Software and Systems Modeling manuscript No.
(will be inserted by the editor)

A UML/OCL Framework for the Analysis of Graph Transformation Rules

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

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

Received: date / Revised version: date

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

We have automated these ideas by providing designers with tools for the graphical specification and analysis of graph transformation rules, including a back-annotation mechanism that presents the analysis results in terms of the original language notation.

Key words Graph Transformation – OCL – Meta-Modelling – Domain Specific Visual Languages – Verification and Validation

1 Introduction

Model-Driven Development [58] (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; and semi-formal to formal. The OMG vision of MDD is called Model-Driven Architecture (MDA) [39] and is founded on standards like QVT [50] for transformations and MOF [44] and OCL [45] for modelling and meta-modelling.

Graph Transformation [21,53] is a declarative, rule-based technique for expressing model transformations. It has been used for specifying in-place transformations like animations [23], simulations [36], optimizations and redesigns [43]. It is now 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) [36], 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 in the MDD approach.

Clearly, the important role of (graph) transformations in MDD needs to be supported by analysis techniques that help designers in determining the quality of such transformations. So far, the main formalization of graph transformation is the so called algebraic approach [21], which uses category theory in order to express the rewriting. Prominent examples of this approach are the double [21] and the single [22] pushout (DPO and SPO), which have developed interesting analysis techniques, e.g. to check independence between pairs of derivations [21,53], or to calculate critical pairs (minimal context of pairs of conflicting rules) [26]. However, graph grammar analysis techniques work with simplified meta-models (so called type graphs), which lack OCL-like constraints for expressing the well-formedness rules of the

Send offprint requests to:
⋆ *Present address:* Estudis d'Informàtica, Multimèdia i Telecomunicacions, Rbla. del Poblenou 156, E-08018 Barcelona, Spain

meta-model. By not considering the meta-model constraints, one could design incorrect rules violating such well-formedness rules. We believe that integrating OCL constraints and graph transformation is crucial for the applicability of the latter in real software MDD projects.

In this paper, our goal is to integrate OCL and graph transformation, and provide more advanced analysis techniques for graph transformations by using OCL as an intermediate representation to express the semantics of graph transformation rules. Then, this OCL representation is used to automatically verify 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 its OCL constraints during the rule analysis. Moreover, it makes available a plethora of tools able to analyze this kind of specifications. A secondary benefit of our approach is that graph transformation is made available to the increasing number of MDA tools that the community is building and vice-versa. For example, by using such MDA tools, it could be possible to (partially) generate code for the transformations [18], or apply metrics and redesigns to the rules [38]. In addition, the OCL specification derived from graph transformation rules could be used as a way to add behaviour to meta-models, and contracts for methods. Finally, an intermediate OCL representation serves as a neutral language for the integration of different transformation languages, approaches and tools.

More in detail, we use OCL to represent fully expressive DPO and SPO rules with negative application conditions and attribute conditions. 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. These properties include 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) and strong executability (if a rule applied to a legal model – which conforms to the meta-model and its well-formedness constraints – always yields a legal model) among others.

In order to improve the usability of our analysis framework, we have hidden the analysis process behind a graphical front-end tool called AToM^3 [34]. This meta-modelling tool generates customized modelling environments for DSVLs, and allows defining model manipulations by graph transformation. In order to analyse the graph transformation rules, an OCL generator provides the input to the UMLtoCSP tool [13] in charge of analyzing the rules' properties. The results are then shown back in AToM^3 using the concrete syntax of the DSVL. In this way, the analysis mechanism is kept transparent to the rule designer. This approach follows the line of *hidden formal methods* [7], which advocate an intuitive presentation of the verification results, probably in terms of the input language.

This paper extends our previous work in [15] by refining the translation patterns and extending them with a set of optimization and simplification rules, by adding validation capabilities to our framework, and by providing a complete tool support as described above. As part of our tool support we also discuss some heuristics that help to scale the analysis when dealing with large meta-models and rules.

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 for rule verification and Section 5 shows how to use our approach for validation purposes. Section 6 presents the tool support for the framework. Section 7 compares with related work and Section 8 ends with the conclusions. An appendix shows the OCL translation of all the example rules.

2 Introduction to Graph Transformation

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.

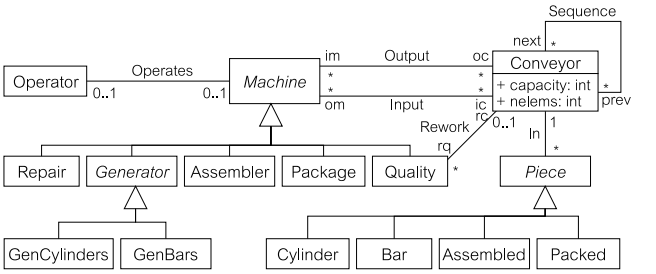

Fig. 1 Meta-model of a DSVL for production systems.

The meta-model defines different kinds of machines (concrete subclasses of *Machine*), which can be connected through conveyors. These can be interconnected and contain pieces (the number of pieces they actually hold is stored in attribute *nelems*), up to its maximum capacity (attribute *capacity*). The OCL invariants on class *Conveyor* 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. 2 shows a production model example conformant to the previous meta-model, expressed using abstract

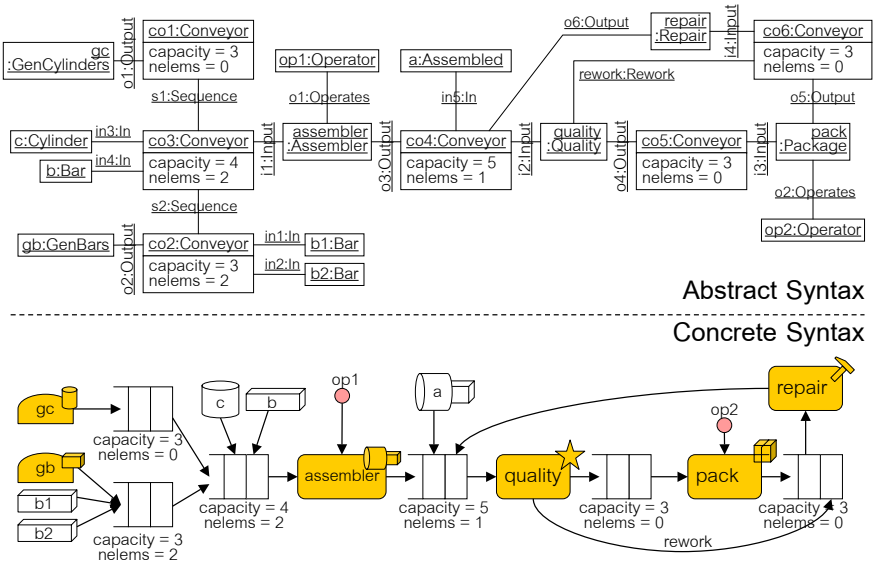

Fig. 2 Example production system model.

syntax on top, and a visual concrete syntax at the bottom. It contains six machines (one of each type), two operators, six conveyors and five pieces. In concrete syntax, 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 an assembler and a package machine respectively. Even though all associations in the meta-model are bidirectional, we have assigned arrows in the concrete syntax, but this does not affect navigability. For the case of the *Input* and *Output* associations, the arrow in the concrete syntax helps identifying the input and output machines.

We use graph transformation [21,22] for the specification of the DSVL operational semantics. A graph grammar is made of a set of rules and an initial graph (*host graph*) to which the rules are applied. Each rule is made of a left and a right hand side (LHS and RHS) graph. 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. This process is called *direct derivation*. 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 in the upper part of Fig. 3, and in abstract syntax to the bottom. Fig. 4 shows its application to a model G (a sub-model of the one in Fig. 2)

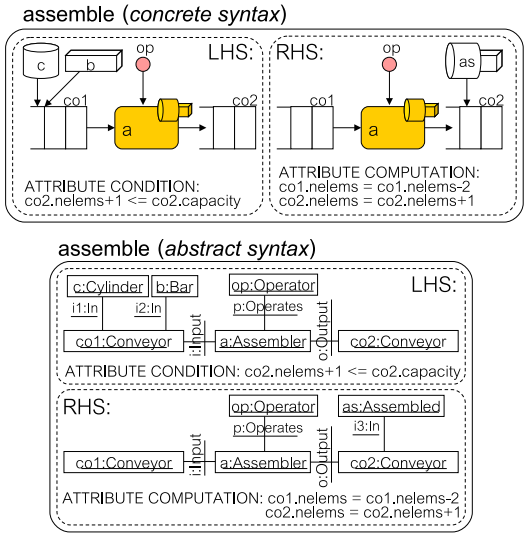

Fig. 3 Rule in concrete (up) and abstract syntax (down).

yielding a model H. First, an occurrence of the LHS is found in the model (dashed area). 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 created. Our rules may include attribute conditions, which must be satisfied by the match, and attribute computations, both expressed in OCL. In Fig. 4, the match of conveyor $co2$ makes the rule's attribute condition $co2.nelems+1 <= co2.capacity$ become $1+1 \leq 5$, and since it is true, the rule is applicable at that match. Attributes referenced to the right of an assignment in an attribute computation refer to the value of the attribute before the rule application. In the figure, the attribute *nelems* of the conveyors matched by the rule are updated.

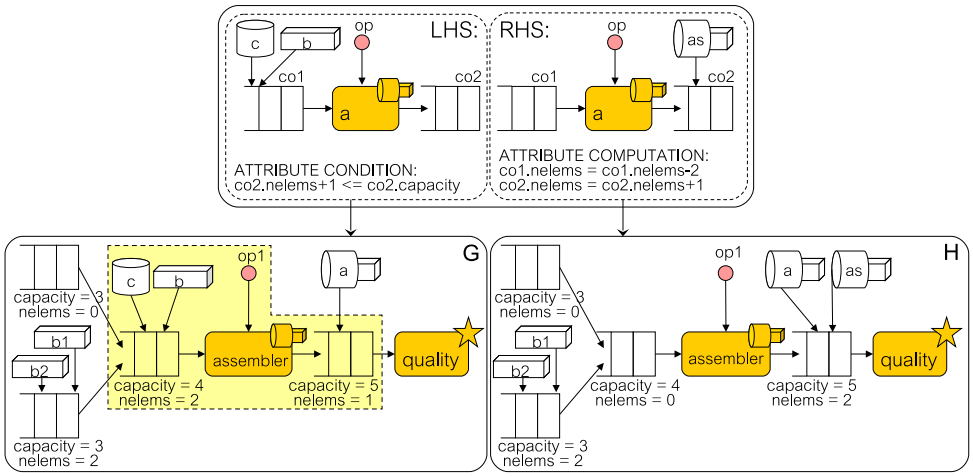

Fig. 4 A direct derivation.

