Transformation techniques for OCL constraints

J. Cabot\textsuperscript{a,}\textsuperscript{*}, E. Teniente\textsuperscript{b}

\textsuperscript{a} Estudis d’Inform\`atica i Multim`edia, Universitat Oberta de Catalunya, Rambla del Poblenou, 156, E08018 Barcelona, Spain
\textsuperscript{b} Dept. Llenguatges i Sistemes Inform\`atics, Universitat Polit\`ecnica de Catalunya, Campus Nord, Ed. Omega, 132, E08034 Barcelona, Spain

Received 11 August 2006; received in revised form 13 April 2007; accepted 14 May 2007
Available online 24 June 2007

Abstract

Constraints play a key role in the definition of conceptual schemas. In the UML, constraints are usually specified by means of invariants written in the OCL. However, due to the high expressiveness of the OCL, the designer has different syntactic alternatives to express each constraint. The techniques presented in this paper assist the designer during the definition of the constraints by means of generating equivalent alternatives for the initially defined ones. Moreover, in the context of the MDA, transformations between these different alternatives are required as part of the PIM-to-PIM, PIM-to-PSM or PIM-to-code transformations of the original conceptual schema.

© 2007 Elsevier B.V. All rights reserved.

Keywords: OCL; Integrity constraints; Transformation

1. Introduction

Integrity constraints are a fundamental part of the definition of a conceptual schema (CS) [11]. In general, 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 [8]. In the UML this is usually achieved by means of invariants written in the OCL [17]. Predefined (graphical) constraints can also be expressed in the OCL [10].

Due to the high expressiveness of the OCL, there are several alternative ways to define the same integrity constraint. For instance, given the CS in Fig. 1, the constraint “the salary of an employee must be higher than the minimum salary of his/her department” may be defined as (among some 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, designers may not be aware of all different alternative definitions and thus they may just choose the one they care about at the moment of defining the constraint. However, the appropriateness of the particular definition chosen depends on the specific purpose for which the constraint is defined (like, for instance, understandability of

\* Corresponding author. Tel.: +34 933263717.
E-mail address: jcabot@uoc.edu (J. Cabot).

0167-6423/S - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.05.001
software requirements specification or efficiency to achieve during automatic code generation). Then, it will usually happen that the designer does not define the constraint in the best way according to the intended purpose. For this reason, it becomes necessary to develop techniques to be able to automatically transform a given integrity constraint into its alternative equivalent syntactic representations.

In the context of the MDA [19], we may identify certain scenarios where such transformations of the integrity constraints are also required. For instance, in PIM-to-PIM transformations, replacing the constraints with alternative representations is required as part of the refactoring operations at the model level [15]. Moreover, in PIM-to-PSM or PIM-to-code transformations, aimed at automatically generating a final implementation of the system directly derived from its specification, the generation of alternative representations may be necessary to achieve better efficiency [2].

In general, an alternative representation for a given constraint may be obtained in two different ways: either by keeping the same context type and replacing the body of the constraint with an equivalent one (as it happens between constraints 1 and 3 of the previous example) or by rewriting the constraint considering a different context than the original one (as it happens with constraints 1 and 2).

In this paper we propose several transformation techniques that allow to obtain a set of alternative constraint representations that are semantically equivalent to a given constraint. The replacement of the constraint body is handled by means of specifying a set of equivalence rules between the different elements and constructs that may appear in the OCL expression defining the body of the constraint. The redefinition of the constraint using an alternative context is formalized as a path problem over a graph representing the CS. Using this graph we identify which entity types are candidates for acting as a new context type for the constraint and, then, we obtain all the possible redefinitions for each of them. Our proposal allows to generate all alternative redefinitions of the given constraint when using a different entity type as a context type but not all possible equivalent bodies for each alternative because of the huge number of equivalences among the different OCL constructs.

In contrast with the extensive research devoted to model transformations, redefinition of OCL expressions has received little attention in the past. In particular, [14] discusses the advantages of changing the context of a constraint but does not define which are the possible new contexts nor provides a method to generate such redefined constraints. Similarly, [6] proposes the context change as one of the possible refactorings to improve the specified OCL expressions but does not provide any method to automatically generate this context change. [9] provides some rules with the purpose of simplifying the constraints but the rules are not aimed at generating several alternative constraint definitions (and again, context changes are not addressed). [15] 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 proposal able to fully deal with transformations of OCL constraints.

The work reported here extends our previous work published at [4] in several directions. First, we also handle integrity constraint redefinitions within the same context type (and not only redefinitions concerning different entity types). Second, we provide a more detailed definition of the techniques that allow computing all alternative changes for a given constraint. Third, we describe the main scenarios in which our transformation techniques are useful. Finally, we portray a prototype tool which implements the techniques presented in this paper.

The structure of the paper is as follows. The next section defines several equivalences between OCL expressions. Then, we propose transformation techniques to change the context of a constraint to a particular entity type (Section 3) and we extend them to any type of the CS (Section 4). Section 5 discusses some scenarios where the provided transformations are especially helpful. Section 6 presents our tool implementation. Finally, we give our conclusions and point out future work in Section 7.
2. Equivalences between OCL expressions

One possible way to generate an alternative representation for an integrity constraint is to replace its body with an equivalent one, while keeping the original context type. For instance, assume we define an integrity constraint in the CS of Fig. 1 to prevent junior employees (those with an age lower than 25) to earn more than the maxJuniorSal value defined for their department. This constraint could be specified in OCL as follows:

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

The previous OCL constraint is equivalent to the following one, defined over the same context type:

```
context Department inv MaxSalary': self.employee -> forAll(e | e.age = 25 or e.salary <= self.maxJuniorSal).
```

Note that the meaning of both constraints is exactly the same. However, in this case, the second expression is simpler since it is shorter and it uses less operators.

