# Analysing Graph Transformation Through OCL

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

[1] Universitat Oberta de Catalunya (Spain), {jcabot,rclariso}@uoc.edu
[2] Universidad Carlos III de Madrid (Spain), eguerra@inf.uc3m.es
[3] Universidad Autónoma de Madrid (Spain), jdelara@uam.es

**Abstract.** In this paper we present an approach to the analysis of graph transformation rules based on an intermediate OCL representation. We translate different semantics of rules into OCL, together with the properties of interest (like rule applicability, conflict or independence). The intermediate representation serves three purposes: (i) allows the seamless integration of graph transformation rules with the MOF and OCL standards, and enables taking into account meta-model and OCL constraints when verifying the correctness of the rules; (ii) permits the interoperability of graph transformation concepts with a number of standards-based model-driven development tools; and (iii) makes available a plethora of OCL tools to actually perform the rule analysis.

## 1 Introduction

Model-Driven Development (MDD) is a software engineering paradigm where models play a fundamental role. They are used to specify, simulate, test, verify and generate code for the application to be built. Most of these activities are model manipulations, thus, model transformation becomes a crucial activity. Many efforts have been spent in designing specialized languages for model transformation, ranging from textual to visual; declarative to imperative through hybrid; semi-formal to formal. The OMG vision of MDD is called Model-Driven Architecture (MDA) and is founded on standards like QVT [17] for the transformations and MOF and OCL for modelling and meta-modelling.

Graph Transformation [8, 18] is a declarative, rule-based technique for expressing model transformations. It has been mainly used for specifying in-place transformations like animations [10], simulations [14], optimizations and redesigns [16]. It is gaining increasing popularity due to its visual form (making rules intuitive) and formal nature (making rules and grammars amenable to analysis). For example, it has been used to describe the operational semantics of Domain Specific Visual Languages (DSVLs) [14], taking the advantage that it is possible to use the concrete syntax of the DSVL in the rules, which then become more intuitive to the designer.

As models and meta-models can be expressed as graphs (with typed, attributed nodes and edges), graph transformation can be used for model manipulation. The main formalization of graph transformation is the so called algebraic approach [8], which uses category theory in order to express the rewriting. Prominent examples of this approach are the double [8] and single [9] pushout (DPO

and SPO), which have developed interesting analysis techniques, e.g. to check independence between pairs of derivations [8, 18], or to calculate critical pairs (minimal context of pairs of conflicting rules) [12]. However, graph grammar analysis techniques work with simplified meta-models (so called type graphs), with no inheritance, cardinalities nor textual OCL-like constraints.

In this paper, our goal is to advance in the integration of graph transformation and MDD. We propose using OCL as an intermediate representation of both the semantics of graph transformation rules and the analysis properties of interest. Representing rules with OCL, concepts like attribute computation and attribute conditions in rules can be seamlessly integrated with the meta-model and OCL constraints during the rule analysis. Specifying the rules and the properties in OCL makes available a plethora of tools (e.g. HOL-OCL [4], USE [11], MOVA [7], UML2Alloy [1] and UMLtoCSP [6]), able to analyze this kind of specifications. A secondary effect is that graph transformation is made available to the increasing number of MDA tools that the community is building and vice-versa. For example, using such tools, it could be possible to generate code for the transformations, or apply metrics and redesigns to the rules.

More in detail, we use OCL to represent (DPO/SPO) rules with negative application conditions and attribute conditions. These rules may have objects with abstract typing, which can be matched to objects of more concrete types [13]. In addition, we have represented a number of analysis properties with OCL, taking into account both the rule structure and the rule and meta-model constraints: rule applicability (whether there is a model satisfying the rule and the meta-model constraints), weak executability (whether the rule's post-condition and the meta-model constraints are satisfiable by some model), strong executability (whether all models satisfying the rule's pre-condition and the meta-model constraints allow a valid rewriting step), correctness preserving (if a rule applied to a legal model always yields a legal model), overlapping rules (whether there is a model in which both rules are applicable), conflicts (there are two applicable rules on the same model, and firing one disables the other), and rule independence (check if the application order of two rules matters). As a proof of concept, we have checked these properties using the UMLtoCSP tool, which internally treats the OCL expressions as a Constraint Satisfaction Problem.

**Paper Organization.** Section 2 introduces graph transformation using a production system example. Section 3 presents our translation of graph transformation rules into OCL. Section 4 shows the encoding of some analysis properties. Section 5 presents the use of the UMLtoCSP tool for checking some properties. Section 6 compares with related work and Section 7 ends with the conclusions.

## 2 Graph Transformation by Example

In this section we give an intuition on graph transformation by presenting some rules that belong to a simulator of a DSVL for production systems. Fig. 1 shows the DSVL meta-model. It defines different kinds of machines (concrete subclasses of *Machine*) that can be connected through conveyors. These can be intercon-

nected and contain pieces (the number of pieces they actually hold is stored in attribute *nelems*), up to its maximum capacity (attribute *capacity*). The last two OCL constraints to the right of the figure guarantee that the number of elements of a conveyor is equal to the number of pieces connected to it and never exceeds its capacity. Human operators are needed to operate the machines, which consume and produce different types of pieces from/to conveyors.

