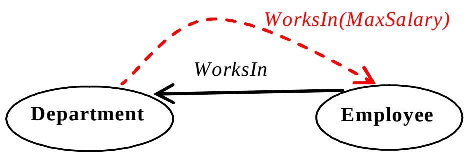


Figure 4.2 – Subgraph for the constraint maxSalary

According to this optimization, table 4.2 summarizes the alternative representations (already simplified) for all constraints of our example.

For instance, the constraint *PossibleEmployee* over the reified type *AssignedTo* is first defined as:

context *AssignedTo* ***inv:*** *self.project->notEmpty() implies self.project->forAll(p|*
*p.employee->forAll(e|e.expirationDate<self.project.dueDat*e)

and then simplified by means of the equivalences in section 3.1.3.

Table 4.2 – Alternative representations for the example constraints

Constraint	**Alternative representations**
MaxSalary	context Department inv: self.employee->forAll(e|e.age>=25 or e.salary<=self.maxJuniorSal)
	context Employee inv: self.age>=25 or self.salary<=self.employer.maxJuniorSal
NotBossFreelance	context Department inv: self.employee->size()>5 implies not self.boss.oclIsTypeOf(Freelance)
	context Employee inv: not self.managed.employee->size()>5 or not self.oclIsTypeOf(Freelance)
	context Freelance[2] inv: not self.managed.employee->size()>5
AtLeastTwo ProjectManagers	context Department inv: self.project->forAll(p|p.employee->select(e|e.category.name="PM")->size()>=2
	context Project inv: self.employee->select(e| e.category.name="PM")->size()>=2
	context Employee inv: self.project.employee->select(e|e.category.name="PM")->size()>=2
PossibleEmployee	context Project inv: self.employee->forAll(e|self.dueDate<e.expirationDate)
	context Employee inv: self.project->forAll(p|p.dueDate>self.expirationDate)
	context AssignedTo inv: self.project.dueDate>self.employee.expirationDate
ValidAge	context Employee inv ValidAge: self.age>16
ValidNHours	context Freelance inv: self.hoursWeek>=5 and self.hoursWeek<=30

5. Finding the best representation for a constraint

In many situations, the designer is interested to find the best alternative representation for an integrity constraint c_1 among all alternative representations we may obtain by means of the method we have proposed.

To obtain the best representation we must be able to compare the set of computed alternative constraints. Given two semantically equivalent constraints c_1 and c_2, we define that c_2 is better than c_1 if the complexity(c_1) > complexity(c_2) where *complexity* is a function that returns the complexity value of an OCL expression, according to a designer-defined complexity model. Clearly, the best alternative representation may in some cases be the original one.

The election of a particular complexity model may be caused by different goals in the simplification process like improving understandability of the resulting representation or selecting the one that involves more efficient constraint checking. For instance, a designer may want to define that c_1 is better than c_2 if it contains a fewer number of navigations.

Very little work has been done in the area of metrics to evaluate OCL expressions. [12] proposes a set of parameters to characterize an OCL expression (number of attributes, navigations, operations…) but does not interpret them. [7] proposes that a constraint is simpler if it contains less navigations and less iterator expressions but does not clarify whether it is preferable a constraint with less navigations than another with less iterator expressions.

[2] The constraint can be defined over *Freelance* as a subtype of *Employee* because the body of the constraint can only be violated by *Freelance* instances

It is out of the scope of this paper to propose a complete complexity model for OCL expressions. However, as an example, we define a partial complexity model to optimize integrity constraint checking since, as discussed in [1], the election of a particular representation affects the efficiency of this process. We consider that a constraint c_1, defined over cet_1, is better than an equivalent constraint c_2, defined over cet_2, if the number of entities accessed when evaluating c_1 over a single instance of cet_1 is lower than the number of entities taken into account when evaluating c_2 over an instance of cet_2.

Obviously, at design time we cannot compute the exact number of entities accessed. Nevertheless, we may still compare alternative representations of the same constraint. For instance, among the alternatives of table 4.2, *MaxSalary* is best redefined over *Employee* because we only need to compare the employee with his/her department instead of all the employees of a department. Moreover, *PossibleEmployee* is best redefined over *AssignedTo* since we only need to access one project and one employee while in the other two options we access all employees of a project or all projects of an employee.

6. Conclusions and further work

We have presented a method that given an OCL constraint generates its alternative representations. Our method considers changes in the body of the constraint (defined as equivalences between expressions) as well as the possibility of redefining the constraint 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.

The main part of 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 and choose the best one using a complexity model provided by the designer.

The method proposed in this paper is useful in several situations like increasing the understandability of integrity constraints, helping its validation, improving the efficiency of integrity checking or learning the OCL language itself.

Further research may involve looking for additional useful equivalences that may improve further the results of our method or the definition of a set of complexity models that would allow obtaining the *best* representation of a given constraint according to the preferences of the designer.

References

1. Cabot, J., Teniente, E.: Computing the Relevant Instances that May Violate an OCL constraint. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05), to appear, (2005)
2. Correa, A., Werner, C.: Applying Refactoring Techniques to UML/OCL Models. In: Proc. 7th International Conference on the Unified Modeling Language (UML'04), Lecture Notes in Computer Science, 3273 (2004) 173-187
3. Embley, D. W., Barry, D. K., Woodfield, S.: Object-Oriented Systems Analysis. A Model-Driven Approach. Yourdon Press Computing Series. Yourdon (1992)
4. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: A. Clark and J. Warmer, (eds.): Object Modeling with the OCL. Springer-Verlag (2002) 85-114
5. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)
6. Jungnickel, D.: Graphs, networks and algorithms. Springer-Verlag (1999)
7. Ledru, Y., Dupuy-Chessa, S., Fadil, H.: Towards computer-aided design of OCL constraints. In: J. Grundspenkis and M. Kirikova, (eds.): CAiSE'04 Workshops Proceedings, Vol. 1. Faculty of Computer Science and Information Technology, Riga Technical University, Riga, Latvia (2004) 329-338
8. McAllister, A.: Complete rules for n-ary relationship cardinality constraints. Data Knowl. Eng. 27 (1998) 255-288
9. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) (2003)
10. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
11. OMG: MDA Guide Version 1.0.1. (2003)
12. Reynoso, L., Genero, M., Piattini, M.: Towards a metric suite for OCL Expressions expressed within UML/OCL models. Journal of Comptuer Science and Technology 4 (2004) 38-44