Backwards reasoning for model transformations: method and applications

Robert Clarisó[*,a], Jordi Cabot[a,b], Esther Guerra[c], Juan de Lara[c]

[a]*Universitat Oberta de Catalunya, Barcelona (Spain)*
[b]*ICREA (Spain)*
[c]*Universidad Autónoma de Madrid (Spain)*

Abstract

Model transformations are key elements of Model Driven Engineering. Current challenges for transformation languages include improving usability (i.e., succinct means to express the transformation intent) and devising powerful analysis methods.

In this paper, we show how backwards reasoning helps in both respects. The reasoning is based on a method that, given an OCL expression and a transformation rule, calculates a constraint that is satisfiable before the rule application if and only if the original OCL expression is satisfiable afterwards.

With this method we can improve the usability of the rule execution process by automatically deriving suitable application conditions for a rule (or rule sequence) to guarantee that applying that rule does not break any integrity constraint (e.g. meta-model constraints). When combined with model finders, this method facilitates the validation, verification, testing and diagnosis of transformations, and we show several applications for both in-place and exogenous transformations.

Key words: Model Transformation, Backwards reasoning, OCL, Weakest Pre-condition, Graph Transformation, Validation and Verification

*Corresponding author: Robert Clarisó, Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Rambla del Poblenou 156, 08018 Barcelona, Spain. phone: (+34)933263410 fax:(+34)933568822 e-mail: rclariso@uoc.edu

Email addresses: `rclariso@uoc.edu` (Robert Clarisó), `jordi.cabot@icrea.cat` (Jordi Cabot), `Esther.Guerra@uam.es` (Esther Guerra), `Juan.deLara@uam.es` (Juan de Lara)

1. Introduction

1.1. Overview

The advent of Model Driven Engineering (MDE) has prompted the need to manipulate models in an automated way. Common manipulations include model-to-model transformations (or exogenous, where typically input and target models in the transformation conform to different meta-models), as well as in-place transformations like refactorings, animations and optimisations. Many transformation languages and approaches have been proposed for both kinds of transformations, where research is mostly directed towards usable languages providing good integration with MDE standards (e.g. UML, MOF, OCL) and supporting some kind of analysis [1].

We propose to use backwards reasoning to achieve both goals. Backwards reasoning methods have been applied in different domains, for example to analyse logic programs [2], Petri nets [3] or timed automata [4]. The unifying idea is that, instead of starting with an initial system configuration and exploring possible reachable states, backwards reasoning assumes some (un)desirable target state and computes the corresponding source state(s). In this paper, we present a method that enables backwards reasoning for model transformations, and show its applications both to achieve a better usability of transformation languages and to provide increased analysis capabilities.

Many model transformation languages are based on rules [5, 6, 7] whose applicability is given by an object pattern complemented with a guard, typically given as an OCL expression. Our backwards reasoning is based on the automated calculation of such guards, given an OCL expression that the model is expected to fulfill after the rule application. Hence, given a constraint C that a model M must satisfy after the application of a rule r, the method generates the weakest constraint C'_r such that if the model satisfies it before applying r, then the resulting model is guaranteed to satisfy C.

The method is agnostic with respect to the particular model transformation language employed, and therefore applicable to many of them – like graph transformation (GT) [6], ATL [5] or ETL [7] – because it only requires the list of atomic actions performed by the rule. Moreover, it can be applied both to in-place and exogenous transformations.

1.2. Running example

As a running example, let us consider an in-place transformation to animate a Domain Specific Visual Language (DSVL) for production systems.

The meta-model for the language is shown to the left of Figure 1. It defines machines with input and output conveyors that can be interconnected. Conveyors may contain pieces, and an OCL constraint ensures that the number of pieces the conveyors actually hold does not exceed their capacity. The right of the same figure shows a model with one machine and two conveyors, in abstract (top) and concrete syntax (bottom). The left conveyor has two raw pieces, while the right one has two processed ones.

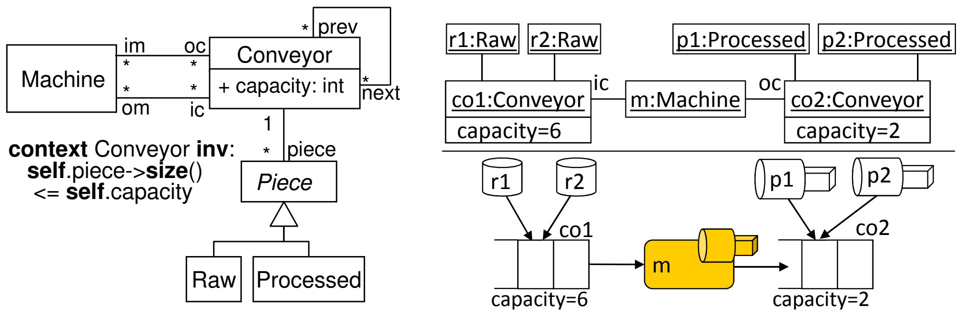

Figure 1: Meta-model (left). A model (right).

In this example, the semantics of the DSVL is defined using GT. In this approach, rules 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 its LHS is found in it. Then, applying the rule consists in deleting the elements of $LHS - RHS$, and creating those of $RHS - LHS$. In this way, the graphical part of the GT rule `process` on the left of Figure 2 describes how machines behave, consuming and producing pieces: the `Raw` piece is deleted and a `Processed` one is created in the output conveyor. Rule `move` in the same figure moves pieces of any kind (we use an "abstract object" labelled `r` of type `Piece`, which can get instantiated for both types `Raw` and `Processed`) between two conveyors.

1.3. Benefits of our backwards reasoning method

To improve the usability of transformation languages, each transformation rule should be consistent with the integrity constraints of the meta-model. Otherwise, users would be forced to use some kind of integrity checking mechanism that verifies that the output model is correct after every rule execution. Hence, the guard of each rule needs to ensure that, for every

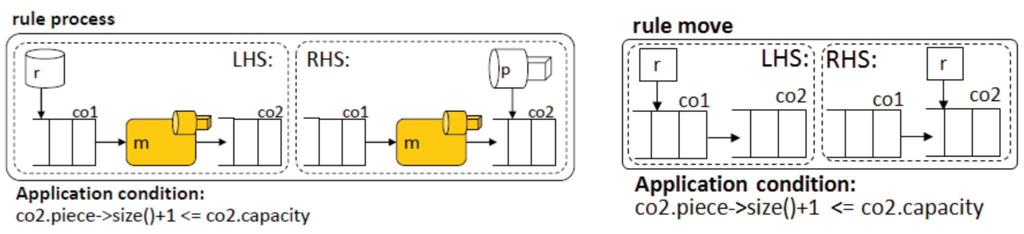

Figure 2: Two simulation rules.

possible model where the rule is applicable, the result after applying the rule satisfies all meta-model invariants (a property called *strong executability* in [8]). For instance, in the running example of Figure 1 the OCL integrity constraint in the meta-model forbids creating pieces in output conveyors that are already full. Below each rule, we provide an application condition that restricts the applicability of the rule to the cases where the output conveyor has enough capacity for the newly created piece.

Unfortunately, in current practice, the engineer has to encode a constraint for the same purpose *twice*: once in the meta-model, and another as the guard of each rule in the transformation to ensure that rule applications do not yield inconsistent models. 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 has any chance to break some meta-model constraint. Then, this work has to be repeated for every rule in the grammar, as done for example in rule `move` of Figure 2.

Instead, our method would derive the application condition for a rule starting from the OCL constraints of the meta-model. This presents several advantages from the point of view of the transformation developer: (i) it notably reduces his work, (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 can be 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 too restrictive conditions 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 of the rules in Figure 2 are not the weakest, as we will show in Section 4. Moreover, this has also a clear advantage at run-time, as tools do not need to implement a roll-back mechanism if some rule application leads to an inconsistent state, and do not even need to check the meta-model constraints at each intermediate state.

Furthermore, combined with techniques for model finding (e.g. [9]), our method enables the analysis of a plethora of correctness properties for the specified transformations. As we will see, several verification and testing procedures are easier to apply once the post-conditions have been advanced which facilitates a more homogeneous analysis and a better tool integration of those procedures with current modeling editors. Besides, we propose the new notion of *transformation diagnosis*, defined as the process of: (i) finding a problem in a transformation, (ii) explaining to the engineer what the problem is, and (iii) proposing some solution. Hence, for example, we are not only able to detect if a rule is not strongly executable, but can explain why (giving a model that makes the rule fail) and propose a solution (giving the weakest pre-condition that makes the rule strongly executable).

1.4. Contributions and structure of this paper

This paper continues the work in [10], where the method was developed and applied to generate rule pre-conditions given some meta-model constraints. In this paper, we make a systematic analysis on the applicability of the method, and show techniques for its application to both in-place and exogenous transformations. Our methods are agnostic with respect to the transformation language, and may use off-the-shelf model finders (like e.g., EMFtoCSP [9], UML2Alloy [11], or the USE Validator [12]) to conduct the analysis. While advancing post-conditions into pre-conditions is a well-known technique in graph transformation [13], it has not been generalized to handling OCL expressions. Moreover, the systematic analysis of backwards reasoning techniques, and the proposal of concrete methods to perform them in model transformation is also novel. Moreover, we provide correctness proofs of the different analysis methods proposed, and most notably we include a correctness proof of the post-condition advancement method.

The rest of the paper is organized as follows. Section 2 summarizes the method for advancing post-conditions into rule pre-conditions. Section 3 overviews some applications of the method for enhanced usability, and validation, verification and diagnosis. Next, Section 4 presents in more detail those applications for in-place and exogenous transformations. Section 5 briefly describes a generic implementation of the advancement procedure. Section 6 compares with related research and Section 7 concludes. An appendix includes the details of the correctness proof for the pre-condition synthesis.

2. Computing OCL Pre-Conditions for Transformation Rules

This section describes how to advance OCL post-conditions into pre-conditions. The method was originally described in [10], but for the sake self-containment, we include a description here as well.

2.1. Overview

Many model transformation systems are defined in terms of rules, which specify a collection of model updates (*actions*) that should be applied whenever the triggering condition for the rule (*enabling condition*) is met. This enabling condition depends on the formalism and may be very complex, e.g. a pattern that needs to be matched in the source model, an explicit invocation that needs to be made from another rule or a combination of several conditions. On the other hand, the actions performed by a rule can be described in an abstract way, independently of the underlying transformation language as a sequence of creation-deletion-update operations on basic model elements (attributes, objects and links). In our approach, we consider the following catalog of atomic actions: (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).

Figure 3 illustrates the overall procedure for advancing OCL post-conditions. The procedure receives two inputs: a *list of atomic actions* (e.g., derived from a transformation rule) and an OCL constraint. The latter is a boolean expression restricting the model after applying the actions. 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 model before applying the actions. We refer to this new expression as the *pre-condition*, and we compute it performing several replacements on the post-condition, which depend on the list of actions performed by the rule. Intuitively, we aim to compute a pre-condition which, evaluated *before* applying the rule, produces the same result as the post-condition *after* applying the rule. This is similar to Dijkstra's notion of *weakest pre-condition*[1] [14].

[1]Given an imperative statement S and a post-condition *Post*, compute the weakest pre-condition that guarantees that *Post* holds after the execution of S.

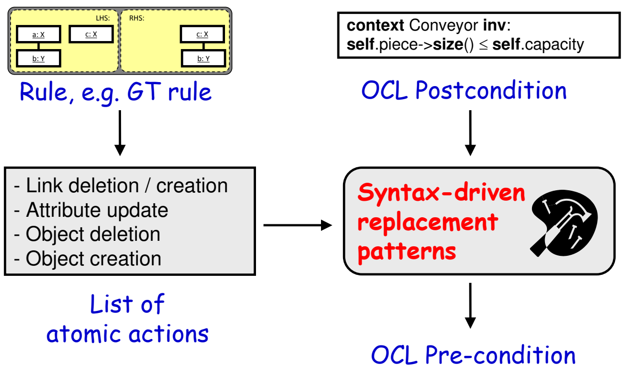

Figure 3: Overall flow for the advancement of OCL pre-conditions.

Hence, given a sequence of actions $s = \langle a_1, ..., a_n \rangle$ and a constraint *Post*, our method (called *adv*) can be formulated as follows:

$$adv(s = \langle a_1, ..., a_n \rangle, Post) = Pre$$

where *Pre* is a constraint s.t. $\forall M, M'$, with $M \overset{s}{\Rightarrow} M'$: $Pre(M) \leftrightarrow Post(M')$. We use the notation $M \overset{s}{\Rightarrow} M'$ to denote that model M is transformed into M' after applying the set of actions in s.

2.2. Advancing OCL Post-conditions

The advantage of abstracting transformation rules as a list of atomic actions is *genericity*: in this way, it is possible to define a procedure to advance preconditions which is independent of the formalism used to define the rule. The only requirement is a formalism-specific pre-processing step that analyzes rules to derive the list of actions.

For instance, in the case of GT, this step would compare the graphs patterns in the LHS and RHS of a rule to identify the changes. For example, rule `process` in Figure 2 performs two atomic graph updates: the deletion of object r and its link co1-r; and the creation of object p and its link co2-p. Rule move removes a link co1-r and creates a link co2-r. A similar approach can be followed for other transformation languages (e.g. see for the declarative part of ATL [15]).

The following section introduces the concept of *replacement pattern* that is the core of the procedure for advancing post-conditions.

2.2.1. Replacement patterns

The computation of the pre-condition from the post-condition [10] applies a set of *textual replacement patterns* on the OCL post-condition. 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.

Patterns are defined in terms of the *abstract syntax tree* (AST) of the OCL post-condition. In this tree, leaves are constants (e.g. 1, 2, "hello"), variable names or type names. Internal nodes of the tree are operators (logic, arithmetic or relational) or an OCL construct (quantifier, operation call, etc). A pattern is characterized by two elements: (1) a *matching criterion* to find relevant OCL subexpressions in the AST and (2) the *replacement OCL expression* used to substitute the match.