There are two main formalizations of algebraic graph transformation [21]: DPO and SPO. From a practical point of view, their difference is that deletion has no side effects in DPO. That is, when a node in the host graph is deleted by a rule, the node can only be connected through those edges explicitly deleted by the rule. When applying the rule in Fig. 4, if piece "b" in model G would be connected to more than one conveyor (should that be allowed by the meta-model), then the rule could not be applied as those additional edges would become dangling in the host graph G after removing "b". This condition is called *dangling edge* condition. Instead, in SPO dangling edges are removed by the rewriting step.

A second difference is related to the injectivity of matches. 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 a match, while SPO allows its application and deletes both nodes. In DPO, this is called the *identification condition*.

Fig. 5 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. Following [21], we take the match of the NAC as injective for both DPO and SPO. Rule "move" uses abstract nodes: piece "p" and machine "m" are abstract, and are visually represented with asterisks. Abstract nodes in a rule can get instantiated to nodes of any concrete subtype [35]. In this way, rules become much more compact. Rule "move" in the example is equivalent to 24 concrete rules, resulting from the substitution of piece and machine by their children concrete classes.

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

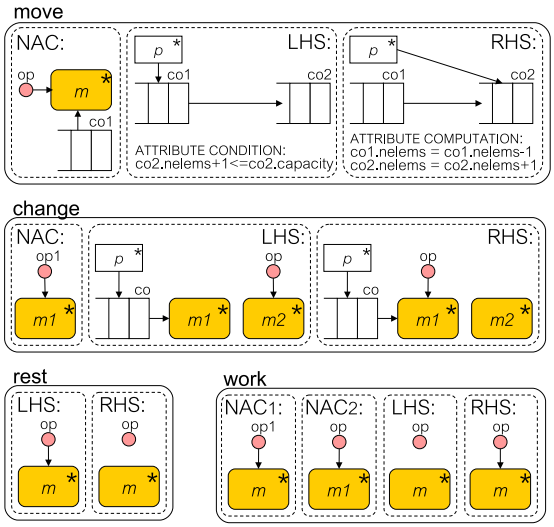

Fig. 5 Additional rules for the DSVL simulator.

break pause of an operator, by deleting its connection to a machine. Finally, rule "work" connects an idle operator (checked by NAC2) to an unattended machine (checked by NAC1).

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 executing the operation as stated by the difference between the rule's RHS and LHS.

More precisely, the input of the procedure is a tuple $(MM, ruleStyle, injMatch, GTS = \{r_j\}_{j \in J})$, where MM is a meta-model possibly restricted by OCL well-formedness rules, $ruleStyle$ and $injMatch$ are two flags indicating DPO or SPO semantics and injectivity of matches respectively, 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 that use the types in MM. Note that they do not necessarily have to satisfy all well-formedness rules in the meta-model (e.g. lower cardinality constraints), as these are patterns, not meant to be complete models. For rules expressing operational semantics, what is important is that the model to which the rules are applied remains consistent. Instances are identified across the models in the rule by their object identifiers, e.g. the elements preserved 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.

The next subsections use this formalization to translate the GTS into 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 of a development process to contain the operations with the behaviour of the system [37]. Alternatively, each operation could be assigned to one of the existing classes. The GRASP patterns (General Responsibility Assignment Software Patterns [37]) can be used to choose the most appropriate class to hold each operation.

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 can 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.

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 LHS 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*, which returns true if a given set of objects complies with the pattern structure defined in LHS and satisfies 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 of type t_2; the right association end to use in the navigation nav_{t2} is extracted from MM according to the type of e and the type of the two participant objects. The 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 $E = \{(L_i, L_j) | L_i, L_j \in L$, and L_i connected to $L_j\}$ 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: $L_1.type$::allInstances()−>exists(L_1 |
  ...
  $L_n.type$::allInstances()−>exists(L_n |
  matchLHSr($L_1, \ldots, L_n$))...)

---

context System::matchLHSr($L_1 : L_1.type$, ..., $L_n : L_n.type$) : Boolean
body: $L_1.nav_{L2.type}$−>includes(L_2) and
  ... and
  $L_i.nav_{Lj.type}$−>includes(L_j) and ATT_{COND}

where $L_i.type$ returns the *type* of the node L_i. Node identifiers are used to name the variable in the quantifier. Note that $L_i.type$::allInstances() returns all direct and indirect instances of $L_i.type$ (i.e. it returns also the instances of its subtypes) and thus abstract objects can be used in the definition of r. When E and ATT_{COND} are empty, the body of *matchLHSr* just returns true.

As an example, the pre-condition for the "rest" rule is the following:

context System::rest()
pre: Operator::allInstances()−>exists(*op* |
  Machine::allInstances()−>exists(m |
  matchLHSrest(op, m)))

---

context System::matchLHSrest(op: Operator, m: Machine) : Boolean
body: op.machine−>includes(m)

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

Many graph transformation tools allow restricting the match to be injective, and this can be enforced in our translation procedure by setting the $injMatch$ flag to true. We can emulate injective matches by adding extra constraints in the pre-condition of the operation stating that every two objects with compatible type should be different. That is, the condition $L_i <> L_j$ is added for $L_i, L_j \in L$, if $L_i.type = L_j.type$ or if one is a subclass of the other. This technique is also used to ensure injectivity of NACs (Section 3.2) and to handle the identification condition (Section 3.4.2).

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 the LHS do not match any of the NACs.

The translation of a NAC pattern is similar to the translation of the LHS: 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. Such $matchNACr$ operation is specified following the same procedure used to define *matchLHSr*. In addition, matches in the NAC must always be injective. Therefore, as part of the translation we must explicitly add conditions ensuring that two nodes of compatible types are not mapped to the same object in the host graph. These have the form $N_i <> L_j$ (or $N_i <> N_j$) for each node N_i in the NAC that is type-compatible with a node L_j in LHS (or another node N_j in the same NAC).

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 in LHS. The extended pre-condition for r (LHS + NAC) is defined as:

context System::r()
pre: $L_1.type$::allInstances()−>exists(L_1 |
...
$L_n.type$::allInstances()−>exists(L_n |
matchLHSr($L_1, \ldots, L_n$)
and not ($N_1.type$::allInstances()−>exists(N_1 |
...
$N_m.type$::allInstances()−>exists(N_m |
matchNACr($L_1, \ldots, L_n, N_1, \ldots, N_m$)
and $N_i <> L_j \ldots$ and $N_i <> N_l$)...))...)

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

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

context System::work()
pre: Machine::allInstances()−>exists(m|
Operator::allInstances()−>exists(op|
matchLHSwork(m,op)
and not Operator::allInstances()−>exists(op1|
matchNAC1work(m,op,op1) and op1 <> op)
and not Machine::allInstances()−>exists(m1|
matchNAC2work(m,op,m1) and m1 <> m))

---

context System::matchLHSwork(m:Machine,
op:Operator): Boolean
body: true

---

context System::matchNAC1work(m:Machine,
op: Operator, op1: Operator) : Boolean
body: m.operator−>includes(op1)

---

context System::matchNAC2work(m:Machine,
op: Operator, m1:Machine) : Boolean
body: m1.operator−>includes(op)

For this rule *matchLHSwork* simply returns true since as long as a machine object and an operator exist in the host graph (ensured by the existential quantifiers in the pre-condition), the LHS is satisfied. The additional conditions imposed by the NACs state that no other operator ($op1$ in the NAC1) can be working on that machine, and that the operator in the LHS cannot be working on a different machine ($m1$ in the NAC2).

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 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 must include the @*pre* keyword. For instance, a post-condition like $o.atr_1 = o.atr_1@pre + 1$ states that the value of atr_1 for object o is increased 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. Then, this initial set of objects will be updated according to the rule definition. Unsurprisingly, this initial condition is expressed

with exactly the same OCL expression used to define the pre-condition[1] (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. Note that when executing the rule it may happen that the set of objects used to satisfy the pre-condition differs from the one used in the match for the post-condition, when there are several possible matches for the rule in the graph. The goal of the pre-condition is just to check that at least one match exists. The post-condition selects one of these matches and changes it. This does not affect the correctness of our approach since graph transformation semantics are non-deterministic. If there are several possible matches, the rule will be applied again on these other matches afterwards. When evaluating the rules (see next section), our checking procedure will try all possible matches to determine their correctness.

Once a set of objects has been selected, it is passed to an auxiliary operation *changeRHSr* in charge of performing the changes defined by the rule. This operation is 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 actions specified by r. Moreover, the ATT_{COMP} expression is added at the end of the procedure, with all references to previous attribute values 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* [54]).

Let $L = \{L_1, \ldots, L_n\}$ and $E = \{(L_i, L_j) | L_i, L_j \in L$, and L_i connected to $L_j\}$ be the set of nodes and edges in LHS, $DN = \{DN_1, \ldots, DN_q\} \subseteq L$ and $DE = \{(L_k, L_l)\} \subseteq E$ the nodes and edges in LHS but not in RHS, and $NN = \{NN_1, \ldots, NN_p\}$ and $NE = \{(NE_x, NE_y) | NE_x, NE_y \in NN \cup (L - DN)$, and NE_x connected to $NE_y\}$ the set of nodes and edges in RHS but not in LHS. Then, according to the previous guidelines, the RHS of r is translated into the following post-condition:

context System::r()
post: $L_1.type$::allInstances@*pre*()−>exists(L_1 |
...
$L_n.type$::allInstances@*pre*()−>exists(L_n |
matchLHSr'($L_1, \ldots, L_n$) and
changeRHSr($L_1, \ldots, L_n$))...)

---

context System::matchLHSr'($L_1 : L_1.type$, ...,
$L_n : L_n.type$) : Boolean
body: $L_1.nav_{L2.type}$@*pre*−>includes(L_2) ...and
$L_i.nav_{Lj.type}$@*pre*−>includes(L_j)
and ATT_{COND}@*pre*

[1] Though looking for a match twice is inefficient, OCL does not offer any mechanism to pass information about variable values from the pre-condition to the post-condition.

