Synthesis of OCL Pre-Conditions for Graph Transformation Rules

Jordi Cabot[1], Robert Clarisó[2], Esther Guerra[3], and Juan de Lara[4]

[1] INRIA - École des Mines de Nantes (France), jordi.cabot@inria.fr
[2] Universitat Oberta de Catalunya (Spain), rclariso@uoc.edu
[3] Universidad Carlos III de Madrid (Spain), eguerra@inf.uc3m.es
[4] Universidad Autónoma de Madrid (Spain), Juan.deLara@uam.es

Abstract. Graph transformation (GT) is being increasingly used in Model Driven Engineering (MDE) to describe in-place transformations like animations and refactorings. For its practical use, rules are often complemented with OCL application conditions. The advancement of rule post-conditions into pre-conditions is a well-known problem in GT, but current techniques do not consider OCL. In this paper we provide an approach to advance post-conditions with arbitrary OCL expressions into pre-conditions. This presents benefits for the practical use of GT in MDE, as it allows: (i) to automatically derive pre-conditions from the meta-model integrity constraints, ensuring rule correctness, (ii) to derive pre-conditions from graph constraints with OCL expressions and (iii) to check applicability of rule sequences with OCL conditions.

1 Introduction

The advent of Model Driven Engineering (MDE) has made evident the need for techniques to manipulate models. Common model manipulations include model-to-model transformations, as well as in-place transformations like refactorings, animations and optimisations. Many transformation languages and approaches have been proposed for in-place transformations, but much research is directed towards usable languages (e.g. able to take advantage of the concrete syntax of the language) providing good integration with MDE standards (e.g. UML, MOF, OCL) and that support some kind of analysis.

Graph transformation (GT) [7] is a graphical and declarative way to express graph manipulations with a rich body of theoretical results developed over the last decades. Its graphical nature has made it a common choice to define in-place manipulations for Domain Specific Visual Languages (DSVLs), taking the advantage that one can use the concrete syntax of the DSVL, making rules intuitive. However, further integration with technologies like OCL is still needed for its practical use in MDE. In this paper, we aim to advance in this direction.

In particular, a recurring problem is the interaction of the applicability conditions of GT rules with the OCL invariants defined in the meta-models. In this way, one may consider that rules should be restricted enough in their applicability conditions to ensure that their application does not violate any meta-model

constraint. If this is not the case, the system would fall in an inconsistent state and hence lead to an incorrect simulation or refactoring. However, it is tedious and error prone for the grammar designer to derive by hand from the meta-model invariants all application conditions required for each rule. Our goal is to automate such task. Hence, given an invariant I that a model M must satisfy after the application of a rule r, we generate the weakest invariant I' s.t. if the model satisfies it before applying r, then the resulting model will satisfy I. This core technique has many applications, like deriving rule pre-conditions from meta-model constraints and from graph constraints [7] equipped with OCL, as well as to test the applicability of rule sequences with arbitrary OCL conditions.
Paper organization. Section 2 presents motivating examples that benefit from our techniques. Section 3 introduces our techniques to advance OCL invariants to rule pre-conditions. Section 4 goes back to the examples, illustrating the advancement of OCL invariants for them. Section 5 comments possible optimisations for the method. Section 6 discusses related research and Section 7 concludes.

2 Motivating Examples

First we consider a DSVL for defining production systems. Its meta-model is shown to the left of Fig. 1. The meta-model defines machines with input and output conveyors, which may contain two different kinds of pieces. An OCL constraint ensures that conveyors do not hold more pieces than their capacity. The center of the same figure shows a model in abstract and concrete syntax.

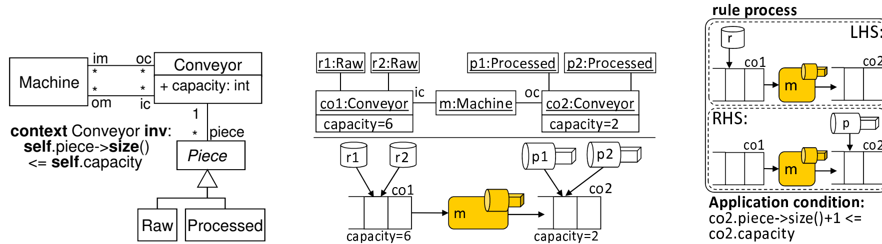

Fig. 1. Meta-model (left). Models in abstract/concrete syntax (center). A rule (right).

Once we have defined the DSVL syntax, we can use GT rules to define its semantics. The rule to the right of Fig. 1 describes how machines behave, consuming and producing pieces. GT rules [7] are made of two graphs, the left and the right hand sides (LHS/RHS), which encode the pre- and post-conditions for rule application. Intuitively, a rule can be applied to a model whenever an occurrence of the LHS pattern is found in it. Roughly, the application of a rule consists in deleting the elements of $LHS - RHS$, and creating those of $RHS - LHS$. In our case, the rule application deletes the `Raw` piece and creates a `Processed` one in the output conveyor.

