From Algebraic Graph Transformation to Adhesive HLR Categories and Systems

Ulrike Prange and Hartmut Ehrig

Technical University of Berlin, Germany
{uprange,ehrig}@cs.tu-berlin.de

Abstract. In this paper, we present an overview of algebraic graph transformation in the double pushout approach. Basic results concerning independence, parallelism, concurrency, embedding, critical pairs and confluence are introduced. As a generalization, the categorical framework of adhesive high-level replacement systems is introduced which allows to instantiate the rich theory to several interesting classes of high-level structures.

1 Introduction to Graph Transformation

Combining the important concepts of graphs, grammars and rewriting, the research area of graph grammars or graph transformation is a discipline of computer science which dates back to the 1970s. Methods, techniques, and results from the area of graph transformation have already been studied and applied in many fields of computer science, such as formal language theory, pattern recognition and generation, compiler construction, software engineering, the modeling of concurrent and distributed systems, database design and theory, logical and functional programming, artificial intelligence, and visual modeling. A detailed presentation of various graph grammar approaches and application areas of graph transformation is given in the handbooks [1, 2, 3].

This wide applicability is due to the fact that graphs are a very natural way of explaining complex situations on an intuitive level. Hence, they are used in computer science almost everywhere, for example for data and control flow diagrams, for entity relationship and UML diagrams, for Petri nets, for visualization of software and hardware architectures, for evolution diagrams of nondeterministic processes, for SADT diagrams, and for many more purposes.

The main idea of graph transformation is the rule-based modification of graphs, as shown in Fig. 1. The core of a rule or production p is a pair of graphs (L, R), called the left-hand side L and the right-hand side R. Applying the rule $p = (L, R)$ means finding a match of L in the source graph G and replacing L by R, leading to the target graph H of the graph transformation. The main technical problems are how to delete L from G and how to connect R with the remaining context leading to the target graph H. In fact, there are several different solutions how to handle these problems, leading to several different graph transformation approaches.

S. Bozapalidis and G. Rahonis (Eds.): CAI 2007, LNCS 4728, pp. 122–146, 2007.

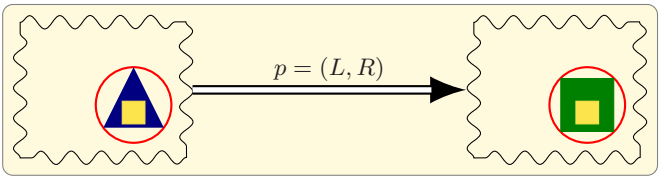


Fig. 1. Rule-based modification of graphs

The algebraic graph transformation approach is based on pushout constructions, where pushouts are used to model the gluing of two graphs along a common subgraph. Intuitively, we use this common subgraph and add all other nodes and edges from both graphs. In the algebraic approach, two gluing constructions are used to model a graph transformation step. For this reason, this approach is also known as the double-pushout (DPO) approach.

Roughly speaking, a production is given by $p = (L, K, R)$, where L and R are the left- and right-hand side graphs and K is the common interface of L and R, i.e. their intersection. The left-hand side L represents the preconditions of the rule, while the right-hand side R describes the postconditions. K describes a graph part which has to exist to apply the rule, but which is not changed. $L \backslash K$ describes the part which is to be deleted, and $R \backslash K$ describes the part to be created.

A direct graph transformation with a production p is defined by first finding a match m of the left-hand side L in the current host graph G and then constructing the pushouts (1) and (2) in Fig. 2. For the construction of the first pushout, however, a gluing condition has to be satisfied, which allows us to construct D such that G is the gluing of L and D via K. The second pushout means that H is the gluing of R and D via K. This means that a direct graph transformation $G \Rightarrow H$ in Fig. 2 consists of two gluing constructions, which are pushouts in the category of graphs and graph morphisms.

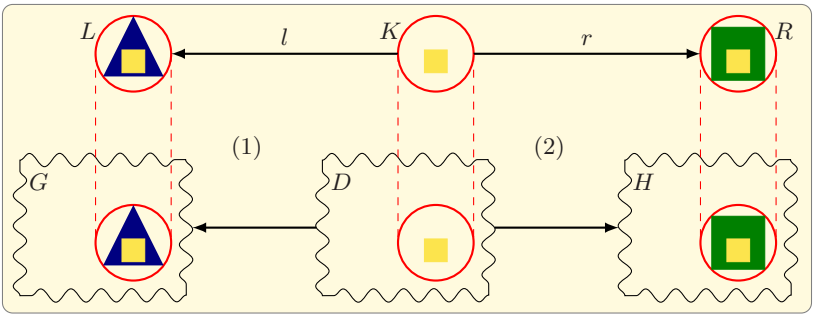


Fig. 2. DPO graph transformation

The algebraic approach to graph transformation is not restricted to (standard) graphs, but has been generalized to a large variety of different types of graphs and other kinds of high-level structures, such as labeled graphs, typed graphs, hypergraphs, attributed graphs, Petri nets, and algebraic specifications. This extension from graphs to high-level structures – in contrast to strings and trees, considered as low-level structures – was initiated in [4, 5] leading to the theory of high-level replacement (HLR) systems. In [6, 7], the concept of high-level replacement systems was joined to that of *adhesive categories* introduced by Lack and Sobociński in [8], leading to the concept of adhesive HLR categories and systems. There are several interesting instantiations of adhesive HLR systems, including not only graph and typed graph transformation systems, but also hypergraph, Petri net, algebraic specification, and typed attributed graph transformation systems.

In addition to pushouts, which correspond to the gluing of graphs, adhesive HLR categories are based on pullbacks, corresponding to the intersection and homomorphic preimages of graphs. The basic axioms of adhesive HLR categories require construction and basic compatibility properties for pushouts and pullbacks. These properties (and a few additional ones) allow to prove several interesting results concerning transformations.

In Section 2, we introduce algebraic graph transformation based on the double pushout approach and present the main results for transformations together with illustrating examples. The categorical framework of adhesive HLR systems is introduced in Section 3. For a more detailed presentation including all the proofs and further results we refer to our book [7].

2 Algebraic Graph Transformation – The Double Pushout Approach

In this section, we introduce graph transformation in the double pushout approach and give an overview of important results. We present the main results with illustrative examples, but give only an intuitive idea of some of the new notions used in these results. A formal definition of these notions and also the proofs of these results are given in [7].

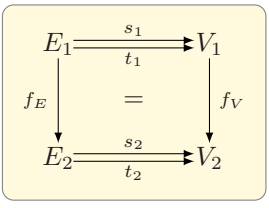
2.1 Graph and Typed Graph Transformation

In this section, we introduce graph and typed graph transformation systems, or (typed) graph transformation systems, for short. In the following, we always use an abbreviated terminology of this kind to handle both cases simultaneously.

A graph has nodes, and edges, which link two nodes. We consider directed graphs, i.e. every edge has a distinguished start node (its source) and end node (its target). We allow parallel edges, as well as loops. Graphs are related by (total) graph morphisms, which map the nodes and edges of a graph to those of another one, preserving the source and target of each edge.

Definition 1 (Graph). *A* graph $G = (V, E, s, t)$ *consists of a set* V *of nodes (also called vertices), a set* E *of edges, and two functions* $s, t : E \to V$*, the source and target functions.*

Given graphs G_1, G_2 *with* $G_i = (V_i, E_i, s_i, t_i)$ *for* $i = 1, 2$*, a* graph morphism $f : G_1 \to G_2$*,* $f = (f_V, f_E)$ *consists of two functions* $f_V : V_1 \to V_2$ *and* $f_E : E_1 \to E_2$ *that preserve the source and target functions, i.e.* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$*.*



If f_V *and* f_E *are both injective (bijective) then* f *is called an injective (isomorphic) graph morphism.*

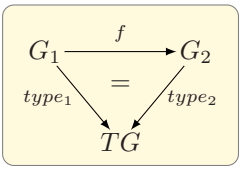
Graphs and graph morphisms form the category **Graphs** *of graphs.*

A type graph defines a set of types, which can be used to assign a type to the nodes and edges of a graph. The typing itself is done by a graph morphism between the graph and the type graph.

Definition 2 (Typed graph). *A* type graph *is a distinguished graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$*.* V_{TG} *and* E_{TG} *are called the vertex and the edge type alphabets, respectively.*

A tuple $G^T = (G, type)$ *of a graph* G *together with a graph morphism type* : $G \to TG$ *is then called a* typed graph*.*

Given typed graphs $G_1^T = (G_1, type_1)$ *and* $G_2^T = (G_2, type_2)$*, a* typed graph morphism $f : G_1^T \to G_2^T$ *is a graph morphism* $f : G_1 \to G_2$ *such that* $type_2 \circ f = type_1$*.*



Typed graphs and typed graph morphisms form the category $\mathbf{Graphs_{TG}}$ *of typed graphs over the type graph* TG*.*

For simplicity, in the following we use the notation G for both graphs and typed graphs.