[2] This second part of the condition is only required when neither o_1 nor o_2 are new objects.

context System::changeRHSr($L_1 : L_1.type$, ...,
$L_n : L_n.type$) : Boolean
body: −− creation of NN nodes
$NN_1.oclIsNew()$ and
$NN_1.oclIsTypeOf(NN_1.type)$ ...and
$NN_p.oclIsNew()$ and
$NN_p.oclIsTypeOf(NN_p.type)$ and
−− removal of DN nodes
$DN_1.type$::allInstances()−>excludes(DN_1) ...and
$DN_q.type$::allInstances()−>excludes(DN_q) and
−− creation of NE links
$NE_1.nav_{NE_2.type}$−>includes(NE_2) and not
$NE_1.nav_{NE_2.type}$@*pre*−>includes(NE_2) ...and
$NE_x.nav_{NE_y.type}$−>includes(NE_y) and not
$NE_x.nav_{NE_y.type}$@*pre*−>*includes*(NE_y) and
−− removal of DE links
$L_1.nav_{L_2.type}$−>excludes(L_2) ...and
$L_k.nav_{L_l.type}$−>excludes(L_l)
−− attribute computation
and ATT_{COMP}

The previous translation pattern assumes a rule without NACs. If it has NACs, we should add OCL code for testing their satisfaction, as described in Section 3.2. Next we show the generated operations for "rest", the translation of the other rules is in the appendix.

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: op.machine−>excludes(m)

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 differences we must consider in our translation are the *dangling edge* and the *identification* conditions.

3.4.1 Dangling edge condition: In DPO, the *dangling edge* condition states that a node can be deleted only if all its incident edges are explicitly deleted by the rule. With SPO semantics, all edges are implicitly removed when deleting the node. This is the common assumption

Table 1 OCL expressions for changeRHSr

Element	∃ in LHS?	∃ in RHS?	Update	OCL Expression
Object o of type t	No	Yes	Insert o	o.oclIsNew() and o.oclIsTypeOf(t)
Object o of type t	Yes	No	Delete o	t::allInstances()−>excludes(o)
Link l between (o_1, o_2)	No	Yes	Insert l	$o_1.nav_{t2}$−>includes(o_2) and not $o_1.nav_{t2}$@pre−>includes(o_2)[2]
Link l between (o_1, o_2)	Yes	No	Delete l	$o_1.nav_{t2}$−>excludes(o_2)

in UML/OCL specifications [54] and thus with SPO we do not need to modify the translation patterns provided so far. For instance, in the "assemble" operation it is assumed that all links connecting objects c and b with other objects are implicitly removed when deleting them.

On the contrary, under DPO semantics we must refine the generated pre-condition for the rule to ensure that the objects deleted by the rule have no other links except 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 *matchLHSr* the following conditions:

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

As an example, the query operation *matchLHSassemble* for rule "assemble" under DPO semantics is defined as follows:

```
context System::matchLHSassemble( c:Cylinder,
                co1:Conveyor, b:Bar, a:Assembler,
                op:Operator, co2:Conveyor ) : Boolean
body: c.conveyor−>includes(co1) and
      c.conveyor−>excluding(co1)−>isEmpty() and
      b.conveyor−>includes(co1) and
      b.conveyor−>excluding(co1)−>isEmpty() and
      co1.om−>includes(a) and
      a.oc−>includes(co2) and
      a.operator−>includes(op) and
      co2.nelems+1 <= co2.capacity
```

3.4.2 Identification condition: The *identification* condition states that two nodes of the LHS cannot be matched to the same object if one of the nodes is deleted and the other preserved. With SPO semantics, the object in the host graph is simply removed. Again, the SPO semantics coincides with the default UML/OCL behaviour. If two OCL variables point to the same object and one of them is used to define that the pointed object is removed, the other automatically becomes undefined. Instead, to enforce the DPO semantics we must ensure that the matching is not injective. Similar to what we have done to ensure injectivity of the LHS and when translating the NACs, we extend the *matchLHSr* operation with an additional condition. Given that L_i and L_j are two nodes in the LHS pattern, such that $L_i.type = L_j.type$ (or, in general, both types are compatible) and L_i but not L_j appears in RHS (or the other way round), the condition $L_i <> L_j$ should be added in *matchLHSr*. This condition forces the problematic existential quantifiers to map to two different objects when evaluating the pre-condition.

3.5 Preservation of Semantics

Altogether, our translation into OCL preserves the semantics of the original rules. A formal proof of such preservation is out of the scope of this paper, instead we provide an intuition. We have to consider two correctness aspects: (i) a rule is applicable iff the generated operation pre-condition is satisfied, and (ii) the effects of a rule are exactly those required by the post-condition. We consider the DPO semantics because it is more restrictive, but a similar reasoning holds for SPO.

For the first aspect, first assume that a rule is applicable. This means that (a) there is a match of the LHS pattern in the graph, (b) the matching objects in the graph satisfy the attribute conditions, (c) if the rule deletes some element, then the dangling condition is satisfied, (d) if the match is non-injective, then the identification condition is satisfied, and (e) all NACs are satisfied. As shown in Section 3.1, the generated OCL pre-condition contains a number of *exists* that locate all potential matching objects, and an auxiliary query operation *matchLHSr* that checks whether the edges are present and the attribute conditions are satisfied by the matching objects. If the dangling edge condition is satisfied, then the deleted objects do not have more connections than those explicitly deleted by the rule, which is exactly what the conditions we generate in Section 3.4.1 do. If the match is non-injective and the identification condition is satisfied, this means that all elements mapped to the same one in the model are preserved. The conditions we generate in Section 3.4.2 ensure that two elements with compatible type, where one is deleted and the other not, have to be different. Hence, if (d) holds, then our conditions are satisfied. Finally, if (e) holds, it means that no matching objects satisfying the conditions are found for any NAC. The conditions we generate in Section 3.2 check that no match exists for the NAC such that the matching objects satisfy the conditions. Altogether we can conclude that if the rule is applicable then our pre-condition is satisfied.

In the case a rule is not applicable, it is because some of the conditions (a-e) fail. If (a) or (b) fails, then our conditions will fail as they will not find any match

satisfying the conditions. If (c) fails, then our dangling conditions will fail as there is some link and hence the *isEmpty()* query will return false. If (d) fails it is because the deleted element could not be identified to the same object as some of the preserved elements. But then, our condition $L_i <> L_j$ will fail. Finally, if (e) fails, then some valid match for the NAC is found by our conditions, and as they are negated, they will also fail.

It is easy to prove the converse of these two properties, i.e. if the pre-condition is true then the rule is applicable, and if it is false the rule is not applicable.

Finally, we show that the effects of a rule are exactly those required by the post-condition. A rule can create and delete objects and edges, as well as change attribute values. The *changeRHSr* operation includes an appropriate OCL expression for each creation and deletion action – check Table 1 – whereas the attribute computations are handled by adding ATT_{COMP}. Hence, we can conclude that the effects of the rule are exactly those required by the post-condition.

3.6 Optimizing the resulting constraints

These general translation patterns can be slightly simplified, yielding optimized OCL constraints, depending on the specific structure of each rule. In what follows we comment some possible simplifications.

- LHS nodes with no edges and no attribute conditions do not need to be passed as parameters for the *matchLHSr* operation. For those objects it is just enough to check their existence in the main pre-condition expression. E.g. in rule "work" we do not need to pass *op* and *m* as parameters for *matchLHSwork*.
- Similarly, LHS nodes not referenced in a NAC pattern do not need to be passed as parameters for the *matchNACr* operation.
- *matchLHSr* and *matchNACr* operations with an empty body (i.e. a body with just the *true* literal expression) can be skipped.
- For a rule *r* not including any *matchLHSr* or *matchNACr* operations, we do not need to nest the *exists* iterators in its pre-condition but just use a conjunction of separated quantifiers, improving the efficiency of its match finding process. For instance, assuming a hypothetical "assign" rule that given a piece and a conveyor puts the piece in the conveyor, the generated pre-condition for "assign" is the following:

```
context System::assign()
 pre: Piece::allInstances()−>exists(p| true)
      and Conveyor::allInstances()−>exists(c| true)
```

- The auxiliary *matchLHSr*, *matchNACr* and *changeRHSr* operations can be reused across different (or the same) rules sharing common patterns. As an example, the NAC1 and NAC2 patterns for the rule "work" and the NAC for the "change" rule (once their irrelevant parameters have been removed using the previous optimizations) can be merged into a single match operation that will be invoked (with different arguments) in the pre-conditions.
- Some conditions in the patterns may be subsumed by the meta-model constraints and thus can be removed from the operations. As an example, consider the condition *c.conveyor−> excluding(co1) −> isEmpty()* (in *matchLHS* for "assemble" under DPO semantics) saying that the piece *c* cannot be related to other conveyors except for *co*1. This condition is already implied by the maximum multiplicity constraint between *Piece* and *Conveyor* in the meta-model, which forces pieces to be related to at most one conveyor.

As an example, once we apply these optimizations to the "work" rule, (part of) its simplified translation is shown in the next table. First, *matchLHSwork* can be eliminated, then the parameters of *matchNAC1work* and *matchNAC2work* can be reduced. Finally, both operations for the NACs can be merged as they share the same pattern and parameters.

```
context System::work()
 pre: Machine::allInstances()−>exists(m|
      Operator::allInstances()−>exists(op|
      not Operator::allInstances()−>exists(op1|
      matchNAC12work(m,op1) and  op1 <> op )
      and not Machine::allInstances()−>exists(m1|
      matchNAC12work(m1,op) and m1 <> m ))
---------------------------------------------------
context System::matchNAC12work(m:Machine,
                               op: Operator ) : Boolean
body:  m.operator−>includes(op)
```

4 Verification of Rule Properties with OCL

Translating a graph grammar into a set of operations with OCL pre/post-conditions allows the analysis of relevant correctness properties of the rules, as described in this section (later on, in Section 6 we show how to automate this analysis process). The properties under analysis will take into account the meta-model invariants that restrict the possible set of legal instantiations of the meta-model, as well as the pre- and post-conditions derived from the rules.

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 r. Regarding post-conditions, we write $\mathsf{POST_r}[I, I']$

to express that an instantiation I' satisfies the post-condition of an operation r given that I was the instantiation before executing the operation. As usual, to avoid the *frame problem* [8] when interpreting $\mathsf{POST_r}[I, I']$, we assume that only the objects referenced in the post-condition can change their state during the operation execution [54].

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 the precision and the complexity of its analysis. The list 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.

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

– **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.

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

– **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).

$$\begin{array}{ccc} I & \xrightarrow{r} & I'_r \\ \downarrow s & & \downarrow s \\ I'_s & \xrightarrow{r} & I'' \end{array} \quad \begin{array}{l} \forall I : (\mathsf{INV}[I] \wedge \mathsf{PRE_r}[I] \wedge \mathsf{PRE_s}[I]) \rightarrow \\ \exists I'_r, I'_s, I'' : \\ (\mathsf{INV}[I'_r] \wedge \mathsf{POST_r}[I, I'_r] \wedge \mathsf{PRE_s}[I'_r] \wedge \\ \mathsf{INV}[I'_s] \wedge \mathsf{POST_s}[I, I'_s] \wedge \mathsf{PRE_r}[I'_s] \wedge \\ \mathsf{INV}[I''] \wedge \mathsf{POST_r}[I'_s, I''] \wedge \mathsf{POST_s}[I'_r, I'']) \end{array}$$

– **Causal Dependence (CD):** Two rules r and s are *causally dependent* iff there is some legal instantiation where one is applicable and the other not, but applying the former makes the latter applicable.