We propose to perform such kind of transformations by means of a set of equivalence rules between OCL expressions, which are described in this section. Each expression on the one side of the equivalence may be replaced with the expression on the other side to generate a new alternative body for the constraint. The set of rules is not exhaustive but it contains those equivalences we believe to be the most usual and/or useful ones according to our own experience. In particular, applying our equivalences, we could perform the transformation from `MaxSalary` to `MaxSalary'` illustrated in the previous example.

Before applying the rules, we need to unfold the OCL expressions to maximize the number of applicable rules. We say that an OCL expression is unfolded when all references to derived elements, query operations and variables resulting from let expressions are replaced with their definition. To guarantee termination, we assume that recursive derived elements are not unfolded. Additionally, we assume that all implicit variables in the OCL expression are made explicit.

The equivalence rules we have considered are defined in Sections 2.1–2.3. We have specified them in such a way that when applied in the left–right direction the equivalences reduce the number of different OCL operations that appear in an OCL expression. For instance, a left–right application of our rules would allow removing the `exists` operation from any OCL expression. As a result of this application we obtain OCL expressions equivalent to the original one but that do not require the full expressivity of the OCL. Hence, methods that provide automatic treatment of OCL expressions may benefit from this approach since they will need to handle only a reduced number of OCL constructs.

On the other hand, when applying the rules in the right–left direction we may obtain more understandable expressions since some of them replace a sequence of several operations with a single operator. For instance, as shown in Section 2.3, we may replace a combination of `not` and `or` operators with the `implies` operator.

In general, designers will choose which rules to apply (and in which direction) depending on their intended goal.

Section 2.1 presents basic equivalence rules. Section 2.2 defines equivalences 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.

2.1. Basic equivalences

Tables 1–3 define a list of basic equivalence rules. Most of these rules are based on the equivalences defined in the OCL standard itself [17]. We have grouped the equivalences by the type of expressions they affect (boolean, collection or iterator expressions). In the rules, the capital letters X, Y and Z represent arbitrary OCL expressions of the appropriate type. The letter o represents an arbitrary object. The expression r₁...rₙ represents a (possibly empty) sequence of navigations.

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 [17]. As an example, a constraint like “all employees must be older than 16” can be expressed as¹:

¹ Type.allInstances() can also be written as Type::allInstances().
context Employee inv ValidAge: Employee.allInstances() ->forAll(e.e.age > 16).

However, since constraints are assumed to be true for all instances of the context type (i.e. for all possible values of the self variable that represents any instance of the context type), the previous constraint could also have been specified as:

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

We propose two equivalences to include/remove the allInstances operation in the expressions that define the body of an integrity constraint definition. They are applicable when the type over which allInstances is applied coincides with the context type (ct) of the constraint. They may not be applied if the constraint already contains any explicit or implicit reference to the self variable.

- ct.allInstances().->forAll(v|Y) <=> Y', where Y' is obtained by replacing all occurrences of v (the iterator variable) in Y with self. As an example, see the previous ValidAge' constraint.
- ct.allInstances().->forAll(v1,v2..vn|Y) <=> ct.allInstances().->forAll(v2..vn|Y') where Y' is obtained by means of replacing all the occurrences of v1 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) of several clauses, each of which is a disjunction (sequence of ORs) of one or more literals, possibly negated. Any logical formula can be translated into a CNF by applying a well-known set of rules.

We propose to apply that set of rules plus an additional rule to deal with the if-then-else construct to (de)normalize any boolean OCL expression in order to generate additional equivalent representations which may improve the results obtained by considering the rest of the rules alone. The rules we propose are the following:

1. Eliminate the if-then-else construct and the implies and xor operators:
   (a) X implies Y ↔ not X or Y
   (b) if X then Y else Z endif ↔ (X implies Y) and (not X implies Z) ↔ (not X or Y) and (X or Z)
   (Y and Z must be boolean expressions)
   (c) X xor Y ↔ (X or Y) and not (X and Y) ↔ (X or Y) and (not X or not Y).

2. Move not inwards until the negations be immediately before literals by repetitively using the laws:
   (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.

1. Repeatedly distribute or over and by means of:
   (a) X or (Y and Z) ↔ (X or Y) and (X or Z).

2.4. Rule application

The different alternative representations of a given OCL expression are obtained by means of applying repetitively the previous equivalence rules. Beginning with an expression exp, the application of a rule r, transforms exp into an equivalent expression exp’. Then, any rule r’ that can be applied over exp or over exp’ generates a new alternative and so forth.

As an example, from the initial MaxSalary constraint definition we may obtain the alternative MaxSalary’ representation by means of the following sequence of rules:

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

2. Removing the allInstances operation:
   context Department inv MaxSalary: not self.employee-＞select(e|e.age＜25)-＞exists(e|e.salary＞self.maxJuniorSal).