The matching criterion of each pattern considers two elements: the *operator* involved in the OCL expression and the *type* of its operands. For example, updates that assign a new value to an attribute named *attr* will only affect expressions that read the value of *attr*. 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'.

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.

Given a list of atomic 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. 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.

The following subsections 2.2.2 to 2.2.6 describe the specific list of replacement patterns for each type of atomic update and subsection 2.2.7 concludes by discussing the limitations of this procedure.

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

Table 1: Replacement patterns for object deletion.

2.2.2. 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., as if x did not exist. This goal can be achieved by excluding object x from any collection where it may belong, i.e. appending “excluding(x)” to any “allInstances()” expression or navigation expression of an appropriate type. Table 1 depicts the complete set of replacement patterns.

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 (avoiding this way the violation of the post-condition):*

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

2.2.3. Attribute updates

Let us consider the update of attribute *attr* (defined in class T) in an object x, 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.

Example 2. *Let us consider a rule that updates the value of attribute* n *of object* x *according to the attribute computation $x.n' = x.n + 1$. Then, the following post-condition:*

```
T.allInstances()−>isUnique(t | t.n)
```

stating that no pair of T objects can share the same value in attribute n *would be advanced as the following pre-condition:*

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.

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 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).

```
T.allInstances()−>isUnique(t | if t = x then x.n + 1 else t.n endif)
```

2.2.4. Link deletion

Let us consider the deletion of the link between objects a and b in an association As (defined between classes T_A and T_B). In OCL, links can only affect navigation expressions. 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 3. *In a rule deleting a link* a-b*, the post-condition:*

```
TA.allInstances()−>exists(x | x.rb−>notEmpty())
```

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

```
TA.allInstances()−>exists(x |
  (if x = a then x.rb−>excluding(b)
   else x.rb endif)−>notEmpty())
```

2.2.5. Link creation

Next, we consider the creation of a link between existing objects a and b in an association As (defined between classes T_A and T_B) . Here 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.

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	$Type(exp) = Set(\mathsf{T}')$, 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).

Example 4. *In a rule adding a link* a-b*, the following post-condition:*

```
TA.allInstances()−>forAll(x | x.rb−>size() ≠ 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:*

```
TA.allInstances()−>forAll(x |
  (if x = a then x.rb−>including(b)
   else x.rb endif)−>size() ≠ 5)
```

2.2.6. Object creation

New objects constitute 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 ≥ 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 ≥ 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 split it in two steps:

- In the first step, “allInstances()” and navigation expressions are modified to introduce an explicit reference to the newly created object. This reference will then be removed in the next step based on the type of expression it appears in. For navigation expressions, the new object reference is introduced using the same patterns seen in Table 4, e.g. for “T.allInstances()” the following pattern is applied:

  ```
  T.allInstances()−>including(b)
  ```

- 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 following example illustrates the replacement patterns for collection and object expressions.

Example 5. *Continuing with the previous scenario, the expression with references to "including(c)" can be transformed into the following (pattern C6, forAll):*

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

Note how this pattern has particularised the quantifier "forAll" for object c *by moving the reference to* c *outside the iterator (since the "forAll" states that all instances of Conveyor must satisfy the condition, this is the same as saying that all instances and* c *must satisfy it, which simplifies the expression). Now pattern 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 for the capacity attribute like: c.x' = 10 the final pre-condition would be:*

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

which basically says that creation of c *is always safe since it does not really add new preconditions once advancing it. Obviously this won't be always the case, e.g. the capacity value could depend on the value of other existing objects which would then need to be taken into account in the analysis of the precondition.*

The following tables 5 and 6 describe the list of replacements for object and collection expressions, respectively, participating in the advancement procedure for object creations. In addition to these patterns, the patterns for link creation should be used as well if the object being created participates in any new links.

Collection expressions can be classified into three categories:

- Simple queries (C1-4): these expressions consider the number of elements in a collection (C1) and whether the collection is empty (C2) or not (C3).

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	b.isOclUndefined()	false
O7	exp−>count(b)	1 (if exp = col−>including(b)); 0 (otherwise)
O8	exp−>includes(b)	true (if exp = col−>including(b)); false (otherwise)
O9	exp−>excludes(b)	false (if exp = col−>including(b)); true (otherwise)
O10	exp−>excluding(b)	col (if exp = col−>including(b)); exp (otherwise)
O11	b = exp or exp = b	true if b = exp; false otherwise
O12	b ≠ exp or exp ≠ b	false if b = exp; true otherwise
O13	**if** exp1 **then** b **else** exp2 **endif**	Expand conditional.
O14	**if** exp1 **then** exp2 **else** b **endif**	Expand conditional.
O15	Set{exp1, . . . , b, . . . , exp2}	Set{exp1, . . . , exp2}−>including(b)
O16	b.oclAsType(A)	b (if T $\sqsubseteq$ A) OclUndefined (otherwise)

Table 5: Replacement patterns for object expressions, where *b* is the identifier of the new object and *exp*, *exp1* and *exp2* are arbitrary expressions.

- Iterators (C5-13): these expressions deal with quantifiers over a collection. OCL quantifiers may compute a truth value, such as “exists” (C5), “forAll” (C6), “one” (C8), or “isUnique” (C9); an element of the collection, such as “any” (C10); a subset of the collection, such as “select” (C11) or “reject” (C12); or a new collection, as in “collect” (C7) or “collectNested” (C13).
- Set operations involving objects or other collections (C14-30): these operations deal mainly with testing the inclusion (C15, C19, C21) or exclusion (C16, C20, C22) of an element in the collection, adding (C17) or (C18) removing elements from the collection, performing set operations like intersection or union (C23-C27) and checking equality (C28, C29).

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. To this end, we denote by $Inst[var, exp]$ the replacement of all references to variable var with the identifier of the new object. Then, a pattern like C5 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 5 and 6.

Object expressions, described in Table 5, are defined similarly. For example, pattern O1 describes an attribute access, simply replaced by the attribute computation expression in the RHS.

2.2.7. Limitations

The method described in this paper supports most of the OCL but the following features from the OCL 2.4 specification are unsupported:

- Calls to recursive query operations.
- OCL collections other than Set (Bag, Sequence, OrderedSet) and their operations (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.

3. Overview of the Method Applicability

This section describes how post-condition advancement can be used to solve common problems in model transformations. This method can be used either to ease the definition of a transformation (by automatically generating pre-conditions from meta-model constraints) or for analysis (by validating or verifying properties of interest).

Figure 4 provides a high-level view of the use of post-condition advancement in combination with model finding techniques. The method can be seen as a black box with two inputs (a model transformation and a post-condition) and one output (a pre-condition). This black box can be used in combination with an *OCL model finder* [16], a tool able to compute models satisfying a set of input OCL expressions [9, 17, 11, 18, 12]. Using a model finder, it is possible to generate the *initial model* (the input of the transformation). The corresponding *final model* (the output of the transformation) can be later computed by executing the transformation.

This approach differs from previous works on the analysis of transformation systems [19, 20, 21, 22], which have explored the notions of *backwards reachability* (is a target model reachable?) and *backwards coverability* (is a

Ref	**Pattern**	**Replacement**
C1	COL_{B}−>size()	col−>size() + 1
C2	COL_{B}−>isEmpty()	false
C3	COL_{B}−>notEmpty()	true
C4	COL_{B}.isOclUndefined()	col.isOclUndefined()
C5	COL_{B}−>exists(x $\mid$ exp)	col−>exists(x $\mid$ exp) **or** $Inst[x, exp]$
C6	COL_{B}−>forAll(x $\mid$ exp)	col−>forAll(x $\mid$ exp) **and** $Inst[x, exp]$
C7	COL_{B}−>collect(x $\mid$ exp)	col−>collect(x $\mid$ exp)−>including($Inst[x, exp]$)
C8	COL_{B}−>one(x $\mid$ exp)	(col−>one(x $\mid$ exp) **and not** $Inst[x, exp]$) **or** (**not** col−>exists(x $\mid$ exp) **and** $Inst[x, exp]$)
C9	COL_{B}−>isUnique(x $\mid$ exp)	col−>isUnique(x $\mid$ exp) **and** col−>select(x $\mid$ exp = $Inst[x, exp]$)−>isEmpty()

C10	COL_{B}−>any(x $\mid$ exp)	**if** col−>exists(x $\mid$ exp) **then** col−>any(x $\mid$ exp)
		else b **endif**

C11	COL_{B}−>select(x $\mid$ exp)	**if** $Inst[x, exp]$ **then** col−>select(x $\mid$ exp) −>including(b)
		else col−>select(x $\mid$ exp) **endif**

C12	COL_{B}−>reject(x $\mid$ exp)	**if** $Inst[x, exp]$ **then** col−>reject(x $\mid$ exp)
		else col−>reject(x $\mid$ exp)−>including(b) **endif**

C13	COL_{B}−>collectNested(x $\mid$ exp)	col−>collectNested(x $\mid$ exp) −>including($Inst[x, exp]$)
C14	COL_{B}−>count(exp)	1 (if exp = b) col−>count(exp) (otherwise)
C15	COL_{B}−>includes(exp)	true (if exp = b) col−>includes(exp) (otherwise)
C16	COL_{B}−>excludes(exp)	false (if exp = b) col−>excludes(exp) (otherwise)
C17	COL_{B}−>including(exp)	col −>including(b) (if exp = b) col−>including(exp)−>including(b) (otherwise)
C18	COL_{B}−>excluding(exp)	col (if exp = b) col−>excluding(exp) −>including(b) (otherwise)
C19	COL_{B}−>includesAll(exp)	col−>includesAll(col') (if exp = COL'_{B}) col−>includesAll(exp) (otherwise)
C20	COL_{B}−>excludesAll(exp)	false (if exp = COL'_{B}) col−>excludesAll(exp) (otherwise)
C21	exp−>includesAll(COL_{B})	false (if exp $\neq$ COL'_{B}) col'−>includesAll(col) (otherwise)
C22	exp−>excludesAll(COL_{B})	false (if exp = COL'_{B}) exp−>excludesAll(col) (otherwise)
C23	COL_{B} − exp	col − col' (if exp = COL'_{B}) (col − exp)−>including(b) (otherwise)
C24	exp − COL_{B}	col' − col (if exp = COL'_{B}) exp − col (otherwise)
C25	COL_{B}−>symDiff(exp) or exp−>symDiff(COL_{B})	col−>symDiff(col') (if exp = COL'_{B}) col−>symDiff(exp) −>including(b) (otherwise)
C26	COL_{B}−>union(exp) or exp−>union(COL_{B})	col−>union (col')−>including(b) (if exp = COL'_{B}) col−>union (exp)−>including(b) (otherwise)
C27	COL_{B}−>intersection(exp) or exp−>intersection(COL_{B})	col−>intersection(col')−>including(b) (if exp = COL'_{B}) col−>intersection(exp) (otherwise)
C28	COL_{B}= exp or exp = COL_{B}	col = col' (if exp = COL'_{B}) false (otherwise)
C29	$\text{COL}_{\text{B}}$$\neq$ exp or exp $\neq$ COL_{B}	col $\neq$ col'(if exp = COL'_{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*).

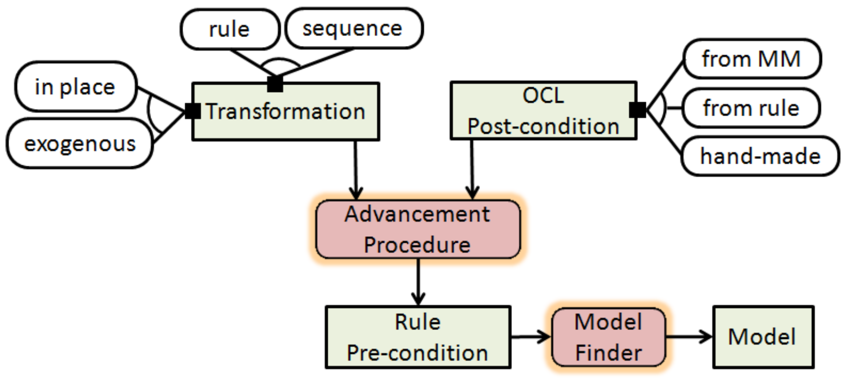

Figure 4: Combining post-condition advancement with model finding.

model including a target submodel reachable?). Some key differences are the following:

- First, our post-condition advancement strategy supports complex OCL post-conditions and application conditions, while previous works impose limitations on the target model/submodel or the (positive and negative) application conditions, e.g. only graph patterns can be used to describe them. In this sense, our approach is both more expressive and more flexible in terms of defining the target post-condition.

- Second, our approach does not require a specialized solver and can take advantage of existing OCL model finders. For instance, in this approach model finders do not need to be aware of the semantics of the transformation language: it is implicitly encoded in the OCL expressions they receive as input. Furthermore, the model finder only considers one model at a time, i.e. the analysis of the initial model or the final model is addressed separately.

- Third, our approach does not focus exclusively on backwards reachability or coverability, problems which have decidability issues [19]. Instead, backwards reachability is only one of the potential applications of this method. However, due to the expressiveness supported by our approach, we are not able to employ optimizations and symbolic techniques which make specialized approaches for backwards reachability more efficient.

- Finally, another advantage is that this architecture provides a uniform framework to address a wide variety of quality issues in model transformation. That is, the basic toolkit depicted in Figure 4 can be applied to a variety of problems depending on the choice of inputs and the usage of the model finder output.

Post-condition advancement can contribute to several types of analysis: *verification* (proving that the transformation is correct), *validation* (exercising the transformation in specific models to assess its correct operation), *testing* (generating test data to automate validation) or *diagnosis* (a combination of the previous goals, where a potential fix for each defect is derived automatically). These application scenarios can be further classified according to several criteria:

1. The input transformation being considered, i.e. whether it is a single rule or a rule sequence[2] and whether it is an in-place or exogenous transformation.
2. The input post-condition, i.e. whether it is a meta-model constraint, a constraint derived from another rule, it is hand-made or it is specific pattern that can be used to check recurring properties.
3. The specific conditions provided to the model finder in addition to the pre-condition or post-condition.
4. How the model finder output is used, i.e. whether a model (or lack of thereof) constitutes an example of a correct scenario, a proof of correctness or a counterexample.

Table 7 summarizes several application scenarios according to these criteria. e.g. executability, deadlocks, rule independence, rule sequences and backward reachability. These scenarios are described in more detail in Section 4.

4. Applications of the Method

This section describes several scenarios where our approach can be applied, both for in-place and exogenous transformations. Without loss of generality, the sample transformations will be described using GT.

[2]Besides rule sequences, this approach may also be applied to study chained transformations.

	in-place	**exogenous**
Constraints from MM	**(V)** Check weak executability of a rule **(V)** Check strong executability of a rule **(D)** Derive weakest pre-condition to ensure rule strong executability	**(D)** Check possible violations of target MM constraints by a rule and strengthen rule applicability to ensure meeting target MM constraints.
Constraints from other rules	**(T)** White-box testing: Test sequence execution **(V)** Detect and analyze deadlocks **(V)** Rule independence	**(T)** White box testing: sequence executability
Hand-made constraints	**(T)** Backwards reachability: finding an initial graph (satisfying C_{pre} leading to a graph fulfilling C_{post}) **(D)** Add pre-condition to a rule so that a state with some properties is not reached	**(T)** Black-box testing: generation of source models to produce certain configuration of elements in the target models

Table 7: Applications of the method. V=Verification, D=Diagnosis, T=Testing.

Before detailing the scenarios of Table 7, we need to introduce some notation. We normally write MM for a meta-model, and $CONST(MM)$ for its set of OCL integrity constraints. A model M typed by MM and satisfying $CONST(MM)$ is written $M \models MM$. Similarly, given a constraint C, we write $M \models C$ if M satisfies C.

To describe in-place transformations, we assume GT rules of the form $p = \langle r\colon L \rightarrow R, ATT_{COND}\rangle$, where L and R are graphs, and ATT_{COND} is a set of OCL constraints expressing attribute conditions. Given a rule p, it is straightforward to derive the set of atomic actions it performs, which can then be serialized to a sequence by placing the actions in any arbitrary order. We call such a sequence $actions(p)$.

We say that a rule p is enabled in a model M, written $M \models p$, if there is an occurrence o of L in M (formally, a graph morphism [6]), where the constraints in ATT_{COND} hold given the identification of objects induced by the occurrence. We sometimes write $M \models_o p$ to denote the particular occurrence o where p is enabled in M. We write $OCC(p, M)$ for the set of all valid occurrences of p in M, and $M \overset{p,o}{\Rightarrow} M'$ for the transformation of model M into M' via the application of rule p at occurrence o. Finally, $enabling(p)$ denotes the constraint ensuring the enabledness of p. This constraint is made of the conjunction of a number of nested existential quantifiers requiring the existence of every element in L connected as specified by L, and the satisfaction of the ATT_{COND} attribute condition (see [8] and the example in Section 4.2). We sometimes write $enabling_L(p)$ for the OCL constraint derived from the rule's LHS, but not including the attribute condition ATT_{COND}. We define the strengthening operator $\tilde{\wedge}$, so that $A \mathbin{\tilde{\wedge}} B$ is a conjunction of B and the inner-most existential quantifier of A. This way, $enabling(p) = enabling_L(p) \mathbin{\tilde{\wedge}} ATT_{COND}$.

Definition 1 (Strengthening operator $\tilde{\wedge}$). *Let A be an OCL constraint defining a condition inside a set of nested existential quantifiers:*

