High-Level Replacement Units and their Termination Properties[1]

Paolo Bottoni[a], Kathrin Hoffmann[a,b], Francesco Parisi Presicce[a,c], Gabriele Taentzer[b]

[a]University of Rome "La Sapienza - Italy, [b]Technical University Berlin - Germany, [c]George Mason University - USA

Abstract

Visual rewriting techniques, in particular graph transformations, are increasingly used to model transformations of systems specified through diagrammatic sentences. Several rewriting models have been proposed, differing in the expressivity of the types of rules and in the complexity of the rewriting mechanism; yet, for many of them, basic results concerning the formal properties of these models are still missing. In this paper, we give a contribution towards solving the termination problem for rewriting systems with external control mechanisms. In particular, we obtain results of more general validity by extending the concept of transformation unit to high-level replacement systems, a generalization of graph transformation systems. For high-level replacement units, we state and prove several abstract properties based on termination criteria. Then, we instantiate the high-level replacement systems by attributed graph transformation systems and present concrete termination criteria. We explore some types of rules and replacement units for which the criterion can be established. These are used to show the termination of some replacement units needed to express model transformations formalizing refactoring.

Keywords. Visual transformations, transformation units, high level replacement, termination, refactoring.

1 Introduction

Visual rewriting techniques are increasingly used to model transformations of systems specified through diagrammatic sentences. Researchers are moving from the specification of static aspects of languages (defined through parsing processes) to the modeling of their dynamics. Graph transformations, in particular, are a widespread formalism with applications to parsing, model animation or transformation. Moreover, a whole new wealth of problems, such as software or model evolution [23, 20, 4, 18], arises from the diffusion of UML as a tool for the specification of both software and general systems.

When specifying such transformations, it is hardly the case that a single, unstructured diagram rewriting system is used to define complex transformations. A typical problem is to steer the progress of the transformation towards some well-defined configuration of the diagram, i.e. state of the system. This

[1]Work partially supported by the EU TMN SeGraVis

may involve the definition of some sequence of rule applications, as well as the prevention of repeated application of the same rule to the same match, or of cyclic repetitions of the same sequence of applications.

In general, guaranteeing such properties of the rewriting process is equivalent to proving its termination, an undecidable problem in its uniform version [21], but which can be studied for individual rewriting systems, following the classical approach – as introduced by Dershowitz and Manna in [8] – of proving termination by constructing a monotone measure function on some multiset, and showing that the value of such a function decreases at each application. Further termination criteria use polynomial orderings, recursive path orderings, etc. [7].

Concretely used approaches, however, exploit some form of control on rule application. Two major control forms are layering and rule expressions. With layering, particularly used in parsing [2, 6], rules are partially ordered according to their content, so that some elements can be produced or removed only at specific stages of the rewriting process. Rule expressions exploit constructs such as iteration and branching to constrain the admissible sequences of rule application [22, 17, 3, 18]. Under these restrictions, the problem is also reduced by having to prove the termination of only those processes which are allowed by the layering condition or by rule expressions.

Different forms of rules can be used even under these control forms, for example to admit negative application conditions, or path expressions by which unlimited context can be taken into account in a rule. These extensions to the basic form of a rule make the problem of determining properties of a rewriting process more complicated. A trade-off must therefore be found between the overall expressiveness of the rewriting system to be used for model transformation and the possibility of assessing properties of the process. In this paper we introduce the notion of high-level replacement units, as an extension of transition units [17], applied to a more abstract version of graph transformation systems, namely high-level replacement systems [12]. In this way, we can also incorporate into a simple rule form some controls traditionally expressed by negative application conditions. Such a transformation approach is practical, yet simple enough for formal reasoning. In particular, we obtain an abstract property that a function has to satisfy in order to be used as a termination criterion for such units.

Paper organization. Section 2 introduces related work, while Section 3 adapts the concept of transformation units to define high-level replacement units. In Section 4, the use of high-level replacement units in model transformation is illustrated by an example in which replacement units are used to specify transformations of UML models consequent to software refactoring. Section 5 discusses an abstract termination criterion for high-level replacement systems which is used to show a number of termination results. Section 6 presents concurrent productions and shows a corresponding termination result. In Section 7, we discuss some concrete termination criteria. In Section 8 some hints towards the application of our approach to special types of rules are given. Finally, conclusions are given in Section 9.

2 Related work

Transformation units have been introduced by Kreowski and Kuske in [17] and extensively used for several types of visual transformations since. Küster *et al.* have considered the role of transformation units in defining transformations of UML models [18]. In particular, they have studied the problem of termination and confluence. Recognising, as demonstrated by Plump in [21], that termination of graph rewriting is undecidable in general, they provide some intuitive consideration on the causes for termination or non-termination of transformation units iterating as long as possible the use of some given rules. However, they do not present results on the iteration of sequences of rules, for which we provide some termination criteria here. Termination criteria for graph transformation have already been considered by Aßmann [1], who sticks to a concrete set of criteria and has not developed a general approach to termination based on criteria, as we do in this paper.

The combined use of negative application conditions, set nodes, and control expressions for the management of visual transformation processes has been proposed in several occasions. In [6], layering conditions were applied to ensure termination of parsing processes. The general problem of proving termination of a given transformation unit is equivalent to introducing some form of local layering, so that the conditions on elimination or insertion of elements in the diagram proceed in accordance with the decrease or increase of the adopted monotone function. The termination conditions given in [6] have been taken up in [9] and formalized for graph transformation systems. Moreover, that work contains an additional set of termination conditions taking negative application conditions on non-deleting rules into account.

3 An algebraic setting for termination

High-Level Replacement Systems [12, 11] are a generalisation of the graph transformation approach and fit into the general approach at the basis of the definition of transformation units [17]. The resulting notion is called high-level replacement unit. Its semantics is given by the set of all possible derivation sequences. Thereafter, high-level replacement units are instantiated by attributed graph transformation.

3.1 High-Level Replacement Units

Let CAT be category with a distinguished morphism class M, such that CAT has pushouts and pullbacks along M-morphisms, i.e. if one of the given morphisms is in M, then also the opposite one is in M, and M-morphisms are closed under pushouts and pullbacks (the notation (CAT, M) will also be used to indicate CAT).

Definition 1 (rule) *A* rule $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$ *is given by two morphisms l and r of M. Let $L(p)$ be the left-hand side and $R(p)$ the right-hand side of rule p.*

A transformation unit controls the rule application by a set of control conditions specified by expressions over rule names.

Definition 2 (control expressions) *The class $\mathcal{C}$ of* control expressions *over $Names$ (representing a set of rule names) is recursively defined by*

- $Names \subseteq \mathcal{C}$,
- $C_1; C_2 \in \mathcal{C}$, *if* $C_1, C_2 \in \mathcal{C}$,
- $C_1 \mid C_2 \in \mathcal{C}$, *if* $C_1, C_2 \in \mathcal{C}$,
- **asLongAsPossible** C **end** $\in \mathcal{C}$, *if* $C \in \mathcal{C}$.

The intended meaning of the operator ; is the application of the expression C_1 followed by the application of the expression C_2. The operator ; is left associative, i.e. $C_1; C_2; C_3 = (C_1; C_2); C_3$. The operator *choice* (|) allows the application of either the expression C_1 or C_2. This operator is both left and right associative, so that $C_1 \mid C_2 \mid C_3 = (C_1 \mid C_2) \mid C_3 = C_1 \mid (C_2 \mid C_3)$. Finally, the intended meaning of the operator **asLongAsPossible** C **end** is the (sequential) application of the expression C as long as its application is possible.