Considered in isolation, the rule could create a piece in a full output conveyor, violating the OCL integrity constraint of the meta-model. Hence, most approaches and tools require the inclusion of an *application condition* in the rule's LHS constraining the application of the rule to the case when the output conveyor has enough capacity (so that the resulting model is consistent with the meta-model). This is what we have done in the figure. However, this is a redundant work, as we are specifying a constraint for the same purpose *twice*: once in the meta-model and another time in the rule. Even worse, the designer has the burden of calculating an application condition that, given the rule's actions, forbids applying the rule if the execution breaks any meta-model constraint.

Hence, one important improvement in current practice is a mechanism to automatically derive OCL application conditions from the meta-model, in such a way that any possible rule application cannot break the meta-model constraints. This presents several advantages: (i) it notably reduces the work of the designer, (ii) it facilitates grammar and meta-model evolution, as a change in the constraints of the latter has less impact on the rules, as many application conditions are automatically derived (iii) it eliminates the risk of not adding appropriate conditions that would cause rule applications to violate the meta-model constraints, and (iv) it eliminates the risk of adding a too restrictive condition that would forbid applying the rule, even when its application would not break any constraint (i.e. a condition that is not the weakest). In fact, the OCL condition added to the rule in Fig. 1 is not the weakest. We solve this issue in Section 4.

The second motivating scenario is a refactoring of class diagrams. We assume the simple meta-model in Fig. 2, where classes contain properties, each of a given data type. For the sake of illustration, an invariant forbids attribute overriding, by disallowing classes to have properties with same name as properties in direct children (note that a real scenario would check this on indirect children too).

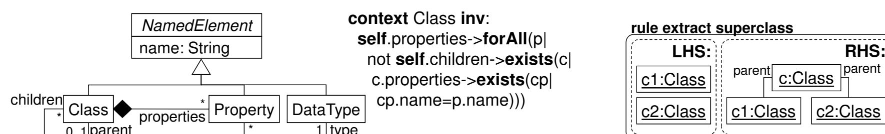

Fig. 2. Simple meta-model for class diagrams (left). Refactoring rule (right).

Fig. 2 (right) depicts rule *extract superclass* that, given two classes, creates a common superclass. As in the previous scenario, we would not like to calculate by hand the extra conditions needed for this rule to be applicable. In this case, the problematic situation is with the "0..1" cardinality constraint in the meta-model that forbids a class to have several parents. The following sections will show that our method treats such cardinality constraints uniformly as OCL constraints [9].

Fig. 3 shows another refactoring rule, *pull-up property*, which given two sister classes having a property with same name and type, pulls up such property to a common parent class. Again, the problem is to calculate the needed applicability

conditions, where the problematic part is the meta-model invariant, as the parent class may have other children containing a property with same name. Finally, it is common to apply two or more refactorings in sequence, e.g. we may only want to extract a common superclass if we can then apply the pull-up property refactoring. Hence, we need to calculate the applicability conditions of a sequence of rules. Neglecting OCL conditions, this can be done by calculating the concurrent rule [7], which includes the pre-conditions and effects of both rules, and is built by gluing the initial rules through common elements. In our case, the glueing is done through elements `c1`, `c2` and `c`, and the resulting rule is shown to the right of Fig. 3. In this scenario we need our method to calculate the OCL application conditions of the concurrent rule. First, the application conditions of the second rule should be advanced to the first, and additional pre-conditions for the resulting rule should be derived from the meta-model constraints.

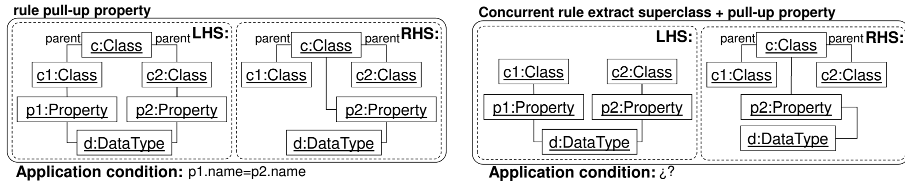

Fig. 3. Refactoring rule (left). Concurrent rule for both refactorings (right).

Altogether, we have seen that the use of GT rules in MDE necessitates techniques to manipulate OCL conditions. The next section presents our method to advance OCL conditions, and then Section 4 will look back again at these examples to explain how our method solves the problems we have pointed out.

3 Translating OCL Post-conditions into Pre-conditions

This section describes how to advance constraints from the RHS of a rule into the LHS. The input of the procedure is a GT rule and an OCL constraint restricting the graph after applying the rule. This constraint can either be (1) an OCL invariant or (2) an arbitrary OCL boolean expression with explicit references to the objects of the RHS using their identifier. We will refer to this expression as the *post-condition*, even though it may not attempt to describe the effects of rule application (e.g. it could be an integrity constraint that should be preserved).