```
Type1.allInstances()->exists(v1 |
Type2.allInstances()->exists(v2 |
...
TypeN.allInstances()->exists(vN | condition ) ...))
```

and let B be another OCL constraint. Then, $A \mathbin{\tilde{\wedge}} B$ is defined as follows:

```
Type1.allInstances()->exists(v1 |
Type2.allInstances()->exists(v2 |
...
TypeN.allInstances()->exists(vN | condition ∧ B ) ...))
```

The applications of our method rely on the correctness of the post-condition advancement procedure (adv). The following lemma defines this correctness notion. A more detailed formalization of this lemma together with a complete proof is included in Appendix A.

Lemma 1 (Correctness of weakest precondition). *Let MM be a meta-model, p a graph transformation rule and C an OCL constraint over MM. Let M_1 and M_2 be any pair of models such that $M_1, M_2 \models MM$, $M_1 \models enabling(p)$ and $M_1 \overset{p}{\Rightarrow} M_2$. Let C' be the weakest precondition computed by our method from rule p and post-condition C, i.e. $C' = adv(actions(p), C)$. Then, $M_1 \models C'$ if and only if $M_2 \models C$.*

In the following subsections, we characterize each application of the method in Table 7 and show how they can be solved using our method and model finders, presenting some examples as well.

4.1. Strong executability of rules

A rule p is called strongly executable (SE) if $\forall M \models MM, \forall o \in OCC(p, M)$: $M \overset{p,o}{\Rightarrow} M'$ implies $M' \models MM$. In the simplest scenario, the rule has no attribute conditions, and the goal is to make it SE by adding conditions that ensure that any application of the rule yields a model that fulfils the cardinality and integrity constraints of the meta-model. In a more general setting, given a rule p with non-empty application conditions, we might want to check whether p is SE, and if not, strengthen its application conditions to make it SE. Definition 2 states the problem in the general case.

Definition 2 (Strengthening to strong executability (SSE)). *Given a rule $p = \langle r\colon L \to R, ATT_{COND}\rangle$, and a meta-model MM, find a rule $p' = \langle r\colon L \to R, ATT'_{COND}\rangle$ s.t. $\forall M \models MM, \forall o \in OCC(p', M)$: $o \in OCC(p, M)$ and ($M \overset{p',o}{\Rightarrow} M'$ implies $M' \models MM$).*

The condition $\forall o \in OCC(p', M)$: $o \in OCC(p, M)$ ensures that the applicability conditions of p' ($enabled(p')$) are stronger than those for p. This is always *true* if the original ATT_{COND} is empty (because both p and p' have the same LHS).

Method 1 describes how to solve the scenario using our advancement procedure *adv*, for which we need a model finder. As previously mentioned, a model finder takes a meta-model and a set of OCL constraints, and generates a model conformant to the meta-model and satisfying the constraint, if it exists within a given bound. In the following examples, we use the Alloy model finder [23], but any other finder could be used as well. While the previous definition does not constrain the structure of the condition ATT_{COND} added to the rule, our advancement procedure constructs the weakest one.

Method 1 (Method for SSE). *Given a rule $p = \langle r\colon L \to R, ATT_{COND}\rangle$, and a meta-model MM:*

1. *Compute $ATT'_{COND} = adv(actions(p), CONST(MM))$.*
2. *Check if $\exists M$ s.t. $M \models MM$ and $M \models enabling(p) \tilde{\wedge} \neg ATT'_{COND}$ (with a model finder)*
3. *If such M exists, then $p' = \langle r\colon L \to R, ATT_{COND} \wedge ATT'_{COND}\rangle$*
4. *else $p' = \langle r\colon L \to R, ATT_{COND}\rangle$*

The main idea of the method is to compute a constraint ATT'_{COND} for the rule p without considering p's attribute conditions, and check whether a model exists that satisfies the original condition of the rule (ATT_{COND}), but not the derived one. Such model is not guaranteed to satisfy MM after the rule application. Therefore, the rule's conditions are strengthened to disallow applying the rule on those models.

Next, we show that using this method, we obtain a rule satisfying Definition 2.

Theorem 1 (Correctness of SSE method). *Given a rule $p = \langle r\colon L \to R, ATT_{COND}\rangle$, and a rule $p' = \langle r\colon L \to R, ATT'_{COND}\rangle$ calculated with Method 1, we have that $\forall M \models MM, \forall o \in OCC(p', M)\colon o \in OCC(p, M)$ and ($M \overset{p',o}{\Rightarrow} M'$ implies $M' \models MM$)*

Proof 1. *Case 1) Let us assume p' has the form $p' = \langle r\colon L \to R, ATT_{COND} \wedge ATT'_{COND}\rangle$. Then, it is clear that $\forall o \in OCC(p', M)\colon o \in OCC(p, M)$. This is so as $enabled(p') \implies enabled(p)$. It remains to check that p' is SE. Rule $p'' = \langle r\colon L \to R, ATT'_{COND}\rangle$ is SE by correctness of our advancement procedure (Lemma 1). Because $enabled(p') \implies enabled(p'')$, we have that p' is applicable in a subset of the models where p'' is applicable. As p'' is SE, then p' is SE as well.*
Case 2) In this case, $p = p'$. By the 2nd step in the method, $\nexists M$ s.t. $M \models MM$ and $M \models enabling(p)\tilde{\wedge}\neg ATT'_{COND}$. This means that $\nexists M$ s.t. $M \models enabling_L(p)\tilde{\wedge}(ATT_{COND} \wedge \neg ATT'_{COND})$, and hence we have that either $ATT_{COND} \implies ATT'_{COND}$ or $enabling_L(p)$ is unsatifiable. In the latter case, p is SE vacuosly (there cannot be a model where p is applicable). Lets consider the first case. Rule $p'' = \langle r\colon L \to R, ATT'_{COND}\rangle$ is SE by Lemma 1. Because $ATT_{COND} \implies ATT'_{COND}$, we have that p is applicable in a subset of the models where p'' is applicable. As p'' is SE, then p is SE as well.

Example 6. Assume we would like to enforce SE for rule `process` of Figure 2. For this purpose:

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

```
Conveyor.allInstances()->forAll(v |
  v.piece->size() ≤ 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, the rule deletes object `r` (together with link `co1-r`) and it creates object `p` (with link `co2-p`).
3. Next, we apply the replacement patterns. After applying patterns OD2, LC1a, C18 and C1 to the first part of the expression, we obtain:

   ```
   Conveyor.allInstances()−>forAll(v |
     if v = co2 then
       v.piece−>excluding(r)−>size() + 1 ≤ v.capacity
     else v.piece−>excluding(r)−>size() ≤ v.capacity
     endif)
   ```

   This result is more complex than the condition stated in rule `process`, because it considers the possibility of a *non-injective* matching, i.e. `co1 = co2`. In such a 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 Figure 2 as it is not obvious from the invariant and the LHS. As a result, condition in Figure 2 was too restrictive and forbade the execution of the rule in a correct scenario. This example illustrates the benefits of automating the approach.
4. Finally, we apply the replacements (patterns OD1, C18, C6, and O2) for the second part of the post-condition. The final result is therefore:

   ```
   Conveyor.allInstances()−>forAll(v |
     if v = co2 then
       v.piece −>excluding(r)−>size() + 1 ≤ v.capacity
     else
       v.piece −>excluding(r)−>size() ≤ v.capacity
     endif) and
     Piece.allInstances()−>excluding(r)−>forAll(z |
       z.conveyor−>size() = 1)
   ```

Altogether, this example shows that adding by hand OCL pre-conditions to ensure SE is error-prone and time-consuming. For instance, Figure 5 shows two sample models, computed using the model finder Alloy, which are incorrectly forbidden by the handwritten attribute conditions of rules

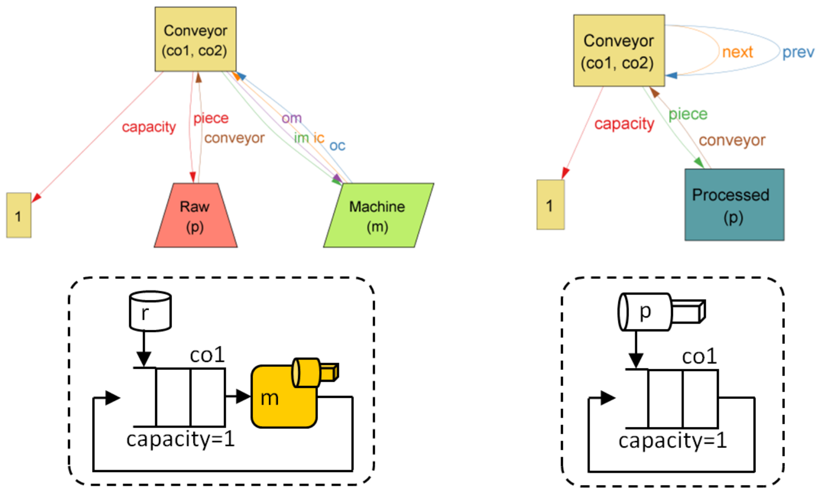

Figure 5: Valid initial models forbidden by the attribute conditions of rule `process` (left) and rule `move` (right). Top: output of the model finder Alloy. Bottom: same output shown in our concrete visual syntax.

`process` and `move`. Our method solves this problem as such pre-conditions are automatically derived from the meta-model integrity constraints.

4.2. White-box testing and sequences

A typical validation scenario concerns the testing of rule application sequences, in order to answer questions like: is it possible to apply the rule sequence $s = \langle p_1; p_2; \ldots; p_n \rangle$ to some starting model, and obtain a model in which C_{post} holds (e.g., there are less than 3 objects of a certain type)? In the more general case, one may also like to describe the starting model: is it possible to apply the sequence s to a model in which C_{pre} holds (e.g., there are exactly 5 objects of a certain type) yielding a model in which C_{post} holds (e.g., there are no more than n objects of a certain type)? These validation tasks are useful for white-box testing of transformations, where one wants to execute a set of predefined rule sequences. We next formulate the problem statement.