3. Removing the exists iterator (rule not X-＞exists(Y) → X-＞forAll(not Y)):
   context Department inv MaxSalary: self.employee-＞select(e|e.age＜25)-＞forAll(e| not (e.salary＞self.maxJuniorSal)).

4. Removing the select iterator (rule X-＞select(Y)-＞forAll(Z) → X-＞forAll(Y implies Z)):
   context Department inv MaxSalary: self.employee-＞forAll(e|e.age＜25 implies not (e.salary＞self.maxJuniorSal)).

5. Transforming to CNF:
   context Department inv MaxSalary: self.employee-＞forAll(e|not (e.age＜25) or not (e.salary＞self.maxJuniorSal)).

6. Removing the not operator (rules not X＜Y→X ≥= Y and not X＞Y→X ＜= Y):
   context Department inv MaxSalary: self.employee-＞forAll(e|e.age＞=25 or e.salary＜=self.maxJuniorSal).

Termination of the transformation process depends on the set of rules chosen by the designer who is taking care of the transformation and on the direction in which they are applied. It is obvious that if the designer allows applying the same rule in both directions, then the generation process may enter into an infinite loop. To prevent this situation the designer should avoid applying a rule that would generate an alternative that has already been considered before. Similarly, confluence (i.e. the existence of a unique rewriting of the original constraint) is only guaranteed if the designer defines a total order regarding the selection of rules to apply over an expression when several rules are applicable.
3. Changing the context type of a constraint

In general, the designer may choose among several entity types when deciding the context of a particular constraint. As we said in the introduction, it may be more useful sometimes to use a certain context, together with a corresponding constraint definition, instead of a different one.

Given two different context types \( ct_1 \) and \( ct_2 \) and a constraint \( c_1 \) defined over \( ct_1 \), we show in this section how to automatically obtain a constraint \( c_2 \) defined over \( ct_2 \) which is semantically equivalent to \( c_1 \). A constraint defines a condition that must always be satisfied by the system state. More precisely, when defined in OCL, each constraint must be true for all instances of the context 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 condition to be evaluated over them is also the same.

In general, it may happen that several semantically equivalent constraints defined over \( ct_2 \) exist. Then, our transformation techniques generate all of them.

We assume in this section that the final context \( ct_2 \) is given by the designer (or by an external method) since our main goal here is to define the conditions under which the constraint redefinition may take place and the procedure to obtain the equivalent OCL expression for the original constraint when defined over \( ct_2 \). These results will then be used in the next section when computing the equivalent OCL expressions for all alternative contexts.

Changing the context type of a constraint makes only sense when the constraint is defined using a single instance of the context type (i.e. when the constraint body contains the \textit{self} variable). Otherwise, i.e. when the constraint is defined with the \textit{allInstances} operation, it is not worthy since its body will always be the same regardless the context chosen.

In Section 3.1, we assume that \( ct_2 \) is any entity type of the CS related with \( ct_1 \) through a sequence of associations. Afterwards, in Section 3.2, we allow \( ct_2 \) to belong to the same taxonomy as \( ct_1 \). Both alternatives are not exclusive since \( ct_2 \) may belong to the same taxonomy as \( ct_1 \) and be also related with it.

3.1. Changing the context between related entity types

This section focuses on the transformation of a constraint \( c_1 \) with context \( ct_1 \) to a semantically equivalent constraint \( c_2 \) with context \( ct_2 \), where \( ct_1 \) and \( ct_2 \) are related through one or more sequence of associations that allow navigating between them.

According to one of the requirements to guarantee the semantic equivalence of \( c_2 \) and \( c_1 \), the context change from \( ct_1 \) to \( ct_2 \) is only possible when there is at least one sequence of associations \( \text{seq}_{\text{ass}} \) relating both types. Otherwise, when the constraint is evaluated over \( ct_2 \) it is not possible to retrieve the instances of \( ct_1 \) that must be taken into account.

Additionally, \( \text{seq}_{\text{ass}} \) has to verify that \( \text{set}_{\text{ct}1} = \text{set'}_{\text{ct}1} \); where \( \text{set}_{\text{ct}1} \) is the population of \( ct_1 \) (the set of instances that \( ct_1 \) restricts) while \( \text{set'}_{\text{ct}1} \) is the set of instances of \( ct_1 \) obtained when navigating from the instances of \( ct_2 \) to \( ct_1 \) through \( \text{seq}_{\text{ass}} \) (i.e. \( \text{set'}_{\text{ct}1} \) is the set of instances of \( ct_1 \) restricted when \( ct_1 \) is defined over \( ct_2 \)). This is required to guarantee that the constraint \( c_2 \) is evaluated exactly over the same set of instances as \( c_1 \). Note that, otherwise, the set of instances \( \text{set}_{\text{ct}1} - \text{set'}_{\text{ct}1} \) would not be restricted by \( c_2 \), therefore making \( c_2 \) not equivalent to \( c_1 \).

We can determine whether \( \text{set}_{\text{ct}1} = \text{set'}_{\text{ct}1} \) by studying the multiplicities of the associations included in \( \text{seq}_{\text{ass}} \).