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

As an example of applicability, Fig. 6 shows a model in which rules "move" and "change" are applicable, since the model (I in the property definition) satisfies the invariants and both rules' pre-conditions. The matches of both rules are enclosed in different dotted polygons. The rules are applicable because the model contains occurrences of both LHSs, but not of the NACs. In fact, the model is also an example of overlapping between the rules. They overlap because they are applicable on the same model. Notice that in this match of the "move" rule, the source and destination conveyors are mapped to the same conveyor object, as there is no constraint forbidding this choice. It is worth noting that this instantiation helps to detect a problem in the 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, or by restricting the match of the LHS to be injective. This situation was detected when validating the system (see Section 5).

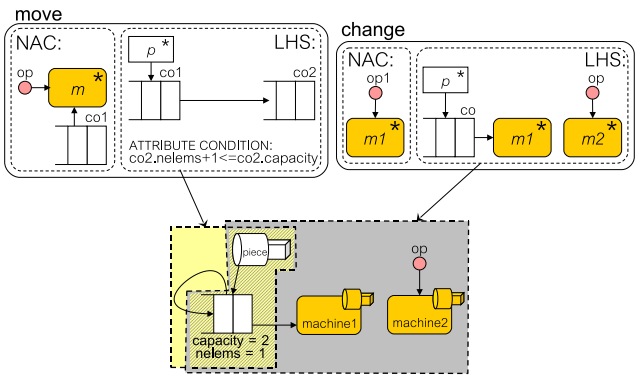

Fig. 6 Overlapping of rules "move" and "change".

The difference between weak and strong executability is that the former requires the existence of just one legal model over which the rule can be successfully executed, while the strong version of the property asks for the executability of the rule in any situation in which its pre-condition is satisfied. If a rule does not satisfy strong executability, it may mean that it is underspecified re-

garding the OCL meta-model invariants. The needed extra constraints in rules can be either NACs or attribute conditions.

For instance, consider the situation in Fig. 7. The picture shows a rule "lighten" that dynamically reconfigures the production plant by adding a new connection from a machine of any type to an empty conveyor, if the machine is already connected to a full conveyor and does not have further outputs. This rule is not strongly executable since there are models satisfying the LHS but where the rule cannot be applied. As an example, below the rule there is a model that satisfies the LHS by mapping the machine in the rule to a quality machine. The rule cannot be executed on this model, as an OCL constraint in the meta-model restricts quality machines to have at most one output. To make the rule strongly executable, one can add an additional NAC with just one machine of type quality mapped to $m1$. This NAC would ensure that the rule is not applied to quality machines. On the contrary, the rule is already weak executable because it can be applied as it is to any machine of type different from quality. For the same reason, the NACs of rules "change" and "work" in Fig. 5 are needed to ensure strong executability.

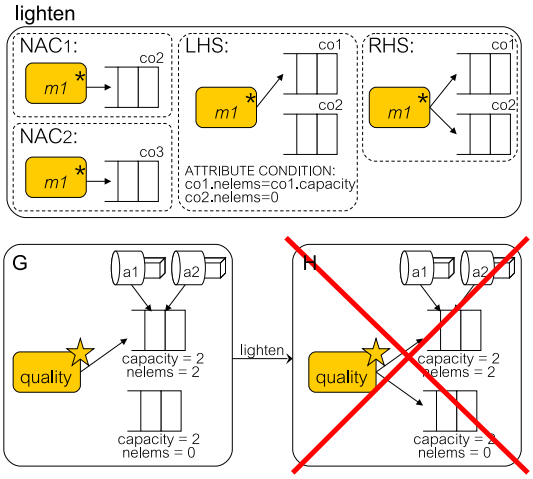

Fig. 7 Non strongly executable rule.

Some other times we need extra attribute conditions in the rules to ensure strong executability. For example, the attribute conditions in rules "assemble" and "move" in Figs. 3 and 5 are necessary to ensure that both rules satisfy strong executability.

The conflict and independence properties are related to the concept of critical pairs. The term *critical pair* is used in graph transformation to denote two direct derivations in conflict (i.e. applying one disables the other), where the starting model is minimal [21,26]. 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, the algebraic approach to graph transformation usually models any attribute computation as a rewriting of edges [21]. This means that any two rules changing the same attribute of a node will be reported as conflicting. In general, this does not imply that one rule disables the other, but however ensures confluence. For example, two rules, one multiplying an attribute `x` by 2, and the other adding 1 to the same attribute, would be reported as a conflict; but no rule disables the other. On the contrary, our conflict condition is more precise about attribute computations and considers the OCL invariants, but by itself does not ensure confluence. In the previous example, if the rule multiplying by 2 is applied first, the attribute `x` becomes $2 \times x + 1$. However, if the rule adding 1 is applied first, we obtain $(x + 1) \times 2$. Hence the transformation would not be confluent. The advantage of our approach is that fewer conflicts will be reported by the conflict property, but confluence has to be checked with the independence property.

The independence property allows applying two rules in any order, obtaining the same result. This is a strong version of the local Church-Rosser theorem in DPO [21], where we require rule independence for every valid model I, and ensures confluence (i.e. same result). In the same way as the technique of critical pairs in graph transformation [1], we do not have to check each possible model in which rules overlap, but only the minimal ones.

The causal dependence property detects whether two rules have some dependency, in such a way that executing one enables the other. In this way, one rule may add one element that the other needs (produce-use dependency), or delete some element that is part of the NAC of the other rule (delete-forbid dependency). For example, consider rules "work" and "rest". The former associates an operator to a machine, while the latter deletes such connection. There is a produce-use dependency between the rules because "rest" needs an edge which "work" produces. These rules have also a produce-use dependency, because "rest" deletes a connection which is in the NAC of "work". Fig. 8 shows two host graphs, together with two derivations that illustrate these two dependencies.

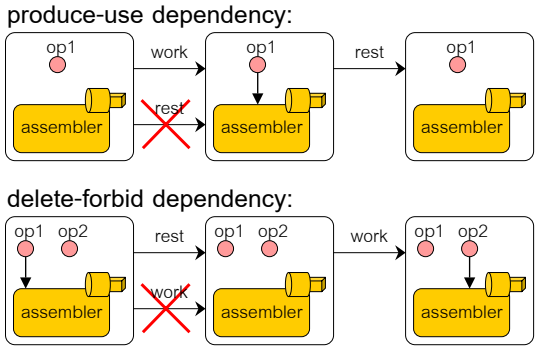

Fig. 8 Causal dependencies.

Finally, note that studying whether all these properties hold in some specific host graph is possible, by replacing the $\exists I$ with some specific model I_1 and then checking if the formula holds for that I_1.

5 Validation of Graph Transformation Systems

The transformation of graph grammars into OCL can also be used for *validation*. While verification attempts to check that the graph transformation rules satisfy some required correctness properties ("is the transformation right?"), validation tries to ensure that the graph transformation rules match the intentions of the designer ("is this the right transformation?").

A validation tool should let the designer inspect the application of graph transformation rules on a given host graph in order to check whether the result is the one expected by the designer. To this end, the designer should be able to perform the following steps:

- define an input model I, instance of the meta-model (i.e. the host graph),
- select the rule to be applied, and
- inspect the result of applying the rule to I.

The UML/OCL definition of the graph transformation rules enables several possible validation flows using the same analysis engine required for verification. The only difference is that now, the input I is fixed by the designer.

This input can be easily characterized by defining an OCL constraint which is only satisfied by the desired input meta-model instance to validate. For example, let us consider how the behaviour of the previous graph transformation rules can be validated in a production system where there is only one operator and no conveyors, machines or pieces. Such production system is characterized by the following OCL constraint:

```
Operator::allInstances() −>size() = 1 and
Machine::allInstances() −>isEmpty() and
Conveyor::allInstances() −>isEmpty() and
Piece::allInstances() −>isEmpty()
```

Now it is possible to validate several properties about the rules by testing their behaviour on that specific input model. In order to do that, it is only necessary to strengthen the pre-condition of the OCL operations generated for the rules. The OCL constraint defining the input graph (denoted as INPUT) is added to the pre-conditions using a conjunction, e.g. for rule "work", the new pre-condition is: $\mathsf{PRE}'_{\mathsf{work}} = \mathsf{PRE}_{\mathsf{work}}$ and INPUT[3]. With this new pre-condition, we can check whether the rule is applicable for this instance, using the same analysis engine as in verification. Similarly, it is possible to check whether the rule is executable and obtain the graph produced by the application of the rule, or check whether two rules overlap for this input instance by strengthening both pre-conditions.

A more interesting possibility is to define partially specified input instances. This can be achieved simply by writing OCL constraints which can be satisfied by more than one instance. For example, the designer might want to study the behaviour of a production system in a scenario where all conveyors are full (the state of other parts of the model is left undefined). This scenario is described by the following constraint:

```
Conveyor::allInstances()−>notEmpty() and
Conveyor::allInstances()−>forAll(c| c.capacity = c.nelems)
```

As before, properties of interest, e.g. is rule "move" applicable if all conveyors are full?, can be checked on this scenario. The advantage is that now we can detect problems in the definition of the graph transformations rules by just characterizing the relevant part of the input state instead of providing its full definition (more time-consuming).

As an example to illustrate the benefits of the validation process, let us consider the rule "work". This rule should provide a mechanism to assign operators to machines, provided that the machine m does not have another operator (NAC1) and that the operator op is not assigned to any other machine (NAC2). The two NACs are intended to capture the situations where this rule cannot be applied. If we validate the rule on the following scenario, where op and m are already connected:

```
Operator::allInstances()−>exists (op |
Machine::allInstances()−>exists (m |
op.machine−>includes(m))) and
Operator::allInstances()−>size() = 1 and
Machine::allInstances()−>size() = 1
```

it turns out that rule "work" is also applicable. However, this is not what we would expect: the rule should not be applicable because it is not possible to add a new connection between op and m. This scenario is not captured by the current NACs because their injective semantics forbids that objects appearing in the NACs but not in the LHS are matched to the same object as any of the objects appearing in the LHS. Therefore, objects op and m from the LHS cannot be equal to objects *op1* and *m1* from the NACs. This means that neither of the NACs forbid the operator and machine matched in the LHS from being connected. It is necessary to add a third NAC forbidding op and m from being previously connected[4]. Note that the rule "work" as shown in Fig. 5 is not strong executable, and thus this error could have also been detected by the verification method of previous section.

[3] The INPUT constraint is not added as an OCL invariant because we want to restrict the input graph, but not the graph produced by the application of the transformation rule.