Definition 3 (high-level replacement unit) *A high-level replacement unit $RU = (P, nm, C)$ in a category CAT, or just* replacement unit*, consists of a finite set P of rules, a bijective function $nm : P \to Names$, and a control expression $C \in \mathcal{C}$ over $Names$.*

High-level replacement units are transformation units in the sense of [17], where we substitute objects in a generic category CAT for graphs in a class. Hence, we have a transformation approach consisting of: 1) the class of objects in CAT; 2) the class of rules in this category; 3) an operator of rule application, specified by the definition of direct derivation as given in Definition 4; 4) a class of control expressions as given in Definition 2; and 5) a class of class expressions, each coinciding with the class of objects in CAT itself, meaning that every object in CAT is an acceptable final object. Hence, a replacement unit is a transformation unit with objects of CAT as initial and terminal graphs. Moreover, the set of imported units for a high-level replacement unit is always empty. In contrast to transformation units, whose semantics is given by the collection of graphs resulting from the application of the unit to an initial graph, the semantics of high-level replacement units is defined by derivation sets.

Definition 4 (match and direct derivation) *Given an object G and a rule $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$, a* match *of p to G is a morphism $m : L \to G$. A* direct derivation *d from G to H by p and match m, $d : G \Rightarrow_{p,m} H$, is given by a double pushout (see Figure 1). Let start and end be two projections from direct derivations to objects such that $start(d) = G$ and $end(d) = H$. A derivation $id : G \Rightarrow_{p_{id},m} G$, is called identical. Given a set P of rules, $Der(P) = \{G \Rightarrow_{p,m} H | G, H \in Obj(CAT) \wedge p \in P\}$ is the set of all direct derivations with rule set P.*

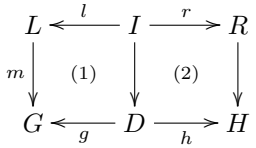

Figure 1: The double pushout approach

In the following, we will also employ the notations $G \Rightarrow_p H$ or $G \Rightarrow H$, when we are not interested in the specific match or rule for the derivation.

Definition 5 (derivation sequence) *Given a set P of rules, a* derivation sequence *on P is defined by a function $s : \mathcal{N} \to Der(P)$ with $start(s(i+1)) = end(s(i))$ for $i \geq 0$. The length of a derivation sequence s is n if $s(i)$ is the identity derivation for all $i > n$, and $end(s(n))$ is called the* derivation result*. The concatenation $s \circ t$ of two derivation sequences s – of length m – and t – such that $start(t(0)) = end(s(m))$ – is the derivation sequence u with $u(i) = s(i)$ for $0 \leq i \leq m$ and $u(m+i) = t(i-1)$ for $0 < i$.*

If t has length n, then the concatenation is finite and $u(m+i) = t(i-1)$ for $0 < i \leq n$.

Definition 6 (derivation subsequence) *A derivation sequence d_1 is a* subsequence *of the derivation sequence d_2 ($d_1 \preceq d_2$), if there exists a derivation sequence d_3 such that $d_1 \circ d_3 = d_2$.*

It is obvious that the relation $\preceq$ is a partial order.

Definition 7 (derivation sets) *A* derivation set *der consists of a number of derivations. If all derivations in der start at object G, it is also called der_G. The* concatenation *of two derivation sets der_1 and der_2 is given by $der_1 \circ der_2 = \{d_1 \circ d_2 | d_1 \in der_1, d_2 \in der_2\}$. The* product *$der^n$ is defined by $der^{n-1} \circ der$ for $n \in \mathcal{N}$. der^0 is the derivation of length 0, indicated by λ, leaving G unaltered. λ is the neutral element of the concatenation operation. The* star product *der^* is defined by $\bigcup_{i \geq 0} der^i$.*

Definition 8 (derived rule) *Given a derivation sequence d of length 1, $d(1)$ is the direct derivation $d : G \Rightarrow_{p,m} H$ as defined in Def. 4, and its* derived rule *is $p_d : G \stackrel{g}{\leftarrow} D \stackrel{h}{\rightarrow} H$. Given two finite derivation sequences $d_1 : G \Rightarrow^* H$ and $d_2 : H \Rightarrow^* K$ with their derived rules $p_{d_1} : G \stackrel{g}{\leftarrow} D_1 \stackrel{h_1}{\rightarrow} H$ and $p_{d_2} : H \stackrel{h_2}{\leftarrow} D_2 \stackrel{k}{\rightarrow} K$, the derived rule of $d = d_1 \circ d_2$ exists and is defined by $p_d : G \stackrel{g \circ c_1}{\leftarrow} D \stackrel{k \circ c_2}{\rightarrow} K$ where c_1 and c_2 are the pullback of h_1 and h_2 respectively, if they exist.*

Definition 9 (semantics of control expression) *Given an object G and a replacement unit $RU = (P, nm, C)$, the semantics of RU applied to G is the set $der(C)_G$ of all possible derivation sequences starting at G and applying rules of P according to C.*

1. $C = nm(p)$: $der(C)_G = \{d \in Der(\{p\}) | start(d) = G\}$,
2. $C = C_1; C_2$: $der(C)_G = \bigcup_{d_1 \in der(C_1)_G} d_1 \circ der(C_2)_H$, *if* $H = end(d_1)$
3. $C = C_1 \mid C_2$: $der(C)_G = der(C_1)_G \cup der(C_2)_G$,
4. $C =$**asLongAsPossible** C' **end**: $der(C)_G = \{d \in der(C')^*_G \mid d$ *is maximal in* $der(C')^*_G$ *wrt.* $\preceq\}$

Coherently with this semantics, a replacement unit of type $nm(p)$ fails if p is not applicable, $C_1; C_2$ fails if either C_1 or C_2 fails, $C_1 \mid C_2$ fails if both C_1 and C_2 fail, and **asLongAsPossible** C'**end** fails in one of the following cases: 1) C' fails and it is not of the form $nm(p); C''$; 2) C' is of this latter form and C'' fails; 3) C' is of one of the forms $nm(p)$ or $E_1 \mid E_2$ and it fails at the first iteration. The rationale for this definition of failure in the case of iteration is that C' is expected to be executable, so that it has to be performed at least once, while if the expression to be iterated is a sequence, it must be executed completely, so that at each iteration, if the first rule in the sequence is executed without failure, the rest of the sequence must be executed as well. Due to the transactional nature of the units, failure restores the situation prior to its attempted application.

3.2 Instantiation by attributed graph transformation

In the following, we recall attributed graph structures as defined in [14], where the category of attributed graphs and morphisms with a distinguished class M of morphisms has been shown to be an adhesive HLR-category. Thus, the assumptions for (CAT, M), as stated in the beginning of Section 3.1 are fullfilled. Furthermore, we know that the Church-Rosser, Parallelism, and Concurrency Theorems hold (see [14]).