Intuitively, if two entity types \( A \) and \( B \) are related through an association \( AB \) with multiplicities \( 0..*:1..* \) (see Fig. 2) it means that each instance of \( A \) is related to at least an instance of \( B \). Thus, if we navigate from all 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 \( A \) 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 \( \text{context A inv: self.a > 0} \) may be translated to: \( \text{context B inv: self.a > forAll(a1 > 0)} \). On the contrary, the constraint \( \text{context B inv: self.b < 5} \) 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 \( b1 \) greater than 5.

Then, we can state that \( \text{set}_{\text{ct}1} = \text{set'}_{\text{ct}1} \) if the value of all minimum multiplicities of the roles used to navigate from \( ct_1 \) to \( ct_2 \) through the associations in \( \text{seq}_{\text{ass}} \) is at least one. This guarantees that the navigation from \( ct_2 \) to \( ct_1 \) reaches all \( ct_1 \) 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 \( A \), but not from \( B \) to \( A \) or \( C \) to \( A \).
For each binary association between two entity types

Given a

Fig. 2. Example of an abstract conceptual schema.

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 $ct_1$ related with some instance of $ct_2$ we can use $ct_2$ as context of $c_1$. Roughly, this may happen when each literal appearing in the body of $c_1$ includes a navigation to $ct_2$. As an example, consider the MaxSalary constraint defined in Section 2. Even though not all departments have employees assigned, the constraint only affects departments with employees (the others always satisfy the constraint). 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 associations from $ct_1$ to $ct_2$ 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, how to find the alternative paths and, for each one of them, how to obtain the new body of the constraint over the new context type.

3.1.1. 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 contains a path from a vertex $v_1$ to a vertex $v_2$ then constraints defined over $v_1$ can be redefined using $v_2$ as a context type.

The graph must be a directed graph (digraph), since being able to change constraints from $ct_1$ to $ct_2$ (i.e. from the vertex representing $ct_1$ to the vertex representing $ct_2$) does not imply that we can also change constraints from $ct_2$ to $ct_1$, the context change is transitive but not symmetric. For instance, consider the graph of Fig. 3, which is the one obtained from the CS of Fig. 2. The graph shows that constraints 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 $B$.

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:

1. All entity types, including reified ones (i.e. association classes), are vertices of $G$.
2. 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.
3. Given a $n$-ary association $As$ among a set of entity types $E_1 \ldots E_n$ we add an edge from $E_i$ to $E_j$ if we can deduce, from the multiplicities of the roles in $As$, that the minimum multiplicity from $E_i$ to $E_j$ is at least one. Although these binary multiplicities are usually left unspecified in class diagrams, [16] shows 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.
4. 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 $E_1 \ldots E_n$ are the participants of the association. We add these edges since an instance of an association class is 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 $E_j \rightarrow E_i$. 

Fig. 3. Example graph.
5. Since subtypes inherit all the associations of their supertypes, for each edge \( A \rightarrow B \) we add an edge \( A_i \rightarrow B \) to each subtype \( A_i \) 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 \( c1 \) to \( c2 \) all constraints defined over \( c1 \) can be redefined using \( c2 \) as a context type.

As we have seen before, a context change from \( c1 \) to \( c2 \) may also be possible when the body of the original constraint only affects those instances of \( c1 \) related with instances of \( c2 \). To deal with these particular cases, we need to add to \( G \) some edges that are specific for certain constraints. For this reason, those edges are labelled with the name of a constraint and paths including them are only valid for changing the context of that particular constraint.

In Fig. 4 we show the CS we will use as a running example in the rest of the paper. It specifies information about departments, their projects and employees and it includes the following six textual constraints. The first two are the constraints \( \text{MaxSalary} \) and \( \text{ValidAge} \) shown in Section 2. The others ensure that departments with more than five employees are not managed by a freelance employee (\( \text{NotBossFreelance} \)), that all projects have at least two project managers (\( \text{AtLeastTwoProjectManagers} \)), that each employee assigned to a project finishes his contract after the due date of the project (\( \text{PossibleEmployee} \)) and that the number of hours per week that freelances work lies between 5 and 30 (\( \text{ValidNHours} \)).

- **context** Department \( \text{inv} \) MaxSalary: self.employee-\( \text{forAll}(e)\text{.age} \geq 25 \) or e.salary \( \leq \) self.maxJuniorSal).
- **context** Employee \( \text{inv} \) ValidAge: self.age \( \geq \) 16.
- **context** Department \( \text{inv} \) NotBossFreelance: self.employee-\( \text{size()} \geq 5 \) implies not self.boss.oclIsTypeOf(Freelance)
- **context** Department \( \text{inv} \) AtLeastTwoProjectManagers: self.project-\( \text{forAll}(p)\text{.employee-}\text{select}(e)\text{.category. name='PM'}\text{.size()} \geq 2)\).
- **context** Project \( \text{inv} \) PossibleEmployee: self.employee-\( \text{forAll}(e)\text{.expirationDate} \geq \) self.dueDate).
- **context** Freelance \( \text{inv} \) ValidNHours: self.hoursWeek \( \geq 5 \) and self.hoursWeek \( \leq 30 \).

Fig. 5 shows the graph obtained from the previous CS. We can draw from it that constraints over Project may be redefined over Employee, Department and Category; constraints over Employee can be redefined 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 labelled with the name of the constraint \( \text{MaxSalary} \) because this is the unique constraint that can be changed from Department to Employee using the association WorksIn.

