Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Electronic Notes in Theoretical Computer Science 127 (2005) 71–86
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Termination of High-Level Replacement Units with Application to Model Transformation*

Paolo Bottoni[a,1] Manuel Koch[b,2] Francesco Parisi-Presicce[a,d,3]
Gabriele Taentzer[c,4]

[a] *Università di Roma "La Sapienza" - Italy*
[b] *Freie Universität Berlin - Germany*
[c] *Technische Universität Berlin - Germany*
[d] *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 basic results concerning the formal properties of these models are still missing for many of them. In this paper, we propose 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. These are used to show the termination of some replacement units needed to express model transformations as a consequence of software refactoring.

Keywords: Transformation units, graph transformation, termination, refactoring.

* Research partially supported by the European Community under RTN Segravis
[1] Email: bottoni@di.uniroma1.it
[2] Email: mkoch@inf.fu-berlin.de
[3] Email: fparisi@ise.gmu.edu
[4] Email: gabi@cs.tu-berlin.de

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2004.08.048

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 modelling 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 [17,15,3,14] 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 may involve the definition of some sequence of rule applications, as well as the prevention of repeated application of a 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 [16], but which can be studied for individual rewriting systems, following the classical approach 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, as introduced by Dershowitz and Manna in [6].

This problem is further complicated by the need for rule expressivity. Indeed, there is always a trade-off between the inherent expressivity (and complexity) of the rewriting relation for a single step and the availability of external control mechanisms steering the rewriting process. Although a number of rewriting models have been proposed, differing in the expressivity of the types of rules and in the complexity of the rewriting mechanism, basic results concerning the formal properties of these models are still missing for many of them. The combination of attributed rules and transformation units employing rule expressions seems to provide a transformation approach which can already be used practically, but which is simple enough for formal reasoning.

In this paper we first study a more abstract version of attributed rules, namely high-level replacement systems [9], to which we extend the notion of transformation unit [13]. We thus obtain an abstract property that a function has to satisfy in order to be used as a termination criterion for such units. In particular, Section 2 introduces related work, while Section 3 adapts the concept of transformation units to define high-level replacement units. In Section 4, a motivating example from software transformation, namely refactoring, is

presented. Section 5 discusses termination criteria for high-level replacement systems and shows some concrete termination results. These are illustrated by presenting some sample transformations in which replacement units are used to specify transformations of UML models consequent to software refactoring. Finally, conclusions are given in Section 6.

2 Related work

Transformation units have been introduced by Kreowski and Kuske in [13] 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 [14]. In particular, they have studied the problem of termination and confluence. Recognising, as demonstrated by Plump in [16], 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 will 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 [5], layering conditions were applied to ensure termination of parsing processes. We can observe how 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. It is to be considered, however, that, rather than in parsing processes, we are interested here in general types of transformations. Such a situation has also been considered in [2], where transformation units were employed to define a semantics for OCL.

3 High-Level Replacement Units

High-Level Replacement Systems [9,7] are an instantiation of the general graph transformation approach, at the basis of the definition of transformation units [13]. 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 a category with one distinguished class M of morphisms.

Definition 3.1 [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 3.2 [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 $C_1, C_2 \in \mathcal{C}$, and
- **asLongAsPossible** C **end** $\in \mathcal{C}$, if $C \in \mathcal{C}$.

The intended meaning of the operator **asLongAsPossible** C is the (sequential) application of the expression C as long as its application is possible.

Definition 3.3 [High-level replacement unit] A high-level replacement unit $RU = (P, name, C)$ in a category CAT, or just *replacement unit*, consists of a finite set P of rules, a bijective function $name : P \rightarrow Names$, and a control expression $C \in \mathcal{C}$ over $Names$.

High-level replacement units are units in the sense of [13]: we have a graph transformation approach consisting of the class of objects in CAT, the class of rules in this category, the application operator as defined in Definition 3.4, a class of control expressions as defined in Definition 3.2 and a class of graph class expressions being the class of objects in CAT itself. 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, the semantics of high-level replacement units is defined by derivation sets.

Definition 3.4 [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 $m : L \rightarrow G$. 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). *start* and *end* are 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.

