*Math. Struct. in Comp. Science* (2014), vol. 24, iss. 4, e240407, 30 pages. © Cambridge University Press 2014
doi:10.1017/S0960129512000369

# Combining termination proofs in model transformation systems

DÉNES BISZTRAY[†] and REIKO HECKEL[‡]

[†]*Technology and Innovation, Nokia Solutions and Networks, Budapest, Hungary*
*Email:* denes.bisztray@nsn.com
[‡]*Department of Computer Science, University of Leicester, Leicester, United Kingdom*
*Email:* reiko@mcs.le.ac.uk

*Received February 2011; revised September 2011*

In model transformations, where source models are automatically translated into target models or code, termination is necessary for the transformation to be well defined. There are a number of specific termination criteria that can be used when specifying model transformations by graph transformation, though termination is undecidable in general. Unfortunately, and particularly for large and heterogeneous specifications, it is often not possible to use a single termination criterion. In this paper, we propose an approach that applies different criteria to suitable subsets of rules so that termination can be shown locally using the most suitable technique for each subset. Global termination then follows if certain causal dependencies between rules in different subsets are acyclic. The theory is developed at the level of typed attributed graphs, and is motivated and illustrated by a case study translating UML activity diagrams to CSP.

## 1. Introduction

Termination is a fundamental property of graph transformation systems, which are used to describe mappings between sets of graphs, as in the case of model transformations. In order to automate development activities such as code generation, refinement or verification, a transformation between source and target models (or code) is required. By representing models as graphs, typed graph transformation systems can be used to define and implement such transformations.

Given a type graph $TG$ representing the union of the metamodels of the source and target, possibly extended by types for traceability relations, the transformation is defined by non-deterministically applying a set of productions $P$ to an input graph typed over the source fragment $TG_S$ of the type graph until no further transformation is possible. The projection of the resulting graph to the target fragment $TG_S$ of $TG$ is the output of the transformation (Ehrig *et al.* 2006). For this approach to be well defined, all transformation sequences starting in any legal input graph $G_0$ must be finite, that is, the transformation system must be *terminating*.

[‡] The work of Reiko Heckel was partly supported by the FP6 IP Sensoria.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

While this property is, in general, undecidable (Plump 1998), it is possible to use proof techniques and specific termination criteria, such as those using termination orders (Baader and Nipkow 1998; Bottoni *et al.* 2005) based on the existence of a well-founded partial order $\succ$ over instance graphs. This ordering reflects the state of completion of the transformation process: so $G \succ H$ if $H$ is closer to termination. To define this order in practice, a metric $\mathcal{E}$ can be given that assigns a natural number to every graph. Then,

$$G \succ H \quad \text{if and only if} \quad \mathcal{E}(G) > \mathcal{E}(H).$$

Termination follows if the metric is monotonically decreasing with the application of productions, that is, if $G$ transforms to $H$ through the application of any production $p$ in the system, then $\mathcal{E}(G) > \mathcal{E}(H)$. Unfortunately, this technique is hard to apply in complex examples because of the lack of guidance in defining metrics satisfying this condition, and this frequently leads to a trial-and-error approach, rather than a systematic solution.

More recent research (Ehrig *et al.* 2005; Varró *et al.* 2006; Levendovszky *et al.* 2007) has focused on the translation of this general technique into more specific forms, which are incomplete and limited to certain classes of systems, but allow a more systematic approach or even automation; see Section 4 for a detailed discussion.

When transformations are complex and heterogeneous, it is less likely that any one of these criteria fits the entire specification, but a combination of them applied to various subsets of transformation productions may well lead to success. In the current paper, we provide conditions under which termination proofs for subsets of productions can be combined to prove termination of the global system. The idea is based on the observation that, especially in complex transformations, there are distinct functional groups of productions working on different parts of the host graph with relatively little interaction between them. While productions are applied non-deterministically, with their execution interleaved, causal dependencies are mainly with productions in the same group. Thus, the termination of one production set is independent of the termination of the others, provided the sets are suitably isolated and existing interdependencies are acyclic.

We will approach the problem in two steps:

(1) We first show the termination of transformations consisting purely of non-deleting productions with self-disabling negative conditions. To do this, we analyse a given unstructured set of productions for *produce–use* dependencies, that is, sequential dependencies between productions that could lead to the generation of a match for one production by the application of another through the creation of new elements. If this relation is acyclic, the system terminates because self-disabling productions can only be applied once at each essential match, while only finitely many matches can be created.

(2) The result is then extended to systems with general (possibly deleting) productions that preserve the input graph. This assumption is in line with the method of model transformation used, which is based on copying information from source to target graphs rather than modifying the source. The input graph is preserved when the productions delete only from the target model being constructed. Assuming that the effects of deletion can be isolated into self-contained production groups, we extend the dependency relation to these groups. The termination follows if the production

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

groups are terminating by themselves and the dependency relation between them is acyclic. The termination of individual production groups can be established by any technique or criteria suitable to the structure of that particular subset of productions.

### 1.1. *Organisation of the paper*

In Section 2, we provide the basic definitions of graph and model transformation. In Section 3, we present our motivating case study, and in Section 4 we discuss the features and shortcomings of existing approaches in relation to this and other examples. In Section 5, we establish the termination criterion for non-deleting graph transformation systems, and then extend it to deleting transformations in Section 6. In Section 7, we apply this approach to the case study. Finally, we present our conclusions in Section 8.

The current paper extends the conference paper Bisztray and Heckel (2010) by revising the theoretical contribution and giving a detailed discussion of the application of the approach to a substantial case study.

## 2. Graph and model transformation

In this section, we recall the fundamental concepts we will require for our development, which is based on the double-pushout approach to typed attributed graph transformation with negative application conditions (Ehrig *et al.* 2006). A graph is a tuple

$$G = (V, E, src, tar),$$

where $V$ is a set of nodes (or vertices), $E$ is a set of edges and $src, tar : E \to V$ associate a source and target node, respectively, with each edge in $E$. Given graphs $G_1$ and $G_2$, a graph morphism is a pair $(f_V, f_E)$ of total functions

$$f_V : V_1 \to V_2$$
$$f_E : E_1 \to E_2$$

such that the sources and targets of edges are preserved.

An E-graph is a graph equipped with an additional set $V_D$ of data nodes (or values) and special sets of edges $E_{EA}$ (edge attributes) and $E_{NA}$ (node attributes) connecting edges in $E$ and nodes in $V$, respectively, to values in $V_D$. An attributed graph is a tuple $(EG, D)$ where $EG$ is an E-graph and $D$ is an algebra over a data signature

$$\Sigma = (S, OP)$$

such that

$$\biguplus_{s \in S} D_s = V_D$$

is the set of data values available for attribution. An attributed graph morphism

$$f : (EG, D) \to (EG', D')$$

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

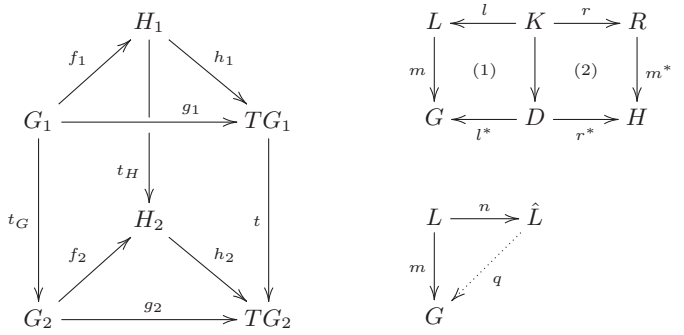



Figure 1. Retyping functor (left), DPO diagram (top right) and satisfaction of negative constraints (bottom right)

is a pair of an E-graph morphism $f_{EG} : EG \to EG'$ and a compatible algebra homomorphism $f_D : D \to D'$. Given an attributed graph

$$AG = (EG, D),$$

we write $AG_G$ to denote the restriction of $EG$ to the underlying graph $G$, disregarding data nodes, node and edge attributes. Analogously, for a morphism $f$, we write $f_G$ to denote the restriction of $f_{EG}$ to the underlying graph.

The category **AGraph** has attributed graphs as objects and attributed graph morphisms as arrows. For an attributed type graph $TG$ (usually over a final $\Sigma$-algebra), we define the category $\mathbf{AGraph_{TG}}$ of $TG$-typed attributed graphs (Ehrig *et al.* 2006).

**Definition 2.1 (typed graphs and retyping functor).** Given a fixed graph $TG$, called a *type graph*, the category of *$TG$-typed graphs* is defined as the comma category $\mathbf{AGraph}_{TG}$. Objects are morphisms in **AGraph** of the form $g : G \to TG$. A morphism $f : g \to h$ from

$$g : G \to TG$$

to

$$h : H \to TG$$

is a morphism in **AGraph** such that

$$h \circ f = g.$$

Given a morphism $t : TG_1 \to TG_2$ between type graphs as in Figure 1 (left), we have a functor $t^\bullet : C_{TG_2} \to C_{TG_1}$ defined on objects by

$$(g_2 : G_2 \to TG_2) \mapsto (g_1 : G_1 \to TG_1)$$

if there is a morphism $t_G : G_1 \to G_2$ such that $t_G, g_1$ is a pullback of $g_2, t$. The functor extends to $TG$-typed morphisms

$$(f_2 : g_2 \to h_2) \mapsto (f_1 : g_1 \to h_1)$$

through the universal property, so all subdiagrams commute and all squares are pullbacks.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

When working in $\mathbf{AGraph}_{TG}$, we will use uppercase letters $G, H, \ldots$ for $TG$-typed graphs. Productions are defined as spans of injective morphisms over the same type graph. Typed attributed graphs are transformed following the double-pushout approach (Ehrig *et al.* 2006) with negative application conditions (Habel *et al.* 1996).