Example 1. In the following, we model a variant of Dijkstra's algorithm for mutual exclusion (see [9]). Given two processes that compete for a resource used by both of them, the aim of the algorithm is to ensure that once one process is using the resource the other has to wait and cannot access it.

There is a global variable *turn* that assigns the resource to any of the processes initially. Each process i has a flag $f(i)$ with possible values 0, 1, 2, initially set to 0, and a state that is initially *non-active*. If the process wants to access the resource, its state changes to *active* and the flag value is set to 1. If the variable *turn* has assigned the resource already to the requesting process, the flag can be set to 2, which indicates that the process is accessing the resource. Then the process uses the resource and is in its critical section. Meanwhile, no other process can access the resource, because the turn variable cannot be changed in this stage of the process. After the critical section has been exited, the flag is set back to 0 and the state to *non-active*. Otherwise, if the resource is assigned to a nonactive process, it can be reassigned and then accessed analogously by the requesting process.

The type graph TG is given in Fig. 3. Each process is typed by P, a resource is typed by R, and T denotes the turn. If the flag of a process is set to 0, we do not depict it in the graph. The flag values 1 and 2 are shown by nodes typed with $F1$ or $F2$, respectively, with a link from the corresponding process to the node and a link to the required resource.

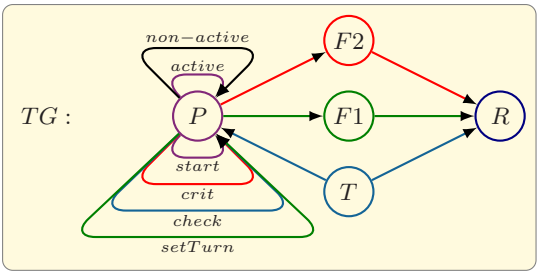


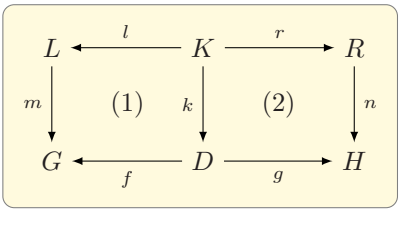
Fig. 3. Example type graph

A typed graph S is given in Fig. 4, containing two nonactive processes that can compete for one resource, where the graph morphism $type : S \to TG$ is given by the labels of the nodes and edges. □

(Typed) graph transformation is based on (typed) graph productions, which describe a general way how to transform (typed) graphs. The application of a (typed) graph production to a (typed) graph is called a direct (typed) graph transformation. This is based on the concept of pushouts which is motivated to be a gluing construction in the introduction.

Definition 3 (Graph production and transformation). *A* (typed) graph production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *consists of (typed) graphs L, K, and R, called the left-hand side, gluing graph, and the right-hand side respectively, and two injective (typed) graph morphisms l and r.*

Given p, a (typed) graph G, and a (typed) graph morphism $m : L \to G$, called match, a direct (typed) graph transformation $G \overset{p,m}{\Longrightarrow} H$ *from G to a (typed) graph H is given by the pushouts (1) and (2), where the (typed) graph morphism n is called comatch.*



A sequence $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ of direct (typed) graph transformations is called a (typed) graph transformation *and is denoted by $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$, we have the identical (typed) graph transformation $G_0 \overset{id}{\Rightarrow} G_0$. Moreover, for $n = 0$ we allow also graph isomorphisms $G_0 \overset{\simeq}{=} G'_0$, because pushouts and hence also direct graph transformations are only unique up to isomorphism.*

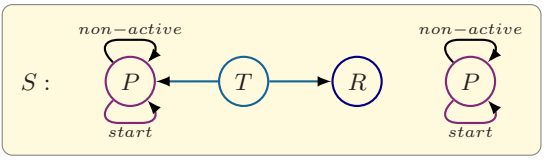


Fig. 4. Example typed graph

Example 2. For our mutual exclusion example, we have five typed graph productions shown in Fig. 5, where all morphisms are inclusions. The typed graph production *setFlag* allows a nonactive process to indicate a request for the resource by setting its flag to 1. The typed graph production *setTurn*1 allows the turn to be changed to an active process if the other process, which has the turn, is nonactive. If the turn is already assigned to the active process, then the turn remains in *setTurn*2. Thereafter, in the typed graph production *enter*, the process enters its critical section. Finally, the process exits the critical section with the typed graph production *exit* and another process may get the turn and access the resource.

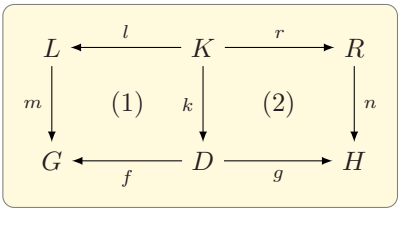
We can apply the typed graph production *setFlag* to the typed graph S given in Fig. 4 with a match m, leading to the direct typed graph transformation $S \overset{setFlag,m}{\Longrightarrow} G_1$ shown in Fig. 6.

If we apply the typed graph productions *setFlag*, *setTurn*1, *enter*, *setFlag*, and *exit* to S, then we obtain the typed graph transformation $S \overset{*}{\Rightarrow} G$ shown in Fig. 7. □

Now we analyze under what conditions a (typed) graph production $p = (L \leftarrow K \rightarrow R)$ can be applied to a (typed) graph G via a match m. In general, the existence of a context graph D that leads to a pushout (1) is required. This allows us to construct a direct (typed) graph transformation $G \overset{p,m}{\Longrightarrow} H$, where, in a second step, the (typed) graph H is constructed as the gluing of D and R via K leading to a pushout (2). Note that the construction of D and H is unique up to isomorphism.

Definition 4 (Gluing condition). *A (typed) graph production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is* applicable *to a (typed) graph G via the match m if the following condition holds:*

p and m satisfy the gluing condition *if all identification points and all dangling points are also gluing points, i.e. $IP \cup DP \subseteq GP$, where*



– *the* gluing points *GP are those nodes and edges in L that are not deleted by p, i.e. $GP = l_V(V_K) \cup l_E(E_K) = l(K)$,*

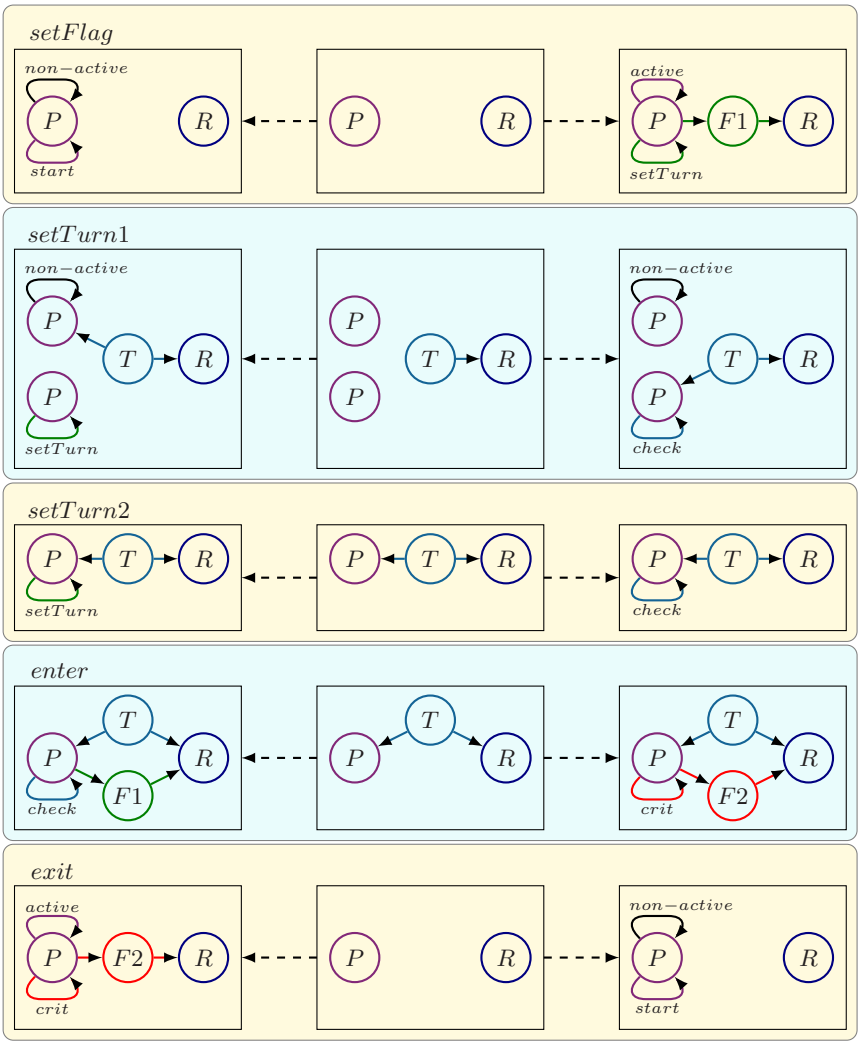


Fig. 5. Example typed graph productions

– *the* identification points *IP* *are those nodes and edges in L that are identified by m, i.e.* $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$,
– *the* dangling points *DP* *are those nodes in L whose images under m are the source or target of an edge in G that does not belong to* $m(L)$, *i.e.* $DP = \{v \in V_L \mid \exists e \in E_G \backslash m_E(E_L) : s_G(e) = m_V(v) \text{ or } t_G(e) = m_V(v)\}$.