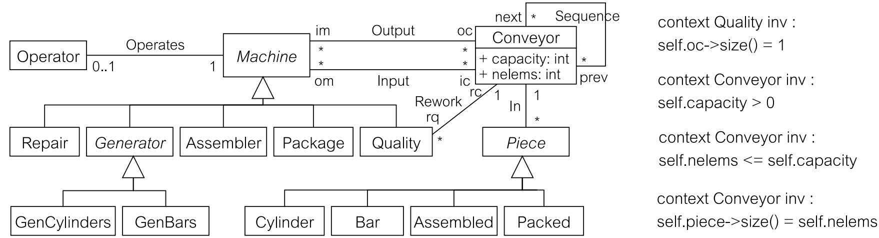


**Fig. 1.** Meta-model of a DSVL for production systems.

Fig. 2 shows a production model example using a visual concrete syntax. It contains six machines (one of each type), two operators, six conveyors and five pieces. Machines are represented as decorated boxes, except generators, which are depicted as semi-circles with an icon representing the kind of piece they generate. Operators are shown as circles, conveyors as lattice boxes, and each kind of piece has its own shape. In the model, the two operators are currently operating a generator of cylindrical pieces and a packaging machine respectively. Even though all associations in the meta-model are bidirectional, we have assigned arrows in the concrete syntax, but of course this does not affect navigability.

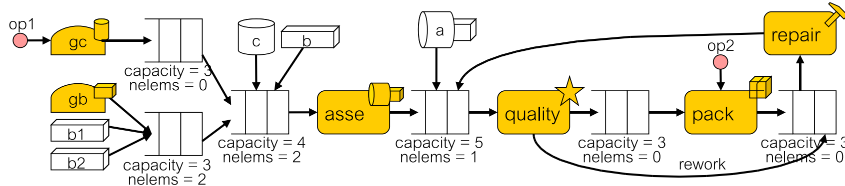


**Fig. 2.** Example production system model.

We use graph transformation techniques for the specification of the DSVL operational semantics. A graph grammar is made of a set of rules and an initial graph to which the rules are applied. Each rule is made of a left and a right hand side (LHS and RHS) graphs. The LHS expresses pre-conditions for the rule to be applied, whereas the RHS contains the rule's post-conditions. In order to apply a rule to the *host graph*, a morphism (an occurrence or match) of the LHS has to be found in it (if several are found, one is selected randomly). Then, the rule is applied by substituting the match by the RHS. The grammar execution proceeds by applying the rules in non-deterministic order, until none is applicable.

Next, we show some of the rules describing the DSVL operational semantics. Rule "assemble" specifies the behaviour of an assembler machine, which converts one cylinder and a bar into an assembled piece. The rule is shown in concrete syntax to the left of Fig. 3, and in abstract syntax to the right. It can be applied if an occurrence of the LHS is found in the model (e.g. it could be applied to the model in Fig. 2). Then, the elements in the LHS that do not appear in the RHS are deleted, whereas the elements in the RHS that do not appear in the LHS are newly created. Our rules may include attribute conditions (which must be satisfied by the match) and attribute computations, both expressed in OCL. Attributes referenced to the right of an assignment in an attribute computation refer to the value of the attribute before the rule application.

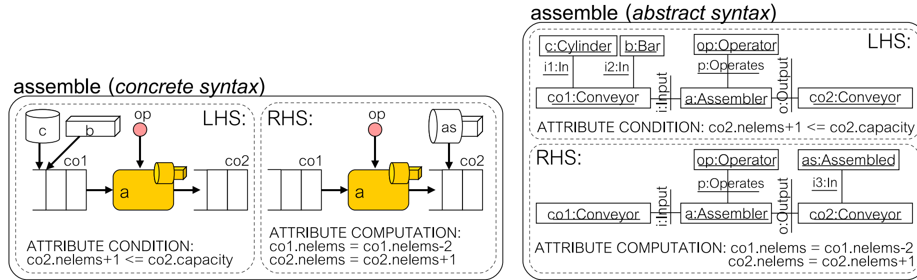


**Fig. 3.** Assemble rule in concrete syntax (left) and abstract syntax (right).

There are two main formalizations of algebraic graph transformation, DPO and SPO. From a practical point of view, their difference is that in DPO, deletion has no side effects. When a node in the host graph is deleted by a rule, it can only be connected with those edges explicitly deleted by the rule. When applying the rule in Fig. 3, if piece "b" in the match is connected to more than one conveyor (should it be allowed by the meta-model), then the rule cannot be applied as edges would become dangling in the host graph. This condition is called *dangling edge* condition. In SPO, dangling edges are removed by the rewriting step. Therefore in DPO, in addition to positive pre-conditions, a LHS also imposes implicit negative pre-conditions in case the rule deletes some node.

A match can be non-injective, which means for example that two nodes with compatible type in the rule may be matched to a single node in the host graph. If the rule specifies that one of them should be deleted and the other one preserved, DPO forbids applying the rule at such match, while SPO allows its application and deletes both nodes. This is called the *identification condition*.