### 3.1.2. Computing alternative paths

Each different path from \( c1 \) to \( c2 \) represents a different way to express the original constraint \( c1 \) in terms of the new context \( c2 \). To compute all alternative paths from \( c1 \) to \( c2 \) we have slightly adapted the depth-first graph searching procedure [12], using \( c1 \) as initial vertex and terminating the search only after all alternative paths reaching \( c2 \) have been generated. To avoid cycles, we do not consider as alternative paths those that contain repeated edges.

For instance, alternative paths from Department to Employee are the following: Department-Manages-Employee and Department-Develops-Project-AssignedTo-Employee. When looking for alternatives for the constraint \( \text{MaxSalary} \) we can also use the edge WorksIn from Department to Employee, and thus, there is an additional path: Department-WorksIn-Employee.
An alternative path may have repeated vertices. However, to simplify our presentation, we will not consider them in the rest of the paper.

3.1.3. Redefining the constraint over the new context type

Given a constraint $c_1$ with a body $X$ defined over a context type $ct_1$, a context type $ct_2$ and a path $p = \{e_1, \ldots, e_n\}$ (where $e_1, \ldots, e_n$ are the edges linking the vertices $\{ct_1, v_2, \ldots, v_n, ct_2\}$), the semantically equivalent constraint $c_2$ defined over $ct_2$ has the form:

$$\text{context } ct_2 \text{ inv } c_2: self.r_1, r_2, \ldots, r_n \rightarrow \text{notEmpty()} \implies self.r_1, r_2, \ldots, r_n \rightarrow \text{forall}(v|X)$$

where all occurrences of $self$ in $X$ have been replaced with $v$ and $r_1, \ldots, r_n$ are the roles that allow navigating from $ct_2$ to $ct_1$ using the associations appearing in $p$. Therefore, $r_1$ represents the navigation from $ct_2$ 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 from $v_2$ to $ct_1$ using $e_1$.

Intuitively, it can be seen that $c_1$ and $c_2$ are equivalent since both apply the same condition to the instances of $ct_1$ (the condition $X$) and apply it over the same set of instances (guaranteed by the graph definition process).

For instance, the constraint $\text{MaxSalary}$ (context $\text{Department}$ inv: $self.\text{employee}\rightarrow \text{forall}(e|e.\text{age} \geq 25 \text{ or } e.\text{salary} \leq \text{self.\text{maxJuniorSal}})$) may be redefined over $\text{Employee}$ because of the path $p = \{\text{WorksIn}\}$. The redefined constraint $\text{MaxSalary'}$ is:

$$\text{context } \text{Employee} \text{ inv: } self.\text{employer}\rightarrow \text{notEmpty()} \implies self.\text{employer}\rightarrow \text{forall}(d|d.\text{employee}\rightarrow \text{forall}(e|e.\text{age} \geq 25 \text{ or } e.\text{salary} \leq d.\text{maxJuniorSal}))$$

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, as ensured by the graph definition process, if an edge $e_i$ links vertices $v_{i+1}$ and $v_i$, there exists the corresponding association 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 the supertype of $E_i$ we need to add “select(oclIsTypeOf($E_i$))” to the corresponding $r_i$. For instance, the constraint $\text{ValidNHours}$ can be translated from $\text{Freelance}$ to $\text{Category}$. However, in the body of the resulting constraint, when navigating from $\text{Category}$ to $\text{Employee}$ we need to select just those employees that are freelances, since these are the only ones affected by the constraint. Then, the final body of $\text{ValidNHours}$ when redefined over $\text{Category}$ is the following:

$$\text{self.}\text{employee}\rightarrow \text{select(e|e.oclIsTypeOf($\text{Freelance}$))}\rightarrow \text{forall}(f|f.\text{oclAsType($\text{Freelance}$)}.\text{hoursWeek} \geq 5 \text{ and } f.\text{oclAsType($\text{Freelance}$)}.\text{hoursWeek} \leq 30)$$

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

1. $self.r_1 \ldots r_n \rightarrow \text{notEmpty()} \rightarrow \text{true}$, if the multiplicity of $self.r_1 \ldots r_n$ is at least one, i.e. if all the minimum multiplicities of $r_1 \ldots r_n$ are at least one. In this case, it is sure that the navigation will return a non-empty set and, thus, there is no need to apply the $\text{notEmpty}$ operation.

2. $self.r_1 \ldots r_n \rightarrow \text{forall}(v|X) \rightarrow X$ (where all the occurrences of $v$ in $X$ are replaced with $self.r_1 \ldots r_n$), if the multiplicity of $self.r_1 \ldots r_n$ is at most one, i.e. if all the maximum multiplicities of $r_1 \ldots r_n$ are at most one. Then, the $\text{forall}$ iterator is no longer necessary.
1. Initial representation after the context change:
   \[ \text{context Employee inv: self.employer->notEmpty() implies self.employer->forAll(d|d.employee->forAll(e|e.age>=25 or e.salary<=d.maxJuniorSal))}. \]

2. Removing the notEmpty operator (rule 1 plus the rules true implies X -> not true or X, not true or X -> false or X and false or X -> X):
   \[ \text{context Employee inv: self.employer->forAll(d|d.employee->forAll(e|e.age>=25 or e.salary<=d.maxJuniorSal))}. \]