Example 3. For the direct typed graph transformation in Fig. 6, we analyze the gluing, identification, and dangling points:

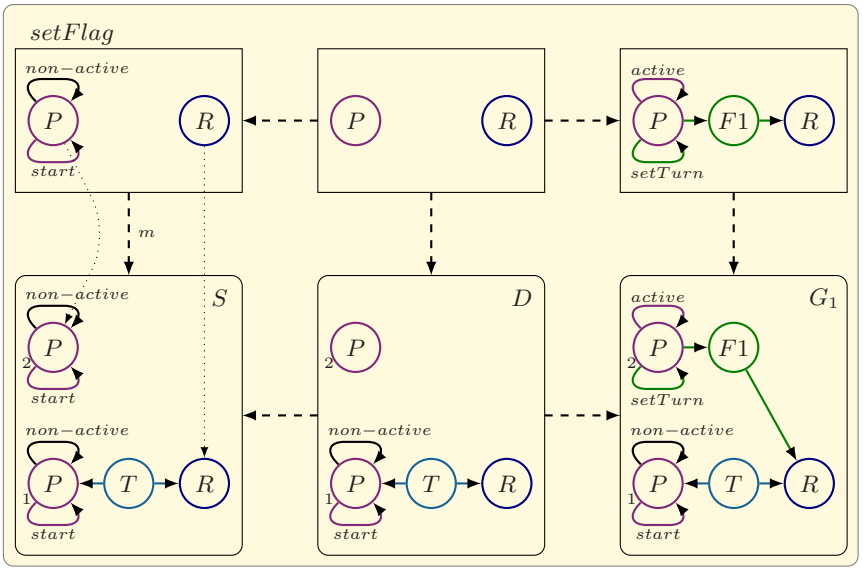


Fig. 6. Example direct typed graph transformation

- $GP = l(K)$, which means that the gluing points in L are both nodes.
- $IP = \varnothing$, since m does not identify any nodes or edges.
- The resource node is the only dangling point: in S, there is an edge from the turn node T (which has no preimage in L) to the resource node R, but there is no edge from or to the upper process node P that is not already in L.

This means that $IP \cup DP \subseteq GP$, and the gluing condition is satisfied by m and *setFlag*.

In contrast, the typed graph production *deleteProcess* given in the top row of Fig. 8 is not applicable to S with the match m'. We have:

- $GP = l(K)$, which means that there are no gluing points in L.
- $IP = \varnothing$, since m' does not identify any nodes or edges.
- The process node in L is a dangling point: in S, there are two loops at this node, which have no preimages in L.

This means that $DP \nsubseteq GP$, and the gluing condition is not satisfied by m' and *deleteProcess*. □

Now we shall define (typed) graph transformation systems and (typed) graph grammars. The language of a (typed) graph grammar consists of those (typed) graphs that can be derived from the start graph.

Definition 5 (Graph transformation system and graph grammar). *A* graph transformation system $GTS = (P)$ *consists of a set of graph productions* P.

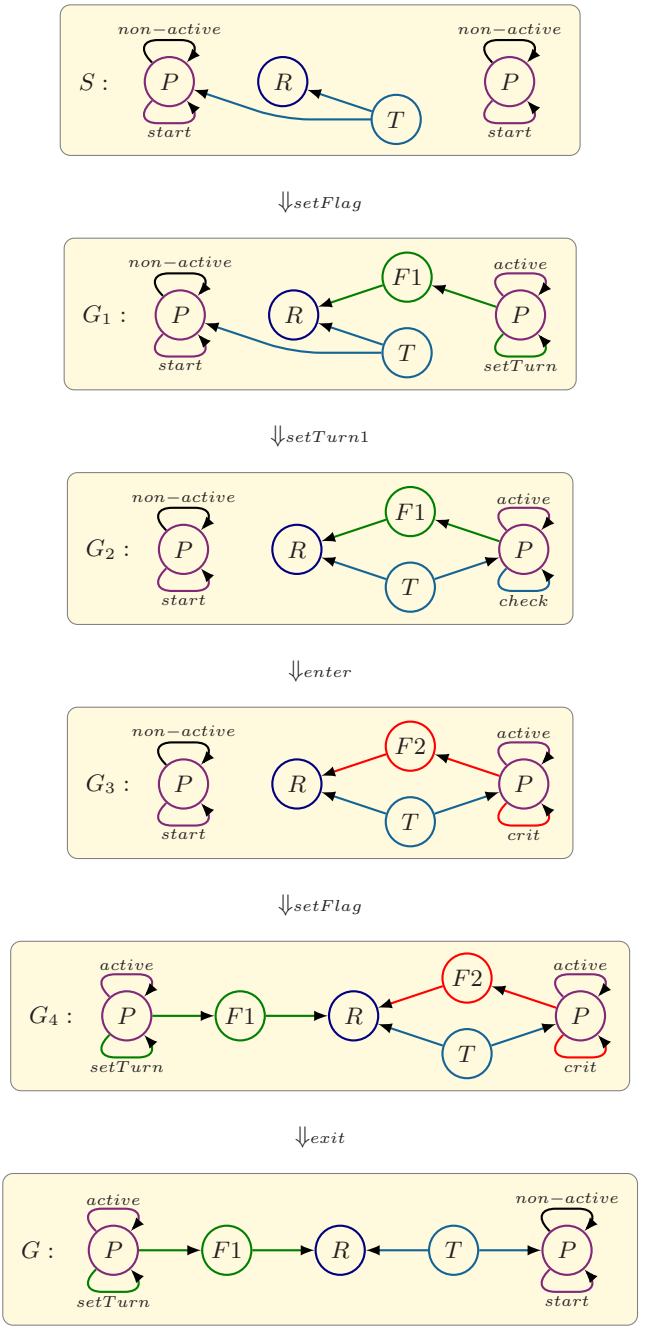


Fig. 7. Example typed graph transformation

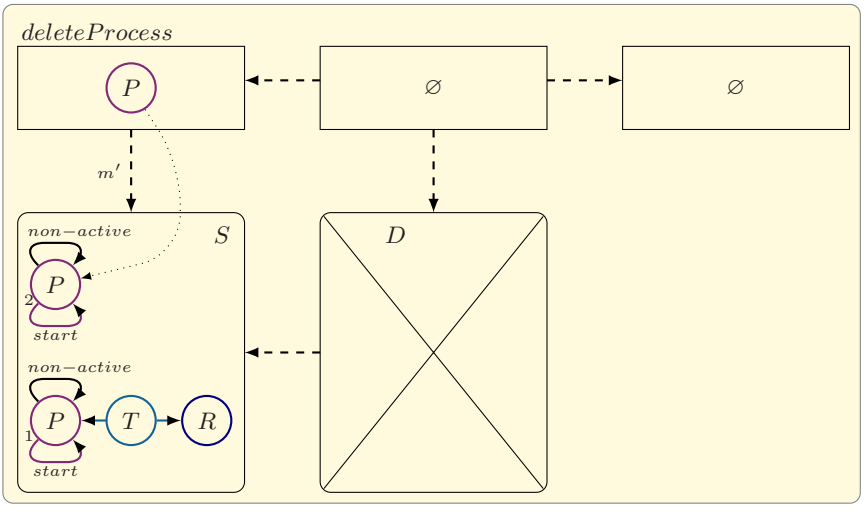


Fig. 8. Example of non-applicability

A typed graph transformation system $GTS = (TG, P)$ *consists of a type graph* TG *and a set of typed graph productions* P.

A (typed) graph grammar $GG = (GTS, S)$ *consists of a (typed) graph transformation system* GTS *and a (typed) start graph* S.

The (typed) graph language L *of a (typed) graph grammar* GG *is defined by*

$$L = \{G \mid \exists \text{ (typed) graph transformation } S \overset{*}{\Rightarrow} G\}.$$

Example 4. Combining the type graph in Fig. 3, the typed graph productions in Fig. 5 and the start graph S in Fig. 4 we have the typed graph grammar $MutualExclusion = (TG, P, S)$ with $P = \{setFlag, setTurn1, setTurn2, enter, exit\}$.