Fig. 4 shows further rules for the DSVL. Rule "move" describes the movement of pieces through conveyors. The rule has a negative application condition (NAC) that forbids moving the piece if the source conveyor is the input to any kind of machine having an operator. Note that this rule uses abstract nodes: piece "p" and machine "m" have abstract types, and are visually represented with asterisks. Abstract nodes in the rule can get instantiated to nodes of any concrete subtype [13]. In this way, rules become much more compact. Rule "change"

models an operator changing to a machine "m1" if the machine has some piece waiting to be processed and it is unattended. Rule "rest" models the break pause of an operator, which is deleted, while rule "work" creates a new operator in an unattended machine.

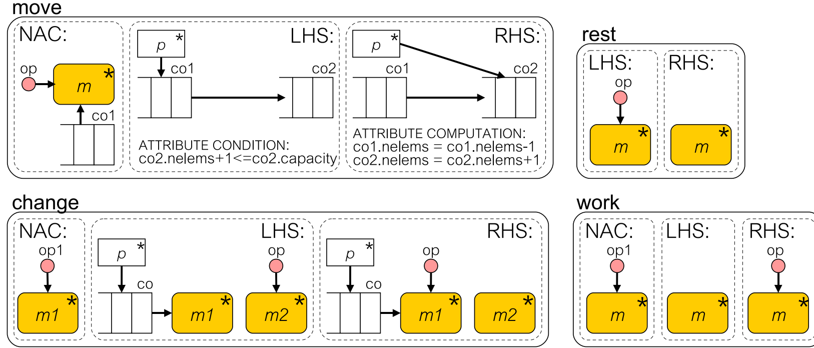


**Fig. 4.** Additional rules for the DSVL simulator.

## 3 From Graph Transformation to OCL

This section presents a procedure to translate graph transformation rules into an OCL-based representation. The procedure takes as input a graph transformation system, made of a set of rules; together with the MOF-compliant meta-model used as a context for the rules. As output, the method generates a set of semantically-equivalent declarative operations (one for each rule) specified in OCL. Declarative operations are specified by means of a contract consisting of a set of pre and post-conditions. Roughly speaking, pre-conditions will define a set of conditions on the source model that will hold iff the rule can be applied, namely if the model has a match for the LHS pattern and no match for any NAC, while post-conditions will describe the new state of the model after the operation execution as stated by the difference between the rule's RHS and LHS.

More precisely, the input of the procedure is a tuple $(MM, ruleStyle, GTS = \{r_j\}_{j \in J})$, where $MM$ is a meta-model (possibly restricted by OCL well-formedness rules), $ruleStyle$ is a flag indicating DPO or SPO semantics, and $GTS$ is a set of graph transformation rules. We represent DPO and SPO rules as $r = (LHS, RHS, ATT_{COND}, ATT_{COMP}, \{NAC^i, ATT^i_{COND}\}_{i \in I})$, where LHS, RHS and $NAC^i$ are models conformant to $MM$. Instances are identified across models by their object identifiers, e.g. the preserved elements by the rule have the same object identifiers in LHS and RHS. $ATT_{COND}$, $ATT^i_{COND}$ and $ATT_{COMP}$ are sets of OCL expressions. The first two contain attribute conditions for the LHS and the i-th NAC, the latter contains attribute computations to state the new values for the attributes in the RHS.

Next subsections use this formalization to translate the $GTS$ in terms of a set of OCL operations. The name of the operations will be the name of the corresponding rule. All operations will be attached to an artificial class $System$,

typically used in the analysis phase to contain the operations describing the behaviour of the system [15]. Alternatively, each operation could be assigned to one of the existing classes in the meta-model (in particular, to one of the classes referred in the LHS or RHS patterns of the rule) following the guidelines provided by the GRASP patterns [15].

### 3.1 Translating the left-hand side

A rule $r$ can be applied on a host graph (i.e a model) if there is a match, that is, if it is possible to assign objects of the host graph to nodes in the LHS such that (a) the type in the host graph is compatible with the type in the LHS, (b) all edges in LHS may be mapped to links in the host graph and (c) the attribute conditions evaluate to true when symbols are replaced by the concrete attribute values in the model. It is possible that the same object is assigned to multiple nodes in LHS (non-injective match) as long as conditions (a-c) are satisfied.

When defining the translation for condition (a) we must explicitly encode the set of quantifiers implicit in the semantics of graph transformation rules: when checking if the host graph contains a match for LHS we have to try assigning each possible combination of objects from compatible types in the model to the set of nodes in the pattern. Thus, we need one quantifier for each node in LHS. In terms of OCL, these quantifiers will be expressed as a sequence of embedded *exists* operators over the population of each node type (retrieved using the predefined *allInstances* operation).

Once we have a possible assignment of objects to the nodes in LHS we must check if the objects satisfy the (b) and (c) conditions. To do so, we define an auxiliary query operation *matchLHSr*. This operation returns true if a given set of objects comply with the pattern structure defined in LHS and satisfy its $ATT_{COND}$ conditions. In particular for each edge $e$ linking two objects $o_1$ (of type $t_1$) and $o_2$ (of type $t_2$) in LHS, *matchLHSr* must define a $o_1.nav_{t2}{-}{>}\ includes(o_2)$ condition stating that $o_2$ must be included in the set of objects retrieved when navigating from $o_1$ to the related objects in $t_2$; the right association end to use in the navigation $nav_{t2}$ is extracted from the $MM$ according to the type of $e$ and the type of the two object participants. $ATT_{COND}$ conditions, already expressed using an OCL-like syntax in $r$, are directly mapped as a conjunction of conditions at the end of *matchLHSr*.