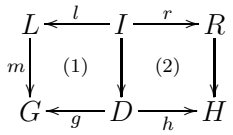

Fig. 1. The double-pushout approach

Definition 3.5 [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 $\forall i > n$, 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 $0 < i$.

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

Definition 3.6 [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 $d_1 \circ d_3 = d_2$. One sees that $\preceq$ is a partial order.

Definition 3.7 [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 empty set. The *star product* der^* is defined by $\bigcup_{i \geq 0} der^i$.

Definition 3.8 [derived rule] Given a direct derivation $d : G \Rightarrow_{r,m} H$ by a double pushout as in Definition 3.4, the *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 $d = d_1 \circ d_2$ 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.

Definition 3.9 [semantics of control expression] Given an object G and a replacement unit $RU = (P, name, C)$, 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.

(i) $C = name(p)$: $der(C)_G = \{d \in Der(\{p\}) | start(d) = G\}$,

(ii) $C = C_1; C_2$: $der(C)_G = \bigcup \{der(C_1)_G \circ der(C_2)_H \mid H = end(d_1), d_1 \in der(C_1)_G\}$,

(iii) $C =$**asLongAsPossible** C' **end**: $der(C)_G = \{d \in der(C')^*_G \mid d$ is maximal in $der(C')^*_G$ wrt. $\preceq\}$

Coeherently with this semantics, we consider that a replacement unit of type $name(p)$ fails if p is not applicable, $C_1; C_2$ fails if either C_1 or C_2 fails, **asLongAsPossible** C' **end** fails if C' fails and it is not of the form $name(p); C''$ or if C' is of this form and C'' fails. 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 present attributed graph structures as defined in [10]. For the category of attributed graph structures and homomorphisms with a distinguished class M of morphisms, the Church-Rosser, Parallelism and Concurrency Theorem have been shown in [10].

Definition 3.10 [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 $OP_D \cap OP_G = \varnothing$. $ASSIG$ 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 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_{DSIG}-indexed set of variables $X = (X_s)_{s \in S_{DSIG}}$, all rule objects are attributed by the term algebra $T_{DSIG}(X)$.

Proposition 3.11 (local C-R, parallelism and concurrency) *The local Church-Rosser theorems I and II, the parallelism theorem and the concurrency theorem as stated in [9] are valid for each graph transformation system based on* **ASSIG-Alg**.

Proof. See [10]. □

4 Examples from UML refactoring

In this section we present an example of refactoring from [11], and model the transformations that the model of a system software must go through to be maintained consistent with the modifications in the code. Elsewhere ([4]) 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 [4], 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 possibly reduces 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 renamings 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 behaviour 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, and to avoid using negative application conditions to check that no subclass has been neglected, we propose an adaptation of the metamodel for class diagrams, to the effect that each class node n is associated with a node of type `ClassDescriptor`, possessing an attribute which keeps a constantly updated list of the direct subclasses of n.

To this end, we modify the rule for class creation, which does not have preconditions, as shown in Figure 2. 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 concrete instances, are indicated by showing their names in *italic*, while variables present in the rule, and subject to a unification process, are written with a capitalized initial. In this version of the rules, which do not present negative application conditions, we do not check for the absence of inheritance cycles or of multiple inheritance.

The elimination of a class is accompanied by the destruction of its associate `ClassDescriptor` node, as shown in Figure 5. As we are using the DPO approach, class elimination is possible only when it has been stripped of all owned features and relations with other classes.

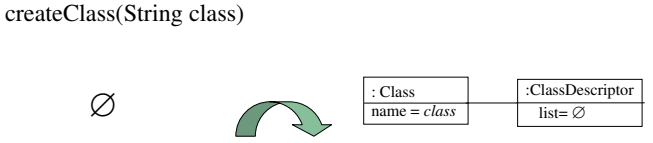

Fig. 2. The rule to create a class, associated with the node recording all its subclasses.