**Definition 2.2 (productions and transformations).** Given a type graph $TG$, a $TG$-typed *graph production*

$$p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right)$$

consists of $TG$-typed graphs $L$,$K$,$R$, called the left-hand side, interface graph and right-hand side, respectively, and two injective morphisms $l$ and $r$ in $\mathbf{AGraph}_{TG}$ that are the identity on the data algebra. A production is non-deleting if $l : K \to L$ is an isomorphism.

Given a $TG$-typed production

$$p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right),$$

a graph $G$ and a morphism $m : L \to G$, called a *match*, a *direct transformation* $G \overset{p,m}{\Longrightarrow} H$ from $G$ to a $H$ exists if a double-pushout (DPO) diagram can be constructed as in Figure 1 (top right), where (1) and (2) are pushouts in $\mathbf{AGraph}_{TG}$ and morphisms $l^*$ and $r^*$ are the identity on the data algebra.

A production is *deterministic on data* if for every potential match, the mapping of attribute values is determined by the graph part, that is, we have for all morphisms $m, m' : L \to G$, that

$$m_G = m'_G$$

implies

$$m = m'.$$

We will assume that all productions are deterministic on data.

The last condition is necessary to avoid non-determinism on the data part, which, typically, can arise from variables that are not bound by an occurrence in the left-hand side and will thus admit an unbounded number of assignments. This could lead to an infinite number of matches for the same production in the same attributed graph, even if the graphical part itself is finite. In the following, we will say an attributed graph $AG$ is *finite* if its graph component $AG_G$ is finite. The above condition ensures that a finite graph can only have finitely many matches for each production that is deterministic on data.

The applicability of productions can be restricted by specifying negative conditions stating the non-existence of certain structures in the context of the match.

**Definition 2.3 (negative application conditions).** A *negative constraint* on a $TG$-typed graph $L$ is a morphism $n : L \to \hat{L}$ in $\mathbf{AGraph}_{TG}$. A morphism $m : L \to G$ satisfies $n$ (written $m \models n$) if and only if there is no $q : \hat{L} \to G$ such that $q \circ n = m$ in $\mathbf{AGraph}_{TG}$ – see Figure 1 (bottom right).

A negative application condition (NAC) on $L$ is a set of constraints $N$. A morphism $m : L \to G$ *satisfies* $N$ (written $m \models N$) if and only if $m$ satisfies every constraint in $N$,

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

that is,

$$\forall n \in N : m \models n.$$

A production with NAC is a pair $\hat{p} = (p, N)$ where

$$p = \left(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\right)$$

and $N$ is a negative application condition over $L$. A *direct transformation using* $\hat{p}$ is a direct transformation $G \overset{p,m}{\Longrightarrow} H$ using the underlying production $p$, such that $m \models N$.

**Definition 2.4 (graph transformation system, transformation).** A *typed graph transformation system (GTS)*

$$\mathcal{G} = (TG, P)$$

consists of a type graph $TG$ and a finite set of $TG$-typed graph productions $P$. A transformation in $\mathcal{G}$ is a sequence of direct transformations

$$s = \left(G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n\right)$$

such that all $p_i$ are in $P$. We also write

$$s : G_0 \overset{\mathcal{G}^*}{\Longrightarrow} G_n$$

for such a sequence, and, given another sequence

$$s' : G'_0 \overset{\mathcal{G}^*}{\Longrightarrow} G'_n$$

with $G_n = G'_0$, we denote composition by

$$s; s' : G_0 \overset{\mathcal{G}^*}{\Longrightarrow} G'_n.$$

This operation is associative and has neutral elements, *viz.* the empty sequences

$$\diamond_G : G \overset{\mathcal{G}^*}{\Longrightarrow} G.$$

GTS $\mathcal{G}$ is *terminating* if there are no infinite transformation sequences

$$G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots$$

from finite graphs $G_0$.

Typed graph transformations provide a formalisation of model transformations. In order to distinguish source and target models, the type graph $TG$ is assumed to be equipped with two subgraphs $TG_S$ and $TG_T$ representing the source and target types, respectively, which are related to $TG$ by injective morphisms $i_S$ and $i_T$, respectively. A set of input graphs $I$ is provided to distinguish those instances of $TG_S$ representing legal source models.

**Definition 2.5 (model transformation system, model transformation).** A *model transformation system*

$$\mathcal{M} = (TG, i_S, I, i_T, P)$$

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

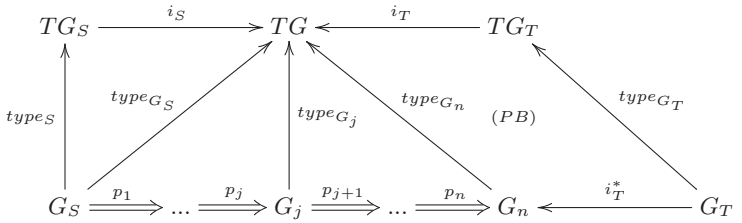



Figure 2. Typing in the model transformation process

consists of a type graph $TG$, injective morphisms

$$i_S : TG_S \to TG$$
$$i_T : TG_T \to TG$$

embedding the source and target type graphs, a set of input graphs $I$ typed over $TG_S$ and a set $P$ of $TG$-typed productions with NACs.

Given a source model $G_S$ in $I$ (and thus typed over $TG_S$), a model transformation is a graph transformation

$$G_S = G_0 \overset{p_1,m_1}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n$$

that terminates in $G_n$, that is, in the GTS $(TG, P)$, there exists no direct transformation

$$G_n \overset{p_{n+1},m_{n+1}}{\Longrightarrow} G_{n+1}.$$

The result of the model transformation is given by the projection $G_T$ of the final graph $G_n$ to the target type graph $TG_T$. This projection, which is defined categorically as a pullback of $i_T$ and $type_{G_n}$ (denoted by $PB$ in Figure 2), removes from $G_n$ all instances of types not in $TG_T$. This means that we will not be concerned with writing productions deleting from $G_n$ any elements not belonging to the target model. Indeed, in most applications we are interested in keeping the source model, for example, so that we can relate source and target elements for traceability. Therefore, while $G_S$ is only typed over source types, intermediate graphs $G_i$ typed over $TG$ combine instances of types from $TG_S$ and $TG_T$ with those of intermediate types (not $TG_S$ nor $TG_T$) representing, for example, traceability relations.

As a consequence of this formalisation of model transformation, our graph transformation productions will normally not delete elements from the source model and will only occasionally delete elements of intermediate or target types. This observation will be important when we ask what termination criteria are realistic for these applications.

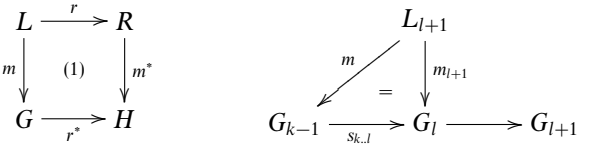
A useful notion in termination criteria is that of an *essential match* (Ehrig *et al.* 2005), which formalises the idea that a production application can be 'delayed' if its match is preserved by another transformation. The idea is that the delayed match is essentially the same as the original one (matching the same graph elements), though in a different graph. The essential match of a step is the first occurrence of this match in the sequence.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

**Definition 2.6 (tracking morphism, essential match).** A graph transformation $G \overset{p,m}{\Longrightarrow} H$ using a non-deleting production

$$p : (L \leftarrow K \rightarrow R)$$

given by the injective morphism $r : L \rightarrow R$ is defined by the pushout (1) of $r$ and $m$. The morphism $r^* : G \rightarrow H$ is called the *tracking morphism* of $G \overset{p,m}{\Longrightarrow} H$.




Given a transformation

$$s = (G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n)$$

that only uses non-deleting productions, we write

$$s_{k..l} = r_l^* \circ \cdots \circ r_k^* : G_{k-1} \rightarrow G_l$$

with $0 < k \leqslant l \leqslant n$ for the composition of the tracking maps of steps $k \ldots l$. A match

$$m : L_{l+1} \rightarrow G_{k-1}$$

is called *the essential match* for the direct transformation

$$G_l \overset{p_{l+1},m_{l+1}}{\Longrightarrow} G_{l+1}$$

with match

$$m_{l+1} : L_{l+1} \rightarrow G_l$$

if

$$s_{k..l} \circ m = m_{l+1}$$

and there is no $j < k$ with

$$n : L_{l+1} \rightarrow G_{j-1}$$

such that

$$s_{j..l} \circ n = m_{l+1}.$$

Intuitively, the essential match for a step in a sequence $s$ is the earliest occurrence of the match for that step. If productions are non-deleting, that match exists throughout the sequence from the point of its creation. This allows us to speak of matches as global entities, independently of the graphs in which they occur. Note that $s_{k..l}$ is injective because all the $r_i$ are injective, and injective morphisms are preserved by pushouts and closed under composition. Therefore, there is exactly one essential match $m$ for each match $n$, and we write

$$s^{-1}(n) =_{def} m.$$

This means that we can use the essential match $m_0$ as a unique representative of the equivalence class of all the matches that are compatible through the tracking morphisms.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

A non-deleting production is *self-disabling* if it has a NAC that prohibits the existence of the pattern the production creates.

**Definition 2.7 (self-disabling production).** Given a non-deleting production

$$p = (L \leftarrow lK \rightarrow rR),$$

a negative constraint $n : L \rightarrow \hat{L}$ for $p$ is *self-disabling* if there exists a morphism $g : \hat{L} \rightarrow R$ such that

$$g \circ n = r.$$

A production $\hat{p} = (p, N)$ is *self-disabling* if its NAC $N$ contains a self-disabling constraint.

The notions of an essential match and self-disablement are only defined for non-deleting productions because a deleting production consumes elements from the match and thus cannot be applied on the same match twice. The following lemma (Ehrig *et al.* 2005) establishes the fact that a self-disabling production, once applied, cannot be applied again on a match with the same essential match (that is, from the same equivalence class of compatible matches).