3. Removing the first forAll (rule 2):
   \[ \text{context Employee inv: self.employer.employee->forAll(e|e.age>=25 or e.salary<=self.employer.maxJuniorSal)}. \]

4. Removing the redundant navigation (rule 3):
   \[ \text{context Employee inv: self->forAll(e|e.age>=25 or e.salary<=self.employer.maxJuniorSal)}. \]
5. Removing the `forall` iterator (rule 2 again):
   
   **context Employee inv:** `self.age` ≥ 25 or `self.salary` ≤ `self.employer.maxJuniorSal`.

3.2. Changing the context within a taxonomy

   Given a constraint `c1` defined over a context type `ct1`, we are now interested in redefining `c1` using `ct2` as a context type, where `ct1` and `ct2` belong to the same taxonomy. This implies that either `ct1` is a subtype of `ct2`, a supertype or both have a common supertype (`ct1` and `ct2` are sibling types).

   When `ct1` is a supertype of `ct2`, the equivalent constraint `c2` defined over `ct2` has as a body: `self.oclIsTypeOf(ct1)` implies `X`, where `X` is the body of `c1`. In this way we ensure that `c2` is only applied over those instances that are instance of `ct1`.

   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.oclAsType(Freelance).hoursWeek > 5 and self.oclAsType(Freelance).hoursWeek < 30.**

   Note that, when accessing an attribute of the subtype, we need to use the `oclAsType` operator to do an explicit cast of the supertype variable.

   If `ct1` is a supertype of `ct2`, the new constraint `c2` is defined in `ct2` with exactly the same body as `c1`. However, `c2` cannot replace `c1` since in general `ct1` may contain instances not appearing in `ct2`. Thus, both constraints are not semantically equivalent. If the set of generalization relationships between `ct1` and its direct subtypes is covering [18] (also called complete) `c1` can be replaced as long as we add a new constraint to each direct subtype of `ct1` with the same body as `c1`. 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.

   When `ct1` and `ct2` share a common supertype the new constraint `c2` can never replace `c1` since not all instances of `ct1` need to be instances of `ct2`. As in the subtype case, the body of `c2` would be `self.oclIsTypeOf(ct1)` implies `X`.

   Before finalizing the context change to a new context entity type `ct` we can apply two simplification rules especially useful for this kind of transformations:

   1. `self.oclIsTypeOf(ct) → true`.
   2. `self.oclAsType(ct).X → self.X`.

4. Computing all alternative context changes for a constraint

   In the previous section we assumed that the final context type `ct2` was given by the designer or by an external method. Computation of alternative constraint representations having `ct2` as a new context type required considering paths from the original context to the vertex representing `ct2` in the graph. Now, to compute all possible context changes for a constraint `c1`, defined over `ct1`, we need to consider at least all possible paths between `ct1` and every different entity type `E` appearing as a vertex of the graph. Moreover, we must reify all the associations appearing in the original CS. When reified, associations appear also as vertices in the graph and turn out to be additional candidate context types for the constraint `c1`.

   Since the number of alternative contexts may become huge when all associations are reified, one may decide to relax the previous requirement and allow the designer to reify only a subset of the associations in the CS.

   As an example, assume that we are interested in obtaining all context changes for the constraint `MaxSalary` (as originally defined in Fig. 4) when reifying only the association `AssignedTo`.

   We must note first that the reification of `AssignedTo` causes the inclusion of a new vertex into the graph. Fig. 8 shows the updated part of the graph of Fig. 5, where new edges for the new vertex `AssignedTo` have been added according to the rules described in Section 3.1.1.

   Now, we have sixteen different alternative representations for `MaxSalary`. In particular, there is one alternative for every path between `Department` and the related types in the graph: `Employee`, `Project`, `Category` and `AssignedTo`. Table 4 shows the list of valid paths (column 2) for each possible final context (column 1).

---

2 Except for those constraints where the body is already defined to apply only over the instances of the subtype `ct2`. 
Each path constitutes a different alternative definition of the original constraint. The OCL expression corresponding to the body of each alternative is computed as explained in Section 3. Then, we can still apply the equivalences of Section 2 to generate additional alternatives by means of changing the body (but not the context) of the obtained constraints. For paths including vertices corresponding to entity types that participate in a taxonomy we must also consider the possible context changes along the taxonomy.

We may reduce the search space by just considering the paths including solely edges representing associations referred in the body of the original constraint. Note that we can discard the other paths because, assuming a simplified definition of the original constraint (which may be automatically obtained by means of applying the rules presented in Section 3.1.3), the alternatives obtained with them are surely 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 (extracted from the path) from the context $ct_2$ of $c_2$ to the context $ct_1$ 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 $ct_2$ to $ct_1$. Note that simplifications over $c_2$ can only be applied when the edges that form the path from $ct_2$ to $ct_1$ 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 consider the graph $G'$, subgraph of $G$, that contains the edges of $G$ representing associations 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 Fig. 9. Using $G'$ we reduce the number of alternative representations from sixteen to only one.