[4] In fact, we ourselves detected this error when "playing" with the rule over different scenarios.

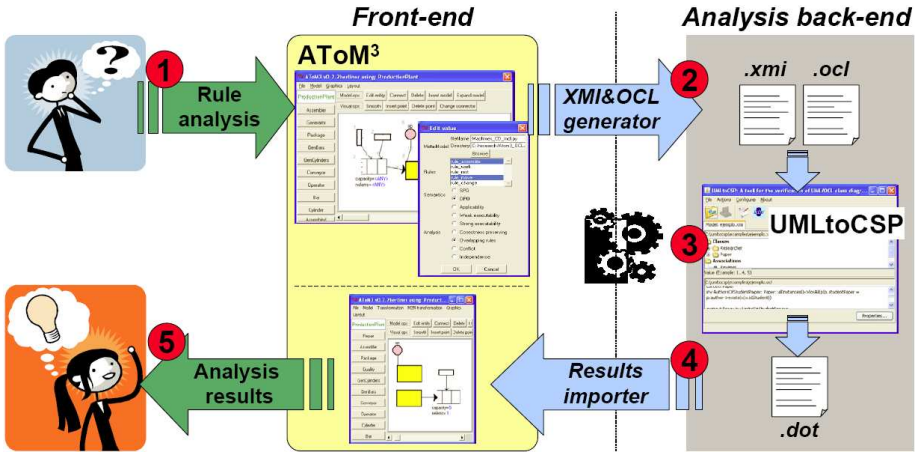

Fig. 9 The architecture for rule verification.

6 Tool Support

In this section, we explain how we have automated the previous ideas using several tools, as shown in Fig. 9.

As a first step, the designer specifies the graph transformation rules and selects the properties to verify using the graphical modelling environment provided by the AToM^3 tool [34]. This tool allows the definition of the syntax of DSVLs using meta-modelling and model manipulation via graph transformation. In this way, the rule designer uses the concrete syntax of the DSVL for rule specification, which is more intuitive and closer to the problem domain. Section 6.1 gives further details on this step.

Then, using the translation we have shown in Section 3, OCL expressions are generated (step 2 of the figure) that can be analysed using different tools (step 3). In particular, our framework uses the UMLtoCSP tool [13], based on constraint programming techniques. Alternative tools that may replace the role of UMLtoCSP within the framework are studied in Subsection 6.4. Finally, our architecture parses the results reported by UMLtoCSP during the analysis (step 4), showing them in the AToM^3 front-end tool, employing the same concrete syntax used in the definition of the DSVL (step 5). In this way, all the verification process is kept transparent to the user, who is not aware of the method and internal formalism used for the verification of the rules. This last step is described in Subsection 6.3.

6.1 Front-end tool

We use the AToM^3 [34] tool for the design of the syntax and semantics of DSVLs. The syntax is specified using meta-modelling, while model manipulations are expressed using graph transformation rules. Constraints can be given in two formats: Python and OCL. The former are meant to be executable inside the tool, while the latter were merely used for documentation up to now.

From the definition of the meta-model and the rules, AToM^3 generates a modelling tool that allows building models conformant to the meta-model, as well as executing the defined graph transformation rules. As an example, Fig. 10 shows the generated modelling environment for the production system example, and a state in the execution of the graph grammar for its simulation.

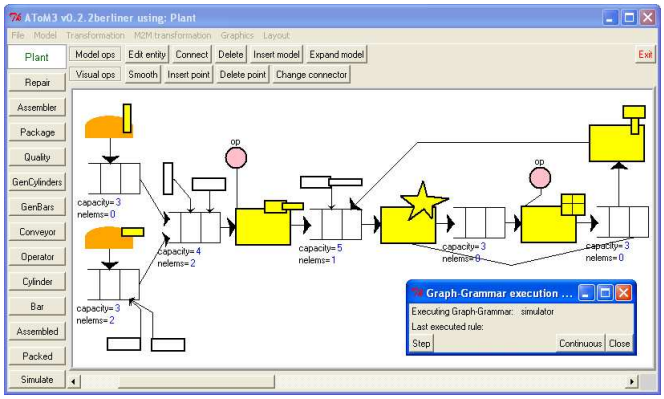

Fig. 10 Modelling environment for the production system.

In order to support the procedure given in previous sections, we have extended AToM^3 with the possibility of exporting the meta-model definition as an XMI file [59], and the OCL expressions derived from the graph grammar rules as a textual file. The code generator uses the OCL version of the constraints in meta-models and rules. The generated meta-model in XMI format includes an additional class *System* which contains the operations for each rule in the grammar. These files can be used by a number of tools to perform different kinds of analysis. The next subsection presents the analysis with the tool UMLtoCSP and Section 6.4 discusses other alternative analysis tools.

6.2 Back-end tool

The correctness properties of graph transformation rules can be studied by analysing their translation into OCL declarative operations. In our architecture, this analysis is performed by UMLtoCSP [13], a tool for the formal verification of UML class diagrams annotated with OCL constraints (invariants, pre-conditions and post-conditions).

UMLtoCSP has been designed to achieve the maximum degree of automation in the verification process without imposing restrictions on the input notation. Rather than verifying custom user-defined properties, UMLtoCSP checks a predefined collection of correctness properties. For example, it can check whether a class diagram is *satisfiable* (i.e. if it is possible to create a non-empty instance without violating any integrity constraint), whether an invariant is *redundant* (the remaining integrity constraints do not allow the invariant to be false), whether an operation is *applicable* (there is an instance which simultaneously satisfies the pre-condition and all integrity constraints) or whether two operations overlap.

The output of UMLtoCSP is an instance of the class diagram, i.e. an object diagram depicting the set of objects in each class, the connections between them and the value of their attributes. For properties that require the existence of one instance (like satisfiability or executability), the output is an *example* that proves the property. On the other hand, for properties that can be disproved by the existence of an instance, the output is a *counterexample* of the property. An example of such properties is redundancy of an invariant, where an instance satisfying all integrity constraints except the invariant certifies that it is not redundant. In any case, the instance can be shown graphically using the graph layout software GraphViz. We have also modified the tool to allow its invocation from the command line, so that it can be used as a back-end from AToM^3.

Internally, the search of an example/counterexample is modelled as a Constraint Satisfaction Problem (CSP). A CSP consists of a set of *variables*, a *domain* for each variable and a set of *constraints* over the variables. Intuitively, the variables characterize the object diagram (number of objects of each class, values of attributes, targets of association ends) while the constraints capture the graphical and textual restrictions (multiplicities, inheritance hierarchies, OCL constraints).

A *solution* to a CSP is an assignment of values to variables such that (a) each value is within the domain of the variable and (b) all constraints are satisfied. The search of a solution is performed automatically by a constraint solver (ECL^{i}PS^e [19] in the case of UMLtoCSP). The solver tries to assign values to variables one at a time, backtracking every time a constraint is violated by a partial assignment.

As an example, we show in Fig. 11 the object diagram automatically computed by UMLtoCSP to prove that rules "change" and "work" overlap[5]. The overlapping between rules "change" and "move" previously depicted in Fig. 6 has also been computed by UMLtoCSP, and it is also an example of conflict, because applying "change" disables "move".

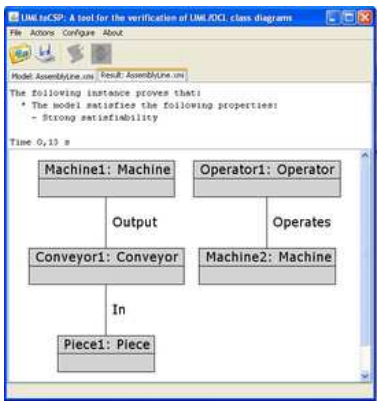

Fig. 11 Example of an overlapping between rules "change" and "work" as computed by UMLtoCSP.

6.2.1 Efficiency issues: The analysis of UML class diagrams is a computationally complex problem. First, reasoning about UML class diagrams without OCL constraints is EXPTIME-complete [6]. The addition of OCL makes the problem undecidable in general.

To avoid undecidability issues, UMLtoCSP offers an automatic procedure for verification which is not *complete*. The search for an example or counterexample occurs within a finite search space that is defined by the user, e.g. by defining the number of objects to be considered. Finding an example (respectively, counterexample) proves (disproves) the property, but if none is found there is no information concerning the validity of the property, other than there is no example/counterexample within the defined search space.

Therefore, from the point of view of completeness, the user is interested in defining a search space which is as broad as possible: a large number of potential objects for each type and a wide range of values for each attribute. However, as a trade-off, the size of the search space to be explored by UMLtoCSP is the grand product of the sizes of all the domains of the variables. For example, given 3 potential objects, each with 2 attributes and 10 possible values for each attribute, the number of possible instantiations of those objects is $10^2 \cdot 10^2 \cdot 10^2 = 10^6$ (2 attributes per object with 10 possible values each where each object is independent of the others). It should

[5] The overlapping property is internally expressed as a satisfiability problem (i.e. two rules overlap if it is possible to create an instance that satisfies the meta-model constraints and the pre-conditions of both rules at the same time, where the pre-conditions are treated as additional invariants to be satisfied by the instance).

be noted that the size of the search space suffers from a combinatorial explosion, growing very quickly as the number of objects to be considered increases.

Although this may seem an important problem that may affect the soundness and scalability of our analysis approach, we would like to remark that limited search spaces are sufficient to identify a high proportion of defects. This observation, also called "*small scope hypothesis*", is the basis of several verification tools such as the Alloy analyzer [27] and it has been validated empirically [3]. In the context of graph transformation rules, the hypothesis implies that analyzing the behaviour of rules for small graphs is sufficient to identify most defects. For example, let us consider the rule "work" from our running example, which assigns idle operators to machines. It seems natural that a small number of operators and machines are required to study the behaviour of this rule, i.e. at most a busy operator, an idle operator, an empty machine and an occupied machine. Therefore, even an inconclusive answer like "considering at most two objects per class, the rule is not applicable" is a signal of a potential error in the rule.

Moreover, the search process does not attempt to explore the search space exhaustively. First of all, the search stops when a solution (example or counterexample) is found, something which typically happens quickly if there exists a solution. Also, the constraint solver used by UMLtoCSP (ECL^{i}PS^e) applies numerous optimizations to guide the search. For instance, constraints are evaluated as early as possible to prune partial solutions which would be unable to satisfy all the constraints. Furthermore, *constraint propagation* is used to infer information about constraints where not all variables have been assigned, e.g. when one term in a product is zero, the result is zero regardless of the value of the other terms. Such optimizations allow the efficient exploration of very large search spaces.