Let $L = \{L_1, \ldots, L_n\}$ denote the set of nodes in LHS and $E = \{(L_i, L_j)\}$ the set of edges. Then, according to the previous guidelines, the LHS pattern of $r$ will be translated into the following equivalent pre-condition:

```
context System::r()
pre: L1.type::allInstances()->exists(L1 |
     ...
     Ln.type::allInstances()->exists(Ln | matchLHSr(L1, ..., Ln) ))
---------------------------------------------------------------
context System::matchLHSr(L1 : L1.type, ..., Ln : Ln.type)
body: L1.nav_L2.type->includes(L2) and
      ... and
      Li.nav_Lj.type->includes(Lj) and ATT_COND
```

where $L_i.type$ returns the *type* of the node and the identifier of the node is used to name the variable in the quantifier. Note that $L_i.type$::allInstances() returns all direct and indirect instances of the $L_i.type$ (that is, it returns also the instances of its subtypes) and thus abstract objects can be used in the definition of $r$.

As an example, the pre-condition for the *rest* rule would be:

```
context System::rest()
pre: Operator::allInstances()->exists(op |
     Machine::allInstances()->exists(m | matchLHSrest(op, m))
---------------------------------------------------------------
context System::matchLHSrest(op: Operator, m: Machine)
body: op.machine->includes(m)
```

where *matchLHSrest* is called for every possible combination of operators and machines in the model (because of the two nested *exists*). If one of such combinations satisfies *matchLHSrest* the pre-condition evaluates to true, meaning that "rest" can be applied on the model.

### 3.2 Translating the negative application conditions

In presence of $NACs$ the pre-condition of $r$ must also check that the set of objects of the host graph satisfying LHS do not match any of the NACs.

The translation of a NAC pattern is almost equal to the translation of a LHS pattern: an existential quantifier must be introduced for each new node in the NAC (i.e. each node not appearing also in the LHS pattern) and an auxiliary query operation (*matchNACr*) will be created to determine if a given set of objects satisfy the NAC pattern. $MatchNACr$ is specified following the same procedure used to define *matchLHSr*.

Within the pre-condition, the translation of the $NACs$ is added as a negated condition immediately after the translation of the LHS pattern.

Let $N = \{N_1, \ldots, N_m\}$ denote the set of nodes in a NAC that do not appear also in LHS. The extended pre-condition for $r$ (LHS + NAC) is defined as:

```
context System::r()
pre: L1.type::allInstances()−>exists(L1 |
    ...
    Ln.type::allInstances()−>exists(Ln | matchLHSr(L1, ..., Ln)
    and not (N1.type::allInstances()−>exists(N1 |
    ...
     Nm.type::allInstances()−>exists(Nm |
     matchNACr(L1, ..., Ln, N1, ..., Nm)) ...)
```

If $r$ contains several $NACs$ we just need to repeat the process for each NAC, creating the corresponding $matchNAC_ir$ operation every time.

As an example, the translation for the LHS and NAC patterns of the "work" rule would be:

```
context System::work()
pre: Machine::allInstances()−>exists(m | matchLHSwork(m)
    and not Operator::allInstances()−>exists(op1|matchNACwork(m,op1) ))
--------------------------------------------------------------------
context System::matchLHSwork(m:Machine):Boolean body: true
--------------------------------------------------------------------
context System::matchNACwork(m:Machine, op1:Operator):Boolean
body: m.operator−>includes(op1)
```

Note that for this rule, *matchLHSwork* simply returns true since as long as a machine object exists in the host graph (ensured by the existential quantifier in the pre-condition), the LHS is satisfied. The additional condition is here imposed by the NAC, stating that no operator may be working on that machine.

### 3.3 Translating the right-hand side

The effect of rule $r$ on the host graph is the following: (1) the deletion of the objects and links appearing in LHS and not in RHS, (2) the creation of the objects and links appearing in RHS but not in LHS and (3) the update of attribute values of objects in the match according to the $ATT_{COMP}$ computations.

Clearly, when defining the OCL post-condition for $r$ we will need to consider not only the RHS pattern (the *new* state) but also the LHS (and NAC) patterns (the *old* state) in order to compute the differences between them and determine how the objects evolve from the old to the new state. In OCL, references to the old state (i.e. references to the values of objects and links in the state *before* executing the operation) must include the @*pre* keyword (for instance, a post-condition expression like $o.atr_1 = o.atr_1@pre + 1$ states that the value of $atr_1$ for object $o$ is increased by one upon completion of the operation)

Therefore, the translation of the RHS pattern requires, as a first step, to select a set of objects of the host graph that are a match for the rule. Unsurprisingly, this initial condition is expressed with exactly the same OCL expression used to define the pre-condition (where the goal was the same: to determine a match for $r$). The only difference is that in the post-condition, all references to attributes,

navigations and predefined properties will include the @*pre* keyword. Next, the selected set of objects are passed to an auxiliary operation *changeRHSr*, in charge of performing the changes defined in the rule.