Definition 3 (Rule sequence testing (RST)). *Given a rule sequence* $s = \langle p_1 = \langle r_1 \colon L_1 \rightarrow R_1, ATT^1_{COND} \rangle; \ldots; p_n = \langle r_n \colon L_n \rightarrow R_n, ATT^n_{COND} \rangle \rangle$*, a meta-model* MM*, and two constraints* C_{pre} *and* C_{post}*, find a model* M *s.t.* $M \models MM$*,* $M \models C_{pre}$ *and* $M \overset{s}{\Rightarrow} M'$*, with* $M' \models MM$*, and* $M' \models C_{post}$*.*

The method to solve this scenario advances C_{post} to rule p_n, then to rule p_{n-1}, and so on, until the first rule in the sequence is reached. In the process, the applicability conditions $\mathit{enabling}(p_i)$ of each rule are taken into account. As stated before, this predicate is made of nested existential quantifiers, and hence it is concatenated with the advanced condition using the strengthening operation $\tilde{\wedge}$.

Method 2 (Method for RST). *Given a rule sequence $s = \langle p_1 = \langle r_1 \colon L_1 \to R_1, ATT^1_{COND}\rangle ; ... ; p_n = \langle r_n \colon L_n \to R_n, ATT^n_{COND}\rangle\rangle$, a meta-model MM, and two constraints C_{pre} and C_{post}, find M as follows:*

```
1. $CC := C_{post}$
2. for $i = n$ to 1
       $CC := enabling(p_i)\ \tilde{\wedge}\ adv(actions(p_i), CC)$
3. Find a model $M$ s.t. $M \models MM$ and $M \models C_{pre} \wedge CC$ (using a model finder).
```

Next theorem shows that the method actually achieves RST, as expressed in Definition 3.

Theorem 2 (Correctness of RST method). *Given a rule sequence $s = \langle p_1 = \langle r_1 \colon L_1 \to R_1, ATT^1_{COND}\rangle ; ... ; p_n = \langle r_n \colon L_n \to R_n, ATT^n_{COND}\rangle\rangle$, a meta-model MM, two constraints C_{pre} and C_{post}, and a model M found using Method 2, then $M \models MM$, $M \models C_{pre}$, and $M \overset{s}{\Rightarrow} M'$, with $M' \models MM$, and $M' \models C_{post}$.*

Proof 2. *We proceed by induction on the rule sequence length, as follows.*

Base case) *$n = 1$. In this case $CC_1 := enabling(p_1)\ \tilde{\wedge}\ adv(actions(p_1), C_{post})$, and let M be a model s.t. $M \models MM \wedge C_{pre} \wedge CC_1$ found in step 3 of Method 2. By Lemma 1, we have that $M' \models C_{post}$ and $M' \models MM$ with $M \overset{p_1}{\Rightarrow} M'$. Please note that, at this stage, the model finder may not find such model M, either if the constraints $MM \wedge C_{pre} \wedge CC_1$ are unsatisfiable (and hence M does not exist), or if the search bounds of the finder are too narrow. We assume now and in following proofs that the search bounds are as wide as necessary.*

Induction step) *Assume Method 2 yields a correct result for sequences s of length up to n. This means that $CC_n := enabling(p_n)\ \tilde{\wedge}\ adv(actions(p_n)$,*

CC_{n-1}), and for a model M s.t. $M \models MM \wedge C_{pre} \wedge CC_n$, we have that $M' \models MM \wedge C_{post}$ with $M \overset{s}{\Rightarrow} M'$.

Let $CC_{n+1} := enabling(p_{n+1}) \tilde{\wedge} adv(actions(p_{n+1}), CC_n)$, and let be a model M with $M \models MM \wedge C_{pre} \wedge CC_{n+1}$. Then, $M \overset{p_{n+1}}{\Rightarrow} M''$ implies $M'' \models MM \wedge CC_n$ by Lemma 1. But this implies that $M''' \models MM \wedge C_{post}$ for $M'' \overset{s}{\Rightarrow} M'''$. Hence, this implies that $M''' \models MM \wedge C_{post}$ with $M \overset{\langle p_{n+1}\rangle \cdot s}{\Longrightarrow} M'''$, and $\langle p_{n+1}\rangle \cdot s$ being the concatenation of rule p_{n+1} to sequence s.

Example 7. Assume we want to test the sequence ⟨`process; move`⟩ starting and finishing in a model with one piece and two conveyors. Our method would first express the requirement on the final model as an OCL constraint, which we denote as C_{post}. The constraint is the following:

```
Piece.allInstances()−>size() = 1 and Conveyor.allInstances()−>size() = 2
```

Then, constraint C_{post} is advanced using this method as a pre-condition for rule `move`, producing a new constraint $C_{pre}^{\texttt{move}}$. Before it can be advanced further within the sequence of rules, two issues need to be resolved:

- This constraint may contain references to objects named in the LHS of the rule.
- We need to make sure that rule `move` was applicable at this point of the sequence.

In order to solve both issues, we need to translate the enabling condition of the LHS of rule `move` as an OCL expression $enabling_L(move)$ (using the algorithm given in [8]). In short, the algorithm defines nested existential quantifiers, one for each object in the LHS. The type of each object in the LHS defines the context type that will be quantified and the name of the object in the LHS is used as the name of the quantified variable. The body of the innermost quantifier describes the graph pattern: each edge $a - b$ becomes a constraint of the form $a.rb{-}{>}\ includes(b)$, where rb is the association end of the association that contains link $a - b$. For instance, the LHS of rule `move` is encoded as:

```
Conveyor.allInstances()−>exists( co1, co2 |
Raw.allInstances()−>exists (r |
  co1.next−>includes(co2) and co1.piece−>includes(r)))
```

Further application conditions (e.g. OCL guards) can be added within the innermost existential quantifier (strengthening the constraint of the graph pattern, i.e. using **and**, which we have written using mathematical notation as $\tilde{\wedge}$ in previous methods and theorems). In this case, we want to ensure that our advanced constraint also holds, so the following constraint would be used:

```
Conveyor.allInstances()−>exists( co1, co2 |
Raw.allInstances()−>exists (r |
  co1.next−>includes(co2) and co1.piece−>includes(r) and
  (co2.piece−>size() + 1 ≤ co2.capacity) and  C_pre^move))
```

This new constraint, which we can call *intermediate*, can now be advanced as a pre-condition of rule `process`, producing a constraint $intermediate_{pre}^{\texttt{process}}$. Again, the result should be enclosed within the OCL condition for the LHS of `process`, to make sure that this rule is enabled at the beginning of the sequence and that any references to elements in the LHS of rule `process` are given a proper meaning. This property would have the following structure:

```
Conveyor.allInstances()−>exists( co1, co2 |
Raw.allInstances()−>exists (r |
Machine.allInstances()−>exists (m |
  co1.piece−>includes(r) and
  m.ic−>includes(co1) and m.oc−>includes(co2) and
  (co2.piece−>size() + 1 ≤ co2.capacity) and intermediate_pre^process)))
```

This constraint, put in conjunction with an OCL condition expressing the requirements expected from the initial model (C_{pre}, a model with one piece and two conveyors in this example), can be used by a model finder to compute an initial model where: (i) the starting model requirements hold, (ii) the first rule is enabled, (iii) each rule in the sequence is enabled after firing the previous rules, and (iv) constraint C_{post} holds after firing the final rule.

If we pass the resulting constraint to a model finder like Alloy this will return a model satisfying the constraint, if such a model exists. The resulting model ensures that the rule sequence is applicable, and can then be used for testing, to check whether the application of the sequence behaves as expected. The top right of Figure 6 shows one model found by Alloy that satisfies the derived OCL constraint. This model satisfies the condition of having one piece and two conveyors (a requirement for the input model). The application of the sequence yields the model to the bottom right of Figure 6, which contains one piece and two conveyors too (the requirement for the output model). On the Alloy instance in the left, the mappings used in the rule firings are identified, e.g. `co1{process}` means that this object is matched as Conveyor `co1` in the LHS of rule `process`. Notice that there is no correspondence for piece `r` in the LHS of rule `move` as that object does not exist yet in the initial model (it is created when rule `process` is fired).

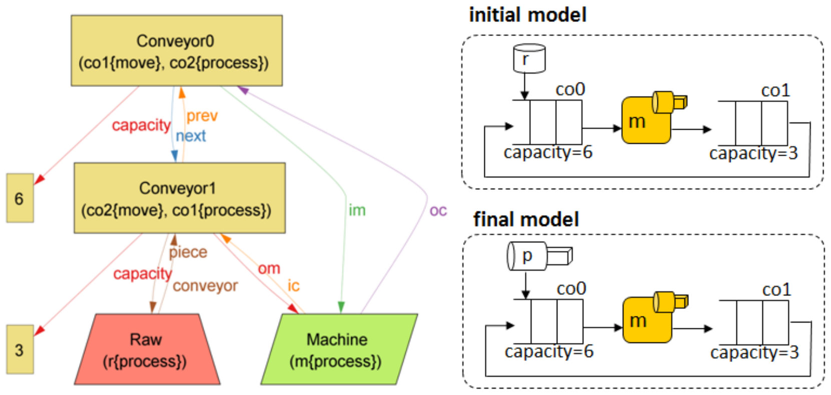

Figure 6: Left: initial model for the sequence ⟨`process; move`⟩ that preserves one piece and two conveyors, as generated by Alloy. Top right: same model using our concrete visual syntax. Bottom right: final model after firing ⟨`process; move`⟩.

Example 8. As a second example, we show the application of the method for white-box testing of exogenous transformations, in particular with Triple Graph Grammar (TGG) operational rules [24]. Their behaviour is similar to that of GT rules but, in this case, rules are made of LHS/RHS triple graphs made of the source and target graphs of the transformation, together with a correspondence graph that contains traces between the other two graphs. Typically, the transformation only creates elements in the target and correspondence graphs, whereas the source graph is just inspected.

The example is a very simplified version of the classical transformation from UML to RDBMS. The triple meta-model for the example, made of the source, target and trace meta-models, is shown to the left of Fig. 7. Thus, the correspondence meta-model in the middle declares two types of traces: `PS` maps packages and schemas, and `CT` maps classes and tables. The dotted arrows specify the allowed references (technically graph morphisms) from the correspondence to the source and target models, and we treat them as normal associations with cardinality 1 on the side of the source/target class, and 0..1 on the side of the trace.

The transformation from UML to RDBMS is shown to the right of Fig. 7. We make the same assumptions as for GT rules regarding the creation of

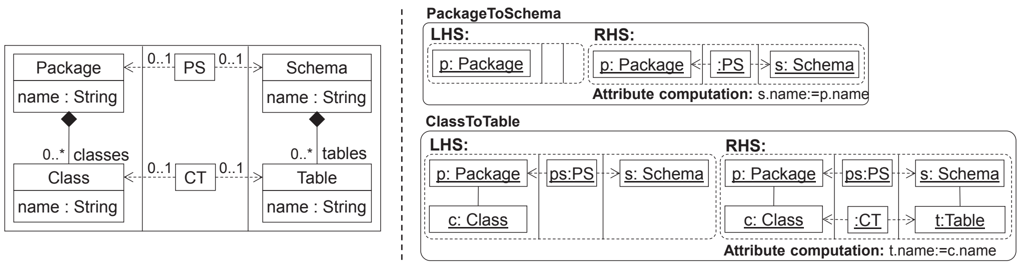

Figure 7: A meta-model triple (left). Operational TGG rules (right).

edges, but do not consider the ones for deletion of objects and edges since exogenous transformations are non-deleting (i.e. rules can only create). Thus, when advancing OCL constraints for exogenous rules, we will not need to consider replacements for deleting operations. In particular, our TGG is made of two operational rules: the first one creates a schema for each package, and the second one creates a table for each class.

Now, we want to obtain a valid UML model that permits firing sequence ⟨`PackageToSchema; ClassToTable`⟩, requiring as the post-condition that there is one schema with no tables:

Schema.allInstances()−>exists(s | s.tables−>isEmpty())

Using the same procedure as in the previous example, the post-condition can be advanced into a pre-condition that ensures the desired outcome for the sequence. Then, a model finder can use this pre-condition to compute sample models. Figure 8 depicts a sample initial and final model, as generated by Alloy. In this scenario, the pre-condition implicitly states that the initial model needs to contain at least two packages, one with a corresponding schema and another without it. Deriving this pre-condition by hand would not be trivial as the target post-condition is not defined in terms of packages but in terms of schemas.

4.3. Backwards reachability

Next, we generalize the previous method with the possibility to perform backwards search. This scenario starts with a final model satisfying C_{post}, and performs a backward exploration applying the available rules backwards, until a model satisfying C_{pre} is found. While in the previous scenario the sequence of rules is fixed a priori, in backwards reachability, all possible sequences of rules are evaluated until reaching a model with the given requirements.

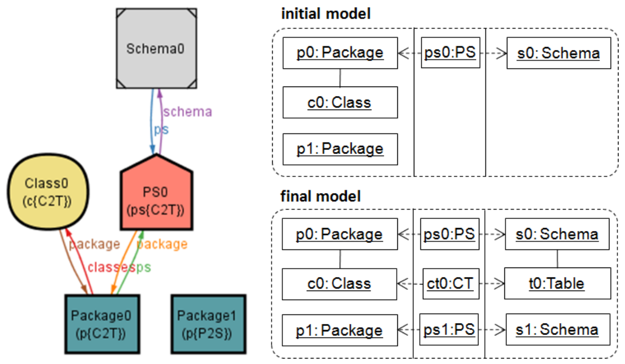

Figure 8: Left: initial model for the sequence ⟨`PackageToSchema; ClassToTable`⟩ that produces a schema with no tables, as generated by Alloy. Top right: same model using our concrete visual syntax. Bottom right: final model after firing the sequence.