To show that this typed graph grammar indeed ensures mutual exclusion, the whole derivation graph is depicted in Fig. 9. The nodes – which stand for the graphs in the typed graph language – show, in an abbreviated notation, the state of the processes. On the left-hand side of each node, the state of the first process is shown, and also its flag value and if the turn is assigned to that process. Analogously, this information for the second process is depicted on the right-hand side. The marked nodes are those nodes where the resource is actually accessed by a process – and only one process can access it at any one time. □

2.2 Overview of Results for (Typed) Graph Transformations

In the following, we present important results for (typed) graph transformations, namely the

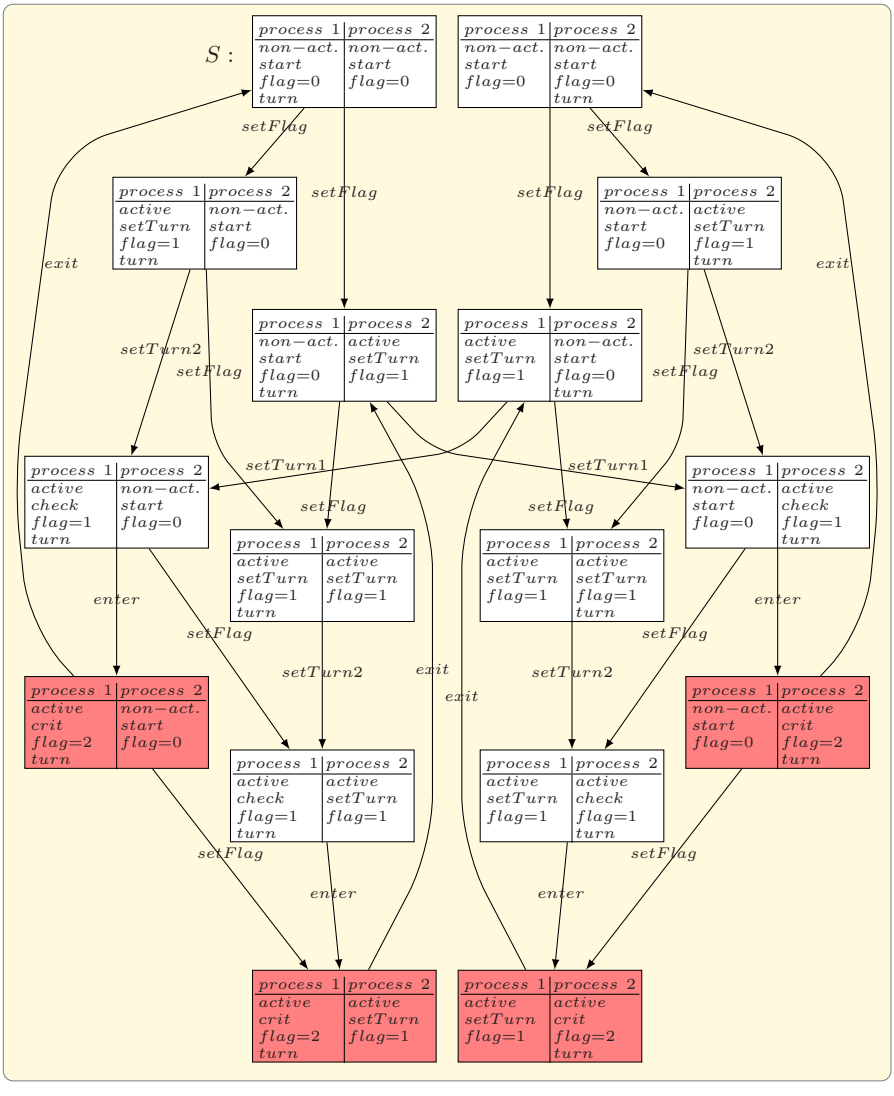


Fig. 9. Example language

– Local Church–Rosser and Parallelism Theorem,
– Concurrency Theorem,
– Embedding and Extension Theorem,
– Critical Pairs and Local Confluence Theorem,
– Graph Constraints and Application Conditions.

Local Church–Rosser and Parallelism Theorem

The first theorem is concerned with parallel and sequential independence of direct (typed) graph transformations. We study under what conditions two direct (typed) graph transformations applied to the same (typed) graph can be applied in arbitrary order, leading to the same result. This leads to the Local Church–Rosser Theorem. Moreover, the corresponding (typed) graph productions can be applied in parallel in this case, leading to the Parallelism Theorem.

Two direct (typed) graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent, if p_1 does not delete anything p_2 uses, and vice versa. This means that all nodes and edges in the intersection of the two matches are gluing items with respect to both transformations, i.e.

$$m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2)).$$

Analogously, two direct (typed) graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ are sequentially independent, if p_1 does not create something p_2 uses, and p_2 does not delete something p_1 uses or creates. This means that all nodes and edges in the intersection of the comatch $n_1 : R_1 \to H_1$ and the match m_2 are gluing items with respect to both transformations, i.e.

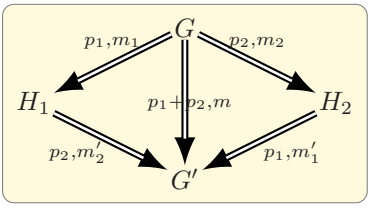
$$n_1(R_1) \cap m_2(L_2) \subseteq n_1(r_1(K_1)) \cap m_2(l_2(K_2)).$$

With this notion of independence, we are able to formulate the Local Church–Rosser and Parallelism Theorem.

Theorem 1 (Local Church–Rosser and Parallelism Theorem). *Given two parallel independent direct (typed) graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$, there is a (typed) graph G' together with direct (typed) graph transformations $H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ and $H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ are sequentially independent.*

Given two sequentially independent direct (typed) graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$, there are a (typed) graph H_2 and direct (typed) graph transformations $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent.

In any case of independence, there is a parallel (typed) graph transformation $G \Rightarrow G'$ via the parallel (typed) graph production $p_1 + p_2$, which is the disjoint union of the (typed) graph productions p_1 and p_2. Vice versa, the parallel (typed) graph transformation $G \Rightarrow G'$ can be sequentialized both ways.



Example 5. We apply the typed graph production *setFlag* twice to the start graph S, first with the match m, and the second time with a different match m'