**Lemma 2.8 (essential match).** Given a transformation

$$s : G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n$$

in a GTS with non-deleting, self-disabling productions only, each production $p_i \in P$ occurs at most once with the same essential match, that is,

$$p_i = p_j$$
$$s^{-1}(m_i) = s^{-1}(m_j)$$

implies $i = j$.

The proof is identical to that given in Ehrig *et al.* (2005), except that we do not require negative constraints, matches and mediating morphisms such as $g$ in Definition 2.7 or $q$ in Figure 1 (bottom right) to be injective. The reason for this lies in the use of attributed graph transformation, where productions contain variables that have to be instantiated by matches, which leads to non-injectivity on the data part. The slight drawback is that constraints with general mediating morphisms are somewhat less expressive than those based on injective ones: for example, they cannot distinguish between the non-existence of one or two identical outgoing edges. As a compromise, we could demand injectivity on the graph part of the morphism alone, leaving the algebra part unrestricted. However, such a complication would only provide minor benefits, but at the cost of sacrificing the categorical nature of the theory.

## 3. Case study

The approach of combining different termination criteria was inspired by the problem of proving the termination of a transformation from UML activity diagrams (OMG 2006) to Communicating Sequential Processes (CSP) (Hoare 1985), which was developed in order

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

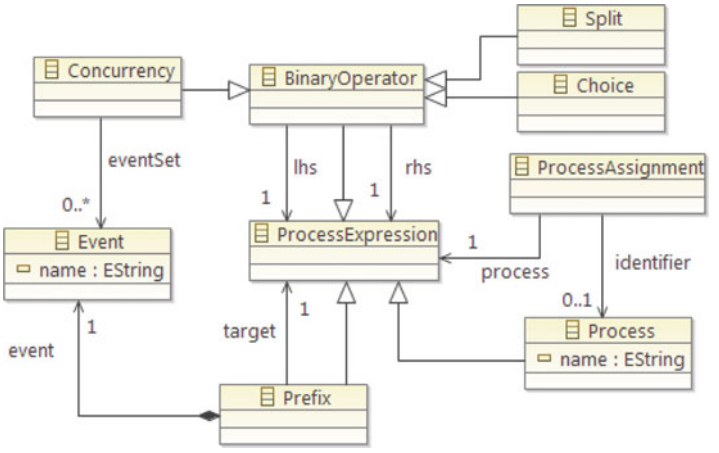



Figure 3. (Colour online) $TG_{CSP}$: CSP metamodel

to verify model refactorings (Varró *et al.* 2008; Bisztray 2009). We will present some of the features of this transformation as an illustration and to provide motivation.

CSP expressions describe interactions between independent processes in terms of an alphabet of events. Processes are defined using recursive equations with guarded expressions – so-called *process assignments*:

$$\overbrace{\underbrace{P =}_{\text{declaration}} \quad \underbrace{(a \rightarrow Q)||(b \rightarrow R)}_{\text{expression}}}^{\text{assignment}}$$

A *declaration* specifies the name of the process. Its behaviour is given by a *process expression*. The syntax of the relevant process assignments is

$$P ::= event \rightarrow P \mid P \,\Box\, Q \mid P \parallel Q \mid SKIP \mid STOP.$$

The prefix $a \rightarrow P$ performs action $a$ and then behaves like $P$. The process $P \,\Box\, Q$ represents external choice between processes $P$ and $Q$. The process $P \parallel Q$ behaves as $P$ and $Q$ engaged in a lock-step synchronisation. $SKIP$ represents successful termination and $STOP$ is a deadlock (Hoare 1985). This textual syntax has a straightforward translation into the type graph $TG_{CSP}$ in Figure 3, which will be the target fragment of the type graph $TG$ of our model.

The metaclasses ProcessAssignment, Process and ProcessExpression capture the notions decribed above. ProcessExpression is an abstract composite class covering different types of expressions. The BinaryOperator is an abstract class representing all process operators taking two operands: the Concurrency and Choice classes representing the parallel composition of processes and external choice, respectively. An example is shown in Figure 4, where process $P$ is initially willing to communicate $a$ and will wait indefinitely for this to happen, then, after $a$, it behaves like $Q$.

In UML, activity diagrams are the language dedicated to workflow modelling. They provide control as well as object flow, signalling and even exception handling, but we will

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

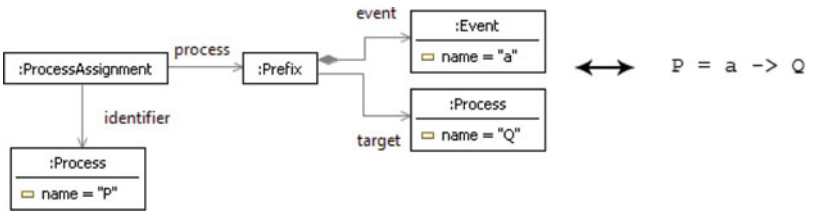



Figure 4. (Colour online) Graph to term: *Prefix*

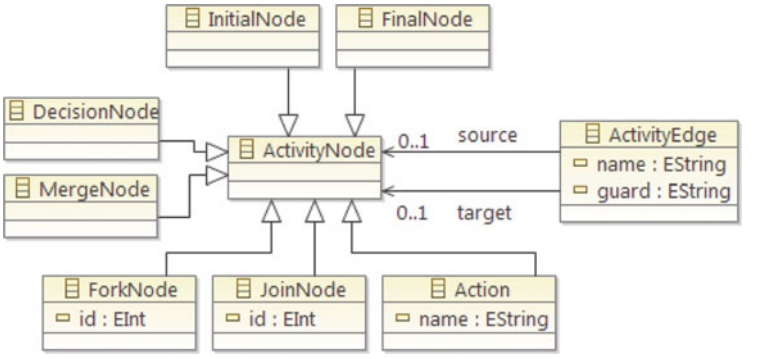



Figure 5. (Colour online) $TG_{AD}$: Activity diagram metamodel

concentrate here on basic control-flow elements. Our simplified metamodel is given by the type graph $TG_{AD}$ in Figure 5.

ActivityNode is an abstract class for points in the flow of an activity connected by edges, including executable nodes and control nodes (OMG 2006). An ActivityEdge is a directed connection representing control flow that has a source and a target ActivityNode, and Action represents a single step within an activity. Although every action in official UML2 is allowed to have multiple incoming and outgoing edges, we restrict them here to having only one incoming and one outgoing edge.

An InitialNode is a control node that starts the flow. A FinalNode stops all executing actions in the activity. A DecisionNode chooses between outgoing flows, while the MergeNode brings together multiple alternate flows. Similarly, the ForkNode splits a flow into multiple concurrent flows, while the JoinNode synchronises multiple flows. A ForkNode has an *id* element, which represents a pairing between a ForkNode and a JoinNode.

Figure 6 shows a simple transformation using the concrete syntax of the participating languages. The idea behind the mapping is to relate an activity edge to a process in CSP. First, all the edges are transformed to the corresponding process declarations, then the various nodes are used to fill the empty process definitions.

Figure 7 shows two basic productions responsible for creating CSP process assignments. The productions are presented in a compact and intuitive format introduced by the Henshin in-place model transformation tool (Arendt *et al.* 2010). The graph objects within the productions are annotated with *create*, *delete*, *preserve* and *forbid* actions. When expanded to the traditional $L \leftarrow K \rightarrow R$ production format, $L$ consists of the

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

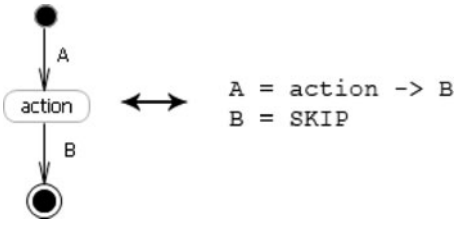
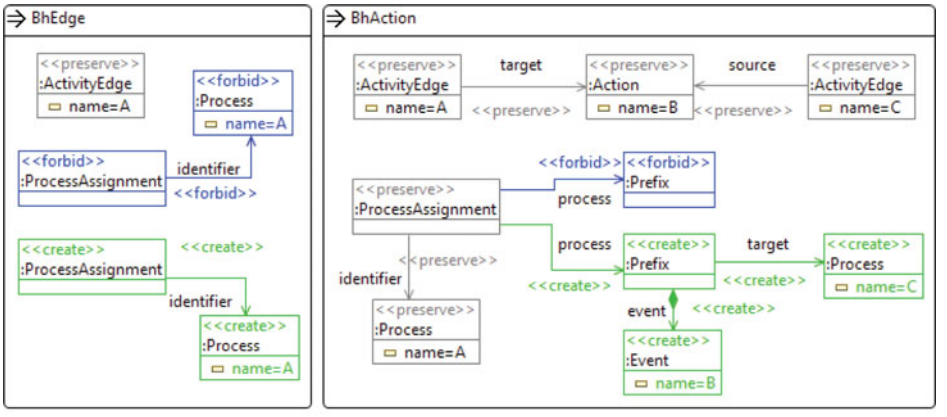



Figure 6. Transformation of basic elements




(a) *BhEdge* (b) *BhAction*

Figure 7. (Colour online) $P_{AD2CSP} = \{\texttt{BhEdge}, \texttt{BhAction}\}$

*preserved* and *deleted* items, $R$ contains the *preserved* and *created* items, $K$ is the set of *preserved* items, and the *forbidden* items form a negative application condition. The *name* is a node attribute, while $A$ and $C$ are variables to be instantiated with values from the host graph.

The `BhEdge` rule in Figure 7(a) transforms an activity edge to an empty process declaration by creating nodes of type ProcessAssignment and Process, with an edge of type identifier for every ActivityEdge node identified by the *name* attribute. The `BhAction` rule creates a process definition from an action by connecting a prefix operator to a ProcessAssignment node for every interconnected activity edge and Action node instance, as shown in Figure 7(b).