*ChangeRHSr* will be defined as a conjunction of conditions, one for each difference between the RHS and LHS patterns. Table 1 shows the OCL expressions that must be added to *changeRHSr* depending on the modifications performed by $r$ on the host graph. Moreover, all $ATT_{COMP}$ are added as additional computations at the end. Again, in the computations all references to previous attribute values are extended with the `@pre` keyword. As usual, we assume in the definition of the post-condition for $r$ that all elements not explicitly modified in the post-condition remain unchanged (*frame problem*).

**Table 1.** OCL expressions for changeRHSr

| **Element** | **∃ in LHS?** | **∃ in RHS?** | **Update** | **OCL Expression** |
|---|---|---|---|---|
| Object $o$ of type $t$ | No | Yes | Inserting $o$ | $o$.oclIsNew() and $o$.oclIsTypeOf($t$) |
| Object $o$ of type $t$ | Yes | No | Deleting $o$ | $t$::allInstances()−>excludes($o$) |
| Link $l$ between $(o_1, o_2)$ | No | Yes | Inserting $l$ | $o_1.nav_{t2}$−>includes($o_2$) |
| Link $l$ between $(o_1, o_2)$ | Yes | No | Deleting $l$ | $o_1.nav_{t2}$−>excludes($o_2$) |

As an example, we show the complete operations generated for the "rest" and "work" rules. The translation for the other rules in the running example can be found in the extended version of this paper [22].

```
context System::rest()
 pre:  Operator::allInstances−>exists(op|
       Machine::allInstances−>exists(m|matchLHSrest(op,m)  ))
 post: Operator::allInstances@pre−>exists(op|
       Machine::allInstances@pre−>exists(m|
       matchLHSrest'(op,m) and changeRHSrest(op,m) ))
--------------------------------------------------------------
context System::matchLHSrest(op: Operator, m: Machine):Boolean
 body: op.machine−>includes(m)
--------------------------------------------------------------
context System::matchLHSrest'(op: Operator, m: Machine):Boolean
 body: op.machine@pre−>includes(m)
--------------------------------------------------------------
context System::changeRHSrest(op: Operator, m: Machine):Boolean
body: Operator::allInstances()−>excludes(op)
```

```
context System::work()
 pre:  Machine::allInstances()−>exists(m|matchLHSwork(m) and not (
       Operator::allInstances()−>exists(op1|matchNACwork(m,op1) ))
 post: Machine::allInstances()@pre−>exists(m|matchLHSwork'(m) and not
       ( Operator::allInstances@pre()−>exists(op1|matchNACwork'(m,op1)) )
       and changeRHSwork(m)
-----------------------------------------------------------------------
context System::matchLHSwork(m:Machine):Boolean body: true
-----------------------------------------------------------------------
context System::matchLHSwork'(m:Machine):Boolean body: true
-----------------------------------------------------------------------
context System::matchNACwork(m:Machine, op1:Operator):Boolean
body: m.operator−> includes(op1)
-----------------------------------------------------------------------
context System::matchNACwork'(m:Machine, op1:Operator):Boolean
body: m.operator@pre−> includes(op1)
-----------------------------------------------------------------------
context System::changeRHSwork(m:Machine):Boolean
 body: op.oclIsNew() and op.oclIsTypeOf(Operator) and
       m.operator−>includes(op)
```

### 3.4 Taking into account DPO and SPO semantics

The behaviour of the rules is slightly different depending on whether DPO or SPO semantics are assumed. The two main differences we must consider in our translation are the *dangling edge* condition and the *identification* condition.

In DPO, the *dangling edge* condition states that when a node is deleted, it can only be connected to other nodes by the edges that are explicitly deleted by the rule. With SPO semantics, all edges are implicitly removed when deleting the node. This is the common assumption in UML/OCL specifications [19] and thus with SPO we do not need to modify the translation patterns provided so far (for instance, in the "rest" operation it is assumed that all links connecting object *op* with other objects are implicitly removed when deleting it). Instead, under DPO semantics we must refine the generated pre-condition to ensure that the objects deleted by the rule have no other links except for those appearing in LHS and not in RHS. Therefore, for each deleted object $o$ instance of a type $t$ and for each type $t_i$ related with $t$ in $MM$ we must include in *matchLHS* the following conditions:

– $o.nav_{ti}$−>$isEmpty()$ (when LHS does not include edges relating $o$ with nodes of type $t_i$)
– $o.nav_{ti}$−>$excludingAll(o_1, o_2, \ldots, o_n)$−>$isEmpty()$ (when LHS includes edges relating $o$ with a set of $\{o_1, o_2, \ldots, o_n\}$ nodes of type $t_i$)

As an example, the query operation *matchLHSrest* for "rest" under DPO semantics would be redefined as follows:

```
context System::matchLHSrest(op: Operator, m: Machine):Boolean
body: op.machine->includes(m) and
      op.machine->excluding(m)->isEmpty()
```

The *identification* condition states that two nodes of the LHS cannot be matched into the same object in the host graph if one of the nodes does not appear in the RHS pattern (i.e. it is deleted). With SPO semantics, the object in the host graph is simply removed. Again, the SPO semantics coincide with the default UML/OCL behaviour. If two OCL variables point to the same object and one of the variables is used to define that the pointed object is removed, the other automatically becomes undefined. Instead, to enforce the DPO semantics we need an additional condition in the *matchLHS* operation. Given that $L_1$ and $L_2$ are two nodes in the LHS pattern, $L_1.type = L_2.type$ and $L_1$ but not $L_2$ appear in RHS (or the other way around), the condition $L_1 <> L_2$ should be added in *matchLHS*. This condition forces the problematic existential quantifiers to map to two different objects when evaluating the pre-condition.