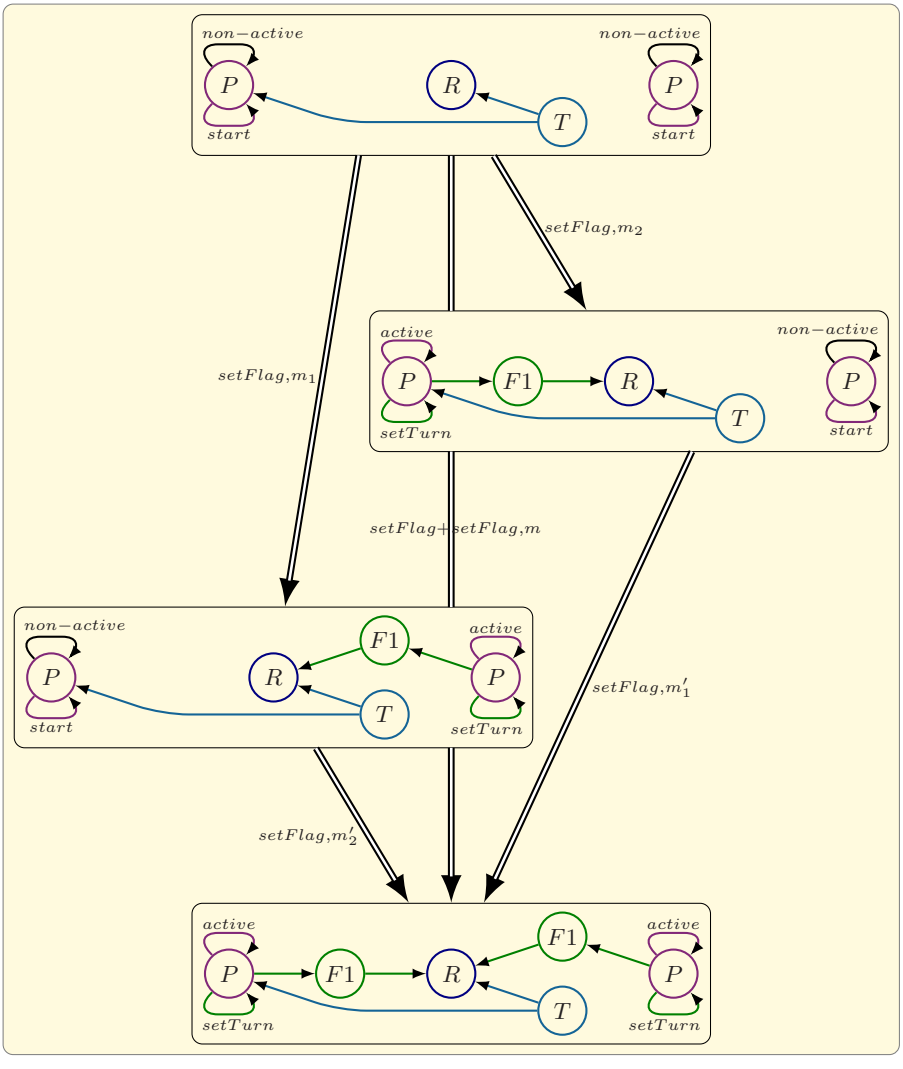


Fig. 10. Example Local Church-Rosser and Parallelism Theorem

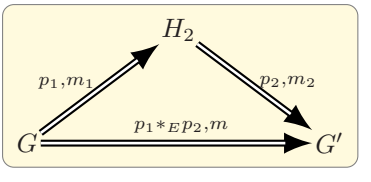
that maps the process node in L to the other process node in S. These two direct typed graph transformations are parallel independent: in the intersection of the matches, there is only the resource node, which is a gluing point with respect to both transformations. Applying the Local Church–Rosser and Parallelism Theorem, we can apply *setFlag* again switching the matches leading to the same typed graph, as well as it is possible to apply $setFlag + setFlag$ directly to S with the same result as shown in Fig. 10. □

Concurrency Theorem

In contrast to the Local Church–Rosser Theorem, the Concurrency Theorem is concerned with the execution of (typed) graph transformations which may be sequentially dependent. This means that, in general, we cannot commute subsequent direct (typed) graph transformations, as done for independent transformations in the Local Church–Rosser Theorem, nor are we able to apply the corresponding productions in parallel, as done in the Parallelism Theorem. Nevertheless, it is possible to apply both transformations concurrently using a so-called E-concurrent (typed) graph production $p_1 *_E p_2$. Given an arbitrary sequence $G \overset{p_1,m_1}{\Longrightarrow} H \overset{p_2,m_2}{\Longrightarrow} G'$ of direct (typed) graph transformations, it is possible to construct an E-concurrent (typed) graph production $p_1 *_E p_2$. The "epimorphic overlap graph" E can be constructed as a subgraph of H from $E = n_1(R_1) \cup m_2(L_2)$, where n_1 and m_2 are the first comatch and the second match, and R_1 and L_2 are the right- and the left-hand side of p_1 and p_2, respectively. Note that the restrictions $e_1 : R_1 \to E$ of n_1 and $e_2 : L_2 \to E$ of m_2 are jointly surjective. The E-concurrent (typed) graph production $p_1 *_E p_2$ allows one to construct a direct (typed) graph transformation $G \overset{p_1 *_E p_2}{\Longrightarrow} G'$ from G to G' via $p_1 *_E p_2$. Vice versa, each direct (typed) graph transformation $G \overset{p_1 *_E p_2}{\Longrightarrow} G'$ via the E-concurrent (typed) graph production $p_1 *_E p_2$ can be sequentialized, leading to an E-related (typed) graph transformation sequence $G \overset{p_1,m_1}{\Longrightarrow} H \overset{p_2,m_2}{\Longrightarrow} G'$ of direct (typed) graph transformations via p_1 and p_2, where "E-related" means that n_1 and m_2 overlap in H as required by E.

Theorem 2 (Concurrency Theorem). *Given two (typed) graph productions p_1 and p_2, and an E-concurrent (typed) graph production $p_1 *_E p_2$, we have:*

- *Given an E-related (typed) graph transformation sequence $G \Rightarrow H \Rightarrow G'$ via p_1 and p_2, then there is a* synthesis construction *leading to a direct (typed) graph transformation $G \Rightarrow G'$ via $p_1 *_E p_2$.*
- *Given a direct (typed) graph transformation $G \Rightarrow G'$ via $p_1 *_E p_2$, then there is an* analysis construction *leading to an E-related (typed) graph transformation sequence $G \Rightarrow H \Rightarrow G'$ via p_1 and p_2.*



Example 6. The first two steps $S \Rightarrow G_1 \Rightarrow G_2$ of the typed graph transformations in Fig. 7 are sequentially dependent, because the *setTurn*-loop needed to apply the typed graph production *setTurn*1 to G_1 is created by *setFlag*. The E-concurrent production for this transformation sequence is shown in the top row of Fig. 11, leading to the depicted E-related typed graph transformation. □

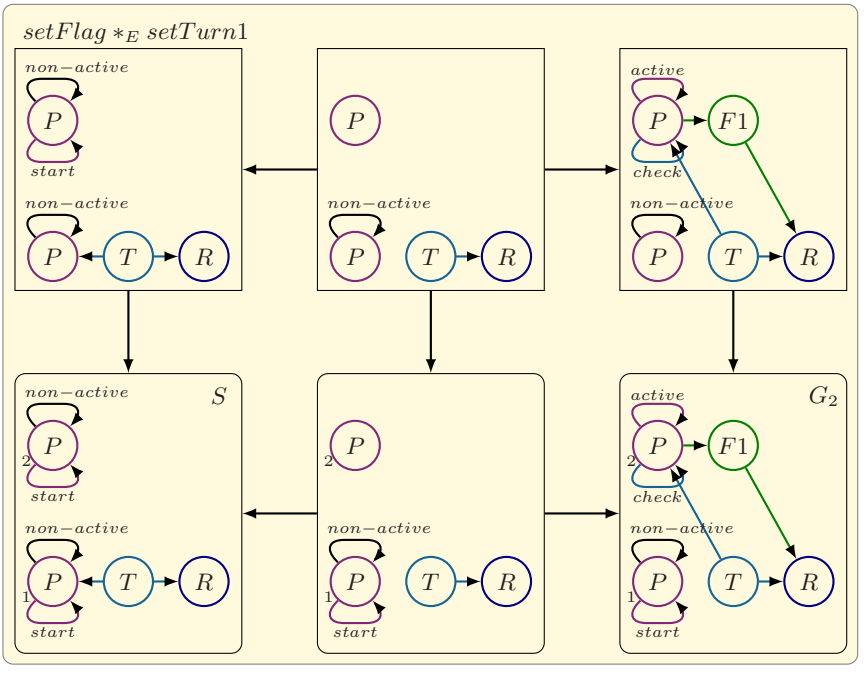
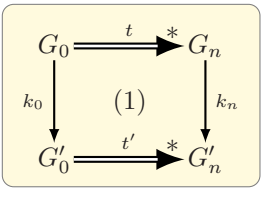


Fig. 11. Example Concurrency Theorem

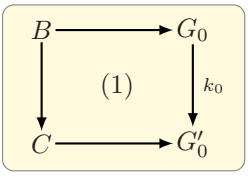
Embedding and Extension Theorem

For the Embedding and Extension Theorem, we analyze under what conditions a (typed) graph transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ can be extended to a (typed) graph transformation $t' : G'_0 \overset{*}{\Rightarrow} G'_n$ via an extension morphism $k_0 : G_0 \to G'_0$. The idea is to obtain an extension diagram (1), where the same (typed) graph productions $p_1, \ldots, p_n$ are applied in the same order in t and t'.



Unfortunately, this is not always possible, but we are able to give a necessary and sufficient consistency condition to allow such an extension. This result is important for all kinds of applications where we have a large (typed) graph G'_0, but only small subparts of G'_0 have to be changed by the (typed) graph productions $p_1, \ldots, p_n$. In this case we choose a suitably small subgraph G_0 of G'_0 and construct a (typed) graph transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ via $p_1, \ldots, p_n$ first. In a second step, we extend $t : G_0 \overset{*}{\Rightarrow} G_n$ via the inclusion $k_0 : G_0 \to G'_0$ to a (typed) graph transformation $t' : G'_0 \overset{*}{\Rightarrow} G'_n$ via the same (typed) graph productions $p_1, \ldots, p_n$.

Now we are going to formulate the consistency condition which allows us to extend $t : G_0 \overset{*}{\Rightarrow} G_n$ to $t' : G'_0 \overset{*}{\Rightarrow} G'_n$ via $k_0 : G_0 \to G'_0$, leading to the extension diagram (1) above. The idea is to first construct a boundary graph B and a context graph C for $k_0 : G_0 \to G'_0$, such that G'_0 is the gluing of G_0 and C along B, i.e. $G'_0 = G_0 +_B C$. In fact, this boundary graph B is the smallest subgraph of G_0 which contains the identification points IP and the dangling points DP of $k_0 : G_0 \to G'_0$, considered as a match morphism. Now the (typed) graph morphism $k_0 : G_0 \to G'_0$ is said to be consistent with $t : G_0 \overset{*}{\Rightarrow} G_n$ if the boundary graph B is preserved by t. This means that none of the (typed) graph production $p_1, \ldots, p_n$ deletes any item of B.