## 4. Related work

Although there are some general theoretical results on termination (Plump 1998; Baader and Nipkow 1998), which are mostly focussed on term graph rewriting, we will concentrate here on contributions directed to supporting termination proofs for model transformations.

*Layered graph grammars* (Ehrig *et al.* 2005) comprise one technique with a comprehensive implementation. The idea is to sort transformation productions into layers. In an ordinary graph transformation system, productions are applied non-deterministically, but

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

in a layered system, only one layer is active at a time, and only the productions of the active layer are applied. Then, only when matches for productions of layer $i$ have been exhausted, do we move on to layer $i + 1$.

Layers are classified into *deletion* and *non-deletion* layers. For each deletion layer, the last creation of a node of a certain type must precede the first deletion of a node with the same type. For a non-deletion layer, if an element of a certain type occurs in the left-hand side of a production, then all elements of that type have already been created in previous layers (Ehrig *et al.* 2005). The proof of termination uses self-disabling productions and the fact that, due to the layers, only finitely many matches are ever created.

However, the transformation in Section 3 cannot be converted into a layered graph grammar. The two productions introduced in Figure 7 could be sorted into either one or two layers, but if they are in the same layer, this cannot be a deletion layer because neither of them is deleting. Neither do they satisfy the conditions for non-deletion layers since the ProcessAssignment instance in the left-hand side of `BhAction` would imply that all ProcessAssignment instances should be created by previous layers, but ProcessAssignment instances are created by `BhEdge` in the same layer.

We might try to solve this problem by sorting the productions into two layers, for example, `BhEdge` for layer one and `BhAction` for layer two, but the classification of layers is still problematic since neither can be a deleting layer (because both productions are non-deleting), but layer two does not satisfy the non-deletion condition since there is not only a ProcessAssignment instance in the left-hand side of `BhAction`, but also a Process instance. Unfortunately, `BhAction` itself creates Process instances. In fact, the generic node types in CSP, such as Process or Event, are created by almost every production in the system. All this means that we cannot apply the technique introduced in Ehrig *et al.* (2005).

Another approach, which was used in Varró *et al.* (2006), verifies termination using *Place-Transition (PT) nets*. The idea is to derive a PT net from a graph transformation system to simulate the original system. This so-called *cardinality net* contains a place $p_y$ for every node or edge $y$ in the type graph, and a transition $t_r$ for every production $r$ in the system. There is an incoming arc $(p_y, t_r)$ if the left-hand side of production $r$ contains an element of type $y$, and there is an outgoing arc $(t_r, p_y)$ if the right-hand side of $r$ contains an element of type $y$. The number of elements of type $y$ in the left- or right-hand side defines the weight of the arc.

This mapping is extended by *permission places* to handle negative application conditions. Permission places count the number of possible applications for a graph transformation production. The cardinality net has a permission place $p_{r,N}$ for every negative constraint $N$ in the application condition of a production $r$. The existence and weights of incoming and outgoing arcs are then dependent on a causal relation between the productions: if the right-hand side of production $r$ contains the forbidden pattern of a constraint $N$ for production $p$, there is an arc $(p_{r,N}, t_r)$ with weight 1. There is an arc of variable weight if a production deletes or creates an object from the pattern of constraint $N$, with the exact weight dependent on the start graph.

A PT net is *partially repetitive* if there is a marking $M_0$ and a firing sequence $s$ from $M_0$ such that some transition occurs in $s$ infinitely often. Thus, if the net is not partially

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

repetitive, there are only finite sequences from any initial $M_0$, which proves termination of the original graph transformation system simulated by the cardinality net. The checking of this condition can be automated, but is based on a manually produced incidence matrix of the cardinality net.

We believe that this approach would be able to handle our case study, though the effort of deriving the cardinality net and matrix manually may be prohibitive and error prone. However, as we point out in the example below, a significant limitation lies in the fact that the approach is largely indifferent to the structure of the start graph.

The technique in Levendovszky *et al.* (2007) proves termination by accumulating the preconditions of all applicable productions in sequence. The Concurrency Theorem (Ehrig *et al.* 2006) then allows us to merge two productions based on their overlap. For example, in a transformation

$$G_0 \overset{p_1}{\Longrightarrow} G_1 \overset{p_2}{\Longrightarrow} G_2,$$

the concurrent production $p_1 *_E p_2$ produced, when applied at the corresponding match, transforms $G_0$ directly into $G_2$. To accumulate preconditions, we build concurrent productions by composing the productions sequentially in all possible ways to create a series

$$p_i^* = (p_1 *_{E_1} p_2) *_{E_2} \ldots *_{E_{i-1}} p_i.$$

If the size of the left-hand sides of the accumulated concurrent productions increases monotonically when new productions are added, termination can be proved. This criterion works for our motivating example if there are only two small productions. However, the full transformation system contains 11 productions with 8–18 elements in each of their left- or right-hand sides, and producing all possible combinations of these by hand is impractical.

Apart from such considerations of scalability, neither Varró *et al.* (2006) nor Levendovszky *et al.* (2007) take account of any structural properties of the input graphs. The system shown in Figure 8 is a toy example illustrating this point. It is responsible for transforming a sequence of nodes into a set of nodes of the same cardinality. In the presence of a cycle in the original sequence, the system does not terminate, but if we limit it to acyclic graphs, termination is obvious.

While Varró *et al.* (2006) will return an indefinite result, the manual nature of Levendovszky *et al.* (2007) may help us discover the termination condition of $GTS_{Counting}$. In analysing possible ways in which productions can be merged, Figure 9 shows an example where the concurrent production incorporates a cycle. In $p_4^*$ a `first` and three `count` productions are merged. The transition from $p_2^*$ to $p_3^*$ adds the $c$ NODE to the left-hand side. In $p_4^*$, the third `count` production overlaps the $c$ and $b$ NODES. $p_5^*$ shows what happens in the case of a cycle: the left-hand side is preserved, but the number of TOKEN nodes increases on the right-hand side. Indeed, Levendovszky *et al.* (2007) ensures termination when the size of the accumulated left-hand sides increases monotonically because in that case it will eventually outgrow any finite input graph. However, this is not the case if, as in our example, the input graph is cyclic.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

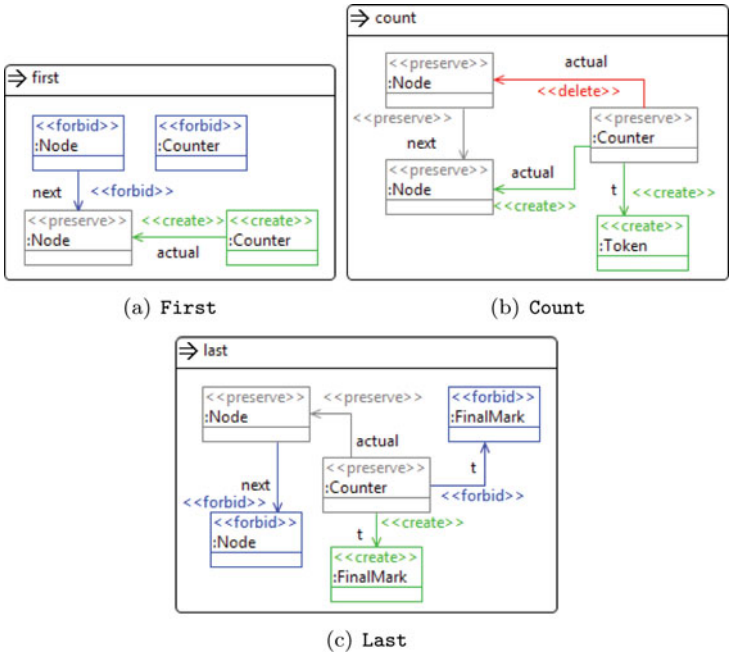



Figure 8. (Colour online) $GTS_{Counting}$ for counting nodes

## 5. Termination of non-deleting graph transformation systems

As pointed out in Section 2, a common approach to model transformation is one where information is copied from source to target models using self-disabling negative application conditions to ensure termination. This means that productions do not need to delete source model elements, and will often be non-deleting overall, with the advantage that negative application conditions, once violated, will not be enabled again in the future. This idea is captured in Lemma 2.8.

In this section, we present a termination criterion for non-deleting graph transformation systems based on analysing the ability of non-deleting productions to create new matches for each other. *Produce–use* dependencies are potential sequential dependencies between productions that are based on a first production creating certain elements that can be used for the match of the second. If this dependency relation is acyclic, the GTS is terminating.

Consider, for example, the `BhEdge` and `BhAction` productions. The former creates the ProcessAssignment and Process nodes with their connecting edge for a given ActivityEdge node in the source model. This structure is needed to find a match for the production `BhAction`. This means that every application of `BhAction` is preceded (though possibly indirectly) by an application of `BhEdge` to the same process assignment. This notion of *precedence* is formalised by the produce–use dependency: the productions $p_1$ and $p_2$ are produce–use dependent if $p_1$ produces nodes and/or edges that can enable the application of $p_2$.