## 4 Formalization of Rule Properties with OCL

Translating a graph grammar into a set of operations with OCL pre/post-conditions allows the analysis of relevant properties of the rules. The properties under analysis will take into account the meta-model invariants that restrict the possible set of legal instantiations (i.e. models) of the meta-model.

The following notation will be used to express these concepts: $I$ denotes an instantiation of the meta-model, while $I'$ represents the modified instantiation after invoking an operation. An instantiation $I$ is called *legal*, noted as $\mathsf{INV}[I]$, if it satisfies all the invariants of the meta-model, i.e. both the graphical restrictions such as multiplicity of roles in associations and the explicit OCL well-formedness rules. By $\mathsf{PRE_r}[I]$ we denote that an instantiation $I$ satisfies the pre-condition of an operation $\mathsf{r}$. Regarding post-conditions, we write $\mathsf{POST_r}[I, I']$ to express that instantiation $I'$ satisfies the post-condition of an operation $\mathsf{r}$ assuming that $I$ was the instantiation before executing the operation.

Two families of properties will be studied. First, it is desirable to verify that for each rule there exists at least one valid model where it can be applied, as otherwise the rule is useless. Second, it is interesting to check whether different rules may interfere among them, making the order of application matter. Within each family of properties, several notions will be presented, each with a trade-off between precision and the complexity of its analysis. The list, with its formalization, is the following:

– **Applicability (AP):** Rule $r$ is *applicable* if there is at least one legal instantiation of the meta-model where it can be applied.

$$\exists I : \mathsf{INV}[I] \wedge \mathsf{PRE_r}[I]$$

– **Weak executability (WE):** $r$ is *weakly executable* if the post-condition is satisfiable in some legal instantiation.

$$\exists I, I' : \mathsf{INV}[I] \wedge \mathsf{INV}[I'] \wedge \mathsf{POST_r}[I, I']$$

– **Strong executability (SE):** $r$ is *strongly executable* if, for any legal instantiation that satisfies the pre-condition, there is another legal instantiation that satisfies the post-condition.

$$\forall I : \exists I' : (\mathsf{INV}[I] \wedge \mathsf{PRE_r}[I]) \rightarrow (\mathsf{INV}[I'] \wedge \mathsf{POST_r}[I, I'])$$

– **Correctness preserving (CP):** $r$ is *correctness preserving* if, applied to a legal instantiation of the meta-model, cannot produce an illegal one.

$$\forall I, I' : (\mathsf{INV}[I] \wedge \mathsf{PRE_r}[I]) \rightarrow (\mathsf{POST_r}[I, I'] \rightarrow \mathsf{INV}[I'])$$

– **Overlapping rules (OR):** Two rules $r$ and $s$ overlap if there is at least one legal instantiation where both rules are applicable.

$$\exists I : \mathsf{INV}[I] \wedge \mathsf{PRE_r}[I] \wedge \mathsf{PRE_s}[I]$$

– **Conflict (CN):** Two rules $r$ and $s$ are in conflict if firing one rule can disable the other, i.e. iff there is one legal instantiation where both rules are enabled, and after applying one of the rules, the other becomes disabled.

$$\exists I, I' : \mathsf{INV}[I] \wedge \mathsf{INV}[I'] \wedge \mathsf{PRE_r}[I] \wedge \mathsf{PRE_s}[I] \wedge \mathsf{POST_r}[I, I'] \wedge \neg\mathsf{PRE_s}[I']$$

– **Independence (IN):** Two rules $r$ and $s$ are *independent* iff in any legal instantiation where both can be applied, any application order produces the same result. Four instantiations of the model will be considered to characterize this property: before applying the rules ($I$), after applying both rules ($I''$), after applying only rule $r$ ($I'_r$) and after applying only rule $s$ ($I'_s$).

$$\begin{array}{ccc} I & \xrightarrow{r} & I'_r \\ \downarrow s & & \downarrow s \\ I'_s & \xrightarrow{r} & I'' \end{array}$$

$$\begin{aligned} \forall I : \exists I'_r, I'_s, I'' : (\mathsf{INV}[I] \wedge \mathsf{PRE_r}[I] \wedge \mathsf{PRE_s}[I]) \rightarrow \\ (\mathsf{INV}[I'_r] \wedge \mathsf{POST_r}[I, I'_r] \wedge \mathsf{PRE_s}[I'_r] \wedge \\ \mathsf{INV}[I'_s] \wedge \mathsf{POST_s}[I, I'_s] \wedge \mathsf{PRE_r}[I'_s] \wedge \\ \mathsf{INV}[I''] \wedge \mathsf{POST_r}[I'_s, I''] \wedge \mathsf{POST_s}[I'_r, I'']) \end{aligned}$$

A model $I$ satisfying $\mathsf{PRE_r}[I]$ may admit non-determinism in the execution of $r$, if it contains more than one match of $r$. The difference between *SE* and *CP* is that the former requires one such application to be valid, while for *CP* all of them have to be valid. If a rule does not satisfy *CP*, it means that it is underspecified regarding the OCL meta-model invariants. Notice that the attribute condition in rule "assemble" in Fig. 3 is necessary to ensure that the rule satisfies *CP*.