Theorem 3 (Embedding and Extension Theorem). *Given a (typed) graph transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ and a (typed) graph morphism $k_0 : G_0 \to G'_0$ which is consistent with respect to t, then there is an extension diagram over t and k_0.*

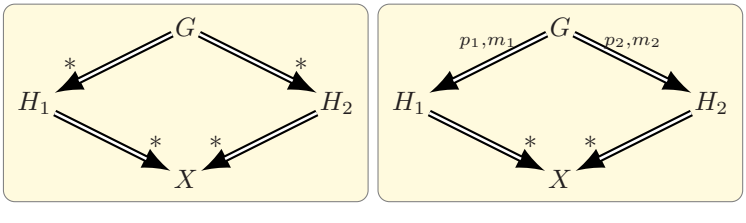
Given a (typed) graph transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ with an extension diagram (1), and the boundary B and the context graph C of $k_0 : G_0 \to G'_0$, then we have:

1. *k_0 is consistent with respect to $t : G_0 \overset{*}{\Rightarrow} G_n$.*
2. *There is a (typed) graph production $der(t) = (G_0 \overset{d_0}{\leftarrow} D_n \overset{d_n}{\rightarrow} G_n)$, called the derived span of $t : G_0 \overset{*}{\Rightarrow} G_n$, leading to a direct (typed) graph transformation $G'_0 \Rightarrow G'_n$ via $der(t)$.*
3. *G'_n is the gluing of C and G_n along B, i.e. $G'_n = G_n +_B C$.*

Example 7. We embed the start graph S, with the typed graph morphism k_0, into a larger context graph H, where an additional resource is available that is also assigned to the first process. The boundary B and context graph C for k_0 are shown in the left-hand side of Fig. 12. Since, in the boundary graph, there is only the first process node, which is preserved by every step of the typed graph transformation $t : S \overset{*}{\Rightarrow} G$, we can extend t over k_0 to H and obtain a typed graph transformation $t' : H \overset{*}{\Rightarrow} H'$ shown in Fig. 12. Note that H' is the gluing of C and G along B. □

Critical Pairs and Local Confluence Theorem

A (typed) graph transformation system is called *confluent* if, for all (typed) graph transformations $G \overset{*}{\Rightarrow} H_1$ and $G \overset{*}{\Rightarrow} H_2$, there is a (typed) graph X together with



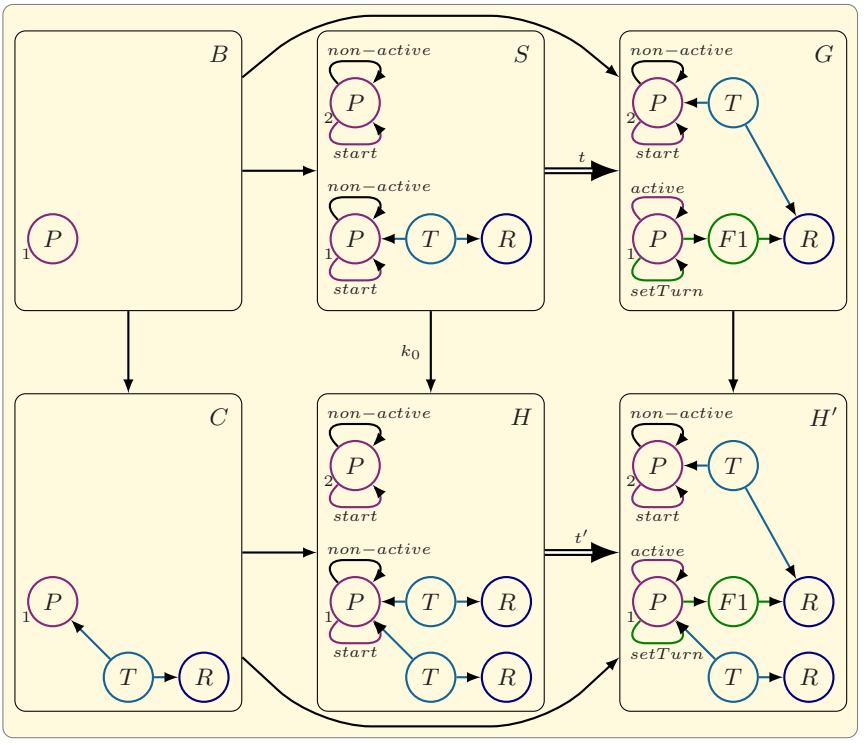


Fig. 12. Example Embedding and Extension Theorem

(typed) graph transformations $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$. *Local confluence* means that this property holds for all pairs of direct (typed) graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$.

Confluence is an important property of a (typed) graph transformation system, because, in spite of local nondeterminism concerning the application of a (typed) graph production, we have global determinism for confluent (typed) graph transformation systems. *Global determinism* means that, for each pair of terminating (typed) graph transformations $G \overset{*}{\Rightarrow} H$ and $G \overset{*}{\Rightarrow} H'$ with the same source graph, the target graphs H and H' are equal or isomorphic. A (typed) graph transformation $G \overset{*}{\Rightarrow} H$ is called *terminating* if no (typed) graph production in the (typed) graph transformation system is applicable to H anymore.

The Local Church–Rosser Theorem shows that, for two parallel independent direct (typed) graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$, there is a (typed) graph G' together with direct (typed) graph transformations $H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ and $H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$. This means that we can apply the (typed) graph productions p_1 and p_2 with given matches in an arbitrary order. If each pair of productions

is parallel independent for all possible matches, then it can be shown that the corresponding (typed) graph transformation system is confluent.

In the following, we discuss local confluence for the general case in which $G \Rightarrow H_1$ and $G \Rightarrow H_2$ are not necessarily parallel independent. According to a general result for rewriting systems, it is sufficient to consider local confluence, provided that the (typed) graph transformation system is terminating.

The main idea is to study critical pairs. A pair $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$ of direct (typed) graph transformations is called a critical pair if it is parallel dependent, and minimal in the sense that the pair (o_1, o_2) of matches $o_1 : L_1 \to K$ and $o_2 : L_2 \to K$ is jointly surjective. This means that each item in K has a preimage in L_1 or L_2. In other words, K can be considered as a suitable gluing of L_1 and L_2. It can be shown that every pair of parallel dependent direct (typed) graph transformations is an extension of a critical pair.

In order to show local confluence, it is sufficient to show strict confluence of all its critical pairs. As discussed above, confluence of a critical pair $P_1 \Leftarrow K \Rightarrow P_2$ means the existence of a (typed) graph K' together with (typed) graph transformations $P_1 \overset{*}{\Rightarrow} K'$ and $P_2 \overset{*}{\Rightarrow} K'$.

Strictness is a technical condition which means, intuitively, that the largest subgraph N of K which is preserved by the critical pair $P_1 \Leftarrow K \Rightarrow P_2$ is also preserved by $P_1 \overset{*}{\Rightarrow} K'$ and $P_2 \overset{*}{\Rightarrow} K'$. In [10], it has been shown that confluence of critical pairs without strictness is not sufficient to show local confluence.

Theorem 4 (Local Confluence Theorem). *A (typed) graph transformation system is locally confluent if all its critical pairs are strictly confluent.*

Example 8. We analyze our typed graph grammar $MutualExclusion$ and take a closer look at the typed graph productions $setFlag$ and $setTurn1$. For a typed graph K that may lead to a critical pair, we have to consider overlappings of the left-hand sides L_1 of $setFlag$ and L_2 of $setTurn1$. The typed graph transformations $K \overset{setFlag}{\Longrightarrow} P_1$ and $K \overset{setTurn1}{\Longrightarrow} P_2$ are parallel dependent if the loop in L_2 typed *non-active* is deleted by $setFlag$. This leads to the two critical overlappings K and K', and we have the critical pairs $P_1 \overset{setFlag}{\Longleftarrow} K \overset{setTurn1}{\Longrightarrow} P_2$ and $P_1' \overset{setFlag}{\Longleftarrow} K' \overset{setTurn1}{\Longrightarrow} P_2'$ shown in Fig. 13.

There are many more critical pairs for other pairs of typed graph transformations in our grammar. All these critical pairs are strictly confluent. Therefore the typed graph transformation system is locally confluent. However, as we can see in the derivation graph, the typed graph grammar is not terminating; nevertheless, it is confluent. □

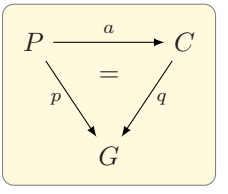
Graph Constraints and Application Conditions

(Typed) graph constraints allow us to formulate properties for (typed) graphs. In particular, we are able to formulate the condition that a (typed) graph G must (or must not) contain a certain subgraph G'. Beyond that, we can require that G contains C (conclusion) if it contains P (premise). Application conditions,

similarly to the gluing condition, allow us to restrict the application of (typed) graph productions. Both concepts are important for increasing the expressive power of (typed) graph transformation systems.