Finally, the analysis of graph transformation rules has some specific traits that make it more amenable to analysis using tools like UMLtoCSP. First, the verification of properties involves one or at most two rules of the graph grammar, but not all of them simultaneously. The number of rules may affect the number of properties to be checked if, for instance, a designer wants to check whether there is conflict among any pair of rules. Nevertheless, the size of the grammar does not have any impact in the complexity of each individual verification problem.

A second trait which helps the analysis is the existence of a LHS, as this provides a strong indicator of the number of objects that will be necessary for its applicability, executability, overlapping or conflict. For instance, in order to check the applicability of a rule, we need to find an instance of the model with as many objects as its LHS. The multiplicities of associations and the invariants of the meta-model may require the creation of additional objects, however the structure of the LHS provides a very good heuristic for choosing the size of the search space.

To illustrate these heuristics, let us consider two examples of overlapping analyzed with UMLtoCSP and presented in Figs. 6 and 11. We can compare the CPU time required for the computation of the overlapping instance in two scenarios: without heuristics, i.e. using the default search space defined in UMLtoCSP; and with heuristics, i.e using the search space (number of objects) inferred from the LHS of the rules. The CPU time in seconds, measured on a Pentium 4 2Ghz with 256Mb of memory, is the following:

Example	**Without heuristics**	**With heuristics**
Fig. 6	43,26 seconds	0,16 seconds
Fig. 11	115,15 seconds	0,34 seconds

Verification can be quickly completed with and without heuristics, but the LHS heuristic provides a significant reduction in CPU time. These results show that, even though the problem is computationally very complex, the proposed approach is efficient enough to be usable within an interactive graphical modelling tool like AToM^3.

6.3 Back-annotation of results

From a user perspective, the adoption of a verification framework such the one presented herein largely depends on usability aspects. In particular, apart from being an automatic process, the framework should also hide the analysis tool and its underlying formalism to the user.

In our framework this is achieved by expressing the input and output of the verification process with the concrete syntax of the DSVL the user has designed. Therefore, the feedback becomes more intuitive and easier to interpret.

As showed in Fig. 9, our architecture hides the verification process by using the graphical interface provided by the AToM^3 tool as a front-end for both defining the DSVL and showing the analysis results. More in detail, the user selects the rules to be analysed and the property to be verified using AToM^3. Then, the XMI/OCL code generator produces two intermediate files, which are automatically fed into UMLtoCSP. The results of the analysis (a GraphViz file) are then parsed and loaded into AToM^3 again, and shown to the user in the concrete syntax of the DSVL. In this way, the designer is not aware of the internal analysis mechanisms.

As an example, Fig. 12 shows the invocation of the rule analysis from AToM^3. The dialog box allows selecting the rules to be analysed, the semantics used for the transformation (DPO or SPO), and the property to be verified. In the figure, the designer has selected to check the overlapping of rules "assemble" and "move" using SPO semantics.

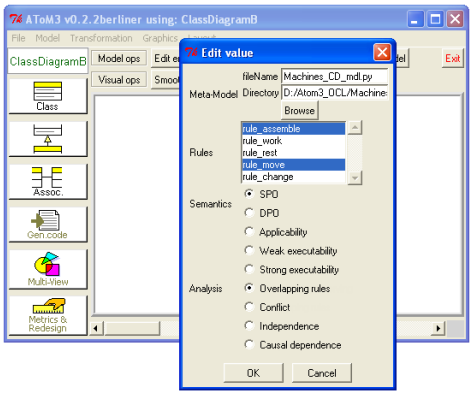

Fig. 12 Invoking the rule analysis from AToM^3.

Fig. 13 shows the result of the analysis, which is a model in the concrete syntax of the DSVL showing an example of overlapping. In case no example/counterexample exists for the given property, the user is warned using a dialog box.

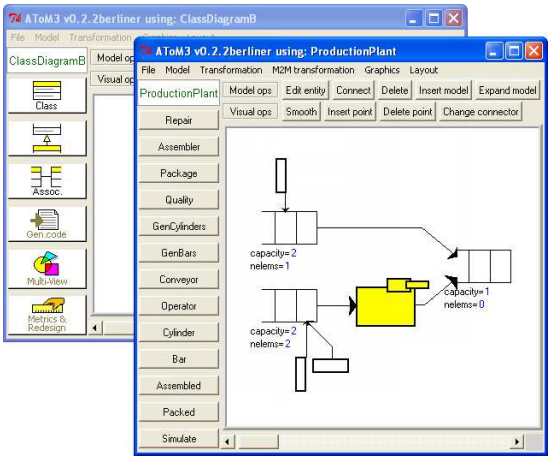

Fig. 13 Analysis results shown back in AToM^3.

6.4 Alternative analysis tools

Even though UMLtoCSP was chosen as the back-end tool, it is not the only candidate tool available. Other existing tools for the validation and verification of UML/OCL models (e.g. [2,10,17,25,47,49]) can be used instead.

Each tool follows a different approach and internal formalism to cope with the verification process, with its own set of advantages and drawbacks: bounded verification, need for user guidance, termination and so forth. Moreover, the richness of constructs in OCL 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 to verify properties that use 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. In the remainder of this section, we present several tools, their capabilities and their suitability to operate in the tool architecture proposed in this paper.

HOL-OCL [10] is a theorem prover for OCL based on the Higher-Order Logic theorem prover Isabelle. It is capable of proving complex properties on UML/OCL specifications like the correctness properties defined in Section 4. The decision procedures provided by HOL-OCL are *complete*: given a UML/OCL specification and a correctness property expressed in Higher-Order Logic, the tool is able to either prove or disprove the property. In this way, the tool is able to formally prove properties using logic deduction rules rather than looking for a counterexample within a bounded search space. However, due to the expressiveness of the underlying logic, some properties are undecidable. This means that, even though HOL-OCL uses heuristics to increase automation, it usually requires user guidance to complete proofs. In this sense, HOL-OCL might fail to hide the details of the verification process from the user. Moreover, there are still some parts of the OCL language not covered by this tool [11].

The Constructive Query Containment (CQC) method is a technique proposed for the analysis of database queries which can also be applied to reason on UML/OCL models [47,49]. Although this approach imposes some restrictions on the expressiveness of OCL constraints (e.g. no arithmetics), this decision procedure is complete. Regarding decidability, as the verification of UML/OCL models is undecidable in general the method may not terminate, but it is possible to analyse the UML/OCL model a priori to ensure the termination of the analysis in some cases [48].

Alloy [27] is a toolkit for the verification of software specifications. Even though it uses its own input notation based on first-order relational logic, there is a graphical front-end (UML2Alloy [2]) which partially maps UML/OCL constructs to their Alloy counterpart. Regarding its functionality, Alloy permits the definition of custom properties which can be checked using SAT solvers and bounded model checkers. Thus, it can check the correctness properties defined in Section 4. However, the Alloy notation provides limited support for arithmetic operators (e.g. no multiplication or division), restricting the set of expressions allowed in attribute conditions. On the other hand, as Alloy supports model checking, we would be able to check more complex properties on sequences of rule firings [5].

USE [25] is a validation environment for UML/OCL models. It provides both textual and graphical interfaces to instantiate classes, establish associations between objects and evaluate OCL constraints. From this point of view, USE can be useful to validate the correctness prop-

erties of graph transformations. However, USE does not support a completely automatic verification process. It can check whether there is an instance satisfying all the invariants, but the user needs to guide this computation by providing a script which specifies, among other things, the structure of the instances to be tested and the order in which constraints should be checked to improve performance. The MOVA tool [17] presents a similar set of advantages and drawbacks.

7 Related Work

There are two main sources of related work in the analysis of graph transformation rules: 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 [21,26,53], but they usually work with simple type graphs (i.e. without OCL constraints). Our work redefines some of these analysis properties, but taking into consideration a full-fledged meta-model which includes OCL integrity constraints, as well as OCL constraints in rules. Some preliminary efforts to integrate graph transformation with meta-modelling can be found in [35], where type graphs were extended with inheritance, and in [55], where edge inheritance and edge cardinalities were incorporated into type graphs. However, none of these works provides analysis capabilities.

With respect to graph-based tools, AGG [1] implements some graph transformation analysis techniques, like dependency analysis, critical pairs, sequence applicability and checks sufficient criteria for grammar termination. However, even though AGG handles type graphs with inheritance and cardinality constraints, it does not consider OCL in meta-models or in rules. Many other graph-based tools are moving towards supporting richer meta-modelling concepts. For example, the TGraph approach [20] allows ordered nodes and edges, and is supported by a number of tools to define, manipulate, analyze, query, visualize and transform graphs. Constraints are formulated using the GReQL query language, while transformations are specified by using the MOLA (Model Transformation Language) transformation engine [29]. Fujaba [24] is another graph-based tool, which allows defining graph transformation rules and includes advanced code generation capabilities. However, it does not provide rule analysis facilities. Altogether, we believe that the approach presented in this paper can be used with other tools and transformation approaches. Moreover, the use of OCL as a common target language enables interoperability of different transformation languages and tools.

Regarding the transformation of graph rules into other domains, their translation into OCL pre- and post-conditions was first presented in [12]. Here we give a more complete OCL-based characterization of rules that considers DPO and SPO semantics, NACs, and that encodes the LHS's matching algorithm as additional preconditions. In [12] the match is passed as parameter to the OCL expressions, assuming a predefined existing external mechanism. In addition, we exploit the resulting OCL expressions in order to enable the tool-assisted analysis of different rule properties.

In [4], rules are analysed using Petri net techniques to check safety properties. In particular, they use an unfolding construction that approximates the original graph grammar, in the sense that for any reachable state in the grammar, there is an equivalent one in the construction (but not the other way round). Hence, this analysis technique is useful to verify reachability properties like deadlocks. Although able to approximate infinite state grammars, as it is, the technique does not cope with attributes or meta-model restrictions.

In [5], rules are translated into Alloy in order to study the applicability of sequences of rules and the reachability of models. Even though Alloy is equipped with a SAT solver (so that similar properties to the ones we verify could be analysed), the properties in [5] are only related to reachability. Moreover, the integration with meta-model integrity constraints is not discussed. In our case, an encoding of reachability properties like the one proposed by the authors is also possible, but left for future work.

In this line of work, other approaches like [51,52, 57] rely on model-checking techniques to analyse reachability and invariants. In particular, in [57], rules, models and meta-models are transformed into Promela for model-checking with SPIN. The Groove tool [51] allows also model-checking of graph grammars, by using a dedicated explicit checker. Finally, in [52] a transformation of rules into the rewriting logic system Maude is proposed, where reachability analysis and LTL model checking is performed, using the Maude model checker. These three approaches do not take into account meta-models with integrity constraints or OCL constraints in rules. However, there are several efforts for supporting OCL in Maude, and integrating OCL in the approach of [52] is feasible. Our use of OCL as intermediate representation has the benefit that it is a tool independent, standard language, and moreover we can easily integrate attribute conditions and meta-model constraints. Finally, the kind of properties we analyse is also different. While these approaches focus on reachability and model checking, our use of UMLtoCSP – and the fact that it formulates the UML/OCL model as a CSP – makes it possible to analyse properties like applicability, overlapping and executability of rules, based on model finding capabilities, in contrast with space-state generation and exploration techniques.