The term *critical pair* is used in graph transformation to denote two direct derivations in conflict (applying one disables the other), where the starting model is minimal [8, 12]. The set of critical pairs gives all *potential* conflicts, and if empty, it means that the transformation is confluent (i.e. a unique result is

obtained from its application). For technical reasons, any attribute computation is usually modeled as a rewriting of edges [8]. This means that any two rules changing the same attribute of the same node will be reported as conflicting. This does not mean that one rule disables the other, but however ensures confluency. On the contrary, our *CN* condition is more precise about attribute computations and considers the OCL invariants, but by itself does not ensure confluency.

*IN* allows applying each pair of rules in any order, obtaining the same result. This is a strong version of the local Church-Rosser theorem in DPO [8], where we require rule independence for every valid model $I$, and ensures confluency.

## 5 Tool Support

Existing tools for the validation (e.g. USE [11] and MOVA [7]) and verification of UML/OCL models (e.g. HOL-OCL [4], UML2Alloy [1] and UMLtoCSP [6]) can be used to prove the correctness properties of graph transformation rules (Section 4) once translated into declarative OCL operations (Section 3).

Each tool follows a different approach to cope with the verification process with its own set of advantages and drawbacks: bounded verification, need of user guidance, support for a different subset of the OCL language, different efficiency levels and so forth. Moreover, the richness of constructs in OCL and the complexity in some features (e.g. undefined values) constitutes a challenge for all existing tools. As a result, the degree of support for some OCL constructs varies from one tool to another. An example is the support for the operator `@pre` in the post-condition, which must be used in the verification of properties that use $\mathsf{POST_r}$. Therefore some tools may have to be adapted for the verification of those properties. All these issues should be taken into account by the designer when choosing a particular tool to verify the graph transformation rules.

As an example, we describe in this section how to use our UMLtoCSP tool in order to check properties of the rules of our running example. UMLtoCSP works by searching a possible instantiation of the (meta-)model consistent with all the invariants. To analyze rules, pre-conditions of the corresponding operations for each rule must be added as additional invariants (with the same body). If after adding these additional invariants a valid instantiation still exists we may conclude that the rule property is satisfied. To check properties involving post-conditions of operations, two valid instantiations must be computed.

First, we use UMLtoCSP to check whether rules “change” and “move” overlap. To prove so, UMLtoCSP automatically computes the match on the left of Fig. 5. Notice that in this match of the “move” operation, the source and destination conveyors are mapped to the same conveyor object, as there is no constraint forbidding this choice. In fact, this instantiation helps us to detect a problem in our system definition: non-injective matches are inadequate for rule “move”, which in this case may be solved by adding an additional invariant to the meta-model stating that a conveyor cannot be next to itself. On the other hand, the image in the right of Fig. 5 depicts a conflict between rules “work”

and "change": both rules can be applied on the model but applying rule "work" would disable rule "change".

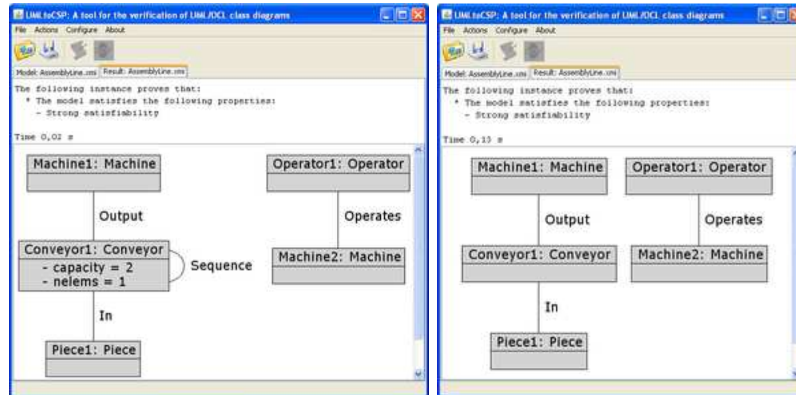


**Fig. 5.** Examples of an overlapping between rules "change" and "move" (left) and a conflict between rules "change" and "work" (right) as computed by UMLtoCSP.

## 6 Related Work

There are two main sources of related work: those analysing rules using DPO and SPO theory, and those that translate rules to other domains for analysis. In the former direction, graph transformation has developed a number of analysis techniques [8, 12, 18], but they usually work with simple type graphs. Our present work redefines some of these analysis properties, but taking into consideration a full-fledged meta-model, as well as OCL constraints in rules. Some preliminary efforts to integrate graph transformation with meta-modelling can be found in [13], where type graphs were extended with inheritance, and [20], were edge inheritance and edge cardinalities were incorporated into type graphs.