The output is an OCL boolean expression which constrains the graph before applying the rule. We refer to this new expression as the *pre-condition*, and we compute it performing several replacements on the OCL constraint being advanced, which depend on the actions performed by the rule. This computation should ensure that in any match where the rule is applicable, the pre-condition is satisfied before applying the rule iff the post-condition is satisfied after applying the rule. Formally, in any application $G \Rightarrow_r G'$ of the rule r on a graph

G producing a graph G', $Pre(G) \leftrightarrow Post(G')$ should hold. This is similar to Dijkstra's notion of *weakest pre-condition*[5].

3.1 Overview

The computation of the weakest pre-condition proceeds in three steps: (1) static analysis of the rule, (2) preprocessing of the post-condition and (3) the final computation of the weakest pre-condition.

Static analysis of the rule: First we examine the rule to identify the list of *atomic graph updates* that occur when the rule is applied. This step is independent on the OCL post-condition being advanced, as we only need to compare the LHS and the RHS. Then, the following atomic graph operations can be identified: (1) deletion/creation of a link between two existing objects; (2) deletion of an object (and all its adjacent links); (3) update of an attribute value; and (4) creation of a new object (plus its adjacent links and attribute initialization).

As an example, the rule of Fig. 1 performs two atomic graph updates: deletion of object `r` and its link `co1-r`; and creation of object `p` and its link `co2-p`.

Preprocessing the OCL post-condition: This step simplifies the OCL post-condition before converting it into a pre-condition. First, if the post-condition is an OCL invariant defined in a context type T, it is implicitly universally quantified over all objects of type T. In order to advance this constraint, it is necessary to make the quantification explicit by performing the following replacement:

context T **inv**: exp $\Rightarrow$ T::allInstances()−>forAll(v | exp')

where exp' is computed by replacing all references to the keyword $self$ in exp with the variable v. After this expansion of invariants, we apply several transformations that simplify the constraint. For example, `if-then-else` expressions nested into other OCL constructs can be expanded as:

(**if** A **then** B **else** C **endif**) op D $\Rightarrow$ **if** A **then** (B op D) **else** (C op D) **endif**

This latter step is also performed whenever a replacement takes place.

Computing the weakest pre-condition: Finally, we transform the post-condition into a pre-condition, by applying a set of *textual replacement patterns* on the OCL post-condition. We have defined a collection of replacement patterns for each atomic graph update. These patterns capture the effect of an atomic graph update and modify the constraint accordingly, synthesizing the equivalent constraint before applying that update. Applying the replacement patterns for all the graph updates the rule performs yields the corresponding pre-condition.

There are two necessary requirements for the replacement we have defined:

– The replacement patterns for each atomic graph update should capture all OCL subexpressions which are relevant to that update. For example, if the update is the deletion of a link, we should consider all OCL navigation expressions that may potentially traverse this link.

– If an OCL constraint is well-formed before applying a replacement pattern, then the result of applying the replacement pattern should also be well-formed. By "well-formed", we refer to syntactical and semantical correctness according to the OCL specification, e.g. correct types for each subexpression.

3.2 Basic replacement patterns

In this section, we focus on the replacement patterns for all atomic graph updates *except* the creation of new nodes (discussed in Subsection 3.3). The application of these replacement patterns works on the *abstract syntax tree* (AST) of the OCL post-condition. The leaves of this tree are constants (e.g. 1, 2, "hello"), variable names or type names. Each internal node is an operator (logic, arithmetic or relational) or an OCL construct (quantifier, operation call, etc).

In order to transform the post-condition for a single atomic update, we perform a bottom-up traversal of the AST: starting from the leaves and looking for matches of the replacement patterns defined for that update. Whenever a match is located in the AST, it is replaced according to the pattern and the traversal continues upwards, until the root of the AST is reached.

In order to identify the OCL subexpressions which are relevant to an atomic graph update, two items are considered: (1) the *operator* involved in the OCL expression and (2) the *type* of its operands. For example, updates dealing with the assignment of a new value to an attribute will affect expressions where the value of this attribute is being accessed. Regarding types, we will use the following notation: $T = T'$ if type T is equal to T', $T \sqsubset T'$ if T is a subtype of T', and $T \sqsubseteq T'$ if T is a subtype or is equal to T'.

Object deletion. Let us consider the deletion of an object x of type T. In order to advance the post-condition, the constraint should be modified to ensure that its evaluation does not take x into account, i.e., it is as if x did not exist.

In OCL, an object x can only be accessed from a collection of objects of its type T, its subtypes or supertypes. This access may use a quantifier (forAll, exists, . . .) or an operation on collections (first, size, . . .). This collection can be computed either from a "Type::allInstances()" expression or from a navigation expression. Therefore, we simulate the deletion by appending "excluding(x)" to any "allInstances()" expression or navigation expression of an appropriate type. Table 1 depicts the complete set of replacement patterns.