Definition 10 (category of attributed graph structure signatures) *A* graph structure signature $GSIG = (S_G, OP_G)$ *is an algebraic signature with unary operations* $op : s \to s'$ *in* OP_G *only. An* attributed graph structure signature $ASSIG = (GSIG, DSIG)$ *consists of a graph structure signature* $GSIG$ *and a data signature* $DSIG = (S_D, OP_D)$ *with attribute value sorts* $S'_D \subseteq S_D$ *such that* $S'_D = S_D \cap S_G$ *and* $OP_D \cap OP_G = \emptyset$. *ASSIG is called well-structured if for each* $op : s \to s'$ *in* OP_G *we have* $s \notin S_D$. *The category of all ASSIG-algebras and ASSIG-homomorphisms* $f = (f_G SIG, f_D SIG) \in M$*is denoted by* **ASSIG-Alg**. *The distinguished class* M *for* **ASSIG-Alg** *is defined by* $f \in M$ *if* f_{GSIG} *is injective.*

Remark: Given an S_D-indexed set of variables $X = (X_s)_{s \in S_D}$, all rule objects are attributed by the term algebra $T_{DSIG}(X)$.

Proposition 1 (local C-R, parallelism and concurrency) *(from [14]). The Local Church - Rosser theorems I and II, the Parallelism theorem, and the Concurrency theorem as stated in [12] are valid for each graph transformation system based on* **ASSIG-Alg**.

4 Examples from Model Refactoring

In this section we present an example of refactoring from [15], and specify the transformations that the model of a system software must go through to be maintained consistent with the modifications in the code. Elsewhere ([5]) we have illustrated how to coordinate transformations occurring in the code – represented at the level of its abstract syntax – with those occurring in the model, represented by graphs, typed according to the UML metamodel.

In [5], we studied how modifications of the code could affect different components of the model, typically class and sequence/collaboration diagrams, and we modelled the necessary coordination of the transformations of such representations through transition units defined on hierarchical distributed graph transformations. However, we are now interested in this example only to illustrate the termination problem of replacement units. As distribution does not add causes for non-termination (actually, it may reduce them), we discuss here a refactoring involving only modifications in class diagrams.

`PullUpVariable(class; attr)` moves the variable named `attr` from subclasses of `class` to `class`. This is used when a variable of the same name with the same type is used throughout all the subclasses of a given class. Previous renaming of such variables may have occurred in order to create the conditions for the application of this refactoring. It is important to note that a requirement for this refactoring, as it guarantees behavior preservation, is that *all* the subclasses of `class` own a copy of this variable. Hence, this refactoring must be performed through the repeated application of a rule which moves the variable from a subclass to its superclass, checking that this occurs for all the (direct) subclasses of the class to which the variable has been moved.

In order to keep the theory simple, we exploit two small adaptations of the metamodel for class diagrams. The first is that each `Class` node n is associated with a node of type `ClassDescriptor`, possessing an attribute *sons* which keeps a constantly updated list of the direct subclasses of n. Without such coding of the subclass relation, we would need an additional process to check that the pulling process has been conducted for all subclasses. We also adopt a small variation on the metamodel contained in the version 2.0 of UML, in which package nodes are defined to maintain a composition relation with all the classes (actually all the types) present in the package. Here, we replace arcs with attribute values, so that the package node maintains an attribute `content` storing a list of the classes in it. Hence, the creation and deletion of a class in a package involves the update of the `content` attribute. This way, we can check that a class is not already present in the package by querying the value of `content`, and without having to recur to a negative application condition.

The rule for class creation results as in Figure 2. Apart from the check on the value of `content` in the *package* node, there are no other preconditions.

Declaring that a class is a subclass of another, or removing the generalisation relation between two classes, produces the effects described by the rules of Figures 3 and 4, respectively. In all the rules discussed in this section, the values of the parameters of the rule, to be matched on attributes of the concrete

instances, are indicated by showing their names in *italic*, while variables present in the rule, and subject to a unification process with values of attributes in the matching subgraph, are written with a capitalized initial. In this version of the rules we do not check for the absence of inheritance cycles or of multiple inheritance. The first condition would need a NAC involving a path expression. The second condition could be checked by a NAC presenting another `specialization-generalization` path leading into a different class.

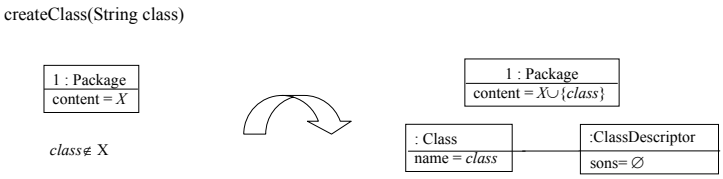

Figure 2: The rule to create a class, associated with the node recording all its subclasses.

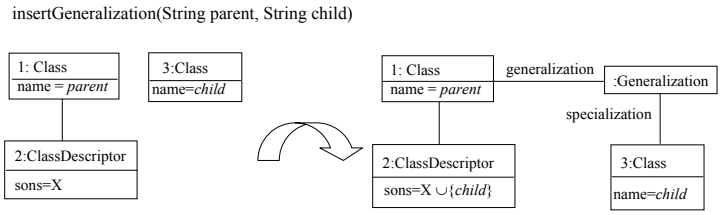

Figure 3: The rule to insert a generalization relation between two classes.

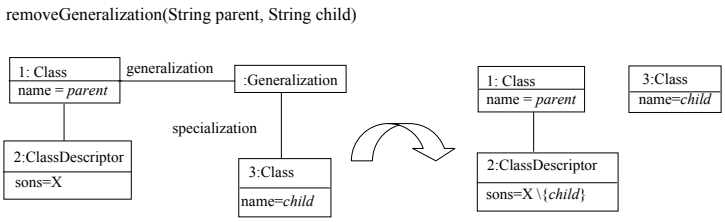

Figure 4: The rule to remove a generalization relation.

The elimination of a class is accompanied by the destruction of its associate `ClassDescriptor` node and the update of package node, as shown in Figure

5. With the DPO approach, class elimination is possible only when it has been stripped of all owned features and relations, including generalization and specialization, with other classes.

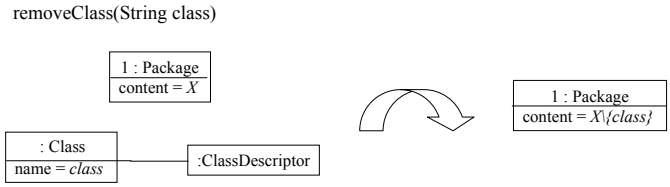

Figure 5: The rule to remove a class.

The replacement unit which performs the pulling up of the variable starts with a rule, called `startCheck`, which marks the class to which the variable must be pulled up by associating with it an `Auxiliary` node, whose `auxSet` attribute will contain all the names of the class from which the pulled up variable has been removed (see Figure 6).

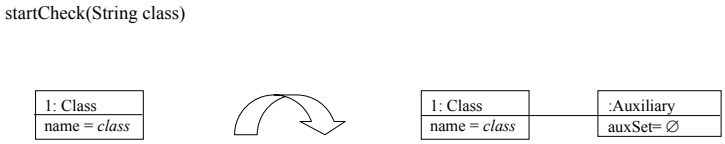

Figure 6: The rule to attach an `Auxiliary` node to the class that the variable must be pulled up to.

The rule `startRefactoring`, presented in Figure 7, extracts the variable from one of the subclasses of `class`. The name of the subclass is added to the `auxSet` attribute of the `Auxiliary` node. Note that at least one such subclass must exist, otherwise the whole replacement unit will fail.

The rule `completeRefactoring`, shown in Figure 8 is then applied as long as possible, i.e. until there are subclasses of `class` owning a variable of the same name and type as the one first moved. For each subclass to which the rule is applied, the name of the class is added to the `auxSet` attribute of the `Auxiliary` node.