Regarding the transformation of graph rules into other domains, their translation into OCL pre- and post-conditions was first presented in [5]. Here we give a more complete OCL-based characterization of rules that considers both DPO and SPO, NACs, and that encodes the LHS's matching algorithm as additional pre-conditions (in [5] the match is passed as parameter to the OCL expressions, assuming an external mechanism). Besides, we exploit the resulting OCL expressions in order to enable the tool-assisted analysis of different rule properties.

Transformations to other domains can be found in [3], where rules are translated into Alloy in order to study the applicability of sequences of rules and the reachability of models; in [2], where rules are translated into Petri graphs to check safety properties; and in [21], where they are transformed into Promela for model-checking. However none of these approaches supports the analysis taking

a meta-model into account or allowing OCL constraints in rules. Besides, our use of OCL as intermediate representation has the benefit that it is tool independent and we can easily integrate attribute conditions and meta-model constraints.

## 7 Conclusions and Future Work

We have presented a new method for the analysis of graph transformation rules that takes into account the (meta-)model structure and well-formedness OCL constraints during the verification process. This way, properties like applicability, which are fundamental to detect inconsistencies in graph transformation rules, can be studied while simultaneously checking for semantic consistency with the meta-model definition. Our method translates the graph transformation rules into an OCL-based representation. Then, the resulting OCL expressions are combined with the OCL constraints specified for the (meta-)models and passed on to existing OCL tools for their joint verification. The translation supports rules with NACs, attribute conditions and distinguishes DPO and SPO semantics.

As future work we would like to develop a plug-in for the graph-based $\text{AToM}^3$ meta-modeling tool [14] to generate OCL expressions from the meta-model and the rules. Additionally, we want to explore alternative applications of our translation. Indeed, once the graph transformation rules are expressed in OCL, we can benefit from all tools designed for managing OCL expressions (spawning from code-generation to documentation, metrics analysis,... ) when dealing with the rules. We also plan to study other properties and to apply this approach to other graph transformation-based techniques, like triple graph grammars.

**Acknowledgments:** Work supported by the Spanish Ministry of Education and Science, projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN2006-09678).

## References

1. Anastasakis, K., Bordbar, K., Georg, G., Ray, I. 2007. *UML2Alloy: A challenging model transformation.* Proc. MODELS'07, pp. 436–450.
2. Baldan, P., Corradini, A., König, B. 2001. *A static analysis technique for graph transformation systems.* Proc. CONCUR'01, LNCS 2154, pp. 381–395. Springer.
3. Baresi, L., Spoletini, P. 2006. *On the use of Alloy to analyze graph transformation systems.* Proc. ICGT'06, LNCS 4178, pp. 306–320. Springer.
4. Brucker, A. D., Wolff, B. 2006. *The HOL-OCL book.* Tech. Rep. 525, ETH Zurich.
5. Büttner, F., Gogolla, M. 2006. *Realizing graph transformations by pre- and postconditions and command sequences.* Proc. ICGT'06, LNCS 4178, pp. 398–413. Springer.
6. Cabot, J., Clarisó, R., Riera, D. 2007. *UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming.* Proc. ASE'07, pp. 547–548.
7. Clavel, M., Egea, M. 2006. *A rewriting-based validation tool for UML+OCL static class diagrams.* Proc. AMAST'06, LNCS 4019, pp. 368–373. Springer.
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer.

9. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A. 1999. *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach.* In [18], pp. 247–312.
10. Ermel, C., Hölscher, K., Kuske, S., Ziemann, P. 2005. *Animated simulation of integrated UML behavioral models based on graph transformation.* Proc. IEEE VL/HCC 2005, pp. 125–133.
11. Gogolla, M., Bohling, J., Richters, M. 2005. *Validating UML and OCL models in USE by automatic snapshot generation.* SoSyM 4(4):386–398. Springer.
12. Heckel, R., Küster, J.-M-., Taentzer, G. 2002. *Confluence of typed attributed graph transformation systems.* Proc. ICGT'02, LNCS 2505, pp. 161–176. Springer.
13. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer G. 2007. *Attributed graph transformation with node type inheritance.* Theor. Comput. Sci. 376(3):139–163.
14. de Lara, J., Vangheluwe, H. 2004. *Defining visual notations and their manipulation through meta-modelling and graph transformation.* J. Vis. Lang. Comput. 15(3-4):309–330. Elsevier.
15. Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* 2004. Prentice Hall, 3$^{rd}$ Edition.
16. Mens, T., Taentzer, G., Runge, O. 2007. *Analysing refactoring dependencies using graph transformation.* SoSyM 6(3):269–285. Springer.
17. QVT standard specification at: `http://www.omg.org/docs/ptc/05-11-01.pdf`
18. Rozenberg, G. (ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific.
19. Sendall, S., Strohmeier, A. 2002. *Using OCL and UML to specify system behavior.* In Object Modeling with the OCL 2002, LNCS 2263, pp. 250–280. Springer.
20. Taentzer, G., Rensink, A. 2005. *Ensuring structural constraints in graph-based models with type inheritance.* Proc. FASE'05, LNCS 3442, pp. 64–79. Springer.
21. Varró, D. 2004. *Automated formal verification of visual modeling languages by model checking.* SoSyM 3(2):85–113. Springer.
22. –. *Analysing Graph Transformation Rules Through OCL (Extended version).* `http://gres.uoc.edu/UMLtoCSP/ICMT08.pdf`.