Definition 4 (Backwards exploration (BE)). *Given a set of rules* $S = \{\langle p_i = \langle r_i \colon L_i \rightarrow R_i, ATT^i_{COND}\rangle\}_{i \in I}$*, a meta-model* MM*, and two constraints* C_{pre} *and* C_{post}*, find a model* M *s.t.* $M \models MM$ *and* $M \models C_{pre}$*, and find a rule sequence* $s = \langle p_r; \ldots; p_k\rangle$ *s.t.* $M \overset{s}{\Rightarrow} M'$*, with* $M' \models MM$ *and* $M' \models C_{post}$*.*

This scenario goes one step further than **RST**, because it seeks a rule sequence in addition to an initial model. In order to solve it, we perform a backwards search starting from C_{post} and ending whenever a model that satisfies C_{pre} is found (see Figure 9). To avoid infinite exploration, we bound the search length to a maximum number of steps. Thus, the method builds iteratively a constraint C_{ji} by advancing C_i through the actions of rule p_j, and then adding its enabling condition $enabling(p_j)$. If there is a model M satisfying $C_{pre} \wedge C_{ji}$ (found using a model finder), then the method returns both the found model M and the sequence $\langle p_j; p_i\rangle$.

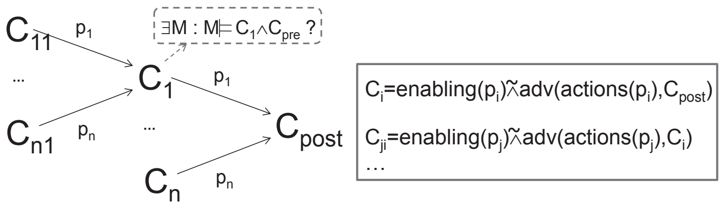
C11
...
Cn1
p1
pn
∃M : M⊨C1∧Cpre ?
C1
...
Cn
p1
pn
Cpost
Ci=enabling(pi)∧adv(actions(pi),Cpost)
Cji=enabling(pj)∧adv(actions(pj),Ci)
...

Figure 9: Method for backwards exploration.

Method 3 (Method for BE). *Given a set of rules $S = \{\langle p_i = \langle r_i \colon L_i \to R_i, ATT^i_{COND} \rangle\}_{i \in I}$, a meta-model MM, a constant MAX bounding the maximum search depth, and two constraints C_{pre} and C_{post}, find a model M and a sequence s as follows:*

$$(M, s) := BEX(S, C_{pre}, C_{post}, \langle\rangle, 1, MAX)$$

where function BEX is defined as follows:

```
**function** $BEX(S, C_{end}, C_{curr}, s_{curr}, clevel, max) : Model \times Sequence$

1. if $\exists M : M \models MM \land C_{curr} \land C_{end}$ return $(M, s_{curr})$
2. if $clevel > max$ return nil
3. for i = 1 to n (with n=—I—, the size of the rule set)
    (a) $(M, s) := BEX(S, C_{end}, enabling(p_i) \tilde{\land} adv(actions(p_i), C_{curr}), \langle p_i \rangle \cdot s_{curr}, clevel + 1, max)$
    (b) if $M \neq nil$ return $(M, s)$
4. return nil
```

where $\tilde{\land}$ is the strengthening operator, and $\langle p \rangle \cdot s$ is the concatenation of rule p to sequence s.

Theorem 3 states that if the BEX function of Method 3 returns a model M and a sequence s, then such pair fulfills the requirements for **BE** given in Definition 4. Alternatively, if BEX returns nil, then a source M satisfying the conditions cannot be reached within the search bounds of MAX.

Theorem 3 (Correctness of BE method). *Let $S = \{\langle p_i = \langle r_i \colon L_i \to R_i, ATT^i_{COND} \rangle\}_{i \in I}$ be a set of rules, MM a meta-model, $MAX \in \mathbb{N}_0$ a constant, and C_{pre} and C_{post} two constraints.*

Assume $(M, s) := BEX(S, C_{pre}, C_{post}, \langle\rangle, 1,\ MAX)$ is the model and sequence returned by the BEX function defined in Method 3. Then, $M \models MM \land C_{pre}$, $|s| \leq MAX$, and $M' \models MM \land C_{post}$, with $M \overset{s}{\Rightarrow} M'$ and where $|s|$ is the length of sequence s.

Alternatively, if $BEX(S, C_{pre}, C_{post}, \langle\rangle, 1, MAX)$ returns nil, then there is no model M and sequence s with $|s| \leq MAX$ s.t. $M \models MM \land C_{pre}$ and $M \overset{s}{\Rightarrow} M'$ and $M' \models MM \land C_{post}$.

Proof 3. *(Sketch) Let us assume that the algorithm returns a non-nil result. If the algorithm returns a non-nil result* (M, s) *in step 1, it means that* $M \models MM \wedge C_{curr} \wedge C_{end}$*,* $C_{curr} = C_{post}$*,* $C_{end} = C_{pre}$ *and* $s = \langle\rangle$*. Therefore the theorem holds, because* $M \overset{\langle\rangle}{\Rightarrow} M' = M$*, and hence* $M \models MM \wedge C_{pre}$ $M' \models MM \wedge C_{post}$*, and* $|\langle\rangle| = 0 \leq MAX$ *(for any* $MAX \in \mathbb{N}_0$*).*

Else, in step 3, a depth first exploration of rule sequences is performed. In each step, $C_{end} = C_{pre}$*, and* C_{curr} *(initially equal to* C_{post}*) is the result of advancing* C_{curr} *via the chosen rule* p_i*, and strengthening* p_i'*s enabling condition:* $C'_{curr} := enabling(p_i)\tilde{\wedge}adv(actions(p_i), C_{curr})$*. Therefore, by Lemma 1, if a model* M *is found s.t.* $M \models C'_{curr}$*, then* $M' \models C_{curr}$ *with* $M \overset{\langle p_i\rangle}{\Rightarrow} M'$*. Therefore, for sequences of length 1, the theorem holds as well, and by iterating these two arguments, the theorem holds for sequences of arbitrary length.*

Lets assume that the algorithm returns a nil result. If this happens at step 2, it means that the search bounds are overstepped. A nil value can also be returned at step 4, which means that at a certain stage in the recursion, every rule was explored, and no model M *was found satisfying the conditions. Because at each step in the recursion the algorithm tries all available rules (and we assume that the model finder is customized with sufficent search bounds), then the nil result is returned because there is no model satisfying the conditions within a scope of* MAX *for the rule sequence.*

Please note that Method 3 performs an exhaustive search of sequence of length MAX. Therefore, if a sequence of length MAX and a model M exist fulfilling Definition 4, then Method 3 will find it. Hence, we can enunciate the following completeness result.

Theorem 4 (Completeness of BE method). *Let* $S = \{\langle p_i = \langle r_i : L_i \to R_i, ATT^i_{COND}\rangle\}_{i\in I}$ *be a set of rules,* MM *a meta-model,* MAX *a constant,* C_{pre} *and* C_{post} *two constraints. Assume there exist two models* M *and* M'*, and* s *a sequence of rules in* S *of length* n *s.t.* $M \overset{s}{\Rightarrow} M'$*, with* $M' \models MM \wedge C_{post}$ *and* $M \models MM \wedge C_{pre}$*, then* $BEX(S, C_{pre}, C_{post}, \langle\rangle, 1, n+1)$ *returns a model* M'' *and a sequence* s' *with* $|s'| \leq n$ *satisfying the conditions in Definition 4.*

Proof 4. *(Sketch) Assume a sequence* s *(with* $|s| \leq MAX$*) and a model* M *exist fulfilling the conditions.* BEX *performs an exhaustive bounded search of sequences of length up to* MAX*. Therefore, the algorithm will find such* *sequence and model, or a shorter sequence and model. In every case, the found sequence and model are correct according to Theorem 3. Please note that we assume that the model finder has suitable bounds for finding a model of the size of M.*

4.4. Detecting and analysing deadlocks

Backwards reachability is a flexible method, as it can be customized with the C_{pre} and C_{post} constraints. For example, we can use backward reachablity to perform a backward exploration from an unwanted deadlock situation (i.e., C_{post} is a constraint encoding that no rule is applicable), into some initial model of interest (satisfying C_{pre}).

Definition 5 (Deadlock exploration (DE)). *Given a set of rules $S = \langle p_i = \langle r_i \colon L_i \to R_i, ATT^i_{COND}\rangle_{i\in I}$, a meta-model MM, and a constraint C_{pre}, find a model M s.t. $M \models MM$ and $M \models C_{pre}$, and find a sequence $s = \langle p_r; ...; p_k\rangle$ s.t. $M \overset{s}{\Rightarrow} M'$, with $M' \models MM$ and no rule in S enabled in M'.*

Method 4 (Method for DE). *Given a set of rules $S = \{\langle p_i = \langle r_i \colon L_i \to R_i, ATT^i_{COND}\rangle\}_{i\in I}$, a meta-model MM, a constant MAX bounding the maximum search depth, and a constraint C_{pre}, find a model M and a sequence s as follows:*

$$(M, s) := BEX(S, C_{pre}, \neg enabling(p_1) \wedge ... \wedge \neg enabling(p_n), \langle\rangle, 1, MAX)$$

Because Method 4 is a special case of Method 3, with $C_{post} = \neg enabling(p_1) \wedge ... \wedge \neg enabling(p_n)$, we have the correctness and completeness results of Theorems 3 and 4.

4.5. Rule independence

Next, we show that our backwards analysis method can be used to test rule independence for pairs of rules having arbitrary attribute OCL conditions. Rules r_1 and r_2 are independent if executing r_1 does not disable r_2 (and vice versa) [6], or formally:

Definition 6 (Rule independence (RI)). *Given two rules $p_i = \langle r_i \colon L_i \to R_i, ATT^i_{COND}\rangle (i = 1, 2)$, and a meta-model MM, p_1 and p_2 are not independent if $\exists M$ s.t. $M \models MM \wedge enabled(r_1) \wedge enabled(r_2)$ and $\langle r_1; r_2\rangle$ is executable, but $\langle r_2; r_1\rangle$ is not.*

To solve this scenario, we can use a model finder to search for a model where both r_1 and r_2 are enabled, $\langle r_1; r_2\rangle$ is executable on M, but $\langle r_2; r_1\rangle$ is not. For the latter two conditions we use our *adv* advancement method.

Method 5 (Method for RI). *Given two rules* $p_i = \langle r_i \colon L_i \rightarrow R_i, ATT^i_{COND}\rangle$ *(i=1, 2), and meta-model* MM*, find a model* M *(with a model finder) s.t.:*

$$\begin{aligned} M \;\models\; & MM \wedge (enabling(r_1)\tilde{\wedge} adv(actions(r_1), enabling(r_2))) \wedge \\ & (enabling(r_2)\tilde{\wedge} adv(actions(r_2), \neg enabling(r_1))) \end{aligned}$$

Next theorem states the correctness of the method for **RI**.

Theorem 5 (Correctness of RI method). *Given two rules* $p_i = \langle r_i \colon L_i \rightarrow R_i, ATT^i_{COND}\rangle$ *(i=1, 2), a meta-model* MM*, and a model* M *s.t.*

$$\begin{aligned} M \;\models\; & MM \wedge (enabling(r_1)\tilde{\wedge} adv(actions(r_1), enabling(r_2))) \wedge \\ & (enabling(r_2)\tilde{\wedge} adv(actions(r_2), \neg enabling(r_1))) \end{aligned}$$

then r_1 *and* r_2 *are not independent.*

Proof 5. *We have the following implications:*

$$\begin{aligned} (enabling(r_1)\tilde{\wedge} adv(actions(r_1), enabling(r_2))) &\implies enabling(r_1) \\ (enabling(r_2)\tilde{\wedge} adv(actions(r_2), \neg enabling(r_1))) &\implies enabling(r_2) \end{aligned}$$

which means that $M \models MM \wedge enabling(r_1) \wedge enabling(r_2)$*, and therefore both* r_1 *and* r_2 *are applicable on* M*.*

In addition, $M \models enabling(r_1)\tilde{\wedge} adv(actions(r_1), enabling(r_2))$*, which means that* r_2 *is applicable on* M'*, with* $M \overset{r_1}{\Rightarrow} M'$*.*

Finally, $M \models enabling(r_2)\tilde{\wedge} adv(actions(r_2), \neg enabling(r_1))$*, and so* r_1 *is not enabled in* M'' *with* $M \overset{r_2}{\Rightarrow} M''$*.*

Please note that, while GT theory has characterized rule independence using categorical reasoning [6], our method has the advantage to consider arbitrary OCL attribute conditions, which is novel to the best of our knowledge.

5. Implementation

As described in Figure 4, the different application scenarios for our method rely on a combination of the advancement procedure with model finders where both can be seen as a black-box from one another. Since several model finders for OCL are already available, to enable our method we only need to provide an implementation of the advancement procedure itself. The *OCL advancement procedure* has been implemented in Java and distributed as an open source Eclipse plugin with the source code freely available in GitHub[3].

Internally, the tool offers a facade class *OCLBackwardReasoning* providing a set of methods to generate a pre-condition out of a given OCL post-condition. Separate methods are in charge of modifying the OCL expression for each atomic action in the (GT) rule. Each method takes as parameters the OCL expression, the model and the variable/s identifying the element to be modified and then delegates the actual advancement task to the appropriate class (e.g., CreateLink, DeleteLink, CreateObject,...).

To detect a particular replacement pattern in a constraint, we rely on the OCL Visitor API provided by the default OCL Eclipse plugin[4]. In our tool we extend the *AbstractVisitor* class to modify the abstract syntax tree of the input OCL expression following the replacement patterns discussed in the paper.

The tool comes also with two examples showing how to use the plugin. The first one implements the running example we are using in this paper while the second one represents a bank management system helpful to illustrate the effect of each individual atomic action separately.