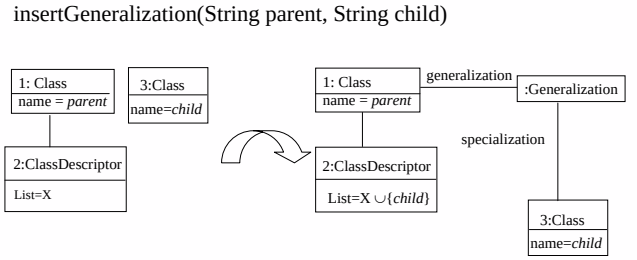

Fig. 3. The rule to insert a generalization relation between two classes.

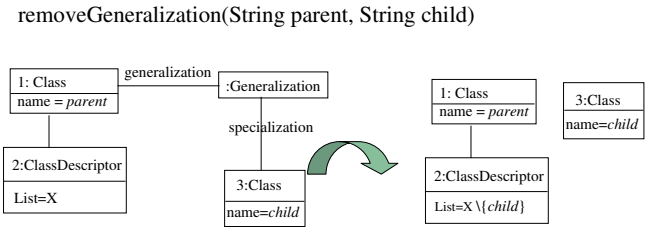

Fig. 4. The rule to remove a generalization relation.

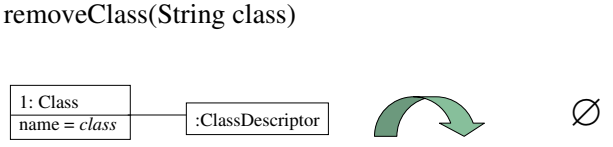

Fig. 5. The rule to remove a class.

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 `list` attribute will contain all the names of the class from which the pulled up

variable have been removed (see Figure 6).

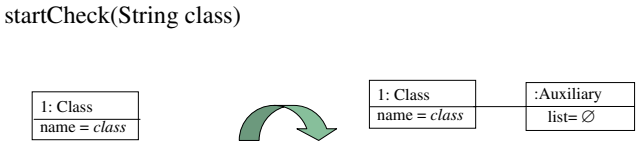

Fig. 6. The rule to attach an `Auxiliary` node to the class that the variable must be pulled up to.

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 `list` attribute of the `Auxiliary` node. Note that at least one such subclass must exist, otherwise the whole replacement unit will fail.

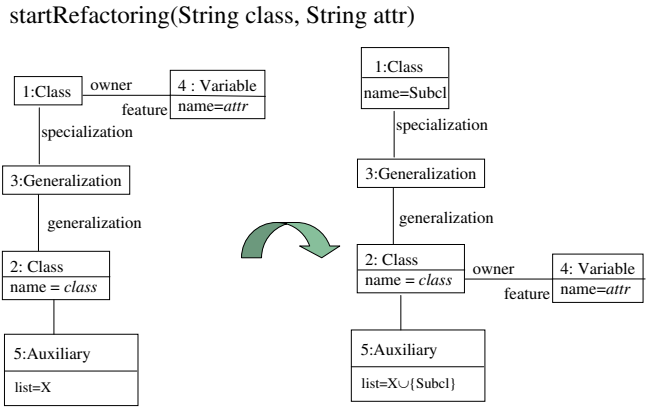

Fig. 7. The rule to pull the variable `attr` from a non deterministically chosen subclass of `class`.

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 `list` 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 `list` attributes for the nodes of type `ClassDescriptor` and `Auxiliary`, associated with `class`. The failure of this rule indicates that some subclass did not possess the vari-

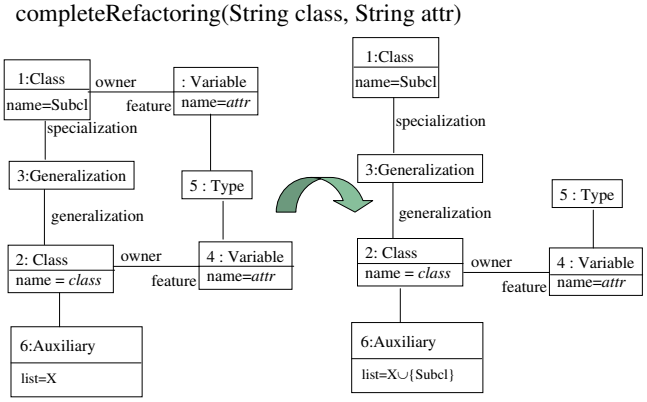

Fig. 8. The rule to pull the variable `attr` from any subclass possessing a copy of the variable to be pulled up.