Finally, the rule `doFinalCheck` (see Figure 9) controls that all the subclasses of `class` have been considered and that from each of them the variables has been pulled up, by comparing the values of the two attributes, `sons` and `auxSet` for the nodes of type `ClassDescriptor` and `Auxiliary`, respectively, associated with `class`. The failure of this rule indicates that some subclass did not possess

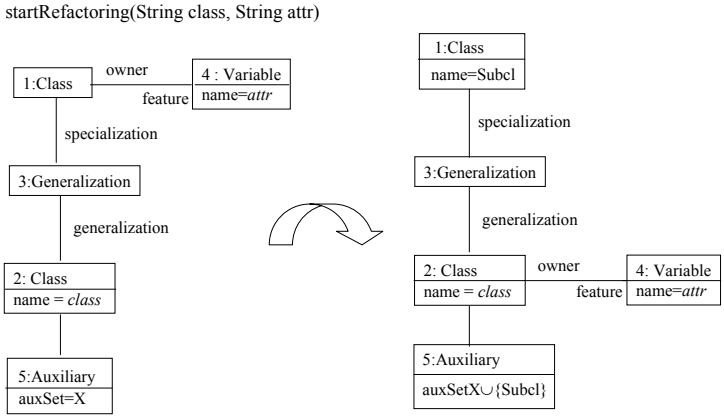

Figure 7: The rule to pull the variable `attr` from a non deterministically chosen subclass of `class`.

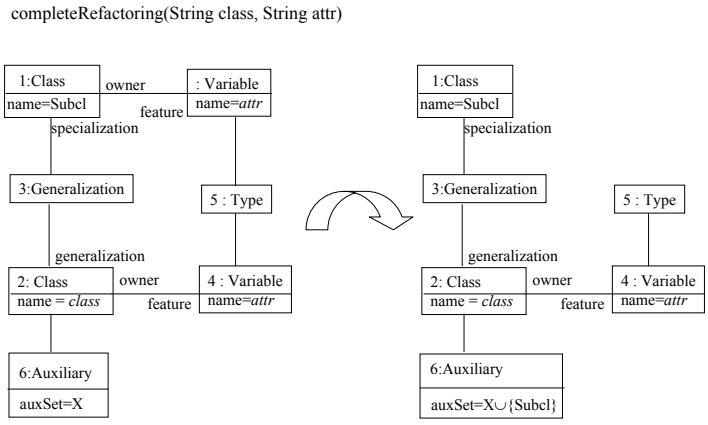

Figure 8: The rule to pull the variable `attr` from any subclass possessing a copy of the variable to be pulled up.

the variable, and makes the whole replacement unit fail. If this occurs, due to the transactional behavior of units, the situation prior to the attempted unit is restored. Hence, either because rule `doFinalCheck` removes the `Auxiliary` node, or because of restoration after failure, no such node exists once the process is completed.

Hence, the complete replacement unit is expressed as:

```
pullUpVariable(String class, String attr) =
```

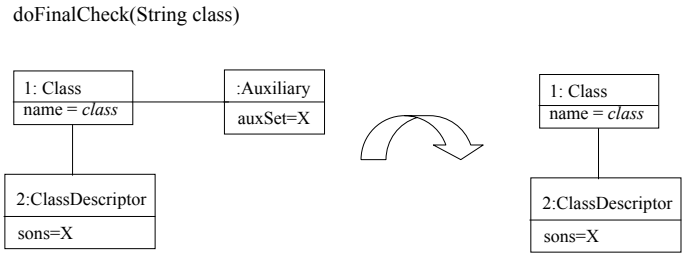

Figure 9: The final rule to check success of the pull up transformation.

```
startCheck(class);
startRefactoring(class, attr);
asLongAsPossible completeRefactoring(class, attr) end;
doFinalCheck(class)
```

5 Termination of Replacement Units

Termination of replacement units is not always guaranteed. If an expression **asLongAsPossible** contains a rule that can be applied indefinitely to the result graphs, the replacement unit does not terminate. Next, we give conditions for the termination of replacement units.

5.1 A General Termination Criterion for High-Level Replacement Units

Let $\mathcal{G}$ be the class of all objects in a category CAT and $\mathcal{P}$ be the set of all the rules on $\mathcal{G}$. In the following, we discuss the notion of termination criterion, by assigning a natural number to each object of CAT.

Definition 11 (termination criterion) *A function $F : \mathcal{G} \to \mathcal{N}$ from objects to natural numbers is a* termination criterion *for (CAT, M) if for any two arbitrary morphisms $a : C \to A$ and $b : C \to B$ in M, the value $F(A +_C B)$ of the pushout object $A +_C B$ of a and b is given by $F(A +_C B) = F(A) + F(B) - F(C)$. Given a rule p with morphisms in M, a termination criterion F for CAT is a termination criterion for p if $F(L(p)) > F(R(p))$.*

The reason for removing the value of F in C from the sum is that it contains the elements common to both A and B, which would otherwise be considered twice. Even if in Definition 11 we have considered p to be a rule in the double pushout approach (DPO), we refer only to its left-hand and right-hand sides. Hence, the same definition could, under the appropriate definition of the class

M, be used also for the single pushout approach (SPO). We develop here the theory only for the DPO approach.

Proposition 2 (termination of direct derivations) *If F is a termination criterion for rule p, then it is also a termination criterion for all the derived rules p_d of all $d : G \Rightarrow_p H$.*

Proof 1 *Since F is a termination criterion for the rule $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$ we have $F(L) > F(R)$. For each direct derivation $G \Rightarrow_p H$ given by a double pushout as in Figure 1 we need $F(G) > F(H)$. From $F(L) > F(R)$, we have $F(L) + F(D) - F(I) > F(R) + F(D) - F(I)$, i.e. $F(L +_I D) > F(R +_I D)$, i.e. $F(G) > F(H)$. Hence, F is a termination criterion for the derived rule $p_d : G \stackrel{g}{\leftarrow} D \stackrel{h}{\rightarrow} H$.*

Termination criteria for specific instances of derived rules can be used as termination criteria for a rule, as shown by the following theorem.

Proposition 3 (termination of direct derivations - 2) *If F is a termination criterion for one derived rule p_d of $d : G \Rightarrow_p H$, then it is a termination criterion for rule p.*

Proof 2 *Let p be the rule $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$. If F is a termination criterion for the derived rule $p_d : G \stackrel{g}{\leftarrow} D \stackrel{h}{\rightarrow} H$ then it must hold that $F(L) + F(D) - F(I) = F(L +_I D) = F(G) > F(H) = F(R +_I D) = F(R) + F(D) - F(I)$. Hence $F(L) > F(R)$ and F is a termination criterion for p.*

Definition 12 (terminating expressions) *Given a replacement unit in CAT $RU = (P, nm, C)$ and a control expression E over $Names$, F is a termination criterion for E if it is a termination criterion for CAT and*

1. *if $E = nm(p) \in Names$, then $F(L(p)) > F(R(p))$;*
2. *if $E = E_1; E_2$, then, for each derived rule p_d of $d \in der(E)$, $F(L(p_d)) > F(R(p_d))$;*
3. *if $E = E_1 \mid E_2$, then, for each derived rule p_{d_i} of $d_i \in der(E_i)$, $i = 1, 2$, $F(L(p_{d_i})) > F(R(p_{d_i}))$;*
4. *if $E =$ **asLongAsPossible** E' **end**, then, for each derived rule p_d of $d \in der(E')$, $F(L(p_d)) > F(R(p_d))$.*

Proposition 4 (termination of sequentially composed derivations) *If F is a termination criterion for E_1 and E_2, then it is also a termination criterion for $E = E_1; E_2$.*

Proof 3 *If F is a termination criterion for the derived rule p_{d_1} of the derivation $d_1 : G \Rightarrow H \in der(E_1)_G$ and for the derived rule p_{d_2} of the derivation $d_2 : H \Rightarrow K \in der(E_2)_H$, then $F(G) > F(H)$ and $F(H) > F(K)$. Hence F is also a termination criterion for the derived rules p_d of all derivations $d \in der(E)_G$.*

Note that the converse is not true, as there may be termination criteria for the composition that are not termination criteria for one of the components. Hence, the existence of different termination criteria for the two components, does not guarantee the existence of a single termination criterion for their composition. As an example, consider the following two rules where CAT is the category of multisets: $p_1 : a \stackrel{l_1}{\leftarrow} a \stackrel{r_1}{\rightarrow} a$ and $p_2 : a \stackrel{l_2}{\leftarrow} \emptyset \stackrel{r_2}{\rightarrow} b$. Here no termination criterion exists for p_1, but any termination criterion for p_2 is also a termination criterion for $E = p_1; p_2$.