In all cases, the weakest pre-condition was computed automatically in a few seconds. For example, the weakest pre-condition used in Example 6 to enforce SE in rule `process` is computed from the post-condition in 0.03s. In the bank management example, we consider the computation of the weakest pre-condition for four post-conditions stating simple integrity constraints, e.g. all accounts should have a non-negative balance. Given four types of atomic actions (link creation, link deletion, attribute update and object creation), the corresponding weakest pre-conditions are generated in 0.05s. These results have been measured on an Intel Core i5 760 2.8Ghz with 4Gb RAM.

[3]https://github.com/SOM-Research/ocl-backwardreasoning

[4]http://www.eclipse.org/modeling/mdt/?project=ocl

6. Related Work

In this section we analyse related works with respect to (1) the advancement method and (2) the use of backwards analysis in model transformations.

6.1. Advancing post-conditions

There are previous works on moving constraints from the RHS to the LHS of GT rules. The idea was first proposed in [13], where post-conditions for rules were derived from global invariants of the form $\forall P \exists Q$, where P and Q are graphs. In [25] the approach was generalized to adhesive high-level replacement systems. These works were extended in [26] to deal with nested conditions of arbitrary depth. This family of conditions has the same expressive power as first-order graph formulas [27]. The work of [28] translates a limited subset of OCL to graph constraints, which could be used by these other works to synthesize local pre-conditions for rules. In [29] an advancement procedure for attributed graph constraints into rule pre-conditions is presented. However, no OCL conditions are considered.

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-modeling environments. Furthermore, the use of OCL enables the application of tools initially targeting the simulation, analysis and verification on UML/OCL models (e.g. [8, 9, 17, 11, 18, 12]).

For transformations not defined as GT, the computation of the weakest pre-condition has also been studied, e.g. to analyse the composition of refactorings [30]. The notion of “backward descriptions” defined in [30] captures our replacement patterns of OCL expressions.

Hoare-style assertions have been extensively used for program verification [31, 32]. In [31] the authors use OCL-light (a simplified version of OCL) assertions to verify Java-light (a simplified version of Java) programs. In particular, the authors present rule for advancing OCL-light expressions upon attribute modification and object creation. In [32] the authors present a proof system for object-oriented programs based on an assertion language comparable to JML [33].

Regarding information systems, in [34] 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 [35] 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.

6.2. Backwards analysis

Some works consider the analysis of rewriting systems by the reverse execution of rules from undesired or bad states [21, 22, 20]. In [20] the authors model ad-hoc protocols using hyperedge replacement, where rules may contain NACs. Then, bad states are described through hypergraph patterns. The analysis consists in a fix-point iteration, in which rules are applied backwards with the aim to reach a pattern describing the set of initial configurations that would yield to the bad state. The algorithm uses over-approximation, because the patterns describe only the minimal requirement that need to be present. In a similar way [22] applies backward analysis to a network protocol described as a hyperedge replacement system. In that work, the authors use the quasi-well order given by the graph minor theorem to view a graph transition system as a well-structured system, which ensures a finite termination of the search. The work in [21] applies a similar backwards traversal for analysing dynamic memory heaps. Heaps are encoded as graphs, in which a preorder is given to ensure termination of the backwards analysis. As a summary, our work is more general than these approaches in the sense that it is formulated independent of the transformation language, and the structures and properties we can handle are richer. Moreover, we have provided several useful analysis properties for model transformations based on backwards analysis.

In [36] the authors compare three different verification mechanisms for model transformations. One of the analysis compared involved using the B method, for which generation of weakest pre-conditions were needed. In [37] a Hoare-style calculus is developed to analyse transformations expressed in (a subset of) QVT Operational.

Rule independence has been widely studied in GT [6], and implemented in tools like Henshin [38]. However, as already mentioned, the GT theory does not handle attribute conditions or attribute computations, but at most negative application conditions. We believe that GT tools specially oriented to MDE would greatly benefit from the integration of our rule independence analysis, since it considers OCL.

7. Conclusions and Future Work

We have presented a technique to automatically synthesize application conditions for model transformation rules. Application conditions are derived from the rule post-conditions such that any occurrence 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 well-formedness constraints defined in the meta-model, from other rules (e.g., when considering rule sequences) or be user specified.

This backwards reasoning process, combined with model finders and exploration mechanisms, facilitates a number of validation, testing and diagnosis tasks on the transformation system, both at the individual rule level and with respect to the interactions between several rules (or sequences of rules). Our method improves GT approaches, which do not consider OCL, leading to a more homogeneous treatment of all these analysis properties and permiting leveraging from standard off-the-shelf model finders. This fact simplifies the practical treatment of the analysis methods, and their integration in existing modeling environments, which could increase the adoption of formal methods in the MDE and GT communities.

Acknowledgements. Work partially funded by the Spanish Ministry of Economy and Competitivity (projects TIN2008-00444, TIN2011-24139 and TIN2014-52129-R), the Community of Madrid with project SICOMORO (S2013/ICE-3006), the EU commission with project MONDO (FP7-ICT-2013-10, #611125) and a research grant from UOC-IN3 (Internet Interdisciplinary Institute). We would like to thank Hamza Ed-Douibi for his work on the tool implementation part, and the reviewers for their useful comments.

References

[1] L. A. Rahim, J. Whittle, A survey of approaches for verifying model transformations, Software and System Modeling 14 (2) (2015) 1003–1028. doi:10.1007/s10270-013-0358-0.
URL `http://dx.doi.org/10.1007/s10270-013-0358-0`

[2] J. M. Howe, A. King, L. Lu, Analysing logic programs by reasoning backwards, in: Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development, Vol. 3049 of LNCS, Springer, 2004, pp. 152–188.

[3] R. Yang, P.-A. Heng, K.-S. Leung, Backward reasoning on rule-based systems modeled by fuzzy Petri Nets through backward tree, in: Computational Intelligence for Modelling and Prediction, Vol. 2 of Studies in Computational Intelligence, Springer, 2005, pp. 61–71.

[4] M. Z. Kwiatkowska, G. Norman, J. Sproston, F. Wang, Symbolic model checking for probabilistic timed automata, Inf. Comput. 205 (7) (2007) 1027–1077.

[5] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Sci. Comput. Program. 72 (1-2) (2008) 31–39, See also `www.eclipse.org/m2m/atl/`.

[6] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer, 2006.

[7] D. S. Kolovos, R. F. Paige, F. Polack, The Epsilon Transformation Language, in: ICMT, Vol. 5063 of LNCS, Springer, 2008, pp. 46–60.

[8] J. Cabot, R. Clarisó, E. Guerra, J. de Lara, A UML/OCL framework for the analysis of graph transformation rules, SoSyM 9 (3) (2010) 335–357.

[9] J. Cabot, R. Clarisó, D. Riera, On the verification of UML/OCL class diagrams using constraint programming, J. Syst. Soft. 93 (2014) 1–23.

[10] J. Cabot, R. Clarisó, E. Guerra, J. de Lara, Synthesis of ocl preconditions for graph transformation rules, in: ICMT, Vol. 6142 of LNCS, Springer, 2010, pp. 45–60.

[11] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, On challenges of model transformation from UML to Alloy, SoSyM 9 (1) (2010) 69–86.

[12] M. Kuhlmann, L. Hamann, M. Gogolla, Extensive validation of OCL models by integrating SAT solving into USE, in: TOOLS, Vol. 6705 of LNCS, Springer, 2011, pp. 290–306.

[13] R. Heckel, A. Wagner, Ensuring consistency of conditional graph rewriting - a constructive approach, Electr. Notes Theor. Comput. Sci. 2 (1995) 118–126.

[14] E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. of the ACM 18 (8) (1975) 453–457.

[15] E. Planas, J. Cabot, C. Gómez, Two basic correctness properties for ATL transformations: Executability and coverage, in: MtATL, Vol. 742, CEUR, 2011, pp. 1–9.

[16] C. A. González, J. Cabot, Formal verification of static software models in MDE: A systematic review, Inf. Soft. Technol. 56 (8) (2014) 821–838.

[17] A. Queralt, E. Teniente, Verification and validation of UML conceptual schemas with OCL constraints, ACM TOSEM 21 (2) (2012) 13:1–13:41.

[18] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, R. Drechsler, Verifying UML/OCL models using boolean satisfiability, in: DATE, IEEE, 2010, pp. 1341–1344.

[19] N. Bertrand, G. Delzanno, B. König, A. Sangnier, J. Stückrath, On the Decidability Status of Reachability and Coverability in Graph Transformation Systems, in: RTA, Vol. 15 of LIPIcs, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 101–116.

[20] M. Saksena, O. Wibling, B. Jonsson, Graph grammar modeling and verification of ad hoc routing protocols, in: TACAS, Vol. 4963 of LNCS, Springer, 2008, pp. 18–32.

[21] P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, A. Rezine, Monotonic abstraction for programs with dynamic memory heaps, in: CAV, Vol. 5123 of LNCS, Springer, 2008, pp. 341–354.

[22] S. Joshi, B. König, Applying the graph minor theorem to the verification of graph transformation systems, in: CAV, Vol. 5123 of LNCS, Springer, 2008, pp. 214–226.

[23] D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2006.

[24] A. Schürr, Specification of graph translators with triple graph grammars, in: WG, Vol. 903 of LNCS, Springer, 1994, pp. 151–163.

[25] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann, Theory of constraints and application conditions: From graphs to high-level structures, Fundamenta Informaticae 74 (1) (2006) 135–166.

[26] A. Habel, K.-H. Pennemann, Nested constraints and application conditions for high-level structures, in: Formal Methods in Soft. and Syst. Mod., Vol. 3393 of LNCS, 2005, pp. 293–308.

[27] A. Habel, K.-H. Pennemann, Correctness of high-level transformation systems relative to nested conditions, Math. Struct. Comp. Sci. 19 (2) (2009) 245–296.

[28] J. Winkelmann, G. Taentzer, K. Ehrig, J. M. Kuster, Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars, ENTCS 211 (2008) 159 – 170.

[29] F. Deckwerth, G. Varró, Attribute handling for generating preconditions from graph constraints, in: ICGT, Vol. 8571 of LNCS, Springer, 2014, pp. 81–96.

[30] G. Kniesel, H. Koch, Static composition of refactorings, Sci. Comput. Program. 52 (1-3) (2004) 9–51.

[31] B. Reus, M. Wirsing, R. Hennicker, A hoare calculus for verifying java realizations of ocl-constrained design models, in: FASE, Vol. 2029 of LNCS, Springer, 2001, pp. 300–317.

[32] K. R. Apt, F. S. de Boer, E. Olderog, S. de Gouw, Verification of object-oriented programs: A transformational approach, J. Comput. Syst. Sci. 78 (3) (2012) 823–852.

[33] G. T. Leavens, A. L. Baker, C. Ruby, JML: A notation for detailed design, in: Behavioral Specifications of Businesses and Systems, Vol. 523 of The Kluwer International Series in Engineering and Computer Science, Springer, 1999, pp. 175–188.

[34] D. Costal, C. Gómez, A. Queralt, E. Teniente, Drawing preconditions of operation contracts from conceptual schemas, in: CAiSE, Vol. 5074 of LNCS, Springer, 2008, pp. 266–280.

[35] A. Mammar, F. Gervais, R. Laleau, Systematic identification of preconditions from set-based integrity constraints, in: INFORSID, 2006, pp. 595–610.

[36] K. Lano, S. Kolahdouz-Rahimi, T. Clark, Comparing verification techniques for model transformations, in: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa '12, ACM, New York, NY, USA, 2012, pp. 23–28.

[37] K. Stenzel, N. Moebius, W. Reif, Formal verification of QVT transformations for code generation, in: MODELS, Vol. 6981 of LNCS, Springer, 2011, pp. 533–547.

[38] K. Born, T. Arendt, F. Heß, G. Taentzer, Analyzing conflicts and dependencies of rule-based transformations in henshin, in: FASE, Vol. 9033 of LNCS, Springer, 2015, pp. 165–168.

[39] M. Gogolla, M. Richters, Expressing UML class diagrams properties with OCL, in: OCL, Vol. 2263 of LNCS, Springer, 2002, pp. 85–114.

[40] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow, Model transformations? Transformation models!, in: MODELS, Vol. 4199 of LNCS, Springer, 2006, pp. 440–453.

[41] H.-J. Kreowski, R. Klempien-Hinrichs, S. Kuske, Some essentials of graph transformation, in: Recent Advances in Formal Languages and Applications, Vol. 25 of Studies in Computational Intelligence, Springer, 2006, pp. 229–254.

A. Appendix: Formal proof

A.1. Outline of the proof

The goal of this appendix is establishing the correctness of the procedure for advancing postconditions, i.e. that the OCL constraint provided as a result is indeed the weakest precondition.

In the context of rule-based model transformation, being the weakest precondition means that evaluating the precondition before applying the rule yields the same result as evaluating the postcondition after applying the rule. The proof will focus on establishing this equivalence for each type of atomic action in a model transformation rule, considering all the replacement patterns defined in the paper.

For each type of atomic action a, the proof uses *structural induction* over the syntax tree of the OCL expression being advanced ($Post$):

Step 1 (base case): Prove that subexpressions of $Post$ which do not match any replacement pattern for a are unaffected by the action and, hence, evaluate to the same result in the precondition.

Step 2 (inductive step): Consider the innermost subexpression e of $Post$ matching a replacement pattern for action a. By structural induction, the advanced subexpressions of e evaluate to the same result in the precondition. Then, we prove that the replacement of e also evaluates to the same result in the precondition.