We have developed this notion for general productions because it will be useful in the next section, where the termination of systems with possibly deleting productions

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

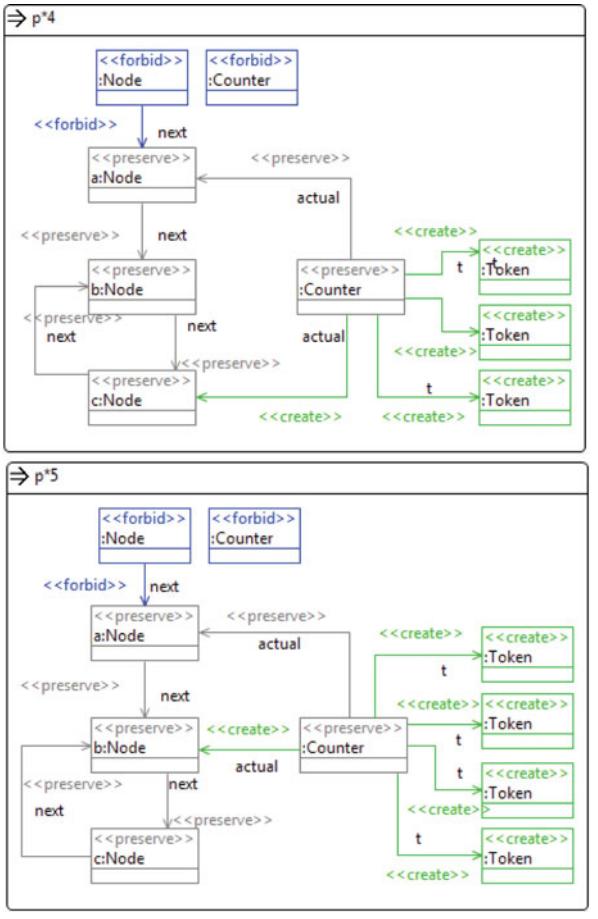



Figure 9. (Colour online) Concurrent productions $p_4^*$ and $p_5^*$

will be considered. It is worth pointing out that even in the general case, produce–use dependencies play a special role among the causal dependencies between productions $p_1$ and $p_2$. If in a *use–delete* situation, $p_2$ deletes elements from a potential match of $p_1$, they are sequentially dependent in the sense that they cannot occur in reverse order. However, production $p_1$ does not represent a prerequisite for $p_2$'s application, that is, its match exists independently. If $p_1$, by deleting elements, allows the negative application condition of $p_2$ to be satisfied, this could be called a *delete–enable* dependency. The intuition here is that matches for $p_2$ are not newly generated by $p_1$, but, instead, existing, but 'dormant', matches are activated by removing forbidden structures, that is, to ensure termination, we will have to guarantee that the number of active and dormant matches is limited overall. Finally, the case of *forbid–create*, where production $p_2$ creates elements that could prevent an application of $p_1$ due to its negative application condition is analogous to use–delete in the sense that the applicability of $p_2$ does not depend on that of $p_1$. Note that in the concrete category of typed attributed graphs, write access to an attribute means deleting a link connecting to its value and creating a new such link, while read access uses that

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

link as part of the match. Therefore, dependencies on attributes are a special case of the above.

**Definition 5.1 (produce–use dependency).** Given a graph transformation system $\mathcal{G} = (TG, P)$, productions $p_1, p_2$ are in *produce–use* dependency, denoted by

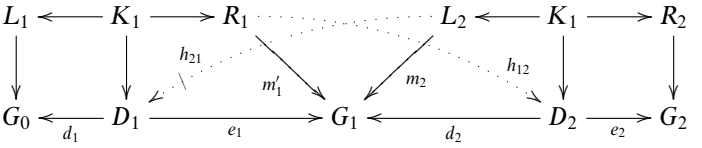
$$p_1 \xrightarrow{pu} p_2$$

if and only if there exists a transformation

$$G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} G_2$$

such that

(1) There is no $h_{21} : L_2 \to D_1$ such that $e_1 \circ h_{21} = m_2$.
(2) There exists $h_{12} : R_1 \to D_2$ such that $d_2 \circ h_{12} = m'_1$.




Condition (1) means that $p_2$ is dependent on $p_1$: certain elements matched by $L_2$ are created by $p_1$ and are therefore not present in the host graph beforehand. Condition (2) ensures that the items created by $p_1$ are not deleted by $p_2$ (that is, they are gluing items). Condition (2) is always satisfied for non-deleting productions, but the definition is required for general productions as explained in Section 6, where dependencies based on deletion will receive special treatment.

Produce–use is a binary relation on productions. It is usually not symmetric, but, since two different productions may be mutually produce–use dependent, that is, $p_1 \xrightarrow{pu} p_2$ and $p_2 \xrightarrow{pu} p_1$ for $p_1 \neq p_2$, neither is *antisymmetric* in general. If the transitive closure $\xrightarrow{pu*}$ of $\xrightarrow{pu}$ is irreflexive, that is, there are no cycles in $\xrightarrow{pu}$, no production is produce–use dependent on itself, either directly or indirectly. Hence, productions cannot mutually produce new matches for each other. The proof of the following theorem is based on this argument.

**Theorem 5.2 (termination of non-deleting GTS).** Given a graph transformation system $\mathcal{G} = (TG, P)$ such that all productions are non-deleting and self-disabling, and the transitive closure of the *produce–use* relation is irreflexive, $\mathcal{G}$ is terminating.

*Proof.* If $\mathcal{G}$ is terminating, there are no infinite transformation sequences

$$s = \left( G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \right)$$

from finite graphs $G_0$. In order to prove termination by creating a contradiction, we assume the existence of an infinite transformation $s$ as above.

Since $P$ contains finitely many productions and $s$ is an infinite sequence, by the Pigeonhole principle (Grimaldi 1998), the productions in the sequence cannot be pairwise

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

distinct, and, indeed, there must be at least one production $p \in P$ that is applied infinitely often.

For each direct transformation

$$G_{i-1} \overset{p,m_i}{\Longrightarrow} G_i,$$

there is an essential match $s^{-1}(m_i)$. From Lemma 2.8, we know that there is only one application of $p$ to that essential match, that is,

$$G_{j-1} \overset{p,m_j}{\Longrightarrow} G_j$$

with

$$s^{-1}(m_j) = s^{-1}(m_j)$$

implies

$$i = j.$$

Therefore, to get an infinite number of applications of $p$, we need an infinite number of distinct essential matches.

Since the initial graph $G_0$ is finite, there are only finitely many matches present at the start, which means that an infinite number of matches for $p$ must be created during the transformation. But new matches are only created due to produce–use dependencies, and to create an infinite number of matches, the relation must have a cycle containing a production $q$ with $q \overset{pu}{\longrightarrow} p$. However, this contradicts the assumption that the transitive closure $\overset{pu*}{\longrightarrow}$ is irreflexive, so there is no infinite sequence $s$. □

## 6. Termination of model transformation systems

Theorem 5.2 does not carry over to systems with general productions. Although deleting productions are also effectively self-disabling in the sense that they cannot be applied twice on the same essential match, a production $p$ may delete elements contained in the negative application condition of another production $q$, thus enabling it on a match that was previously disabled. Even if a self-disabling NAC in $q$ is violated by $q$'s action of creating certain elements, these may be deleted again by $p$, and so on, leading to a cycle.

Hence, in order to extend the termination criterion presented in Section 5, we will isolate deletion-based dependencies by considering connected components with respect to this relation. Termination can then be shown locally for each component by whatever means appropriate, and if produce–use dependencies between productions in different components are acyclic, termination follows from an argument similar to that of Theorem 5.2.

In line with our discussion of model transformation in Section 2, we will make use of the distinction between source and target metamodels: in particular, the fact that the source model may be subject to additional constraints and should be preserved by the transformation.

**Definition 6.1 (protective model transformation system).** A model transformation system

$$\mathcal{M} = (TG, i_S, I, i_T, P)$$

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

is *protective* if for all productions

$$p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right) \in P,$$

their reduction

$$i_S^\bullet(L) \xleftarrow{i_S^\bullet(l)} K \xrightarrow{i_S^\bullet(k)} i_S^\bullet(R)$$

through the inclusion on type graphs $i_S : TG_S \to TG$ yields a pair of isomorphisms.

This means that productions in $\mathcal{M}$ have no effect on the source part of the model.

As a motivating example, we will now add the two new productions `BhDecision1` and `BhDecision2` to the productions of $P_{AD2CSP}$ in Figure 7, as shown in Figure 10. Since a decision node has one incoming edge and multiple outgoing activity edges (OMG 2006), the productions will convert a 1-to-$n$ junction into a binary tree, bottom up, as shown in Figure 11. First, an arbitrary ActivityEdge node is matched to create the lowest element of the tree. The tree is then built up by adding elements one by one. BhDecision2 is deleting: as shown in Figure 10(b), the process edge going directly from the ProcessAssignment node to the preserved Choice node is deleted. Hence, Theorem 5.2 does not apply to $P_{AD2CSP}$ because it contains deleting productions. The strategy to deal with the deleting productions is to isolate them.

$P_{AD2CSP}$ can be sorted into subsets responsible for transforming certain elements. For example, `BhDecision1` and `BhDecision2` will not interfere with the elements matched by `BhAction`. This means that deletion is contained in the component { `BhDecision1`, `BhDecision2` }: productions delete only what has been created by other productions in the same component. This means that productions should be in the same component if they are produce–delete dependent.