In [16], we applied similar techniques to the analysis of declarative model-to-model transformation specifications, in particular triple graph grammars (TGGs) and QVT-relational transformations. However, there the in-

termediate representation was based on the definition of invariants between the source and target meta-models (more suited to the relational semantics of declarative transformations) instead of pre/post-condition contracts. The properties that we are interested in both cases are also different. For relational transformations we are more interested in checking whether the transformation is total, exhaustive or injective.

In our approach we analyse rules (or pairs of rules) in isolation, but we do not consider sequences of rules yet. Some recent works have started to analyse sequences, such as [33], where sufficient conditions for sequence applicability are stated, or [46] where the minimal graph enabling (or forbidding) the application of a sequence is calculated. We leave the study of rule sequences for future work.

Finally, this paper focuses on the analysis of graph transformation rules. However, some properties under consideration are also relevant to other transformation paradigms, e.g. the potential dependencies among different transformation steps. For example, two areas where dependency analysis is especially relevant are *program refactoring* [41], where different code restructuring operations may enable/disable each other, and *aspect-oriented development* [40], where aspects encode cross-cutting concerns in a software system and unexpected interactions may arise when composing different aspects. Graph transformation approaches and tools like AGG can be applied to this dependency analysis [40,42]. However alternative approaches for other transformation notations have also been proposed [30]. For example, the Condor tool [31,32] analyses dependencies among aspects formalized as *conditional transformations*, i.e. program transformations where a pre-condition controls whether the transformation is applied or not. Using a logic-based inference engine, Condor considers notions like rule conflict, causal dependence and rule sequences (which are not studied in this paper). Nevertheless, the graph transformation notation considered in our paper takes into account not only the structure of the graph but also the values of attributes. This type of analysis is outside the scope of Condor, as it is intended as an efficient light-weight approach.

8 Conclusions and Future Work

In this paper we have presented a new method for the analysis of the operational semantics of DSVLs expressed as graph transformation rules. Our method takes into account the (meta-)model structure of the DSVL and its well-formedness OCL constraints during the verification. 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, injective and non-injective matches, attribute computations and distinguishes DPO and SPO semantics.

We have automated the verification process by developing a bridge between AToM^3 (for specifying the meta-model and the graph transformation rules) and UML-toCSP (for verifying the resulting UML/OCL translation). We have also implemented a back-annotation mechanism that hides the analysis method and permits the designer to work in terms of the original DSVL only.

We believe this translation can be useful for other purposes as well. 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. Moreover it promotes the interoperability of transformation languages and tools.

As future work we would like to extend our analysis framework with the inclusion of model transformation testing capabilities [9], where relevant test models are automatically synthesized from our translation and passed on to the testing tool. We believe that our intermediate representation can also be useful to express model transformations specified with other model transformation languages (e.g. QVT). In this sense, we plan to explore the benefits of using our method when trying to compare and combine partial model transformations expressed in different languages.

We would also like to explore the automatic synthesis of attribute conditions and NACs for the rules, given the rule's actions and the meta-model constraints, in a similar way as graph constraints are converted into rule's pre-conditions [21]. The derived conditions should constrain the rule to ensure that it is strongly executable (in the sense of the definition of Section 4), improving this way the user experience when using the rules. Finally, we plan to combine our approach with others devoted to the verification and testing of rule sequences.

Acknowledgements. We thank the referees for their useful comments, which helped us to improve the paper. This work is sponsored by the Spanish Ministry of Science and Innovation, under projects "MODUWEB" (TIN2006-09678), "METEORIC" (TIN2008-02081) and "Design and construction of a Conceptual Modeling Assistant" (TIN208-00444/TIN - Grupo Consolidado), and UOC-IN3 research grant.

References

1. AGG graph transformation analysis tool: `http://tfs.cs.tu-berlin.de/agg/`
2. Anastasakis, K., Bordbar, K., Georg, G., Ray, I. 2007. *UML2Alloy: A challenging model transformation.* Proc. MODELS'07, LNCS 4735, pp. 436–450. Springer.

3. A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. *Evaluating the "Small Scope Hypothesis"*. Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.
4. Baldan, P., Corradini, A., König, B. 2001. *A static analysis technique for graph transformation systems*. Proc. CONCUR'01, LNCS 2154, pp. 381–395. Springer.
5. Baresi, L., Spoletini, P. 2006. *On the use of Alloy to analyze graph transformation systems*. Proc. ICGT'06, LNCS 4178, pp. 306–320. Springer.
6. D. Berardi, D. Calvanese, and G. D. Giacomo. 2005. *Reasoning on UML class diagrams*. Artificial Intelligence. Vol. 168(1-2), pp. 70-118. Elsevier.
7. Berry, D. M. Formal methods: the very idea. Some thoughts about why they work when they work. *Science of Computer Programming* 2002; 42(1): 11-27.
8. Borgida, A., Mylopoulos, J., Reiter, R. On the Frame Problem in Procedure Specifications. *IEEE Trans. Software Eng.* 1995; 21(10): 785-798.
9. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y. 2006. *Metamodel-based Test Generation for Model Transformations:an Algorithm and a Tool*. Proc. ISSRE'06, pp. 85–94, IEEE Computer Society.
10. Brucker, A. D., Wolff, B. 2006. *The HOL-OCL book*. Tech. Rep. 525, ETH Zurich.
11. Brucker, A. D., Wolff, B. 2009. *Semantics, Calculi, and Analysis for Object-oriented Specifications*. Acta Informatica 56(4):255-284.
12. Büttner, F., Gogolla, M. 2006. *Realizing graph transformations by pre- and postconditions and command sequences*. Proc. ICGT'06, LNCS 4178, pp. 398–413. Springer.
13. 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.
14. Cabot, J., Clarisó, R., Riera, D. 2008. *Verification of UML/OCL Class Diagrams Using Constraint Programming*. MoDeVVa 2008, ICST Workshop, pp. 73–80.
15. Cabot, J., Clarisó, R., Guerra, E., de Lara, J. 2008. *Analysing Graph Transformation Rules through OCL*. Proc. ICMT 2008, LNCS 5063, pp. 229-244. Springer.
16. Cabot, J., Clarisó, R., Guerra, E., de Lara, J. 2008. *An Invariant-Based Method for the Analysis of Declarative Model-to-Model Transformations*. Proc. MODELS 2008, LNCS 5301, pp. 37-52. Springer.
17. 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.
18. Dresden OCL Toolkit. `http://dresden-ocl.sourceforge.net/` (visited October 2008).
19. The ECL^{i}PS^e Constraint Programming System, `http://www.eclipse-clp.org`.
20. Ebert, J., Riediger, V., Winter, A. 2008. *Graph Technology in Reverse Engineering. The TGraph Approach*. Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics. pp. 67–81.
21. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.
22. 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 [53], pp. 247–312.
23. 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.
24. Fujaba tool suite home page: `http://wwwcs.uni-paderborn.de/cs/fujaba/`
25. Gogolla, M., Bohling, J., Richters, M. 2005. *Validating UML and OCL models in USE by automatic snapshot generation*. SoSyM 4(4):386–398. Springer.
26. Heckel, R., Küster, J.-M-., Taentzer, G. 2002. *Confluence of typed attributed graph transformation systems*. Proc. ICGT'02, LNCS 2505, pp. 161–176. Springer.
27. Jackson D. 2002. *Alloy: a lightweight object modelling notation*, ACM TOSEM, vol. 11(2), pp. 256–290.
28. Jouault, F., Allilairem, F., Bézivin, J., Kurtev, I., Valduriez, P. 2006. *ATL: a QVT-like transformation language*. OOPSLA Companion, pp. 719–720, ACM.
29. Kalnins, A., Barzdins, J., Celms, E. 2004. *Model Transformation Language MOLA*, Proc. MDAFA'04, LNCS 3599, pp. 62–76. Springer.
30. Katz, S. 2005. *A survey of verification and static analysis for aspects*. Technical Report AOSD-Europe Milestone M8.1, AOSD-Europe-Technion-1, Technion Israel.
31. Kniesel, G., Bardey, U. 2006. *An analysis of the Correctness and Completeness of Aspect Weaving*. WCRE'06, pp. 324–333, IEEE Computer Society.
32. Kniesel, G. 2009. *Detection and Resolution of Weaving Interactions*. Proc. TAOSD, LNCS 5490, pp. 135–186. Springer.
33. Lambers, L., Ehrig, H., Taentzer, G. 2008. *Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences*. Proc. GT-VMT'08, Electronic Communications of the EASST, Vol(10).
34. de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling*. Proc. FASE'02, LNCS 2306, pp. 174–188.
35. 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.
36. 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.
37. Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. 2004. Prentice Hall, 3^{rd} Edition.
38. Markovic, S., Baar, T. 2008. *Refactoring OCL annotated UML class diagrams*. SoSyM 7(1):25–47. Springer.
39. Mellor, S. J., Scott, K., Uhl, A., Weise, D. 2004. *MDA Distilled*. Addison-Wesley Object Technology Series.
40. Mehner, K., Monga, M., and Taentzer, G. 2006. *Interaction Analysis in Aspect-Oriented Models*. RE'06, pp. 66–75, IEEE Computer Society.
41. Mens, T. and Tourwé, T. 2004. *A Survey of Software Refactoring*. IEEE Trans. Softw. Eng., 30(2):126–139, IEEE.
42. Mens, T., Kniesel, G., Runge. O. 2006. *Transformation dependency analysis – a comparison of two approaches*. LMO'06 12:167–182. Hermes Science Publishing.
43. Mens, T., Taentzer, G., Runge, O. 2007. *Analysing refactoring dependencies using graph transformation*. SoSyM 6(3):269–285. Springer.