Notice that, due to how navigation expressions are being modified, these replacement patterns are implicitly encoding the deletion of all edges where x may participate. Therefore, replacement patterns for link deletion only need to be applied if both objects are preserved in the RHS.

Example 1. In a rule deleting an object x of type T, the post-condition:

```
T::allInstances()−>exists(t | t.isGreen)
```

is advanced to a pre-condition demanding some object *other than* x to be green thus ensuring that the rule is only applied when x is not the only green object:

```
T::allInstances()−>excluding(x)−>exists(t | t.isGreen)
```

LHS: x:T
RHS:

Ref	Pattern	Conditions	Replacement
OD1	A::allInstances()	$T \sqsubseteq A$ or $A \sqsubset T$	A::allInstances()−>excluding(x)
OD2	exp.role	"role" is an association end of type A, with $T \sqsubseteq A$ or $A \sqsubset T$	exp.role−>excluding(x)

Table 1. Replacement patterns for object deletion.

Attribute updates. Let us consider the update of attribute *attr* of an object x of type T, such that the new value is given by an OCL expression *new_val_exp*. In OCL, the value of an attribute can only be accessed through an expression of type AttributeCallExp, e.g. "object.attribute". Intuitively, to advance any post-condition, it is necessary that every time we refer to the attribute *attr* of an object of type T, we use *new_val_exp* instead, but only if we are referring to x. In Table 2, we present the replacement patterns that implement this concept.

LHS: x:T
RHS: x:T
Attribute computation:
x.attr' = new_value_exp

Ref	Pattern	Conditions	Replacement
At1	x.attr	None	new_val_exp
At2	exp.attr	Type(exp) $\sqsubseteq$ T	**if** exp = x **then** new_val_exp **else** exp.attr **endif**

Table 2. Replacement patterns for an update of an attribute.

Link deletion. Let us consider the deletion of a link of association As between objects a (of type T_A) and b (of type T_B). We only consider scenarios where neither a nor b are deleted by this rule, because if any of them is deleted, the situation is already captured by the replacement patterns for object deletion.

Hence, we only need to modify navigation expressions traversing association As, so that they do not take the link $a - b$ into account. This can be implemented by appending "excluding(a)" to navigations going from T_B to T_A and "excluding(b)" to navigations going from T_A to T_B, as described in Table 3.

Example 2. In a rule deleting a link a-b, the post-condition:

T_A::allInstances()−>exists(x | x.rb−>notEmpty())

states that at least one T_A object is connected to a T_B object. Advancing the invariant considers a a special case, as it may be connected to b in the LHS:

T_A::allInstances()−>exists(x | (**if** x = a **then** x.rb−>excluding(b) **else** x.rb **endif**)−>notEmpty())

Link creation. Finally, we consider the creation of a link of an association As between objects a (of type T_A) and b (of type T_B). We assume that both a and b exist in the LHS, as the creation of objects is discussed in Section 3.3. We have to simulate the existence of an edge $a - b$ in navigation expressions that traverse association As. This is done by appending "including(b)" to navigations going from T_A to T_B, or "including(a)" to expressions going from T_B to T_A.

LHS: a:TA —ra rb— b:TB
RHS: a:TA b:TB

Ref	Pat.	Conditions	Replacement
LD1a	exp.rb	Type(exp) $\sqsubseteq$ $\mathsf{T_A}$	**if** exp = a **then** exp.rb−>excluding(b) **else** exp.rb **endif**
LD2a	exp.rb	Type(exp) = Set(T′) with $\mathsf{T'} \sqsubseteq \mathsf{T_A}$	(exp−>excluding(a).rb)−> union(a.rb−>excluding(b))

Table 3. Replacement patterns for link deletion, for navigations $\mathsf{T_A} \to \mathsf{T_B}$ (the symmetric patterns LD1b and LD2b for navigations $\mathsf{T_B} \to \mathsf{T_A}$ are omitted for brevity).

Example 3. In a rule adding a link a-b, the following post-condition:

$\mathsf{T_A}$::allInstances()−>forAll(x | x.rb−>size() $\neq$ 5)

states that no object of type $\mathsf{T_A}$ can be connected to exactly 5 T_B objects. It would be advanced as follows, by treating object a in a distinct way:

$\mathsf{T_A}$::allInstances()−>forAll(x |
(**if** x = a **then** x.rb−>including(b) **else** x.rb **endif**)−>size() $\neq$ 5)

LHS: a:TA b:TB
RHS: a:TA —ra rb— b:TB

Ref	Pattern	Conditions	Replacement
LC1a	exp.rb	Type(exp) $\sqsubseteq$ $\mathsf{T_A}$	**if** exp = a **then** exp.rb−>including(b) **else** exp.rb **endif**
LC2a	exp.rb	Type(exp) = Set(T′) with $\mathsf{T'} \sqsubseteq \mathsf{T_A}$	**if** exp−>includes(a) **then** exp.rb−>including(b) **else** exp.rb **endif**

Table 4. Replacement patterns for link creation, for navigations $\mathsf{T_A} \to \mathsf{T_B}$ (the symmetric patterns LC1b and LC2b for navigations $\mathsf{T_B} \to \mathsf{T_A}$ are omitted for brevity).