**Definition 6.2 (produce–delete dependency).** Productions $p_1$ and $p_2$ are *produce–delete* dependent, written $p_1 \xrightarrow{pd} p_2$, if there exists a transformation

$$G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$$

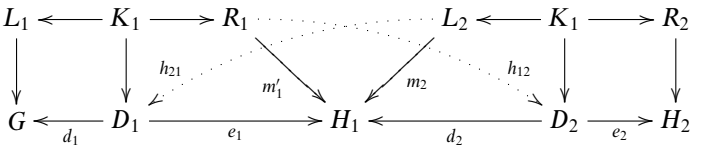
such that there exists neither $h_{12} : R_1 \to D_2$ with

$$d_2 \circ h_{12} = m_1'$$

nor $h_{21} : L_2 \to D_1$ with

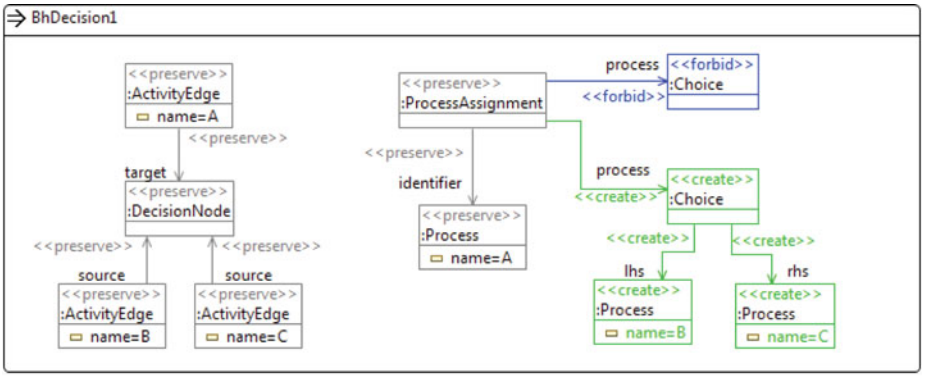
$$e_1 \circ h_{21} = m_2.$$

Graphically,

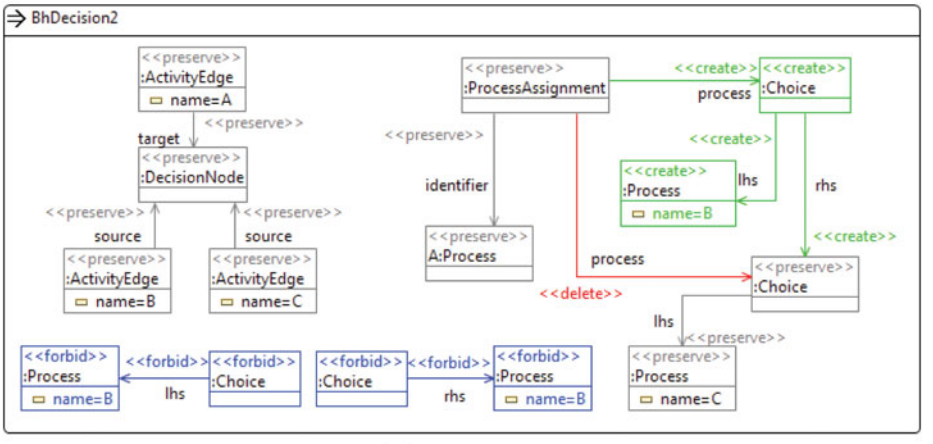



This means that $p_2$ deletes elements created by $p_1$. Another way in which deletion can enable another production is through removing parts of a forbidden pattern.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press




(a) BhDecision1




(b) BhDecision2

Figure 10. (Colour online) Productions BhDecision1 and BhDecision2

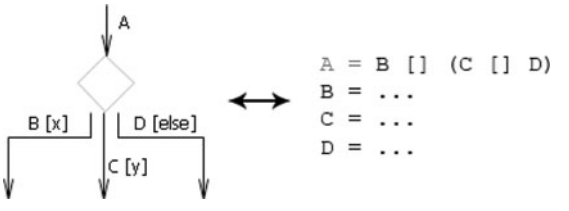



Figure 11. Transformation of the decision node

**Definition 6.3 (delete–enable dependency).** Productions

$$\hat{p}_1 = (p_1, N_1)$$
$$\hat{p}_1 = (p_2, N_2)$$

are *delete–enable* dependent, written $\hat{p}_1 \overset{de}{\longrightarrow} \hat{p}_2$, if there exists a transformation

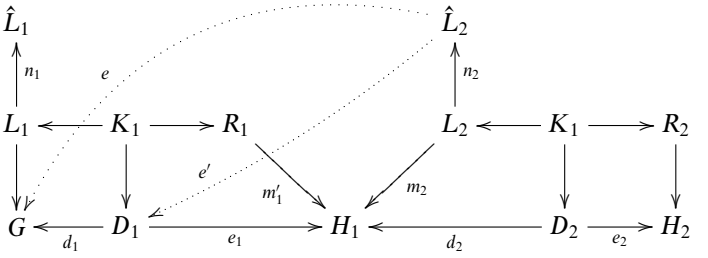
$$G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$$

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

and a constraint $n_2 \in N_2$ with $e : \hat{L}_2 \to G$, but no $e' : \hat{L}_2 \to D_1$ with

$$d_1 \circ e' = e.$$

This means that $p_1$ deletes part of an occurrence of a forbidden pattern $\hat{L}_2$ prohibiting the application of $p_2$ at match $e \circ n_2$.




**Definition 6.4 (deletion components).** For a source-protective model transformation system

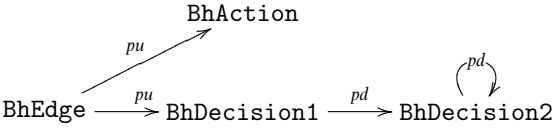
$$\mathcal{M} = (TG, i_S, I, i_T, P),$$

we define the equivalence relation $\overset{d^*}{\longleftrightarrow}$ on $P$ as the symmetric, transitive and reflexive closure of $\overset{pd}{\longrightarrow} \cup \overset{de}{\longrightarrow}$. The equivalence classes with respect to $\overset{d^*}{\longleftrightarrow}$ are called the *deletion components of* $P$.

Given two such deletion components $P_1, P_2 \subseteq P$, they are in produce–use dependency, written $P_1 \overset{pu}{\longrightarrow} P_2$, if there exist productions $p_1 \in P_1$ and $p_2 \in P_2$ such that $p_1 \overset{pu}{\longrightarrow} p_2$.

A deletion component $D \subseteq P$ is terminating if for all finite graphs $G$ with $G|_{TG_S} \in I$, there is no infinite transformation starting in $G$ using productions from $D$ only.

Thus, deletion components are connected components in the graph with productions in $P$ as nodes and $\overset{pd}{\longrightarrow}$ and $\overset{de}{\longrightarrow}$ relations as edges. As an example, consider the following dependency graph over productions $P_{CSP2AD}$:




The PROCESS and PROCESSASSIGNMENT nodes created by `BhEdge` may trigger an application of `BhAction` or `BhDecision1`, so `BhAction` and `BhDecision1` are produce–use dependent on `BhEdge`. Since they are not delete–enable or produce–delete dependent on each other, according to Definition 6.4, they are not $\overset{d^*}{\longleftrightarrow}$-related, and hence not in the same component. The dependency between `BhDecision1` and `BhDecision2` is more interesting since they seem to be produce–use dependent because `BhDecision2` matches the CHOICE and PROCESS nodes created by `BhDecision1`. However, according to Condition (2) in Definition 5.1, the connected CHOICE and PROCESS nodes created by `BhDecision1` should be gluing items in `BhDecision2`. However, this is clearly not the case, and this is why Condition (2) is necessary: the deleting production has to be grouped with the productions

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

that create the elements it deletes. As a result, we have three components:

$$\{\texttt{BhEdge}\}, \{\texttt{BhAction}\}, \{\texttt{BhDecision1}, \texttt{BhDecision2}\}$$

It is important to recognise that components are not the same as the layers of Ehrig *et al.* (2005). The most obvious difference is the production application strategy in layered graph transformation. Layers are executed one after the other sequentially, the transition to layer $n+1$ occurring only after all productions have been applied to all possible matches in layer $n$. Production application in a model transformation system with components is standard, that is, productions are applied non-deterministically, and the application of productions of different components can interleave. For instance, after an application of `BhEdge` and `BhDecision1`, there is no restriction on `BhDecision2`. We can apply `BhEdge` or `BhDecision1` again on a different match.

We will now extend Theorem 5.2 to model transformation systems with deleting productions. The idea is that arbitrary termination criteria can be used to prove the termination of the individual deletion components. Once their termination has been determined, we use the transitive closure of the produce–use relation on components to show that they cannot produce matches for each other in a cyclic manner.

The following is a generalisation of the notion of an essential match, where we drop the limitation to non-deleting productions and allow for the possibility of negative applications conditions. This will be used to formalise the idea of one production being able to create matches for another, which will be an essential ingredient in the proof of the main theorem (Theorem 6.7).

**Definition 6.5 (essential match for general productions).** For a transformation

$$s; t_2 : G_0 \overset{\mathcal{G}^*}{\Longrightarrow} G_2,$$

the *anticipated match for step*

$$t_2 = \left(G_1 \overset{\hat{p}_2, m_2}{\Longrightarrow} G_2\right)$$

*with respect to s*, written

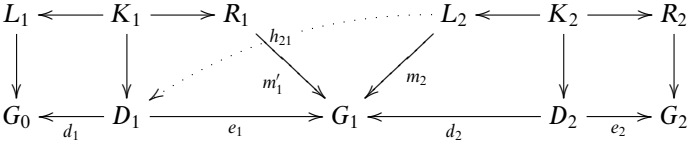
$$am(s, t_2) = m : L_2 \to G_0,$$

is defined by the following cases:

(1) $am(\diamond_{G_1}, t_2) = m_2$ (see Definition 2.4);
(2) $am(t_1, t_2) = m_{21}$ if $t_2$ is independent of

$$t_1 = \left(G_0 \overset{\hat{p}_1, m_1}{\Longrightarrow} G_1\right),$$

that is, in the diagram




there is a morphism $h_{21} : L_2 \to D_1$ with

$$e_1 \circ h_{21} = m_2$$

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

such that match

$$m_{21} = d_1 \circ h_{21}$$

satisfies the negative application condition of $\hat{p}_2$;

(3) $am(s; t_1, t_2) = am(s, t_2')$ if

$$am(t_1, t_2) = m_2'$$

and

$$t_2' = \left( G_0 \overset{\hat{p}_2, m_2'}{\Longrightarrow} G_1' \right)$$

Given a transformation $s = s_1; t; s_2$, the *essential match* for $t$ in $s$ is given by $am(s_{12}, t)$, where $s_{12}$ is the largest suffix of $s_1$ such that the anticipated match of $t$ with respect to $s_{12}$ exists. This means that for all

$$s_{11}'; s_{12}' = s_1$$

with $am(s_{12}', t)$, there exists $s'$ such that

$$s_{12} = s'; s_{12}'.$$

In a sequence $s$, we say that step $t_1$ *via* $p_1$ creates a match for step $t_2$ *via* $p_2$ if

$$s = s_1; t_1; s_2; t_2$$

such that the essential match for $t_2$ is $am(s_2, t_2)$. Production $p_1$ creates a match for $p_2$ if there exists a sequence $s$ with steps $t_1, t_2$ as above such that $t_1$ creates a match for $t_2$.