Proposition 5 (termination of choices) *If F is a termination criterion for E_1 and E_2, then it is also a termination criterion for $E = E_1 \mid E_2$.*

Proof 4 *If F is a termination criterion for the derived rule p_{d_1} of the derivation $d_1 : G \Rightarrow H_1 \in der(E_1)_G$ and for the derived rule p_{d_2} of the derivation $d_2 : G \Rightarrow H_2 \in der(E_2)_H$, then $F(G) > F(H_1)$ and $F(G) > F(H_2)$. Hence F is also a termination criterion for the derived rules p_{d_i} of all derivations $d_i \in der(E_i)_G$, for $i = 1, 2$.*

Proposition 6 (termination of as long as possible loops) *If F is a termination criterion for E', it is also a termination criterion for $E =$* **asLongAsPossible** *E'* **end**.

Proof 5 *If F is a termination criterion for each derived rule $p_{d'}$ of derivations $d' : G \Rightarrow H \in der(E')_G$, then $F(L(p_{d'})) > F(R(p_{d'}))$. Applying E' as long as possible, we get a derivation sequence $s : \mathcal{N} \rightarrow der(E')$ with $F(start(s(i))) > F(start(s(i+1)))$ for $i \geq 0$. Since $\mathcal{N}$ has no infinite descending sequence, there must exist $m \in \mathcal{N}$ such that for $j > m$, $F(start(s(j))) = F(start(s(j+1)))$ $= F(end(s(j)))$ so that $s(j)$ is the identity derivation, i.e. all derivations in $der(E)$ terminate.*

Theorem 1 (terminating derivations) *All derivation sequences over rules in $P \in \mathcal{P}$ are terminating if there is a termination criterion F which holds for all $p \in P$.*

Proof 6 *Given any derivation sequence $s : \mathcal{N} \rightarrow Der(P)$, due to Proposition 2, we know that $F(start(s(i))) > F(start(s(i+1)))$ for $i \geq 0$. Thus, there must be an $m \in \mathcal{N}$ such that for $j > m$, $s(j)$ is the identity derivation, i.e. all derivations in $der(E)$ are terminating.*

The result of Theorem 1 is adapted to replacement units where control conditions are used, in the following corollary.

Corollary 1 (termination of replacement units I) *Given a replacement unit $RU = (P, nm, C)$, all derivations in $der(C)$ terminate, if there is a termination criterion F which holds for all $p \in P$.*

The following theorem shows that the termination criterion need not be unique over a whole control expression.

Theorem 2 (Termination of replacement units II) *A replacement unit* $RU = (P, nm, C)$ *terminates if for each subexpression of* C *of the form* **asLongAsPossible** C' **end** *there is a termination criterion* F*.*

Proof 7 *The proof is by induction on the structure of the expression* C*:*

1. *Base step:*
   C *is a rule name. In this case, since each single rule application terminates,* RU *is terminating.*

2. *inductive step:*

   - $C = C_1; C_2$
     By induction hypothesis, both C_1 *and* C_2 *define sets of only finite derivation sequences* $der(C_1)$ *and* $der(C_2)$*; hence also* $der(C_1; C_2)$ *contains only finite derivations.*
   - $C = C_1 \mid C_2$
     By induction hypothesis, both C_1 *and* C_2 *define sets of only finite derivation sequences* $der(C_1)$ *and* $der(C_2)$*; hence also* $der(C_1 \mid C_2)$ *contains only finite derivations, as it contains the union of two finite derivation sequences.*
   - $C =$ **asLongAsPossible** C' **end**
     By induction hypothesis, C' *has a termination criterion* F *which, by Proposition 6, is also a termination criterion for* C*. Hence* RU *terminates.*

Hence, Theorem 2 states that a replacement unit is terminating if, for each **asLongAsPossible**-subexpression of C, there is a suitable termination criterion. The important aspect of this is that these criteria may differ from subexpression to subexpression.

6 Special termination criteria for sequential compositions

It is interesting to observe that some tricky situations may occur. Consider for example the following two rules for a multiset rewriting system, a case of high-level replacement system where CAT is the category of multisets: $p_1 : a \stackrel{l_1}{\leftarrow} \emptyset \stackrel{r_1}{\rightarrow} b$ and $p_2 : b \stackrel{l_2}{\leftarrow} \emptyset \stackrel{r_2}{\rightarrow} a$. If we now consider the replacement unit **asLongAsPossible** (**asLongAsPossible** p_1 **end**; **asLongAsPossible** p_2 **end**) **end**, we can easily see that this is not terminating, even though the two individual loops are. This is due to the fact that there is no possible termination criterion for the subexpression **asLongAsPossible** p_1 **end**; **asLongAsPossible** p_2 **end**, as every rule derived from it reduces either to the identity rule or to a rule which creates a finite number of additional a elements. Indeed, each instance of a will be rewritten to an instance of b and then each instance of b will turn into an a

again. If the initial object G contained some bs, these will also be turned into as after the first iteration over p_2.

On the other hand, we want to investigate under which conditions an arbitrary sequence of rules can satisfy a termination criterion. To this end, we want to explore criteria for a sequential expression of the type $p_1; p_2; \ldots; p_k$.

First we review the concept of sequentially dependent derivation sequences as defined in [12] and reformulated in [13] in the context of adhesive HLR-categories [19]. In the following, we assume that our HLR-systems are adhesive[2]. The dependency of two productions $p_1 : L_1 \stackrel{l_1}{\leftarrow} I_1 \stackrel{r_1}{\rightarrow} R_1$ and $p_2 : L_2 \stackrel{l_2}{\leftarrow} I_2 \stackrel{r_2}{\rightarrow} R_2$ is given by an object K and morphisms $K \rightarrow R_1$ and $K \rightarrow L_2$. The intention of a concurrent production is to combine the effects of p_1 and p_2 in a single production.