3.3 Replacement patterns for creating objects

New objects create a challenge, because there is no placeholder to designate them in the LHS. For example, a constraint like the following:

Conveyor::allInstances()−>forAll(x | x.capacity $\geq$ 0)

restricts all objects of type Conveyor. If a new Conveyor c is created by the rule, it should also satisfy this constraint. However, as new objects do not exist in the LHS, we cannot refer to them using an identifier. Thus, the expression:

Conveyor::allInstances()−>including(c) −>forAll(x | x.capacity $\geq$ 0)

is an invalid pre-condition, as identifier c is meaningless before rule application.

As a result, the transformation for advancing post-conditions becomes more complex in rules that create objects. Hence, we have to split it in two steps:

– In the first step, described in Table 5, we modify "allInstances()" and navigation expressions to take into account the newly created object. This transformation introduces references to the identifier of the new object that need to be removed in the next step. These direct references (and also those appearing previously in the post-condition) may be located in two types of expressions: collections including the new object and object expressions.

LHS: a:TA

RHS: a:TA ra rb b:TB

Ref	Pattern	Conditions	Replacement
OC1	T::allInstances()	$T_B \sqsubseteq T$	T::allInstances()−>including(b)
OC2	exp.rb	Type(exp) $\sqsubseteq T_A$	**if** exp = a **then** exp.rb −>including(b) **else** exp.rb **endif**
OC3	exp.rb	Type(exp) = Set(T'), with $T' \sqsubseteq T_A$	**if** exp−>includes(a) **then** exp.rb −>including(b) **else** exp.rb **endif**
OC4	b		See Tables 6 and 7.

Table 5. Replacement patterns for object creation.

– The second step removes direct references to the new object by a set of replacements that either (i) move the reference upwards in the AST of the OCL expression, (ii) particularize OCL quantifiers that affect the new object, or (iii) rewrite the expression to avoid the reference. The iterative application of those replacements yields an equivalent expression without direct references.

The remainder of this section focuses on the second step. Tables 6 and 7 describe the replacements for collection expressions and object expressions, respectively. Due to space constraints, the collection of patterns is incomplete.

Collection expressions can be classified into three categories: simple queries (C1-3), iterators (C4-10) and operations involving objects or other collections (C11-19). For example, C2 indicates that the query “isEmpty()” can be replaced by “false” when it is applied to a collection containing the new object.

The transformation of iterators combines the evaluation of the expression on the new object and on the remaining elements of the collection. We denote by $Inst[var, exp]$ the replacement of all references to variable *var* with the identifier of the new object in the RHS. Then, a pattern like C4 for the existential quantifier establishes that either the old elements of the collection *or* the new object satisfy the condition. Applying *Inst* introduces references to the new object, which are, again, further simplified using the patterns from Tables 6 and 7.

Finally, collection expressions with subexpressions of type object or collection are *synchronisation points*: the replacement to be applied depends on whether the other subexpression also contains the new object. For example, if the object b is created, an expression like “b = b” is replaced by “true”. However, we need to process both branches of the equality before reaching this conclusion. Hence, replacements should be applied bottom-up in the AST, stopping at synchronisation points until all sibling subexpressions in the AST have been processed.

Object expressions, described in Table 7, are defined similarly. For example, pattern O1 describes an attribute access, simply replaced by the attribute computation expression in the RHS. Again, object expressions which operate with other objects or collections are synchronisation points.

Example 4. Continuing with the previous example, the expression with references to “including(c)” can be transformed into the following (pattern C5, forAll):

Conveyor::allInstances()−>forAll(x | x.capacity $\geq$ 0) **and** (c.capacity $\geq$ 0)

This pattern has particularised the quantifier “forAll” for object c. Now pat-

tern O1 replaces "c.capacity" by the value given to this attribute in the RHS, removing the last reference to the object. For example, if the RHS includes an attribute computation like: c.x' = 10, the final pre-condition would be:

Conveyor::allInstances()−>forAll(x | x.capacity $\geq$ 0) **and** (10 $\geq$ 0)