Case (1) in Definition 6.5 states that the anticipation of a match across an empty sequence is possible and results in the same match. Case (2) anticipates a match of $t_2$ across another step $t_1$ if $t_2$ does not depend on $t_1$. Case (3) provides an inductive extension to a sequence of arbitrary finite length by repeatedly dropping steps $t_1$ if allowed by Case (2). The essential match of $t$ is the result of applying this procedure to the largest possible sequence $s_{12}$ preceding $t$ in $s$. The following lemma provides the main technical ingredient allowing us to generalise the proof of our main theorem (Theorem 6.7) from non-deleting to general systems.

**Lemma 6.6 (enabling productions).** Given a model transformation system

$$\mathcal{M} = (TG, i_S, I, i_T, P)$$

with productions $p_1, p_2 \in P$, if $p_1$ creates a match for $p_2$, then $p_1 \xrightarrow{pu} p_2$ or there exists a deletion component in $D \subseteq P$ such that $p_1, p_2 \in D$.

*Proof.* We will show that a match is created by $p_1$ for $p_2$ if $p_2$ is produce–use, produce–delete or delete–enable dependent on $p_1$ (any other dependency does not create new matches for $p_2$, but only prevents the application of $p_2$ after $p_1$). The lemma will then follow from Definition 6.4.

Production $p_1$ creates a match for $p_2$ if $am(u, t_2)$ is the essential match for $t_2$ in $s; t_1; u; t_2$. According to Definition 6.5, this is the case if $t_2$ is independent of all the steps in $u$, but not of $t_1$. This means, referring to the diagram in Definition 6.5, that either there is no

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

morphism $h_{21} : L_2 \to D_1$ with

$$e_1 \circ h_{21} = m_2,$$

or match

$$m_{21} = d_1 \circ h_{21}$$

does not satisfy the negative application condition of $\hat{p}_2$. In the first case, the steps are either produce–use or produce–delete dependent, and in the second case, they are delete–enable dependent. □

We will now state our main result: the characterisation of termination in terms of the termination of deletion components and the acyclicity of their produce–use dependencies. In order to help these local termination proofs, we can assume that input graphs satisfy certain constraints, thus limiting the set $I$ to a proper subset of the $TG_S$-typed graphs.

**Theorem 6.7 (termination of protective MTS).** Given a protective model transformation system

$$\mathcal{M} = (TG, i_S, I, i_T, P)$$

where all productions are self-disabling (see Definition 2.7), if the transitive closure of the produce–use dependency relation on components is irreflexive and each deletion component in $P$ is terminating, then $\mathcal{M}$ is terminating, that is, for all finite start graphs $G_0 \in I$, there is no infinite transformation starting in $G_0$.

*Proof.* In order to prove termination by creating a contradiction, we assume the existence of an infinite transformation $s$ starting in a finite graph $G_0 \in I$.

Since $P$ has finitely many productions and $s$ is an infinite sequence, there must be at least one production $p \in P$ that is applied infinitely often. Since all productions are either self-disabling or deleting, no production can be applied twice at the same match. There are only finitely many matches in any finite graph, so fresh matches must be generated indefinitely. This means that $p$ is part of a cycle $p_0 p_1 \dots p_n \in P$ with $p_n = p_0$ and $n \geqslant 1$ such that $p_i$ creates a match for $p_{i+1}$.

The cycle $p_0 p_1 \dots p_n$ separates into $k$ fragments $p_{i,1} \dots p_{i,l_i}$ $(1 \leqslant i \leqslant k)$ such that all $p_{i,1} \dots p_{i,l_i}$ are in the same component, while $p_{i,l_i}$ and $p_{i+1,1}$ are in different ones. Since $p_{i,l_i}$ creates a match for $p_{i+1,1}$ while belonging to a different component, by Lemma 6.6, they must be produce–use dependent. This leads to a cycle of produce–use dependencies on components, which contradicts the assumption unless $k = 1$. This means that all productions $p_0 p_1 \dots p_n$ are from the same component $C$.

We are going to show that it is possible to filter out from the infinite transformation $s$ an infinite transformation $s|_C$ containing only productions from $C$. By assumption, individual components terminate on graphs $G$ with $G|_{TG_S} \in I$. This condition holds for $G_0$ and is preserved by productions in the protective system $\mathcal{M}$. Therefore, all graphs in the sequence share this property. An infinite sequence using productions in $C$ from a graph $G$ reachable from a start graph $G_0 \in I$ thus represents a contradiction.

In order to construct sequence $s|_C$, we consider

$$s = s_1; t_1; s_2; t_2; s_3; t_3; \dots$$

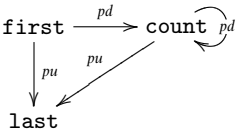
https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

decomposed into steps $t_i$ using productions from the cycle and intermediate transformations $s_i$ such that each $t_i$ creates the match for $t_{i+1}$. If we could now drop all the $s_i$, we would be left with the infinite sequence $t_1; t_2; \ldots$ of steps in $C$ only. However, this will not be possible in general because a step $t_j$ may depend on a step $u_i$ within $s_i, (i < j)$. If $u_i$ is a step in the same component $C$, we just keep it as part of the sequence to be constructed.

We show that the set of all $i \in n$ such that some $t_j$ depends on $u_i$ and such that $u_i$ is a step in a different component is finite. We can then pick the largest index $l$ from that set and build the infinite sequence starting from $t_l; t_{l+1}; \ldots$. So we now assume that this set is infinite. This means that there are infinitely many steps in $s$ outside $C$ that are not dependent on steps in $C$ because otherwise the produce–use relation would be cyclic on components. So, by similar arguments to those above, these steps must form a cycle inside another component $D$.

If $D$ has finitely many outside dependencies, we can form the infinite sequence of steps in $D$, which contradicts the assumption of local termination. If $D$ has infinitely many outside dependencies, the game continues until the last component is reached. This cannot have any outside dependencies at all, since otherwise the produce–use relation would have a cycle, so we again have an infinite sequence of steps in a single component, and thus a contradiction. □

This theorem can be applied to the node-counting example in Figure 8. First we need to sort the productions {`first`,`count`,`last`} into deletion components. The only deleting production is `count`, which deletes edges created by `first` and `count`. According to Definition 6.4, these productions must be in the same component. `last` is a non-deleting production enabled by `first` and `count`. The following graph shows these dependencies:




Thus, we have a decomposition into two deletion components {`first`, `count`} and {`last`} with an acyclic produce–use dependency between them. The fact that the components terminate on acyclic input graphs follows from Levendovszky *et al.* (2007), as shown in Section 4. Hence, the entire system terminates on acyclic input graphs.

The definitions and results of the current paper can be transferred to the framework of adhesive categories. Specifically, finitary $\mathcal{M}$-adhesive categories (Braatz *et al.* 2010), where each object can only have a finite number of sub-objects, provide a generalisation of the notion of a finite (attributed) graph. With additional assumptions on the left-hand sides of productions, we can guarantee that each finite graph is the target of a finite number of matches only, thus allowing us to generalise the reasoning in the proof of Theorem 6.7.

## 7. Applications and discussion

In the previous sections we have set out the foundations of our approach. In order to apply the criteria established in Theorems 5.2 and 6.7, we need to carry out four tasks:

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

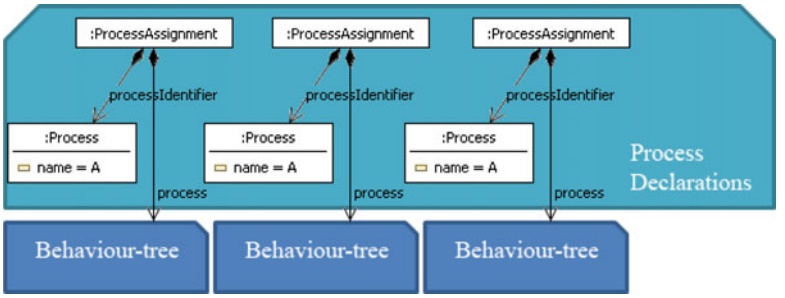



Figure 12. (Colour online) The structure of the CSP graph

(1) Determine the produce–use, produce–delete and delete–enable dependencies.
(2) Compute the deletion components as connected components of produce–delete and delete–enable dependencies.
(3) Verify the acyclicity of produce–use on components.
(4) Prove the termination of deletion components on graphs with valid source model part.

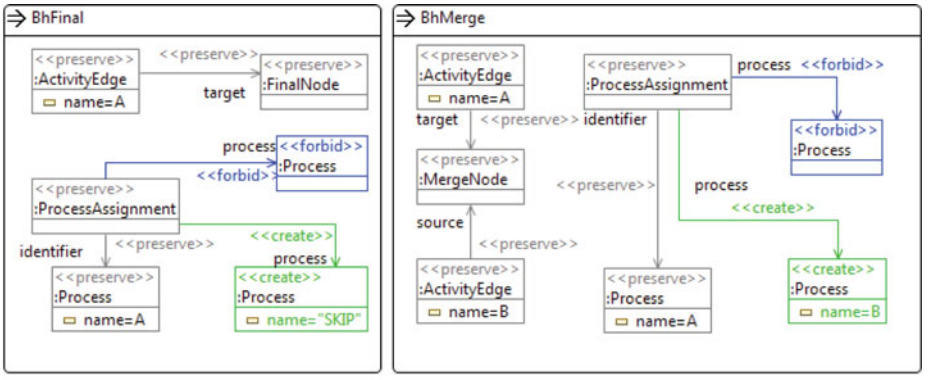
The first task can be automated by the Attributed Graph Grammar System (AGG) (Ermel *et al.* 1999), which provides the analysis of sequential dependencies between productions as a basic analysis capability. Currently, the classification of such dependencies into the three categories has to be performed manually, but it would be easy to automate this too. The same holds for tasks (2) and (3). The problem of termination for individual components is undecidable in general, but termination criteria can be applied in many cases, as discussed in Section 4, or the components may be small enough to prove termination manually.