Section A.2 will provide definitions used throughout the proof. Section A.3 will establish the correctness for the patterns for link creation and deletion. Section A.4 considers the replacement patterns for attribute updates. Section A.5 discusses the patterns for object deletion. Then, Section A.6 studies the patterns creation of objects. This proof is more involved, as the process for applying matches is iterative and it is necessary to discuss, for instance, the termination of the procedure. Finally, Section A.7 concludes by considering the correctness of the overall procedure, i.e. the composition of different atomic actions.

A.2. Definitions

A meta-model MM is a tuple $MM = \langle CLASS, ATT, ASSOC, \prec \rangle$ where:

- $CLASS$ is a finite set of *class* names.

- ATT is a set of *attributes* for each class. Each attribute is a triple $\langle at, c, t\rangle$ where at is an attribute name, $c \in CLASS$ and t is the type of the attribute (either Integer, Real, Boolean or String).
- $ASSOC$ is a set of *binary associations*. Each association is a triple $\langle as, c_1, c_2, r_1, r_2, m_1, m_2\rangle$ where as is a unique association name and $c_1, c_2 \in CLASS$, r_1 and r_2 are distinct *role names* and m_1 and m_2 are sets of natural numbers defining *multiplicities*.
- $\prec$: $CLASS \rightarrow CLASS$ is the *generalization hierarchy*, a partial function mapping each class to its superclass.

This definition is a simplification of the formal semantics proposed in the OCL specification as the models under consideration do not include multiple inheritance, n-ary associations or associative classes. Note that features such as associative classes, non-recursive queries and n-ary associations can be simplified into an equivalent model with additional OCL constraints [39].

A model M conforming to a meta-model MM ($M \models MM$) is an instance of the meta-model defined as $\langle Oid, Obj, Att, Links\rangle$ where

- Oid is the set of unique object identifiers participating in the model.
- $Obj : oid \rightarrow CLASS$ is a mapping defining the base type of each object.
- $Att : oid \times \langle at, c, t\rangle \rightarrow domain(t)$ is a mapping assigning a value of the suitable type to each attribute.
- $Link$ is the set of links, a set of pairs (oid_1, oid_2) such that classes $Obj(oid_1)$ and $Obj(oid_2)$ belong to a binary association in MM and the set of links respect the multiplicities for each role.

A model transformation, denoted as $M \Rightarrow M'$, converts a *source* model M into a *target* model M'. Source and target may conform to the same meta-model (*in-place* transformation) or to different meta-models (*exogenous* transformation). For exogenous transformations, it is possible to define a *transformation model* [40] which includes both the source and target meta-models. Hence, without loss of generality, we can consider that there is a common meta-model for source and target models (which will be the transformation model in the case of exogenous transformations).

Model transformations consist of a set of *rules*. A rule defines an *enabling condition*, a set of patterns and/or constraints that need to be satisfied for the rule to be applicable. We denote with $M \models \mathit{enabling}(r)$ the fact that rule r is enabled in a model M. A rule can be enabled in different submodels of a model M: o denotes the particular occurrence where the rule is enabled and $M \overset{r,o}{\Rightarrow} M'$ denotes the application of rule r in occurrence o of model M, producing a model M'. The rule transforms the source model by performing a sequence $\langle a_1, \ldots, a_n \rangle$ of *atomic actions*. Actions can be of four types: deletion of an object (and adjacent links); deletion of a link among two objects; creation of a link among two objects; update of an attribute value; or creation of an object (and adjacent links).

An *OCL constraint* is a boolean expression defined over the elements of the meta-model MM, which can be evaluated in concrete models yielding `true` or `false`. The structure of an OCL constraint is defined according to the following grammar, which is a subset of the full OCL 2.4 grammar considering the limitations of our approach described in Section 2.2.7:

$$
\begin{array}{rcl}
\text{OclConstraint} & ::= & \text{Exp} \\
\text{Exp} & ::= & \text{VariableExp} \mid \text{IsOfExp} \mid \text{IfExp} \mid \\
 & & \text{LoopExp} \mid \text{LiteralExp} \mid \text{NavigationCallExp} \mid \\
 & & \text{AttributeCallExp} \mid \text{OperationCallExp} \mid \text{TypeExp} \\
\text{BaseType} & ::= & \texttt{Integer} \mid \texttt{Real} \mid \texttt{Boolean} \mid \texttt{String} \mid \texttt{identifier} \\
\text{TypeExp} & ::= & \text{BaseType} \mid \texttt{Set}\ (\ \text{BaseType}\ ) \\
\text{VariableExp} & ::= & \texttt{identifier} \\
\text{IsOfExp} & ::= & \text{Exp} \ {-}{>}\ \ \text{IsOfOp}\ (\ \text{TypeExp}\ ) \\
\text{IsOfOp} & ::= & \texttt{oclIsKindOf} \mid \texttt{oclIsTypeOf} \\
\text{IfExp} & ::= & \texttt{if}\ \text{Exp}\ \texttt{then}\ \text{Exp}\ \texttt{else}\ \ \text{Exp}\ \texttt{endif} \\
\text{LoopExp} & ::= & \text{Exp}\ {-}{>}\ \text{IteratorName}\ (\ \text{VariableExp} \mid \text{Exp}\ ) \\
\text{IteratorName} & ::= & \texttt{exists} \mid \texttt{forAll} \mid \texttt{isUnique} \mid \texttt{select} \mid \texttt{reject} \mid \\
 & & \texttt{any} \mid \texttt{collectNested} \\
\text{LiteralExp} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{string} - \texttt{literal} \mid \texttt{int} - \texttt{literal} \mid \\
 & & \texttt{float} - \texttt{literal} \mid \text{SetLiteral} \\
\text{SetLiteral} & ::= & \texttt{Set}\{\ \text{ExpList}\ \} \\
\text{ExpList} & ::= & \text{EmptyExpList} \mid \text{NonEmptyExpList} \\
\text{EmptyExpList} & ::= & \\
\text{NonEmptyExpList} & ::= & \text{Exp} \mid \text{Exp}\ ,\ \text{NonEmptyExpList} \\
\text{NavigationCallExp} & ::= & \text{Exp}\ .\ \texttt{identifier} \\
\text{AttributeCallExp} & ::= & \text{Exp}\ .\ \texttt{identifier} \\
\text{OperationCallExp} & ::= & \text{Exp}\ .\ \texttt{identifier}\ (\ \text{ExpList}\ )
\end{array}
$$

An OCL constraint can be seen as an abstract syntax tree (AST) adhering to this grammar. In this AST, each subtree is an OCL subexpression, internal nodes represent operations or quantifiers and leaves are literal values or references to model elements. Given an OCL expression exp, $eval(exp, M)$ denotes the result of evaluating expression exp in a model M.

A.3. Proof for link creations and deletions

Notation: In this section, we consider a meta-model MM, which includes two classes A and B and an association $Assoc$ among them, with role names ra and rb respectively. Then, we consider two models M_1 and M_2

such that $M_1, M_2 \models MM$, and M_2 is obtained from M_1 by adding a link in association $Assoc$, connecting two objects a (of type A) and b of type B.

Lemma 2. *Let C be an OCL constraint over MM. If C contains no navigations through association Assoc, then $eval(C, M_1) = eval(C, M_2)$.*

Proof 6. *(By cases) In the subset of the OCL language under consideration, there are several types of expressions. The value of navigation expressions (type AssociationEndCallExp) is a collection which will include the objects in the created link. However, any other OCL expressions are unaffected by the change: expressions with a constant value (LiteralExp and its subtypes); attribute accesses (AttributeCallExp); variables (VariableExp), as there are no associative classes; composite expressions such as if-then-else (IfExp) or iterators/quantifiers (LoopExp); and operations (OperationCallExp), which include arithmetic, boolean and relational operators. Some other features, like non-recursive queries (also OperationCallExp) and "let" expressions (LetExp) can be expanded and removed. Finally, other features like messages (OclMessageExp) and association classes (AssociationClassCallExp) are not included in our OCL subset. Thus, if an expression has no navigation expressions, its value will be unmodified by the creation of links.* □

Lemma 3. *Let exp be an OCL expression over MM, whose result data type is A or Set(A), and such that $eval(exp, M_1) = eval(exp, M_2)$. Then:*

1. *$eval(a.rb, M_2) = eval(exp', M_1)$*
   where exp' is (a.rb–>including(b)).
2. *$eval(exp.rb, M_2) = eval(exp', M_1)$*
   where exp' is
   *(**if** exp–>includes(a) **then** exp.rb–>including(b)*
   ***else** exp.rb **endif**).*

Proof 7. *(Direct proof)*

(1) Evaluating a.rb in M_2 yields a set of objects which includes b. The same result can be computed in M_1 by explicitly adding b to the result of a.rb. □

(2) The evaluation of expression exp in M_1 may include a or not. If it does, then b should be added to the result of the navigation. Otherwise, the result should be the same as in M_2. □

The proof for navigations in the opposite direction, from B to A using role name ra, uses the same argument.

Lemma 4. *Let C be an OCL expression over model MM. Then, the application of the replacement rules for link creation (from Table 4) in a bottom-up progression in the abstract syntax tree of C produces a constraint C' such that $eval(C', M_1) = eval(C, M_2)$*

Proof 8. *(By structural induction) If C does not contain any navigation expression, then it will not be modified by the replacement patterns ($C' = C$). Also, by Lemma 2, we have that $eval(C, M_1) = eval(C, M_2)$. Therefore, this Lemma holds for the base case where there are no navigation expressions.*

*If there are navigation expressions, let exp.role be the innermost navigation expression in C. Being the innermost one, its subexpression exp will have no navigations and due to Lemma 2, $eval(exp, M_1) = eval(exp, M_2)$. The expression exp.role will be replaced by the replacement pattern exp' = (**if** exp−>includes(a) **then** exp.rb−>including(b) **endif**). Due to Lemma 3, this expression exp' ensures that $eval(exp', M_1) = eval(exp, M_2)$. By induction, the same reasoning can be applied to the closest navigation expression enclosing this one, as its subexpression will satisfy the condition of Lemma 3, that the subexpression has the same value before and after creating the link.* □

A similar reasoning can be used to prove the replacement patterns for the deletion of links. It should be noted that it is not sufficient to change the pattern from "including(b)" to "excluding(b)". This is sufficient for the case when the subexpression is a single object: if it is a, the removal of the link is simulated by the "excluding(b)" clause. However, if the source is a set of objects of type A, it may happen that another object in the set other than a is connected to b both in M_1 and M_2. Adding "excluding(b)" to this expression would have the unintended side-effect of removing that link as well. Thus, the replacement pattern becomes more complex: it is necessary to distinguish among objects navigated from a (where the clause "excluding(b)" should be appended) and the navigations of the remaining objects, whose navigation should be unchanged. This is the motivation behind the replacement pattern used for sets:

$$(exp{-}{>}\ excluding(a)).rb{-}{>}\ union(a.rb{-}{>}\ excluding(b))$$

A.4. Proof for attribute updates

Notation: In this section, we consider a meta-model MM, which includes a class A with an attribute at. Then, we consider two models M_1 and M_2 with $M_1, M_2 \models MM$, such that M_2 is obtained from M_1 by changing the value of an attribute at of an object a of type A to a new constant value val.

Lemma 5. *Let C be an OCL constraint over MM. If C contains no accesses to the value of attribute at, then $eval(C, M_1) = eval(C, M_2)$.*

Proof 9. *(By cases) Only expressions of type AttributeCallExp access the value of an attribute. If there are no attribute access expressions involving attribute at of type A (or attribute at in a superclass of A), the expression will evaluate to the same result in M_1 as in M_2.* □

Lemma 6. *Let exp be an OCL expression defined over model M, with result type A and such that $eval(exp, M_1) = eval(exp, M_2)$. Then:*

1. *$eval(a.at, M_2) = eval(val, M_1)$.*
2. *$eval(exp.at, M_2) = eval(exp', M_1)$*
   where exp' is
   *(**if** exp = a **then** val **else** exp.at **endif**).*

Proof 10. *(Direct proof)*

(1) Evaluating a.at in M_2 yields the value val (by the definition of M_2). This is equivalent to evaluating the constant value val in M_1. □

(2) The evaluation of expression exp in M_1 may be equal to a or not. If it is equal, the result should be val and otherwise, it should provide the same result as in M_2 (exp.at). This is simulated with the if-then-else expression, which therefore produces the same result as evaluating exp.at in M_2. □

Lemma 7. *Let C be an OCL expression over model M. Then, the application of the replacement rules for attribute updates (from Table 2) in a bottom-up progression in the abstract syntax tree of C produces a constraint C' such that $eval(C', M_1) = eval(C, M_2)$.*

Proof 11. *(By structural induction) If C does not contain any attribute access, then it will not be modified by the replacement patterns ($C' = C$). Also, by Lemma 5, we have that $eval(C, M_1) = eval(C, M_2)$. Therefore, this Lemma holds for the base case where there are no attribute accesses.*

*If there are attribute access expressions, let exp.at be the innermost attribute access in C. Being the innermost one, its subexpression exp will have no attribute accesses and due to Lemma 5, $eval(exp, M_1) = eval(exp, M_2)$. This expression will be replaced by the replacement pattern $exp' \equiv$ (**if** exp = a **then** val **else** exp.at **endif**). Due to Lemma 6, this expression exp' ensures that $eval(exp', M_1) = eval(exp, M_2)$. By induction, the same reasoning can be applied to the closest attribute access enclosing this one, as its subexpression will satisfy the condition of Lemma 6, that the subexpression has the same value before and after updating the attribute.* □

A.5. Proof for object deletions

Notation: In this section, we consider a meta-model MM, which includes a class T, and an object x of type T. Then, we consider two models M_1 and M_2 with $M_1, M_2 \models MM$, such that M_2 is obtained from M_1 by removing object x and all links where x participates.