44. MOF 2.0 standard specification at: http://www.omg.org/spec/MOF/2.0/
45. OCL 2.0 standard specification at: http://www.omg.org/technology/documents/formal/ocl.htm
46. Pérez Velasco, P.P., de Lara, J. 2008. *Using Matrix Graph Grammars for the Analysis of Behavioural Specifications: Sequential and Parallel Independence*. Electr. Notes Theor. Comput. Sci. 206, pp. 133–152.
47. Queralt, A., Teniente, E. 2006. *Reasoning on UML Class Diagrams with OCL constraints*. ER 2006, LNCS 4215, pp. 497–512. Springer.
48. Queralt, A., Teniente, E. 2008. *Decidable Reasoning in UML Schemas with Constraints*. CAISE 2008, pp.: 281-295. Springer.
49. Queralt, A., Teniente, E. 2008. *Validation of UML Conceptual Schemas with Operations*. CAISE Forum 2008, pp. 101–104. CEUR Workshop Proceedings.
50. QVT standard specification at: http://www.omg.org/docs/ptc/05-11-01.pdf
51. Rensink, A. 2008. *Explicit State Model Checking for Graph Grammars*. Concurrency, Graphs and Models, LNCS 5065, pp. 114–132. Springer.
52. Rivera, J. E., Guerra, E., de Lara, J., Vallecillo, A. 2008. *Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude*. Proc. SLE'08, LNCS 5452, pp. 54–73. Springer.
53. Rozenberg, G. (ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.
54. 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.
55. Taentzer, G., Rensink, A. 2005. *Ensuring structural constraints in graph-based models with type inheritance*. Proc. FASE'05, LNCS 3442, pp. 64–79. Springer.
56. A. Schürr. 1994. *Specification of graph translators with triple graph grammars*. In *WG'94*, *LNCS* 903, pp. 151–163. Springer.
57. Varró, D. 2004. *Automated formal verification of visual modeling languages by model checking*. SoSyM 3(2):85–113. Springer.
58. Völter, M., Stahl T. 2006. *Model-Driven Software Development*. Wiley.
59. XML Metadata Interchange (XMI), v2.1.1 standard specification at: http://www.omg.org/cgi-bin/doc?formal/2007-12-01

Appendix

This appendix provides the (non-simplified) OCL translations for the rules "assemble", "work", "move" and "change", not included in Section 3. For the "assemble" rule, the *matchLHS* operation is provided in two different versions, depending on the DPO or SPO semantics.

Rule assemble

context System::assemble()
pre: Cylinder::allInstances()−> exists(c|
Conveyor::allInstances()−> exists(co1, co2|
Bar::allInstances()−> exists(b|
Assembler::allInstances()−> exists(a|
Operator::allInstances()−> exists(op|
matchLHSassemble(c,co1,b,a,op,co2))))))
post: Cylinder::allInstances@pre()−> exists(c|
Conveyor::allInstances@pre()−> exists(co1, co2|
Bar::allInstances@pre()−> exists(b|
Assembler::allInstances@pre()−> exists(a|
Operator::allInstances@pre()−> exists(op|
matchLHSassemble'(c,co1,b,a,op,co2) and
changeRHSassemble(c,co1,b,a,op,co2))))))

SPO Semantics
context System::matchLHSassemble(c:Cylinder,
co1:Conveyor, b:Bar, a:Assembler,
op:Operator, co2:Conveyor): Boolean
body: c.conveyor−> includes(co1)
and b.conveyor−> includes(co1) and
co1.om−> includes(a) and a.oc−> includes(co2)
and a.operator−> includes(op) and
co2.nelems+1 <= co2.capacity

---

context System::matchLHSassemble'(c:Cylinder,
co1:Conveyor, b:Bar, a:Assembler,
op:Operator, co2:Conveyor): Boolean
body: c.conveyor@pre−> includes(co1) and
b.conveyor@pre−> includes(co1) and
co1.om@pre−> includes(a) and
a.oc@pre−> includes(co2)
and a.operator@pre−> includes(op) and
co2.nelems@pre+1 <= co2.capacity@pre

DPO Semantics
context System::matchLHSassemble(c:Cylinder,
co1:Conveyor, b:Bar, a:Assembler,
op:Operator, co2:Conveyor): Boolean
body: c.conveyor−> includes(co1) and
c.conveyor−> excluding(co1)−> isEmpty() and
b.conveyor−> includes(co1) and
b.conveyor−> excluding(co1)−> isEmpty() and
co1.om−> includes(a) and a.oc−> includes(co2)
and a.operator−> includes(op) and
co2.nelems+1 <= co2.capacity

---

context System::matchLHSassemble'(c:Cylinder,
co1:Conveyor, b:Bar, a:Assembler,
op:Operator, co2:Conveyor): Boolean
body: c.conveyor@pre−> includes(co1) and
c.conveyor@pre−> excluding(co1)−> isEmpty()
and b.conveyor@pre−> includes(co1) and
b.conveyor@pre−> excluding(co1)−> isEmpty()
and co1.om@pre−> includes(a) and
a.oc@pre−> includes(co2) and
a.operator@pre−> includes(op) and
co2.nelems@pre+1 <= co2.capacity@pre

context System::changeRHSassemble(c:Cylinder,
co1:Conveyor, b:Bar, a:Assembler,
op:Operator, co2:Conveyor): Boolean
body: Cylinder::allInstances()−> excludes(c) and
Bar::allInstances()−> excludes(b) and
co2.nelems=co2.nelems@pre+1 and
co1.nelems=co1.nelems@pre-2 and
as.oclIsNew() and as.oclIsTypeOf(Assembled) and
as.conveyor−> includes(co2)

Rule work

context System::work()
pre: Machine::allInstances()−>exists(m|
Operator::allInstances()−>exists(op|
matchLHSwork(m,op)
and not Operator::allInstances()−>exists(op1|
matchNAC1work(m,op,op1) and op1 <> op)
and not Machine::allInstances()−>exists(m1|
matchNAC2work(m,op,m1) and m1 <> m))
post: Machine::allInstances@pre()−>exists(m|
Operator::allInstances@pre()−>exists(op|
matchLHSwork'(m,op)
and not Operator::allInstances@pre()−>exists(op1|
matchNAC1work'(m,op,op1) and op1 <> op)
and not Machine::allInstances@pre()−>exists(m1|
matchNAC2work'(m,op,m1) and m1 <> m)
and changeRHSwork(m,op)))

---

context System::matchLHSwork(m:Machine,
op:Operator): Boolean
body: true

---

context System::matchLHSwork'(m:Machine,
op:Operator): Boolean
body: true

---

context System::matchNAC1work(m:Machine,
op: Operator, op1: Operator): Boolean
body: m.operator−>includes(op1)

---

context System::matchNAC1work'(m:Machine,
op: Operator, op1: Operator): Boolean
body: m.operator@pre−>includes(op1)

---

context System::matchNAC2work(m:Machine,
op: Operator, op1: Operator): Boolean
body: op.machine−>includes(m1)

---

context System::matchNAC2work'(m:Machine,
op: Operator, op1: Operator): Boolean
body: op.machine@pre−>includes(m1)

---

context System::changeRHSwork(m:Machine,
op:Operator): Boolean
body: m.operator−>includes(op)
and not m.operator@pre −>includes(op)

Rule move

context System::move()
pre: Conveyor::allInstances()−>exists(co1,co2|
Piece::allInstances()−>exists(p|
matchLHSmove(co1,co2,p) and not
(Machine::allInstances()−>exists(m |
Operator::allInstances()−>exists(op|
matchNACmove(co1,co2,p,m,op)))))
post: Conveyor::allInstances@pre()−>exists(co1,co2|
Piece::allInstances@pre()−>exists(p|
matchLHSmove'(co1,co2,p) and not
(Machine::allInstances@pre()−>exists (m |
Operator::allInstances@pre()−>exists(op |
matchNACmove'(co1,co2,p,m,op)))) and
changeRHSmove(co1,co2,p)))

---

context System::matchLHSmove(co1:Conveyor,
co2: Conveyor, p:Piece): Boolean
body: p.conveyor−>includes(co1) and
co1.next−>includes(co2) and
co2.nelem+1 <= co2.capacity

---

context System::matchLHSmove'(co1:Conveyor,
co2: Conveyor, p:Piece): Boolean
body: p.conveyor@pre−>includes(co1) and
co1.next@pre−>includes(co2) and
co2.nelem@pre+1 <= co2.capacity@pre

---

context System::matchNACmove(co1:Conveyor,
co2: Conveyor, p: Piece, m: Machine,
op: Operator): Boolean
body: co1.om−>includes(m) and
m.operator−>includes(op)

context System::matchNACmove'(co1:Conveyor,
co2: Conveyor, p: Piece, m: Machine,
op: Operator): Boolean
body: co1.om@pre−>includes(m) and
m.operator@pre−>includes(op)

---

context System::changeRHSmove(co1:Conveyor,
co2: Conveyor, p:Piece): Boolean
body: p.conveyor−>excludes(co1) and
p.conveyor−>includes(co2) and
not p.conveyor@pre−>includes(co2) and
co1.nelems=co1.nelems@pre−1 and
co2.nelems=co2.nelems@pre+1

Rule change

```
context System::change()
pre:  Piece::allInstances() ->exists (p |
      Conveyor::allInstances() ->exists (co |
      Machine::allInstances()->exists (m1,m2 |
      Operator::allInstances()->exists (op |
      matchLHSchange(p,co,m1,m2,op) and not
        (Operator::allInstances()->exists (op1 |
        matchNACchange(p,co,m1,m2,op,op1)
        and op<>op1))))))
post: Piece::allInstances@pre() ->exists (p |
      Conveyor::allInstances@pre() ->exists (co |
      Machine::allInstances@pre()->exists (m1,m2 |
      Operator::allInstances@pre()->exists (op |
      matchLHSchange'(p,co,m1,m2,op) and not
        (Operator::allInstances@pre()->exists (op1 |
          matchNACchange'(p,co,m1,m2,op,op1)
          and op1<>op)) and
      changeRHSchange(p,co,m1,m2,op)))))
```

```
context System::matchLHSchange( p: Piece,
                co: Conveyor, m1: Machine,
                m2: Machine, op: Operator ): Boolean
body: p.conveyor->includes(co) and
      co.om->includes(m1) and
      op.machine->includes(m2)
```

```
context System::matchLHSchange'( p: Piece,
                co: Conveyor, m1: Machine,
                m2: Machine, op: Operator ): Boolean
body: p.conveyor@pre->includes(co) and
      co.om@pre->includes(m1) and
      op.machine@pre->includes(m2)
```

```
context System::matchNACchange( p: Piece,
                co: Conveyor, m1: Machine,
                m2: Machine, op: Operator,
                op1: Operator ): Boolean
body: m1.operator-> includes(op1)
```

```
context System::matchNACchange'( p: Piece,
                co: Conveyor, m1: Machine,
                m2: Machine, op: Operator,
                op1: Operator ): Boolean
body: m1.operator@pre-> includes(op1)
```

```
context System::changeRHSchange( p: Piece,
                co: Conveyor, m1: Machine,
                m2: Machine, op: Operator ): Boolean
body: op.machine->excludes(m2)
      and op.machine->includes(m1)
      and not op.machine@pre-> includes(m1)
```