Definition 6 (Graph constraint). *An* atomic (typed) graph constraint *is of the form $PC(a)$, where $a : P \to C$ is a (typed) graph morphism.*

A (typed) graph constraint *is a Boolean formula over atomic (typed) graph constraints. This means that true and every atomic (typed) graph constraint are (typed) graph constraints, and, for (typed) graph constraints c and c_i with $i \in I$ for some index set I, $\neg c$, $\wedge_{i\in I} c_i$, and $\vee_{i\in I} c_i$ are (typed) graph constraints.*



A (typed) graph G satisfies *a (typed) graph constraint c, written $G \models c$, if*

- *$c = true$;*
- *$c = PC(a)$ and, for every injective (typed) graph morphism $p : P \to G$, there exists an injective (typed) graph morphism $q : C \to G$ such that $q \circ a = p$;*
- *$c = \neg c'$ and G does not satisfy c';*
- *$c = \wedge_{i\in I} c_i$ and G satisfies all c_i with $i \in I$;*
- *$c = \vee_{i\in I} c_i$ and G satisfies some c_i with $i \in I$.*

Now we introduce application conditions for a match $m : L \to G$, where L is the left-hand side of a (typed) graph production p. The idea is that the (typed) graph production cannot be applied at m if m violates the application condition.

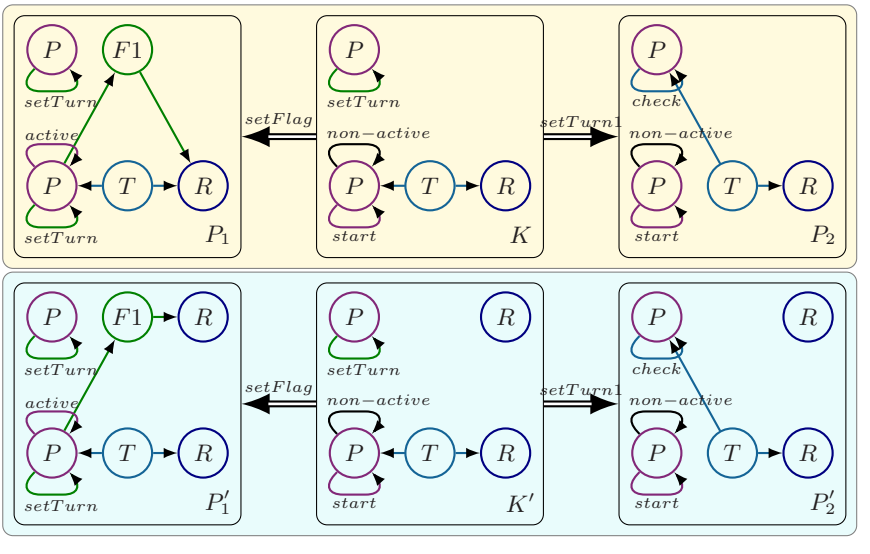
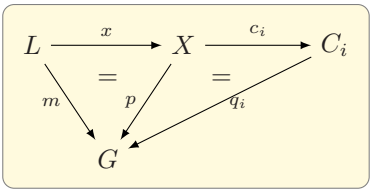


Fig. 13. Example critical pairs

Definition 7 (Application condition). *An* atomic application condition *over a (typed) graph L is of the form $P(x, \vee_{i\in I} x_i)$, where $x : L \to X$ and $x_i : X \to C_i$ with $i \in I$ for some index set I are (typed) graph morphisms.*

An application condition *over L is a Boolean formula over atomic application conditions over L. This means that true and every atomic application condition are application conditions, and, for application conditions acc and acc_i with*



$i \in I$, $\neg acc$, $\wedge_{i\in I} acc_i$, and $\vee_{i\in I} acc_i$ are application conditions.

A (typed) graph morphism $m : L \to G$ satisfies *an application condition acc, written $m \models acc$, if*

- *$acc = true$;*
- *$acc = P(x, \vee_{i\in I} x_i)$ and, for all injective (typed) graph morphisms $p : X \to G$ with $p \circ x = m$, there exists an $i \in I$ and an injective (typed) graph morphism $q_i : C_i \to G$ with $q_i \circ x_i = p$;*
- *$acc = \neg acc'$ and m does not satisfy acc';*
- *$acc = \wedge_{i\in I} acc_i$ and m satisfies all acc_i with $i \in I$;*
- *$acc = \vee_{i\in I} acc_i$ and m satisfies some acc_i with $i \in I$.*

Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, an *application condition* $A(p) = (A_L, A_R)$ for p consists of a left application condition A_L over L and a right application condition A_R over R. A direct (typed) graph transformation $G \overset{p,m}{\Rightarrow} H$ with a comatch $n : R \to H$ satisfies the application condition $A(p) = (A_L, A_R)$ if $m \models A_L$ and $n \models A_R$. Otherwise, p cannot be applied to G via m.

A widely used variant of application conditions are negative application conditions. A *negative application condition* is of the form $NAC(x)$, where $x : L \to X$ is a (typed) graph morphism. A (typed) graph morphism $m : L \to G$ *satisfies* $NAC(x)$ if there *does not* exist an injective (typed) graph morphism $p : X \to G$ with $p \circ x = m$. A negative application condition $NAC(x)$ is equivalent to an application condition of the form $P(x, \vee_{i\in I} x_i)$ with an empty index set I.

Example 9. We consider the typed graph constraint $PC(a : P \to C)$ in Fig. 14 for the typed graphs of the graph grammar $MutualExclusion$. A typed graph G satisfies this constraint if, for each resource node R, there is a turn variable that connects it to a process. The start graph S obviously satisfies this constraint – there is only one resource, which is connected to the first process node.

For an example of an application condition, we add a new production *addResource* to our typed graph grammar $MutualExclusion$, as shown in Fig. 15. This production inserts a new resource node and a new turn node, connected to a given process. For the application of this production, we define the left negative application condition $NAC(x)$ as depicted. With $NAC(x)$, we forbid the possibility that the process that the turn will be connected to is already active. □

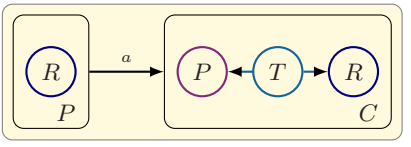


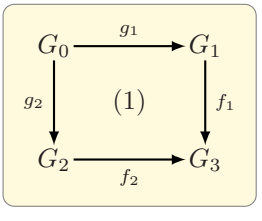
Fig. 14. Example typed graph constraint

It is possible to construct for each (typed) graph constraint an equivalent right application condition and for each right application condition an equivalent left application condition. This allows us to make sure that a derived (typed) graph H satisfies a given (typed) graph constraint $PC(a)$, provided that the match $m : L \to G$ of the direct (typed) graph transformation $G \overset{p,m}{\Longrightarrow} H$ satisfies the corresponding left application condition *acc*.

3 Transformations in Adhesive HLR Systems

In this section, we generalize the basic concepts of the algebraic approach from graphs in Section 2 to high-level structures. The concept of weak adhesive high-level replacement (HLR) categories is introduced as a suitable categorical framework for graph transformation in this more general sense.

In addition to pushouts we also need pullbacks. The intuitive idea of a pullback G_0 of injective morphisms $f_1 : G_1 \to G_3$ and $f_1 : G_2 \to G_3$ is that G_0 is the intersection of G_1 and G_2 with injective morphisms $g_1 : G_0 \to G_1$ and $g_2 : G_0 \to G_2$ leading to the commutative diagram (1).



If f_1 is an inclusion and f_2 an arbitrary morphism then G_0 can be considered as the preimage $f_2^{-1}(G_1)$.

The intuitive idea of weak adhesive HLR categories is that of categories with suitable pushouts and pullbacks which are compatible with each other. More precisely, the definition is based on van Kampen squares.

The idea of a van Kampen (VK) square is that of a pushout which is stable under pullbacks, and, vice versa, that pullbacks are stable under combined pushouts and pullbacks.

Definition 8 (Van Kampen square). *A pushout (1) is a* van Kampen square *if, for any commutative cube (2) with (1) in the bottom and where the back faces are pullbacks, the following statement holds: the top face is a pushout iff the front faces are pullbacks.*

It might be expected that, at least in the category **Sets**, every pushout is a van Kampen square. Unfortunately, this is not true. However, at least pushouts along injective functions or monomorphisms are VK squares in **Sets** and several other categories.