Lemma 8. *Let C be an OCL expression. If there are no navigation expressions to type T (or its subclasses), and the method "allInstances()" is not invoked on class T (or its sub and superclasses), then $eval(C, M_1) = eval(C, M_2)$.*

Proof 12. *(By cases) A difference in $eval(C, M_1)$ and $eval(C, M_2)$ would mean that the evaluation of C takes into account the object x or one of its links. This may happen in one of the following ways:*

- *The links between x and other objects can be accessed only through navigation expressions, as discussed in Lemma 2.*
- *x can also be accessed through the set of instances of T, its subtypes or supertypes. Therefore, any call to "allInstances()" (OperationCallExp) in one of those types will produce a collection that includes x in M_1 and not in M_2.*
- *C may refer directly to an object using an identifier (VariableExp) from the RHS of the rule. Only the RHS is available because C is the post-condition being advanced, thus, it can only use identifiers of objects which exist after applying the rule. Then, as x is deleted by the rule, its identifier does not appear in the RHS so it is not possible to use its identifier in C.*

- *Local variables introduced by iterators (also of type VariableExp) may only refer to x if the collection over which they are defined contains x. Thus, their value will not depend on the existence of x unless x is already present in the subexpression computing the collection.*

- *Other expressions are either constants (LiteralExp) or composite expressions like if-the-else (IfExp), iterators (LoopExp) and all operators (OperationCallExp), whose value may depend on x only if one of its subexpressions depends on x.*

Therefore, only the operation "allInstances()" and navigation expressions may be affected by the deletion of x. Any expression which has none of these elements will evaluate to the same value with or without x. □

Lemma 9. *Let C be an OCL navigation expression of the form exp.roleT, with roleT being an association end of class T, and exp such that $eval(exp, M_1) = eval(exp, M_2)$. Then $eval(C, M_2) = eval(C{-}{>}\ excluding(x), M_1)$.*

Proof 13. *(Direct proof) The value of exp.roleT in M_1 and M_2 can only differ in the presence (or not) of x in M_1. Adding "excluding(x)" to the expression will make the result equal whether x appears in M_1 ("excluding" will remove it, making it equal to the result in M_2) or not ("excluding" will not modify the result, which was already equal to the result in M_1).* □

Lemma 10. *Let C be an OCL expression over MM of the form $A.allInstances()$, with A being either type T or one of its subtypes or supertypes. Then $eval(C, M_2) = eval(C{-}{>}\ excluding(x), M_1)$.*

Proof 14. *(Direct proof) The result of C does not include x in M_2 (as x is deleted). On the other hand, if A is type T or a supertype, then the result of C will include x in M_1. Similarly, if A is a subtype of T, then the result of C* may *include x in M_1. Other than the presence of x, the collection resulting from M_1 and M_2 is equal. Therefore, appending "excluding(x)" in M_1 yields the same result in M_1 and M_2.* □

Lemma 11. *Let C be an OCL expression over the meta-model MM. Then, the application of the replacement rules for object deletion (from Table 1) in a bottom-up progression in the abstract syntax tree of C produces a constraint C' such that $eval(C', M_1) = eval(C, M_2)$.*

Proof 15. *(By structural induction) The proof follows the argument of the previous Lemmas 4 and 7. The base case where there are no matches for the replacement pattern satisfies this Lemma, as proved by Lemma 8. Expressions with matches for the replacement rules are analyzed recursively using the results of Lemmas 9 and 10. As a result, the property is proved inductively.* □

A.6. Proof for object creation

Notation: In this section, we consider a meta-model M, which includes a class B and an object b of type B. Then, we consider two models M_1 and M_2 with $M_1, M_2 \models MM$, such that M_2 is obtained from M_1 by creating the object b and some links among b and other preexisting objects. Let $Assoc$ be the set of associations where one of these links is created.

Lemma 12. *Let C be an OCL constraint that does not contain any call T.allInstances (where T is B or a supertype of B) nor any navigation to an object of type B (or a subtype) through any association in Assoc, nor any direct reference to identifier b. Then $eval(C, M_1) = eval(C, M_2)$.*

Proof 16. *(By cases) In order for C to have a different value, it is necessary to consider the new object b in some way. A first potential way of accessing b is through a variable (VariableExp). In these expressions, it is only necessary to consider variable references that directly use the identifier b: as b is created by the rule, other objects in the LHS cannot be matched with b, even if its types are compatible [41]. Therefore, it is only necessary to watch for occurrences of the identifier b while other identifiers or variables cannot be referring to b.*

Aside from direct references through the identifier b, it is still possible to access it in two ways:

- *A call to allInstances on type B or a supertype will include b. It is not necessary to consider subtypes of type B, as the creation of an object in a GT rule needs to specify the base type of the object (thus, if the object is created in type B it cannot be an instance of any of its subtypes).*
- *A navigation that includes b in the result: in this case, the navigation should occur in one of the associations in Assoc.*

Therefore, any constraint that does not include these expressions will remain unaffected by the creation of a new object. □

Lemma 13 (Completeness). *Replacement rules from Tables 5 and 6 consider any possible expression where a reference to b may appear.*

Proof 17. *(By cases) A reference to b may appear in expressions whose type is either T (Table 5) or Set(T) (6), with T being B or one of its subclasses or superclasses.* □

Lemma 14 (Termination). *The iterative application of the rules from Tables 5 and 6 terminates after a finite number of iterations, producing an OCL constraint with no references to b.*

Proof 18. *(By cases) The application of the replacement patterns from Tables 6 and 5 requires finding a matching expression referencing the identifier b. We will show that the iterative application of these replacement patterns eventually reduces the number of occurrences of the identifier b in the OCL expression.*

These replacement patterns can be classified into three categories:

1. *Patterns that delete a reference to b, e.g. by replacing the expression by another one without b such as a constant [C1-3,C11-13,C15-19,O1,O3-4,O6-O11].*
2. *Patterns that move a reference to b upward inside the syntax tree, either by simplifying the expression or changing the order of the operations [C14, O2, O5, O12].*
3. *Patterns that expand a quantifier, replicating its body expression and replacing the quantified variable by b [C4-C10].*

Replacement patterns of type 3 are the only ones that can introduce new references to b. The number of times they can be applied is bounded by the number of quantifiers in the expression, so the number of references to b that can be introduced in the OCL expression is bounded a priori. Furthermore, other replacement rules cannot introduce new quantifiers in the OCL expression.

Regarding patterns of type 2, they can only be applied a finite number of times, as their application moves the reference to b one level higher in the abstract syntax tree and the height of the abstract syntax tree of the expression is finite. Due to Lemma 13, these patterns cover every possible expression where may b appear. Therefore, as patterns of type 2 can only be applied a finite number of times, eventually, they will lead to an expression where a replacement pattern of type 1 or 3 can be applied.

Finally, replacement patterns of type 1 can only be applied as long as there are occurrences of b in the OCL expression: each time one of them is applied, the reference to b disappears. Hence, a pattern of type 1 may not enable any other replacement pattern.

To sum up, all patterns can be applied a finite number of times: type 3 patterns are bounded by the number of quantifiers in the expression, type 2 patterns by the height of the abstract syntax tree and type 1 patterns by the number of references to the object b. □

Due to space concerns, a complete proof of the correctness for each individual replacement pattern in Tables 5 and 6 is out of the scope of this proof. Intuitively, by examining each pattern it is possible to determine that the replacement expression is equivalent to the original expression: they both produce an expression of the same type and evaluating to the same value. Let us consider the proof of three patterns to illustrate the process.

- **Pattern C2**:
  "col−>including(b)−>isEmpty()" becomes "false".

  The original expression and the replacement are of type boolean. Given that we assume that the collection "col" will include at least object b, then the result of the query isEmpty() will always be false. Thus, the original expression is equivalent to the replacement "false".

- **Pattern C4**:
  "col−>including(b)−>exists(x | exp)" becomes "col−>exists(x | exp) **or** $Inst[x, exp]$".

  The original expression and the replacement are both of type boolean. In order to preserve the semantics of the original expression, the resulting expression should evaluate to true if and only if (a) either the new object satisfies the quantified subexpression *exp* or (b) the collection contains an object other than b which satisfies the expression. The proposed translation formalizes this intended semantics.

- **Pattern O10**:
  "$b = exp$" becomes "true" if $exp = b$ and "false" otherwise.

  The original expression and the replacement are both of type boolean. Using the proposed order of application for replacement patterns (Subsection 2.2.6), *exp* should either be the identifier b or an expression whose subexpressions do not include any reference to b (as all matches in subexpressions have been applied) nor any quantified variable that could be assigned to b (as any quantifier enclosing "$b = exp$" has been processed before). Given these assurances and given that b is a new object, the only way that exp could evaluate to b is if it is b itself, otherwise it will necessarily be different.

A.7. Composition of atomic actions

Notation: In this section, let MM be a meta-model, C be an OCL constraint over MM and a_1 and a_2 two atomic actions. Moreover, let M_1 and M_2 be two models of MM, such that (i) the atomic actions a_1 and a_2 can be applied to M_1 (e.g. objects to be deleted exists) and (ii) M_2 is computed from M_1 by applying the atomic actions a_1 and a_2 in sequence. Furthermore, let us denote as $adv(a, C)$ the OCL constraint computed by advancing post-condition C considering the atomic action a.

Definition 7 (Non-overlapping actions). *A set of atomic actions are non-overlapping if and only if:*

- *If one action is the creation/deletion of a link, no other event may create/delete the same link or create/delete the objects participating in the link.*
- *If one action is the modification of an attribute value, no other event may modify the same attribute or create/delete the object where it belongs.*
- *If one action is the deletion of an object, no other event may create/delete the same object, modify its attributes or create/delete a link to that object.*
- *If one event is the creation of an object, no other event may create/delete the same object, modify its attributes or delete a link to that object. However, other object creation actions may create new links to this object.*

As an example, the set of atomic actions computed from a graph transformation rule is non-overlapping. This is due to the semantics and well-formedness rules of GT rules [6]: it is not possible to define a rule which creates and destroys the same object. The rationale behind each condition in the definition lies in the formalisation of the atomic actions: object creation includes the initialization of all attributes and the creation of links where the object participates and object deletion includes the deletion of all links where it participates. The last condition of this definition allows GT rules that create, for example, two new objects *and* a link among them: the first action will create the first object while the second one will introduce the object and the link.

Lemma 15 (Concurrent application). *If M_2 is computed by applying non-overlapping actions a_1 and a_2 to M_1, the result of applying a_2 and a_1 to M_1 is also M_2.*

Proof 19. *(By contradiction) Let us assume that $a_2(a_1(M_1))$ and $a_1(a_2(M_2))$ differ. This difference can be located in:*

- *The value of an attribute at in an object x: this can only happen if a_1 and a_2 are both attribute updates for $x.at$ or if a_1 destroys x while a_2 creates it. In both scenarios, a_1 and a_2 would be overlapping according to Definition 7, contradicting our premise that they were non-overlapping.*
- *The existence of an object x: this can only happen if a_1 deletes the object while a_2 creates it or vice versa (otherwise, x would be present or missing in both instances). This contradicts our premise that a_1 and a_2 are non-overlapping.*
- *The existence of a link $x - y$: similarly, this can only happen if either of the actions deletes the link while the other creates it, or if one event destroys x or y and the other re-creates the destroyed object. In either case, a_1 and a_2 would need to be overlapping.*

□

The conclusion of this Lemma is that the order of application of atomic actions (as they are defined in this paper) is irrelevant as long as they are non-overlapping.

Lemma 16 (Composition of non-overlapping atomic actions). *For any OCL constraint C defined over a metamodel MM, models M_1 and M_2 such* *that $M_1, M_2 \models MM$ and non-overlapping actions a_1 and a_2 over model M_1 such that $a_1(a_2(M_1)) = M_2$, the following holds:*

$$\begin{aligned} eval(adv(a_2, adv(a_1, C)), M_1) &= \\ eval(adv(a_1, adv(a_2, C)), M_1) &= \\ eval(C, M_2) & \end{aligned}$$

Proof 20. *Let us consider that the actions a_1 and a_2 are applied in this order: $M_1 \overset{a_1}{\Rightarrow} M' \overset{a_2}{\Rightarrow} M_2$, where M' is the intermediate model. Lemma 15 and a similar argument can be used for the reverse ordering of actions.*

Previous lemmas of the proof ensure that the weakest precondition $adv(a_i, C)$ is correct for individual atomic actions a_i of each type, i.e. attribute updates (Lemma 7), link creations and deletions (Lemma 4), object deletions (Lemma 11) and object creations (Section A.6). Thus, we know that the weakest precondition for action a_2 is correct: $eval(adv(a_2, C), M') = eval(C, M_2)$.

For the same reason, we know that for any constraint C', the following holds: $eval(adv(a_1, C'), M_1) = eval(C', M')$. In particular, this will hold for $C' = adv(a_2, C)$. Replacing C' with this constraint in the previous equality yields: $eval(adv(a_1, adv(a_2, C)), M_1) = eval(adv(a_2, C), M')$.

According to our first argument, $eval(adv(a_2, C), M') = eval(C, M_2)$, so we have proved that $eval(adv(a_1, adv(a_2, C)), M_1) = eval(C, M_2)$. □

The previous results can be combined to prove the overall correctness of the procedure for weakest precondition synthesis.

Theorem 6 (Correctness of weakest precondition). *Let MM be a metamodel, p a model transformation rule causing atomic actions $\langle a_1, a_2, \ldots, a_n \rangle$, and an OCL constraint C over MM. Let M_1 and M_2 be two models with $M_1, M_2 \models MM$, such that $M_1 \models enabling(p)$ and $M_1 \overset{p}{\Rightarrow} M_2$. Then, $C' = adv(a_1, adv(a_2, \ldots adv(a_n, C)))$ satisfies that $eval(C', M_1) = eval(C, M_2)$.*

Proof 21. *This theorem is the generalization of Lemma 16 to sequences of n atomic actions. The same argument used in the proof of Lemma 16 for pairs of actions can be extended to this more general case.* □