Definition 13 (K-concurrent production [12]) *Let $p_1 : L_1 \stackrel{l_1}{\leftarrow} I_1 \stackrel{r_1}{\rightarrow} R_1$ and $p_2 : L_2 \stackrel{l_2}{\leftarrow} I_2 \stackrel{r_2}{\rightarrow} R_2$ be productions and K be an object together with morphisms $K \rightarrow R_1$ and $K \rightarrow L_2$.*

- *The pair $(k_1 : K \rightarrow R_1,\ k_2 : K \rightarrow L_2)$, or short (k_1, k_2), is called a dependency relation for (p_1, p_2) if the pushout object H^* of $K \rightarrow R_1$ and $K \rightarrow L_2$ exists and if there are unique (up to isomorphism) pushout complements of $I_1 \rightarrow R_1 \rightarrow H^*$ and $I_2 \rightarrow L_2 \rightarrow H^*$.*
- *Given a dependency relation (k_1, k_2) the K-concurrent production $p_1 *_{(k_1,k_2)} p_2$ of p_1 and p_2 is given by the following construction (see Fig. 10):*
  1. *Let H^* be the pushout object in diagram (1).*
  2. *Let I_1^* and I_2^* be the pushout complements in diagrams (2) and (2')respectively.*
  3. *Let L^* and R^* be the pushout object in diagram (3) and (3') respectively.*
  4. *Let I^* be the pullback object in diagram (4) with $I^* \rightarrow L^* = I^* \rightarrow I_1^* \rightarrow L^*$ and $I^* \rightarrow R^* = I^* \rightarrow I_2^* \rightarrow R^*$.*

Next we investigate some special cases of dependency relations.

Proposition 7 (Parallelism) *Let $\emptyset$ be the initial object in CAT. For $K = \emptyset$ we have $p_1 *_{(\emptyset,\emptyset)} p_2 \cong p_1 + p_2$, i.e. the $(\emptyset, \emptyset)$-concurrent production is isomorphic to the parallel production (see Figure 11). The derivation sequence $G \Rightarrow_{p_1,m_1} H \Rightarrow_{p_2,m_2} X$ is sequentially independent if and only if it is K-dependent for $K = \emptyset$, i.e. if the application of the first rule does not create matches for the second rule.*

Proof 8 *See [12].*

[2]This assumption is not very restrictive, as most of the HLR conditions in [11] are valid in adhesive categories.

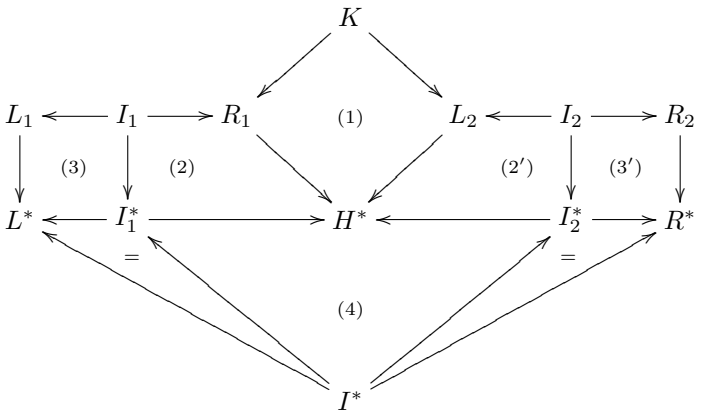

Figure 10: K-concurrent production

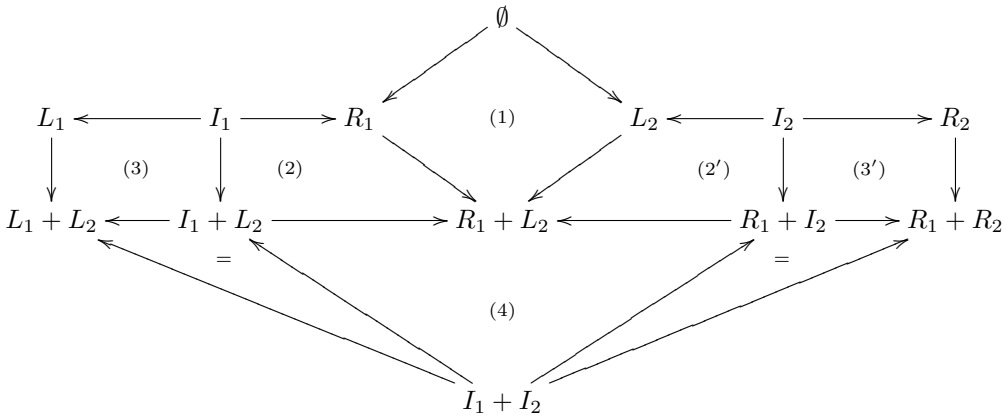

Figure 11: Parallelism as special case of K-concurrency

Proposition 8 (termination of sequentially composed derivations) *Let F be a termination criterion for a category (CAT, M), P a set of rules with objects and morphisms in (CAT, M), and $C = p_1; p_2$ a control expression, with $p_1, p_2 \in P$. Then C terminates if $F(L(p_1)) + F(L(p_2)) > F(R(p_1)) + F(R(p_2))$.*

Proof 9 *Let $C = p_1; p_2$ with $p_1 : L_1 \stackrel{l_1}{\leftarrow} I_1 \stackrel{r_1}{\rightarrow} R_1$ and $p_2 : L_2 \stackrel{l_2}{\leftarrow} I_2 \stackrel{r_2}{\rightarrow} R_2$. Furthermore let $(k_1 : K \rightarrow R_1,\ k_2 : K \rightarrow L_2)$ a dependency relation for (p_1, p_2) and $p = p_1 *_{(k_1,k_2)} p_2$ the (k_1, k_2)-concurrent production (see Def. 13). Then we need to show*

$$F(L(p)) > F(R(p)) \Leftrightarrow F(L(p_1)) + F(L(p_2)) > F(R(p_1)) + F(R(p_2))$$

which is performed in the following steps:

$$
\begin{array}{ll}
 & F(L(p)) > F(R(p)) \\
\Leftrightarrow & F(L^*) > F(R^*) \\
\Leftrightarrow & F(L_1 +_{I_1} I_1^*) > F(R_2 +_{I_2} I_2^*) \\
\Leftrightarrow & F(L_1) + F(I_1^*) - F(I_1) > F(R_2) + F(I_2^*) - F(I_2) \\
\Leftrightarrow & F(L_1) + F(I_1^*) - F(I_1) + F(L_2) + F(R_1) > \\
 & F(R_2) + F(I_2^*) - F(I_2) + F(L_2) + F(R_1) \\
\Leftrightarrow & F(L_1) + F(R_1 +_{I_1} I_1^*) + F(L_2) > F(R_2) + F(L_2 +_{I_2} I_2^*) + F(R_1) \\
\Leftrightarrow & F(L_1) + F(H^*) + F(L_2) > F(R_2) + F(H^*) + F(R_1) \\
\Leftrightarrow & F(L_1) + F(L_2) > F(R_2) + F(R_1) \\
\Leftrightarrow & F(L(p_1)) + F(L(p_2)) > F(R(p_1)) + F(R(p_2))
\end{array}
$$

The proof mainly makes use of the construction of the (k_1, k_2)-concurrent production $p_1 *_{(k_1,k_2)} p_2$ in Def. 13 and the termination criterion in Def. 11.

The associativity of the ; operator suggests the possibility of expanding the construction to obtain a single rule from a sequence of rules. The following Proposition specifies a termination criterion for such a sequence.