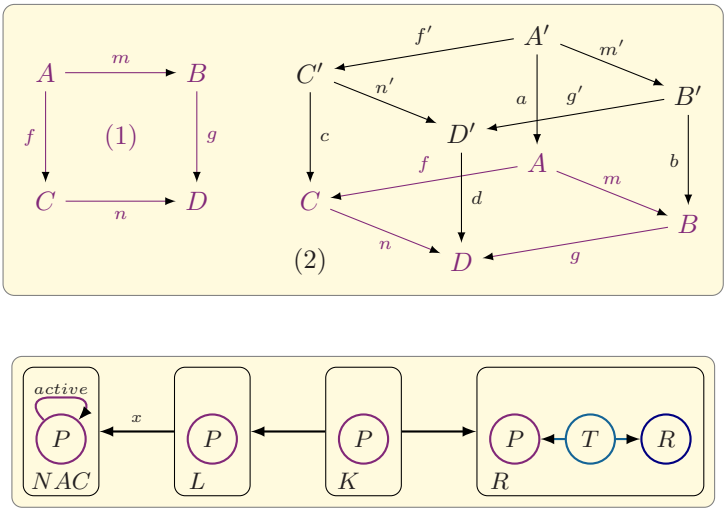


Fig. 15. Example negative application condition

Definition 9 (Weak adhesive HLR category). *A category* $\mathbf{C}$ *with a morphism class* $\mathcal{M}$ *is called a* weak adhesive HLR category *if:*

1. $\mathcal{M}$ *is a class of monomorphisms closed under isomorphisms, composition* $(f : A \to B \in \mathcal{M}, g : B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M})$*, and decomposition* $(g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M})$*.*
2. $\mathbf{C}$ *has pushouts and pullbacks along* $\mathcal{M}$*-morphisms, and* $\mathcal{M}$*-morphisms are closed under pushouts and pullbacks.*
3. *Pushouts in* $\mathbf{C}$ *along* $\mathcal{M}$*-morphisms are weak VK squares, i.e. the VK square property holds for all commutative cubes with* $m \in \mathcal{M}$ *and* ($f \in \mathcal{M}$ *or* $b, c, d \in \mathcal{M}$)*.*

For historical reasons, these categories are called weak adhesive HLR categories. In [11] and related work, adhesive categories are used as the categorical framework for deriving process congruences from reaction rules. The step from adhesive to adhesive HLR categories is justified by the fact that there are some important examples – such as algebraic specifications and typed attributed graphs – which are not adhesive categories. However, they are adhesive HLR categories for a suitable subclass $\mathcal{M}$ of all monomorphisms. Thus, the main difference between adhesive HLR categories and adhesive categories is that a distinguished class $\mathcal{M}$ of monomorphisms is considered instead of all monomorphisms, so that only pushouts along $\mathcal{M}$-morphisms have to be VK squares. Another important example – the category **PTNets** of place/transition nets with the class $\mathcal{M}$ of injective morphisms – fails to be an adhesive HLR category, but is a weak adhesive HLR category. This justifies the step to weak adhesive HLR categories.

Weak adhesive HLR categories are closed under product, slice, coslice, functor, and comma category constructions. This means that we can construct new weak adhesive HLR categories from given ones.

Theorem 5 (Construction Theorem). *If* $(\mathbf{C}, \mathcal{M}_1)$ *and* $(\mathbf{D}, \mathcal{M}_2)$ *are weak adhesive HLR categories, then the following categories are weak adhesive HLR categories:*

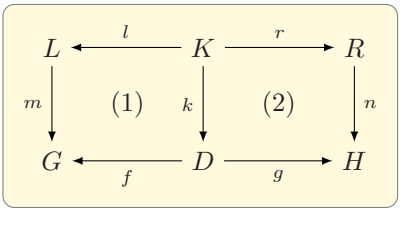
1. *the* product category $(\mathbf{C} \times \mathbf{D}, \mathcal{M}_1 \times \mathcal{M}_2)$,
2. *the* slice category $(\mathbf{C}\backslash X, \mathcal{M}_1 \cap \mathbf{C}\backslash X)$,
3. *the* coslice category $(X\backslash\mathbf{C}, \mathcal{M}_1 \cap X\backslash\mathbf{C})$,
4. *the* functor category $([\mathbf{X}, \mathbf{C}], \mathcal{M} - functor\ transformations)$,
5. *the* comma category $(ComCat(F, G; \mathcal{I}), (\mathcal{M}_1 \times \mathcal{M}_2) \cap Mor_{ComCat})$, *where* $F : \mathbf{C} \to \mathbf{X}$ *preserves pushouts along* $\mathcal{M}_1$*-morphisms and* $G : \mathbf{D} \to \mathbf{X}$ *preserves pullbacks (along* $\mathcal{M}_2$*-morphisms).*

Examples for weak adhesive HLR categories are the categories **Sets** of sets, **Graphs** of graphs, $\mathbf{Graphs_{TG}}$ of typed graphs, **Hypergraphs** of hypergraphs, **ElemNets** of elementary Petri nets, **PTNets** of place/transition nets and **AHLNets** of algebraic high-level nets, all together with the class $\mathcal{M}$ of injective morphisms, as well as the category **Spec** of algebraic specifications with the class $\mathcal{M}_{strict}$ of strict injective specification morphisms and the category $\mathbf{AGraphs_{ATG}}$ of typed attributed graph with the class $\mathcal{M}_{D-iso}$ of injective graph morphisms with isomorphic data part. After proving that **Sets** is a weak adhesive HLR category, the proofs for most of these categories can be done using the Construction Theorem.

Analogously to the (typed) graph case, we can define productions, transformations, and adhesive HLR systems and grammars, where we replace injective morphisms by $\mathcal{M}$-morphisms.

Definition 10 (Adhesive HLR system and grammar). *A* production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of objects* L, K, *and* R, *and two morphisms* l *and* r *with* $l, r \in \mathcal{M}$.

Given a production p, *an object* G, *and a morphism* $m : L \to G$, *called match, a* direct transformation $G \overset{p,m}{\Longrightarrow} H$ *from* G *to an object* H *is given by the pushouts (1) and (2).*



An adhesive HLR system $AS = (\mathbf{C}, \mathcal{M}, P)$ *consists of a weak adhesive HLR category* $(\mathbf{C}, \mathcal{M})$ *and a set of productions* P.

An adhesive HLR grammar $AG = (AS, S)$ *consists of an adhesive HLR system* AS *and a start object* S.

The language L *of an adhesive HLR grammar* AG *is defined by*

$$L = \{G \mid \exists\ transformation\ S \overset{*}{\Rightarrow} G\}.$$

Under a few additional conditions, it has been shown in [7] that all the results for (typed) graph transformations given in Subsection 2.2 are valid in adhesive HLR systems. Hence they can be applied to all the examples of weak adhesive HLR categories discussed above.

4 Conclusion

In this paper, we have given an overview of several concepts and results of algebraic graph transformation based on gluing constructions and the double pushout approach. Basic results concerning independence, parallelism, concurrency, embedding, critical pairs and confluence have been introduced and explained by examples.

As a generalization, we have defined the categorical framework of adhesive high-level replacement systems for unified constructions and proofs, which allows to instantiate the rich theory not only to graphs and typed graphs, but also to many different high-level structures. As a consequence we obtain a rigorous approach to various transformation systems providing as fundamental results the Local Church-Rosser and Parallelism, Concurrency, Embedding and Extension, and the Local Confluence Theorems.

For a detailed presentation of all the concepts, results and proofs we refer to our book [7].

References

[1] Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, vol. 1. World Scientific, Singapore (1997)
[2] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
[3] Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Concurrency, Parallelism and Distribution, vol. 3. World Scientific, Singapore (1999)
[4] Ehrig, H., Habel, A., Kreowski, H.-J., Parisi-Presicce, F.: From Graph Grammars to High Level Replacement Systems. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars and Their Application to Computer Science. LNCS, vol. 532, pp. 269–291. Springer, Heidelberg (1991)
[5] Ehrig, H., Habel, A., Kreowski, H.-J., Parisi-Presicce, F.: Parallelism and Concurrency in High-Level Replacement Systems. MSCS 1(3), 361–404 (1991)
[6] Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive High-Level Replacement Categories and Systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 144–160. Springer, Heidelberg (2004)
[7] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
[8] Lack, S., Sobociński, P.: Adhesive Categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)

[9] Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Mateo, CA (1996)
[10] Plump, D.: On Termination of Graph Rewriting. In: Nagl, M. (ed.) WG 1995. LNCS, vol. 1017, pp. 88–100. Springer, Heidelberg (1995)
[11] Sobociński, P.: Deriving Process Congruences from Reaction Rules. PhD thesis, BRICS (2004)