Table 4
Valid paths for MaxSalary

<table>
<thead>
<tr>
<th>Context</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Department – Manages – Employee</td>
</tr>
<tr>
<td></td>
<td>Department - WorksIn –Employee</td>
</tr>
<tr>
<td></td>
<td>Department – Develops – Project – AssignedTo – Employee</td>
</tr>
<tr>
<td></td>
<td>Department – Develops – Project – AssignedProject – AssignedTo – AssignedEmployee – Employee</td>
</tr>
<tr>
<td>Category</td>
<td>Department – Manages - Employee - BelongsTo – Category</td>
</tr>
<tr>
<td></td>
<td>Department - WorksIn -Employee -BelongsTo – Category</td>
</tr>
<tr>
<td></td>
<td>Department – Develops – Project – AssignedTo - Employee -BelongsTo – Category</td>
</tr>
<tr>
<td></td>
<td>Department – Develops – Project – AssignedProject – AssignedTo – AssignedEmployee – Employee - BelongsTo – Category</td>
</tr>
<tr>
<td>Project</td>
<td>Department – Develops – Project</td>
</tr>
<tr>
<td></td>
<td>Department – Manages – Employee – AssignedTo – Project</td>
</tr>
<tr>
<td></td>
<td>Department – Manages – Employee – AssignedEmployee – AssignedTo – AssignedProject – Project</td>
</tr>
<tr>
<td></td>
<td>Department – WorksIn – Employee – AssignedTo – Project</td>
</tr>
<tr>
<td></td>
<td>Department – WorksIn – Employee – AssignedEmployee – AssignedTo – AssignedProject – Project</td>
</tr>
<tr>
<td>AssignedTo</td>
<td>Department – Manages – Employee – AssignedEmployee – AssignedTo</td>
</tr>
<tr>
<td></td>
<td>Department - WorksIn –Employee – AssignedEmployee – AssignedTo</td>
</tr>
<tr>
<td></td>
<td>Department – Develops – Project – AssignedProject – AssignedTo</td>
</tr>
</tbody>
</table>
Fig. 9. Subgraph for the constraint MaxSalary.

Table 5
Alternative representations for the example constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Alternative representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxSalary</td>
<td><strong>context</strong> Department inv: self.employee-&gt;forAll(e.age&gt;=25 or e.salary&lt;=self.maxJuniorSal)</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> Employee inv: self.age&gt;=25 or self.salary&lt;=self.employer.maxJuniorSal</td>
</tr>
<tr>
<td>NotBossFreelance</td>
<td><strong>context</strong> Department inv: self.employee-&gt;size()&gt;5 implies not self.boss.oclIsTypeOf(Freelance)</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> Employee inv: self.managed.employee-&gt;size()&lt;=5 or not self.oclIsTypeOf(Freelance)</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> Freelance inv: self.managed.employee-&gt;size()&lt;=5</td>
</tr>
<tr>
<td>AtLeastTwoProjectManagers</td>
<td><strong>context</strong> Department inv: self.project-&gt;forAll(p</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> Project inv: self.employee-&gt;select(e</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> Employee inv: self.project-&gt;forAll(p</td>
</tr>
<tr>
<td>PossibleEmployee</td>
<td><strong>context</strong> Project inv: self.employee-&gt;forAll(e</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> Employee inv: self.project-&gt;forAll(p</td>
</tr>
<tr>
<td></td>
<td><strong>context</strong> AssignedTo inv: self.project.dueDate&gt;self.employee.expirationDate</td>
</tr>
<tr>
<td>ValidAge</td>
<td><strong>context</strong> Employee inv: self.age&gt;16</td>
</tr>
<tr>
<td>ValidNHours</td>
<td><strong>context</strong> Freelance inv: self.hoursWeek&gt;=5 and self.hoursWeek&lt;=30</td>
</tr>
</tbody>
</table>

According to this optimization, Table 5 summarizes the alternative representations (already simplified with the rules of Section 3.1.3) for all constraints of our example. Note that for some constraints (as ValidAge and ValidNHours) the original representation is the only alternative. Notice also that NotBossFreelance can be defined over Freelance as a subtype of Employee because the body of the constraint can only be violated by Freelance instances.

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.dueDate)  

and then simplified by means of removing the notEmpty operator (self.project has always a multiplicity value of 1), the first forAll (for the same reason) and, finally, applying the specific rules proposed for reified types.

5. Application scenarios for the constraint transformation techniques

There are at least two promising scenarios in which the transformation techniques proposed in this paper may be useful:

1. At the modeling level, since in information systems engineering it is required to specify a set of models with different objectives [13] and thus in which the best way to express the constraints can vary. These different representations should be automatically generated.
In the context of the MDA [19], such a need is clearly envisaged in PIM-to-PIM transformations. In particular, several refactoring operations (see, [22] or [15] for examples) have been proposed to improve the design and structure of PIMs. Clearly, OCL expressions may be affected by these refactorings and, as a consequence, they need to be transformed to keep the consistency of the new generated PIM model.

As a simple example, when a supertype is removed and all its properties are assigned to its subtypes we need to redefine the constraints having the supertype as a context type in terms of its subtypes. For instance, that may happen in our running example if we remove the Employee type and move all its attributes and associations to Freelance and Regular. In this case we also need to change the context of the constraint ValidAge from Employee to both Freelance and Regular, and this can be done with the transformation techniques proposed in Section 3.2.

[15] mentions other issues that may be raised by refactoring operations. To avoid problems with OCL expressions, [15] allows to perform only the refactoring operations under certain strong restrictions which are not required when our transformation techniques are available. Therefore, our techniques may help to overcome the limitations of this method in this respect.