Ref	Pattern	Replacement
C1	COL_B−>size()	col−>size() + 1
C2	COL_B−>isEmpty()	false
C3	COL_B−>notEmpty()	true
C4	COL_B−>exists(x | exp)	col−>exists(x | exp) **or** $Inst[x, exp]$
C5	COL_B−>forAll(x | exp)	col−>forAll(x | exp) **and** $Inst[x, exp]$
C6	COL_B−>collect(x | exp)	col−>collect(x | exp)−>including($Inst[x, exp]$)
C7	COL_B−>one(x | exp)	(col−>one(x | exp) **and not** $Inst[x, exp]$) **or** (**not** col−>exists(x | exp) **and** $Inst[x, exp]$)
C8	COL_B−>isUnique(x | exp)	col−>isUnique(x | exp) **and** col−>select(x | exp = $Inst[x, exp]$)−>isEmpty()
C9	COL_B−>any(x | exp)	**if** col−>exists(x | exp) **then** col−>any(x | exp) **else** b **endif**
C10	COL_B−>select(x | exp)	**if** $Inst[x, exp]$ **then** col−>select(x | exp) −>including(b) **else** col−>select(x | exp) **endif**
C11	COL_B−>count(exp)	1 (if exp = b) col−>count(exp) (otherwise)
C12	COL_B−>includes(exp)	true (if exp = b) col −>includes(exp) (otherwise)
C13	COL_B−>excludes(exp)	false (if exp = b) col−>excludes(exp) (otherwise)
C14	COL_B−>including(exp)	col −>including(b) (if exp = b) col−>including(exp)−>including(b) (otherwise)
C15	COL_B−>excluding(exp)	col (if exp = b) col−>excluding(exp) −>including(b) (otherwise)
C16	COL_B−>includesAll(exp)	col−>includesAll(col') (if exp = col'−>including(b)) col−>includesAll(exp) (otherwise)
C17	exp−>includesAll(COL_B)	false (if exp $\neq$ col'−>including(b)) col'−>includesAll(col) (otherwise)
C18	COL_B= exp or exp = COL_B	col = col' (if exp = col'−>including(b)) false (otherwise)
C19	$\text{COL}_\text{B}$$\neq$ exp or exp $\neq$ COL_B	col $\neq$ col'(if exp = col'−>including(b)) true (otherwise)

Table 6. Replacement patterns for collection expressions, where b is the identifier of the new object, *col* and *col'* are collection expressions, *exp* is an arbitrary expression, and COL_B is a shorthand for *col*−> *including*(b).

3.4 Putting everything together

Given a list of atomic graph updates corresponding to a rule, advancing a post-condition consists on applying the replacement patterns for each update in sequence. The order of application of the replacements is irrelevant for two reasons.

Ref	Pattern	Replacement
O1	b.attrib	attribute_condition(attrib)
O2	b.role	Set{$\mathsf{a}_1, \ldots \mathsf{a}_N$}, where $\mathsf{a}_1, \ldots \mathsf{a}_N$ are the identifiers of the objects linked to x through "role" in the RHS
O3	b.oclIsTypeOf(A)	true if T = A; false otherwise
O4	b.oclIsKindOf(A)	true if $\mathsf{T} \sqsubseteq$ A; false otherwise
O5	b.allInstances()	T::allInstances()−>including(b)
O6	exp−>count(b)	1 (if exp = col−>including(b)) 0 (otherwise)
O7	exp−>includes(b)	true (if exp = col−>including(b)) false (otherwise)
O8	exp−>excludes(b)	false (if exp = col−>including(b)) true (otherwise)
O9	exp−>excluding(b)	col (if exp = col−>including(b)) exp (otherwise)
O10	b = exp or exp = b	true if b = exp; false otherwise
O11	b ≠ exp or exp ≠ b	false if b = exp; true otherwise
O12	Set{ exp1, . . . , b, . . . , exp2}	Set{ exp1, . . . , exp2}−>including(b)

Table 7. Replacement patterns for object expressions, where *b* is the identifier of the new object and *exp*, *exp1* and *exp2* are arbitrary expressions.

First, each intermediate replacement produces a valid OCL expression. Second and most important, there is no overlap between updates: link creation/deletion updates are applied only when no object is being created/deleted, otherwise we use the object creation/deletion patterns.

Finally, we provide some remarks on the *correctness* and *completeness* of the method. Our method ensures that the resulting expression is well-formed, because each pattern ensures type consistency by replacing each matching expression with another one of a compatible type. Regarding the equivalence of pre- and post-conditions, a complete proof is out of the scope of this paper.

The method supports most of the OCL but the following features from the OCL 2.0 specification are unsupported:

- Calls to recursive query operations.
- OCL collections other than Set (i.e. Bag, Sequence, OrderedSet) and their operations (i.e. first, last, append, prepend, insertAt, sortedBy, at, indexOf, subSequence, subOrderedSet, asBag/Sequence/OrderedSet)
- The type Tuple and operations involving tuples, e.g. the cartesian product.

4 Back to the Examples

This section illustrates the transformation method using the examples we presented in Section 2. Firstly, we consider rule *process* in Fig. 1.

1. First we pre-process the OCL invariants, rewriting them to a global scope:

   Conveyor::allInstances()−>forAll(v | $v.piece$−>$size() \leq v.capacity$) **and**
   Piece::allInstances()−>forAll(z | $z.conveyor$−>$size() = 1$)

   where the second clause, constraining pieces, is derived from the cardinality constraints on the association between conveyors and pieces.