Proposition 9 (Sequential composition) *Let F be a termination criterion for a category (CAT, M), P a set of rules with objects and morphisms in (CAT, M), and $C = p_1; p_2; \ldots; p_n$ a control expression, with $p_i \in P$ for $i = 1, .., n$. Then C terminates if $\sum_{i=1}^{n} F(L(p_i)) > \sum_{i=1}^{n} F(R(p_i))$.*

Proof 10 *The proof proceeds by induction on the length n of the sequence and exploits the associativity of the sequential composition. We only need to consider the case for $n \geq 3$, since case $n = 1$ reduces to $C = p_1$ which terminates for $F(L(p_1)) > F(R(p_1))$, and case $n = 2$ has been proved in Proposition 8. Let us assume that the theorem holds for each sequence of length n and prove that it holds for a sequence of length $n+1$ for which $\sum_{i=1}^{n+1} F(L(p_i)) > \sum_{i=1}^{n+1} F(R(p_i))$. Due to the (left) associativity of operator ;, we have $p_1; p_2; \ldots; p_{n+1} = (p_1; p_2); \ldots; p_{n+1}$ and we can apply the pushout construction in Def. 13, to rules p_1 and p_2 to obtain a rule $p^* = p_1 *_{(k_1,k_2)} p_2$ so that $C = p^*; \ldots; p_{n+1}$.*

The construction in Figure 10 produces a rule $p^* : L^* \stackrel{l}{\leftarrow} I^* \stackrel{r}{\rightarrow} R^*$. From the four equalities given by the pushout constructions (2), (2′), (3) and (3′):

1. $F(L(p^*)) = F(L(p_1)) + F(I_1^*) - F(I_1)$
2. $F(H^*) = F(R(p_1)) + F(I_1^*) - F(I_1)$
3. $F(H^*)) = F(L(p_2)) + F(I_2^*) - F(I_2)$
4. $F(R(p^*)) = F(R(p_2)) + F(I_2^*) - F(I_2)$

one obtains $F(L(p^*)) = F(L(p_1)) + F(H^*) - F(R(p_1))$ and $F(R(p^*)) = F(R(p_2)) + F(H^*) - F(L(p_2))$. Therefore, $F(L(p^*)) + \sum_{i=3}^{n+1} F(L(p_i)) = F(L(p_1)) + F(H^*) - F(R(p_1)) + \sum_{i=3}^{n+1} F(L(p_i)) > F(R(p_2)) + F(H^*) - F(L(p_2)) + \sum_{i=3}^{n+1} F(R(p_i)) = F(R(p^*)) + \sum_{i=3}^{n+1} F(R(p_i))$.

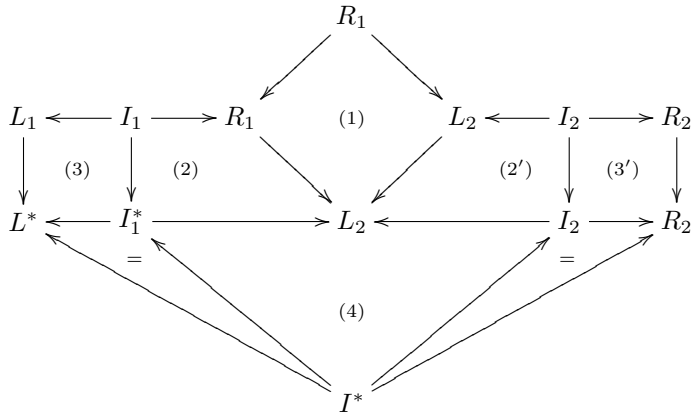

Figure 12: $K = R_1$ as special case of K-concurrency

Hence, we have a sequence of length n satisfying the criterion, and from the induction hypothesis the whole sequence satisfies the criterion.

In Proof 10 above, one sees that the cases depicted in Figures 11, 12, and 13 are all special cases of the most general situation treated in the proof. In particular one has:

- for $L_2 \cap R_1 = \emptyset$ (see Figure 11), $F(L(p^*)) = F(L(p_1)) + F(L(p_2))$ and $F(R(p^*)) = F(R(p_1)) + F(R(p_2))$;
- for $R_1 \subset L_2$ (see Figure 12), $F(L(p^*)) = F(L(p_1)) + F(L(p_2)) - F(R(p_1))$, while $F(R(p^*)) = F(L(p_2))$;
- for $L_2 \subset R_1$ (see Figure 13), $F(L(p^*)) = F(L(p_1))$ and $F(R(p*)) = F(R(p_2)) + F(R(p_1)) - F(L(p_2))$.

where we use the symbol $\subset$ to indicate the existence of an injective, non surjective mapping of G_1 into G_2 and the symbol $\cap$ to indicate the pullback for G_1 and G_2.

7 Concrete termination criteria for Attributed Graph Transformation

We show now how some functions which naturally arise from counting elements in a graph can be used to establish criteria for termination.

Definition 14 (Concrete termination criteria) *Let $n : \mathcal{G} \to \mathcal{N}$ be a function returning the number of nodes in G, i.e. $n(G) = |G_N|$, and $e : \mathcal{G} \to \mathcal{N}$ a function computing the number of edges in G, i.e. $e(G) = |G_E|$, for each graph G in the category* **ASSIG-Alg**. *If s is a sort in S_G, the function $t_s : \mathcal{G} \to \mathcal{N}$ yields, for each graph A in* **ASSIG-Alg**, *the number of elements in $A_s^{S_G}$.*

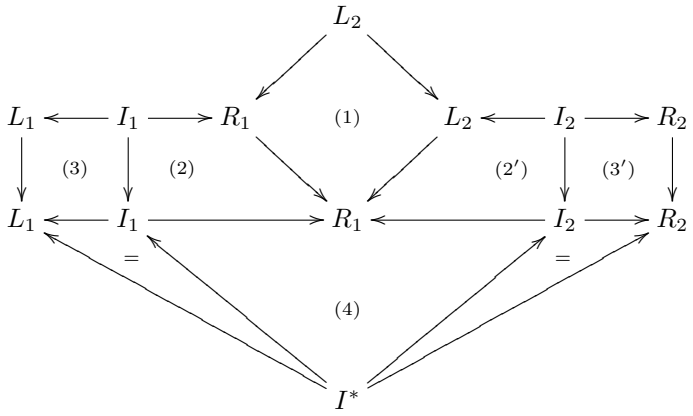

Figure 13: $K = L_2$ as special case of K-concurrency

We show that n, e, and t_s can be used as termination criteria within the category **ASSIG-Alg**.

Proposition 10 *The functions n, e, and t_s, for each $s \in S_G$, are termination criteria for category* **ASSIG-Alg** *as defined in Definition 11.*

Proof 11 *We can prove that the functions n, e, and t_s satisfy the termination criterion for the pushout construction on two morphisms $a : C \to A$ and $b : C \to B$ where $a \in M$ and b is arbitrary, so that* a fortiori *it holds when $b \in M$, which is what is required by Definition 11. Since the graph part of a is injective, the pushout construction glues graphs A and B only at elements of the graph C, by taking, for nodes and edges separately, the disjoint union of B and the part of A not in the image of C under a i.e. $D = B \uplus (A - a(C))$. Thus $n(D) = n(A +_C B) = n(A) + n(B) - n(C)$, $e(D) = e(A +_C B) = e(A) + e(B) - e(C)$, and $t_s(D) = t_s(A +_C B) = t_s(A) + t_s(B) - t_s(C)$ for each $s \in S_{GS}$.*

Now, to show that the replacement unit of Section 4 is terminating, we only have to check the termination of *completeRefactoring* for any possible choice of *class* and *attr*, as this is the only rule to be iterated. At each application of this rule, a node of type `Variable` is removed (together with the edges connecting it to nodes of type `Class` and `Type`). Hence, both functions n and e above can be used as termination criteria to prove termination of this sub-unit.

8 Discussion of further termination criteria

In this Section we point at some directions for the extension of the presented approach to special cases. In particular, we deal with Contextual Layered Graph Grammars and rules with negative application conditions (NACs).