The average size of deletion components depends on the amount of deletion in the system. In exogenous one-way model transformations, such as those transforming platform-independent models into platform specific ones, the source model is kept unchanged, so deletion may only be necessary when temporary structures are updated, for example, to keep track of the progress of a transformation. One-way transformations generated from triple graph grammar productions (Schürr 1994) satisfy this condition naturally.

In our own motivating case study, *viz.* the transformation of activity diagrams to CSP (Varró *et al.* 2008; Bisztray 2009), productions are indeed designed according to the structure of triple graph grammars, even if they were never formally used in the implementation. Since the applicability of productions is controlled by self-disabling negative constraints over the creation of elements in the target model, the source model can be retained and the system remains protective, which allows us to apply Theorem 5.2. Also, deleting productions are the exception, so deletion components are indeed small, with three components having two productions each, and all other components just containing a single production.

For the structure of the transformation, consider Figure 12 showing the abstract syntax tree of a CSP specification. Process declarations at the root are connected to their behaviour subtrees. The transformation builds these trees top-down: elements closer to

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press




(a) *BhFinal* (b) *BhMerge*

Figure 13. (Colour online) `BhFinal` and `BhMerge`

the root are created before their branches. Process declarations are created by production `BhEdge` from control flow edges of the activity diagram. The declarations are then completed, with one behaviour subtree corresponding to one activity diagram decision node, forknode or action.

This structure is reflected in the deletion components derived from the dependencies: the set of productions $P_{AD2CSP}$ splits into components responsible for transforming a certain type of element of the activity diagram, while not interfering with the behaviour subtrees created by other components. For example, the productions for building the subtree associated with a decision node will not modify a subtree that describes the behaviour of a fork node.

The role of `BhEdge` and `BhAction` was introduced in Section 3. Rule `BhFinal`, shown in Figure 13(a), completes a declaration by a $SKIP$ process. `BhFork1` and `BhFork2` (and `BhForkNoJoin1`, `BhForkNoJoin2`) transform the fork nodes, and their structure is similar to the decision nodes (see Figure 10), but matching ForkNodes instead of DecisionNodes and creating Concurrency instead of Choice operators. The component comprising `BhForkNoJoin1` and `BhForkNoJoin2` deals with forks without any synchronisation. The `BhMerge` production in Figure 13(b) merges multiple flows back into one process, and the analogous production `BhJoin` synchronises incoming flows.

We used AGG to generate the three dependency relations, straining the tools capabilities in the process – memory problems meant the analysis had to be restarted repeatedly for particularly large and complex pairs of productions. The resulting dependencies are shown in Figure 14.

Deletion components are determined by collecting productions that are related by produce–delete (pd) or delete–enable dependencies into one component. Apart from components that only contain a single production, we have the deletion components

$$\{\texttt{BhDecision1}, \texttt{BhDecision2}\}$$
$$\{\texttt{BhFork1}, \texttt{BhFork2}\}$$
$$\{\texttt{BhForkNoSync1}, \texttt{BhForkNoSync2}\}.$$

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

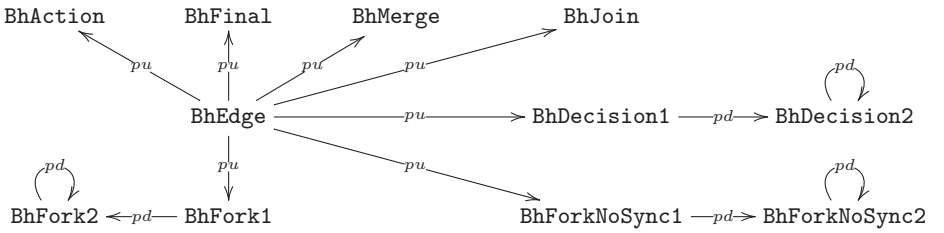



Figure 14. Dependency graph generated using AGG

There are no directed produce–use (pu) cycles in the graph, so the transitive closure of this relation on components is irreflexive. Also, all productions are self-disabling and preserve the source model. This means that we can apply Theorem 5.2 if we are able to show that components terminate individually.

We will consider component $\{$BhDecision1, BhDecision2$\}$ first. BhDecision1 is a non-deleting production creating behaviour-tree $A_d$. Unless the root of the CHOICE tree is deleted, this production cannot be applied again to the same essential match given by the DECISIONNODE and PROCESSASSIGNMENT nodes for which the CHOICE node has been created. BhDecision2 is a deleting production building a binary tree. BhDecision1 can only be applied once on the same DECISIONNODE, so BhDecision2 is the only production able to alter $A_d$. As seen in Figure 10(b), the production's only action is to replace the deleted PROCESS edge. Thus, if one DECISIONNODE is connected to $n$ outgoing edges, this production may be applied precisely $n-2$ times. Since $G_0$ is finite and there is only a finite number of DECISIONNODES, BhDecision2 can only be applied finitely often.

The termination of other deletion components follows by similar arguments, and termination then follows from Theorem 6.7.

## 8. Conclusions

In the current paper we have addressed the termination problem for model transformation systems with a distinguished source and target model such that the source model is preserved. We first established a criterion for non-deleting graph transformation systems with self-disabling rules only. Generalising from this, we then established a technique to combine existing proof techniques for termination that can be applied to components of rules where the effect of deletion has been isolated. The conditions are geared towards the restrictions likely to match applications in practice, specifically, for one-way exogenous transformations.

Apart from providing more specific tool support and further evaluating the approach, for example, by seeking applications to industrial model transformation languages, future work will investigate the possibility of a structured *design* of model transformation systems as opposed to a component structure *extracted* from the system through dependency analysis. Thus, the approach has the potential to turn into a design method enabling structured reasoning on complex sets of rules.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

## References

Arendt, T., Biermann, E., Jurack, S., Krause, C. and Taentzer, G. (2010) Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Petriu, D. C., Rouquette, N. and Haugen, O. (eds.) Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). *Springer-Verlag Lecture Notes in Computer Science* **6394** 121–135.

Baader, F. and Nipkow, T. (1998) *Term Rewriting and All That*, Cambridge University Press.

Bisztray, D. (2009) *Compositional Verification of Model-Level Refactorings Based on Graph Transformations*, Ph.D. thesis, Department of Computer Science, University of Leicester.

Bisztray, D. and Heckel, R. (2010) Combining termination criteria by isolating deletion. In: Ehrig, H., Rensink, A., Rozenberg, G. and Schrr, A. (eds.) Graph Transformations – Proceedings 5th International Conference (ICGT 2010). *Springer-Verlag Lecture Notes in Computer Science* **6372** 203–217.

Bottoni, P., Koch, M., Parisi-Presicce, F. and Taentzer, G. (2005) Termination of high-level replacement units with application to model transformation. *Electronic Notes in Theoretical Computer Science* **127** (4) 71–86.

Braatz, B., Ehrig, H., Gabriel, K. and Golas, U. (2010) Finitary m-adhesive categories. In: Ehrig, H., Rensink, A., Rozenberg, G. and Schrr, A. (eds.) Graph Transformations – Proceedings 5th International Conference (ICGT 2010). *Springer-Verlag Lecture Notes in Computer Science* **6372** 234–249.

Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D. and Varró-Gyapay, S. (2005) Termination criteria for model transformation. In: Cerioli, M. (ed.) Proceedings International Conference on Fundamental Approaches to Software Engineering (FASE 2005). *Springer-Verlag Lecture Notes in Computer Science* **3442** 49–63.

Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. (2006) *Fundamentals of Algebraic Graph Transformation*, EATCS Monographs in Theoretical Computer Science, Springer-Verlag.

Ermel, C., Rudolf, M. and Taentzer, G. (1999) The AGG approach: Language and environment. In: *Handbook of Graph Grammars and Computing by Graph Transformation – volume 2: Applications, Languages and Tools*, World Scientific 551–603.

Grimaldi, R. P. (1998) *Discrete and Combinatorial Mathematics: An Applied Introduction*, 4th edition, Addison Wesley Publishing Company.

Habel, A., Heckel, R. and Taentzer, G. (1996) Graph grammars with negative application conditions. *Fundamenta Informaticae* **26** (3/4) 287–313.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*, Prentice Hall.

Levendovszky, T., Prange, U. and Ehrig, H. (2007) Termination criteria for dpo transformations with injective matches. *Electronic Notes in Theoretical Computer Science* **175** (4) 87–100.

OMG (2006) Unified Modeling Language, version 2.1.1. `http://www.omg.org/technology/documents/formal/uml.htm`.

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

Schürr, A. (1994) Specification of graph translators with triple graph grammars. In: Mayr, E. W., Schmidt, G. and Tinhofer, G. (eds.) Graph-Theoretic Concepts in Computer Science. Proceedings 20th International Workshop (WG'94). *Springer-Verlag Lecture Notes in Computer Science* **903** 151–163.

Varró, D. *et al.* (2008) Transformation of UML models to CSP: A case study for graph transformation tools. In: Schrr, A., Nagl, M. and Zndorf, A. (eds.) Applications of Graph Transformations with Industrial Relevance – Revised Selected and Invited Papers from Third

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press

International Symposium, AGTIVE 2007. *Springer-Verlag Lecture Notes in Computer Science* **5088** 540–565.

Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U. and Taentzer, G. (2006) Termination analysis of model transformations by Petri nets. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L. and Rozenberg, G. (eds.) Proceedings Third International Conference on Graph Transformation (ICGT 2006). *Springer-Verlag Lecture Notes in Computer Science* **4178** 260–274.

https://doi.org/10.1017/S0960129512000369 Published online by Cambridge University Press