2. Now we extract the atomic actions the rule performs. In our case, it deletes object `r` (together with link `co1-r`) and it creates object `p` (with link `co2-p`).
3. Next, we apply the replacement patterns. We start by the replacements due to object deletion, which in addition incorporate the deletion of all adjacent edges (hence deletion of both `r` and `co1-r`). The resulting expression is:

   Conveyor::allInstances()−>forAll(v |
     v.piece−>excluding(r)−>size() $\leq$ v.capacity) **and**
   Piece::allInstances()−>excluding(r)−>forAll(z | z.*conveyor*−>*size*() = 1)

   where we have applied patterns *OD2* and *OD1*.
4. In order to apply the patterns for the creation of object `p` and its adjacent link `co2-p`, we will consider both conditions in the conjunction separately. In the first condition, the expression v.piece matches the pattern OC2:

   Conveyor::allInstances()−>forAll(v |
     (**if** v.piece = co2 **then** v.piece −>including(p)
     else v.piece **endif**)−>excluding(r)−>size() $\leq$ v.capacity)

5. Before continuing, we can expand the conditional expression:

   Conveyor::allInstances()−>forAll(v |
     if v = co2 **then** v.piece−>including(p)−>excluding(r)−>size() $\leq$ v.capacity
     else v.piece−>excluding(r)−>size() $\leq$ v.capacity **endif**)

6. Now we can apply pattern C15 (excluding), and then pattern C1 (size):

   Conveyor::allInstances()−>forAll(v |
     if v = co2 **then** v.piece−>excluding(r)−>size() + 1 $\leq$ v.capacity
     else v.piece−>excluding(r)−>size() $\leq$ v.capacity **endif**)

   Notice that this result is more complex than the condition stated in Fig. 1 (right), because it considers the possibility of a *non-injective* matching, i.e. `co1 = co2`. In this case, there is no size problem as the rule creates *and* deletes the piece on the same conveyor. This is achieved implicitly by the conditional and the call to "excluding(r)": if `co1 = co2`, then "v.piece" contains piece `r` and it is removed by the call "excluding(r)"; otherwise, "v.piece" remains unaltered. This case was not considered by the designer of the rule in Fig. 1 as it is not obvious from the invariant and the LHS. As a result, condition in Fig. 1 was too restrictive and forbade the execution of the rule in a correct scenario. This example illustrates the benefits of automating the approach.
7. Finally, we apply the replacements for the second part of the post-condition:

   Piece::allInstances()−>excluding(r) −>forAll(z | z.conveyor−>size() = 1)

   ↓ Pattern OC1 (allInstances)

   Piece::allInstances()−>including(p) −>excluding(r)
     −>forAll(z | z.conveyor−>size() = 1)

   ↓ Pattern C15 (excluding)

   Piece::allInstances()−>excluding(r)−>including(p)
     −>forAll(z | z.conveyor−>size() = 1)

   ↓ Pattern C5 (forAll)

   Piece::allInstances()−>excluding(r) −>forAll(z | z.conveyor−>size() =1)
   and (p.conveyor−>size() = 1)

↓ Pattern O2 (navigation)

Piece::allInstances()−>excluding(r) −>forAll(z | z.conveyor−>size() = 1) **and** (Set{co1}−>size() = 1)

↓ Set{co1}−>size() = 1 : the second condition in the **and** is true

Piece::allInstances()−>excluding(r) −>forAll(z | z.conveyor−>size() = 1)

The complete pre-condition generated from our post-condition is therefore:

Conveyor::allInstances()−>forAll(v | **if** v = co2 **then** v.piece −>excluding(r)−>size() + 1 $\leq$ v.capacity **else** v.piece −>excluding(r)−>size() $\leq$ v.capacity **endif**) **and** Piece::allInstances()−>excluding(r)−>forAll(z | z.conveyor−>size() = 1)

5 Optimizations

When the post-condition *Post* being advanced is an integrity constraint, the problem becomes different to that of computing the weakest pre-condition: as *Post* is an invariant, it also holds before applying the rule. This can simplify the pre-condition being advanced, as it only needs to check the property *incrementally*, i.e. on the subgraph being modified by the rule. For instance, the conveyor capacity constraint could be reduced to:

*co*2.piece−>excluding(r)−>size() +1 $\leq$ *co*2.capacity

since adding a piece to a conveyor (in this case *co*2) can only violate the invariant on that conveyor. Therefore, it is useless to iterate through all existing conveyors when checking rule applicability. As before, the *excluding*(r) operation is added to handle the special case in which *co*1 and *co*2 are matched to the same object in the host graph.

Extending our approach with the synthesis of incremental constraints would follow the general algorithms for deriving incremental constraints for UML/OCL models presented in [3], which we do not show for space constraints.