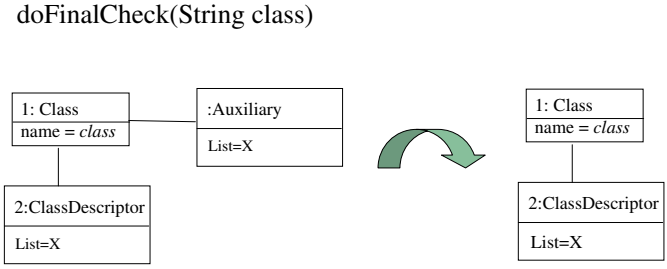

Fig. 9. The final rule to check success of the pull up transformation.

able, and makes the whole replacement unit fail. It is to be noted that if this occurs, due to the transactional behaviour 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) =
    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 set 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 5.1 [termination criterion] A function $F : \mathcal{G} \to \mathcal{N}$ from objects to natural numbers is a *termination criterion* for CAT 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 common elements to A and B, which would otherwise be considered twice. Even if in Definition 5.1 we have considered p to be a rule in the double pushout approach, 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. We develop here the theory only for the DPO approach, but will sketch out a consequence on the definition of concrete criterion for SPO.

Proposition 5.2 (direct derivation) *If F is a termination criterion for a rule p, then it is also a termination criterion for the derived rules p_d of all $d : G \Rightarrow_p H$.*

Proof. 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 $G \Rightarrow_r H$ given by a double pushout as in Figure 1 we need $F(G) > F(H)$. From $F(L) > F(R)$, we have $F(L) + F(C) - F(I) > F(R) + F(C) - F(I)$, i.e. $F(L +_I C) > F(R +_I C)$, i.e. $F(G) > F(H)$. Hence, F is a termination criterion for the derived rule $p_d : G \stackrel{g}{\leftarrow} C \stackrel{h}{\rightarrow} H$. □

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

Proposition 5.3 (termination of derived rules) *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 the rule p.*

Proof. 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 $p_d : G \stackrel{g}{\leftarrow} C \stackrel{h}{\rightarrow} H$ then it must hold that $F(L) + F(C) - F(I) = F(L +_I C) = F(G) > F(H) = F(R +_I C) = F(R) + F(C) - F(I)$. Hence $F(L) > F(R)$ and F is a termination criterion for p. □

Definition 5.4 [terminating expressions] Given a replacement unit in CAT $RU = (P, name, C)$ and an expression E over $Names$, F is a termination criterion for E if it is a termination criterion for CAT and

(i) if $E = name(p) \in Names$, then $F(L(p)) > F(R(p))$;

(ii) if $E = E_1; E_2$, then, for each derived rule p_d of $d \in der(E)$, $F(L(p_d)) > F(R(p_d))$;

(iii) 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 5.5 (sequential composition) *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. 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.

Proposition 5.6 (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. 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 be an $m \in \mathcal{N}$ 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 5.7 (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. Given any derivation sequence $s : \mathcal{N} \rightarrow Der(P)$, we know that $F(start(s(i))) > F(start(s(i+1)))$ for $i \geq 0$, due to proposition 5.2. 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 5.7 is adapted to replacement units where control conditions are used, in the following corollary.

Corollary 5.8 (termination of replacement units I) *Given a replacement unit $RU = (P, name, C)$, all derivations in $der(C)$ terminate, if there is a termination criterion F which holds for all $p \in P$.*

Proof. Direct consequence of Theorem 5.7. □

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

Theorem 5.9 (Termination of replacement units II) *A replacement unit $RU = (P, name, C)$ terminates if each* **asLongAsPossible***-subexpression C' of C has a termination criterion F.*

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

(i) Base step:
C is a rule name: in this case, since each single rule application terminates, then RU is terminating.

(ii) 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.
- $C =$ **asLongAsPossible** C' **end**
  By induction hypothesis, C' has a termination criterion F which, by proposition 5.6, is also a termination criterion for C. Hence RU terminates.

□

Hence, Theorem 5.9 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.

5.2 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 5.10 [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 $A_s^{S_G}$.