8.1 Contextual Layered Graph Grammars

Contextual layered graph grammars (CLGGs) have been used in parsing, as they provide a natural way to steer the parsing process, thereby reducing its non-determinism and its complexity. A contextual layered graph grammar is a construct $CLGG = (S, T, P, cl, dl, rl)$, where S is a labelled graph, called the *initial* graph, T is a set of node and edge *types* of labels and P is a set of rules. The *layering functions* cl, dl, and rl assign a *creation* and a *deletion* layer to elements of T and a unique layer to each rule $p \in P$, respectively.

In [6], the following concrete termination criterion for CLGGs was discussed.

Definition 15 (Termination criteria for CLGG) *A contextual layered graph grammar* $CLGG = (S, T, P, cl, dl, rl)$ *terminates if for all* $p \in P$*: if* $rl(p) = k$*, then*

- p *deletes at least one node or edge.*
- p *deletes only nodes and edges of a type* t *with* $dl(t) \leq k$*, and*
- p *creates only nodes and edges of a type* t *with* $cl(t) > k$*.*

The layering condition above guarantees the termination of the process, by producing a parsing derivation, or proving that the sentence cannot be parsed. The existence of the layering function rl allows the partitioning of the set P into a collection of sets $\{P_1, \ldots, P_k\}$. Rules in a set P_i can be used only after rules from the set P_{i-1} have been used and are no longer applicable. Moreover, after using a rule from P_j, $j \geq i$, no rule from P_{i-1} can be applied any longer.

The effect of a contextual layered graph grammar can be achieved via a control expression. In particular, given a CLGG such that $P = \{P_1, \ldots, P_k\}$, with each layer P_i containing the set of rules $\{p_{i,1}, \ldots, p_{i,n_i}\}$, this terminates if and only if the control expression $C =$ **asLongAsPossible** $(p_{1,1} \mid \ldots \mid p_{1,n_1})$ **end**; $\ldots$; **asLongAsPossible** $(p_{k,1} \mid \ldots \mid p_{k,n_k})$ **end** terminates. To prove termination of the control expression, we need to accommodate labelled rewriting rules in the framework of high-level replacement systems, so that a suitable function F can be defined which takes into account the layer to which a label begins.

This should provide a generalisation to the approach in [9], which ensures termination of a CLGG by separately considering layers with and without deletion and defining concrete termination criteria for the two cases.

8.2 Negative Applicative Conditions

NACs for graph transformation have been introduced in [16] and have proven useful when applying graph transformation to practical problems. Rules with NACs can be formalized as shown in Figure 14. Here, the morphism n is an injective total morphism, so that the rule is applicable only if match m cannot be extended to a match m' such that $n \circ m' = m$. Several objects N_i, and the associated morphisms n_i, can be associated with one L, indicating that no extension of m should exist for any i.

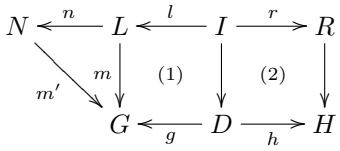

Figure 14: A DPO rule with negative application condition

While there is no particular restriction on the form of NACs, we claim that the criterion illustrated in this paper can be extended to prove termination of rules with NACs characterized by having $I = L$, and $N = R$. As in this case no element of L is deleted and the same rule could be applied again, the NAC checks that the effects for that match have already been produced, thus forbidding the repeated application of the same rule on the same match again. By exploiting the recent incorporation of NACs into the high-level replacement framework [10], we can extend the termination criterion for rules with NACs of this type so that $F(L) > F(R) - F(N)$, in order to consider the reduction in the number of possible matches subsequent to the application of a rule. This can also be the basis for a more general treatment of replacement units involving rules with NACs.

9 Conclusions

Termination is an important issue for model transformations. Specifying them by graph transformation in the double pushout approach has the advantage that they are precisely defined and can be formally analyzed.

In this paper, we discuss the termination of transformations and propose a general termination criterion for high-level replacement systems, a generalization of graph transformation systems. Since model transformations can become complex, we consider not only the application of single rules, but also replacement units where rule applications are restricted according to an additional control flow. For the description of the control flow we allow application of single rules, sequential composition of rule expressions, choice between two rules, and iterations applying an expression as long as possible. This paper contains a number of results concerning termination of replacement units.

We plan to extend the presented results in several ways, such as studying additional operators for control expressions, e.g. if-then-else expressions, priorities, etc. Moreover, we plan to study wider criteria to establish termination of sequential compositions of rules, also taking into account different orderings, as studied in [7], and to extend the types of NACs for which we can prove termination.

Finally, as more and more types of rewriting systems get incorporated into the high-level replacement framework, concrete termination criteria must be identified, which might also lead to a more general definition of the requests for a termination criterion.

Acknowledgments We thank the anonymous referees for several useful observations on a previous version of this paper.

References

[1] U. Assmann. Graph rewrite systems for program optimization. *ACM TOPLAS*, 22(4):583–637, 2000.

[2] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 105–181. World Scientific, 1999.

[3] P. Bottoni, M. Koch, F. Parisi Presicce, and G. Taentzer. Automatic consistency checking and visualization of OCL constraints. In *UML 2000 - The Unified Modeling Language*, pages 294–308. Springer, 2000.

[4] P. Bottoni, F. Parisi-Presicce, and G.Taentzer. Specifying Integrated Refactoring with Distributed Graph Transformation. In *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 220–235. Springer, 2004.

[5] P. Bottoni, P. Parisi-Presicce, and G. Taentzer. Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations. In P. v. Bommel, editor, *Transformation of Knowledge, Information, and Data: Theory and Applications*, pages 95–125. Idea Group Publishing, 2004.

[6] P. Bottoni, G. Taentzer, and A. Schürr. Efficient parsing of visual languages based on critical pair analysis (and contextual layered graph transformation). In *IEEE Symposium Visual Languages*, pages 59–61. IEEE Press, 2000.

[7] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1& 2):69–115, 1987. Corrigendum: 4,3 (Dec. 1987), 409-410.

[8] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[9] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In *Proc. Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 49–63. Springer Verlag, 2005.

[10] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, volume 3256 of *LNCS*, pages 287–303. Springer, 2004.

[11] H. Ehrig, M. Gajewsky, and F. Parisi-Presicce. High-Level Replacement Systems applied to Algebraic Specifications and Petri Nets. In *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 3: Concurrency, Parallelism and Distribution*, pages 341–400. World Scientific, 2000.

[12] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.

[13] H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement categories and systems. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, volume 3256 of *LNCS 3256*, pages 144–160. Springer, 2004.

[14] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004.

[15] M. Fowler, K. Beck, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[16] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[17] H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11:690–723, 1999.

[18] J. M. Küster, R. Heckel, and G. Engels. Defining and Validating Transformations of UML Models. In *Proc. HCC 2003*, pages 145–152. IEEE Computer Society, 2003.

[19] S. Lack and P. Sobocinski. Adhesive categories. In I. Walukiewicz, editor, *FOSSACS 2004*, volume 2987 of *LNCS*, pages 273–288. Springer, 2004.

[20] T. Mens, S. Demeyer, and D. Janssens. Formalising Behaviour Preserving Program Transformations. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of ICGT'02*, volume 2505 of *LNCS*, pages 286–301. Springer, 2002.

[21] D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[22] A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 487–550. World Scientific, 1999.

[23] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML models. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference*, pages 134–148. Springer, 2001.