Additionally, we could also regard our transformation techniques as refactoring operations that improve the understandability of the OCL expressions. In this sense, we provide more powerful refactorings than the ones proposed in [6].

2. For modeling and code generation frameworks, since it is essential to be able to manage and to verify constraints from first level models to their implementation.

The main goal of PIM-to-PSM and PIM-to-code transformations is to (semi)automatically obtain the final implementation of the system from the initial PIM specification. Therefore, and due to the automatic nature of this process, the simplicity of the integrity constraints defined at the PIM level has a direct impact on the efficiency of the resulting implementation. As we have shown in [2], our techniques can be used to increase the efficiency of the final implementation by generating equivalent but more efficient constraints than the original ones written by the designer.

Moreover, it may happen that the final technology platform chosen does not offer a predefined mechanism to directly implement integrity constraints (as it turns out in object-oriented programming languages). In this case the constraints must be implemented by means of alternative constructs. A natural way to do it is to include integrity constraint checking in the contracts of the system operations that may cause its violation. For instance, in our example, an operation RaiseSalary defined in the entity type Employee must check that the new salary does not violate the MaxSalary constraint. To be able to easily include this check in the contract of RaiseSalary, the constraint should be defined in terms of the employee instances and, thus, one could use our transformation techniques to convert MaxSalary from its original version (using Department as a context type) to a new version defined using Employee as a context type.

As far as validation is concerned, our techniques could be used to detect redundancies among a given set of integrity constraints. For instance, we could determine that two or more constraints are equivalent by means of redefining them over the same context type, processing their body with the rules of Section 2 in the left–right direction and, finally, comparing the resulting expressions. In some cases, a simple look at the expression finally obtained could be enough to determine whether some constraints are equivalent like it happens, for instance, with the three different versions of MinSalary shown in the introduction. When this does not happen, our transformation may help existing model checkers to provide better results.

In addition to the previous significant scenarios, we also see a couple of minor situations in which our transformation techniques can be helpful. First, we could use them to assist designers in the definition of integrity constraints by providing them with alternative ways of specifying each integrity constraint and letting them chose the one they prefer. The same ideas would also facilitate students learning of the OCL. Second, by using the rules of Section 2 in the left–right direction we get equivalent expressions that use only a subset of the OCL. Then, tools that implement transformations defined using OCL or OCL-like languages (as in the QVT standard [21]) or tools that translate OCL expressions to Java code (see [3] for a survey) would not need to address the full expressivity of the OCL.

6. Tool implementation

We have implemented a prototype tool [5] for the transformation techniques presented in this paper. Given an XMI file [20] representing a CS and a set of OCL integrity constraints in textual form (parsed using the Dresden OCL
toolkit [7]), our tool generates all possible context changes for those constraints. The generated constraints are shown to the designer and may be stored, if required, in an output text file.

As a first step, the input constraints are preprocessed by means of applying the equivalence rules of Section 2 in the left to right direction. Since each rule has been implemented in a separate Java class, our tool could be easily extended with the inclusion or the removal of new equivalence rules according to the designers’ interest.

Then, and according to the input CS, the graph representing the CS is created and all possible paths are computed. As an example, we show in Fig. 10 the results of processing our running example with the tool.

Finally, the user may select some (or all) the constraints in order to generate their alternative representations following the previous paths and the taxonomic relationships. These alternative representations are simplified (with the equivalences of Section 3.1.3) and shown to the user. Along with the final constraints, the tool also provides information about the path and the rules applied to obtain them (see Fig. 11).

7. Conclusions and further work

We have proposed several transformation techniques that allow obtaining a set of semantically equivalent representations for a given OCL constraint. The techniques consider both changes in the body of the constraint 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 proposal able to generate all alternative representations of a given integrity constraint in terms of different context types.

Although we have focused on integrity constraints, most of the equivalences of Section 2 are useful for any kind of OCL expressions while the context changes of Section 3 are partially applicable to pre and postcondition expressions (which, in fact, are represented as stereotyped constraints in the UML metamodel) as well.

The main part of our proposal is formalized as a path problem over a graph representing the conceptual schema. The graph is created such that every path between two vertices corresponds to a different way 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 are able to compute the different alternative representations of a given OCL constraint.

The proposed techniques contribute to software development in two different ways. First, at the modeling level, these techniques are helpful to assist the designer during the definition of an appropriate constraint representation for
each kind of model. Second, during code generation, our techniques are necessary when managing the constraints from first level models to their implementation.

Further research may involve looking for additional useful equivalences that may improve further the results of our techniques (for instance, integrating some of the equivalences presented in [9]). Moreover, we would like to formally prove the correctness of the proposed transformations. This could be done, for instance, by means of adapting some existing theorem prover (such as Hol-OCL [1]) to deal with our generic transformations. Finally, we aim at defining a set of complexity models that, depending on the designers’ goal, allow to (semi)automatically obtain the best representation of a given constraint.

Acknowledgements

We would like to thank the anonymous referees and the people of the GMC group (J. Conesa, D. Costal, C. Gómez, A. Olivé, A. Queralt, R. Raventós and M. R. Sancho) for their many useful comments on previous versions of this paper. This work was partially supported by the Ministerio de Ciencia y Tecnología under project TIN2005-06053.

References