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

Proposition 5.11 *The functions n, e, and t_s, for each $s \in S_G$, satisfy the termination criterion in Definition 5.1.*

Proof. We can prove that the functions $n(G)$, $e(G)$, and t_s satisfy the 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 5.1. 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 $s \in S_{GS}$. □

A concrete criterion, other than simple counting of nodes and edges, can be obtained for any rule p in the SPO approach by considering the function $F_p(G)$ which counts the number of matches and partial matches for L, the left-hand side of p, in G. The class M is here considered that of partial morphisms m such that if $i \neq j$, then $m(i) \neq m(j)$. Hence, we consider the partial morphism $L \to R$, given by the rule p, and the match $L \to G$. The pushout construction produces the graph H as the result of the application of p. Now, if $F_p(L) > F_p(R)$, we have $F_p(H) = F_p(G) + F_p(R) - F_p(L)$, so that H has fewer total or partial matches for p than G. This means that at each application of p the number of possible future applications of it decreases, so that it can be applied only a finite amount of time, as the original graph G was finite.

5.3 Termination of Sample UML Refactorings

The replacement unit of Section 4 is terminating. Indeed, 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 looped on. 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 of Section 5.2 can be used as termination criteria to prove termination of this sub-unit.

6 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 are concerned with 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 do not only consider the application of single rules but 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, and loops 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. optional rule applications, if-then-else expressions, priorities, etc. By doing so, we could show that the termination of layered graph transformation to be used for graph parsing would be a special case of the results for our framework. Layered graph transformation systems can be considered as a special case of high-level replacement units where the control expressions are sequential compositions of as-long-as-possible loops applying a set of rules each. Moreover, we plan to study wider criteria to establish termination of sequential compositions of rules.

Furthermore, we would like to take negative application conditions (NACs) into account. NACs for graph transformation have been introduced in [12] and have proven useful when applying graph transformation to practical problems. Recently ([8]), they have been incorporated into the high-level replacement framework in. We would like to build up on this approach to formulate termination results for replacement units taking NACs into account.

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

References

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

[2] Bottoni, P., M. Koch, F. Parisi Presicce and G. Taentzer, *Automatic consistency checking and visualization of OCL constraints*, in: *UML 2000 - The Unified Modeling Language* (2000), pp. 294–308.

[3] Bottoni, P., F. Parisi-Presicce and G.Taentzer, *Specifying Integrated Refactoring with Distributed Graph Transformation*, in: *Applications of Graph Transformations with Industrial Relevance*, LNCS **3062** (2004), pp. 220–235.

[4] Bottoni, P., 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* (2004), to appear.

[5] Bottoni, P., 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* (2000), pp. 59–61.

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

[7] Ehrig, H., 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. Vol 3: Concurrency, Parallelism and Distribution*, World Scientific, Singapore, 2000 pp. 341–400.

[8] Ehrig, H. and A. Habel, *Constraints and Application Conditions: From Graphs to High-Level Structures*, in: *Proc. Int. Conf. on Graph Transformation 2004*, 2004, to appear.

[9] Ehrig, H., A. Habel, H.-J. Kreowski and F. Parisi-Presicce, *Parallelism and concurrency in High Level Replacement Systems*, Math. Struc. in Comp. Science **1** (1991), pp. 361–404.

[10] Ehrig, H., U. Prange and G. Taentzer, *Fundamental Theory of Typed Attributed Graph Transformation*, in: *Proc. Int. Conf. on Graph Transformation 2004*, 2004, to appear.

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

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

[13] Kreowski, H.-J., S. Kuske and A. Schürr, *Nested graph transformation units*, Int. Journal on Software and Knowledge Engineering **7** (1997), pp. 479–502.

[14] Küster, J. M., R. Heckel and G. Engels, *Defining and Validating Transformations of UML Models*, in: *Proc. HCC 2003* (2003), pp. 145–152.

[15] Mens, T., S. Demeyer and D. Janssens, *Formalising behaviour preserving program transformations*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *Proc. ICGT02*, 2002, pp. 286–301.

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

[17] Sunyé, G., D. Pollet, Y. L. 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* (2001), pp. 134–148.