6 Related Work

There are previous works on moving constraints from the RHS to the LHS of GT rules. The idea was first proposed in [12], where post-conditions for rules where derived from global invariants of the form $\forall P \exists Q$, where P and Q are graphs. The approach generated a set of post-conditions for each rule from such invariants, and then applied the rule "backwards" to obtain the desired set of pre-conditions. In [6] the approach was generalized to adhesive high-level replacement systems. Again, constraints are graphical graph patterns which can be universally or existentially quantified, or combined using boolean operators. These works were extended in [10] to deal with nested conditions of arbitrary depth. This family of conditions has the same expressive power as first-order graph formulas [11].

These approaches have two main limitations w.r.t. our new technique: (1) lack of expressivity in the post-condition expressions (e.g. OCL expressions such

as numerical constraints on attributes or cardinalities of collections are not supported) and (2) complexity of the advancement procedure (the procedure is described by categorical operations and needs an additional method to simplify redundant graph patterns as otherwise the graph constraints may become too large) that makes difficult their application in practice. In contrast, our technique is especially tailored to consider OCL expressions, and hence is very appropriate for its use in meta-modelling environments. Furthermore, the use of OCL allows the application of tools for the simulation, analysis and verification on UML/OCL models [1, 2, 8, 15]. Regarding the drawbacks of our proposal, it is not complete (some OCL constraints have no translation for rules that create objects) and we do not have a formal proof of its correctness yet.

The work of [16] translates a subset of OCL to graph constraints, which can be used to synthesize local pre-conditions for rules. However, the covered OCL subset is limited, and their techniques suffer the drawbacks of [6, 11, 12].

For transformations not defined as GT, the computation of the weakest precondition has also been studied, e.g. to analyse the composition of refactorings [13]. The notion of "backward descriptions" defined in [13] captures our replacement patterns of OCL expressions.

Regarding information systems, in [4] the authors study the generation of weakest pre-conditions for basic operations that perform a single change on the system state, e.g. instance creation or attribute update. Rather than studying the generation of weakest pre-conditions for arbitrary operations and constraints (as it is done in this paper), a fixed catalog of typical integrity constraints as well as patterns for determining the weakest pre-condition with respect to each kind of basic operation are defined. The same problem, advancing integrity constraints as pre-conditions, is studied in [14] for set-based invariants described in B. This family of constraints (constraints involving intersections, unions, differences and tests for membership) is a subset of those considered in this paper, e.g. cardinalities of sets are not supported.

7 Conclusions and Future Work

We have presented a technique to automatically synthesize application conditions for GT rules. Application conditions are derived from the rule post-conditions such that host graphs satisfying the applicability conditions will surely be consistent with all post-conditions at the end of any possible rule execution. Rule post-conditions may come from the rule itself or, for instance, from the well-formedness constraints defined in the meta-model. As a salient feature of our approach, post-conditions may include arbitrary OCL expressions and hence is a step towards the integration of GT and MDE.

As further work we would like to adapt this technique to other transformation languages (e.g. QVT and ATL), to combine it with techniques to advance graphical post-conditions and to reuse it as part of more complex scenarios like the automatic generation of concurrent rules. We also plan to study in more

detail the possible optimizations commented above to simplify and improve the efficiency of the generated conditions and to provide tool support.

Acknowledgements. Work funded by the Spanish Ministry of Science and Innovation through projects "Design and construction of a Conceptual Modeling Assistant" (TIN2008-00444/TIN - Grupo Consolidado), "METEORIC" (TIN2008-02081), mobility grants JC2009-00015 and PR2009-0019, and the R&D program of the Community of Madrid (S2009/TIC-1650, project "e-Madrid"). We would like to thank the anonymous referees and the members of the Conceptual Modeling Group (GMC) at UPC for their useful comments.

References

1. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. A UML/OCL framework for the analysis of graph transformation rules. *Soft. and Syst. Mod.*, To appear, 2010.
2. J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *MoDeVVa 2008. ICST Workshop*, pages 73–80, 2008.
3. J. Cabot and E. Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.
4. D. Costal, C. Gómez, A. Queralt, and E. Teniente. Drawing preconditions of operation contracts from conceptual schemas. In *CAiSE'2008*, volume 5074 of *LNCS*, pages 266–280. Springer, 2008.
5. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
6. H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae*, 74(1):135–166, 2006.
7. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, 2006.
8. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Soft. and Syst. Mod.*, 4(4):386–398, 2005.
9. M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. volume 2263 of *LNCS*, pages 85–114. Springer, 2002.
10. A. Habel and K.-H. Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 293–308, 2005.
11. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. Comp. Sci.*, 19(2):245–296, 2009.
12. R. Heckel and A. Wagner. Ensuring consistency of conditional graph rewriting - a constructive approach. *ENTCS*, 2, 1995.
13. G. Kniesel and H. Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.
14. A. Mammar, F. Gervais, and R. Laleau. Systematic identification of preconditions from set-based integrity constraints. In *INFORSID'2006*, pages 595–610, 2006.
15. A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In *ER'2006*, volume 4215 of *LNCS*, pages 497–512. Springer, 2006.
16. J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. Kuster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *ENTCS*, 211:159 – 170, 2008.