Some Essentials of Graph Transformation*

Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
{kreo,rena,kuske}@informatik.uni-bremen.de

Summary. This chapter introduces rule-based graph transformation, which constitutes a well-studied research area in computer science. The chapter presents the most fundamental definitions and illustrates them with some selected examples. It presents also the concept of transformation units, which makes pure graph transformation more feasible for specification and modeling aspects. Moreover, a translation of Chomsky grammars into graph grammars is given and the main theorems concerning parallelism and concurrency are presented. Finally, an introduction to hyperedge replacement is given, a concept which has nice properties because it transforms hypergraphs in a context-free way.

1 Introduction

Graphs are a well-established means in computer science for representing data structures, states of concurrent and distributed systems, or more generally sets of objects with relations between them. Famous examples of graphs are Petri nets, flow diagrams, Entity-Relationship diagrams, finite automata, and UML diagrams.

In many situations one does not only want to employ graphs as a static structure, but also to transform them e.g. by firing transitions in the case of Petri nets or UML state diagrams, or by generating or deleting objects and links in the case of UML object diagrams. The area of graph transformation brings together the concepts of graphs and rules with various methods from the theory of formal languages and from the theory of concurrency, and with a spectrum of applications, see the three volumes of the Handbook of Graph

* Research partially supported by the EC Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modeling Techniques) and the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

Z. Esik and C. Martin-Vide (Eds.): Recent Advances in Formal Languages and Applications, Vol. 25 of Studies in Computational Intelligence, pp. 229-254, 2006.

Grammars and Computing by Graph Transformation as an overview [Roz97, EEKR99, EKMR99].

In this chapter, we give a survey of some essentials of graph transformation including

- a translation of Chomsky grammars into graph grammars in Section 5 showing the computational completeness of graph transformation,
- the basic notions and results on parallelism and concurrency in graph transformation in Section 6, and
- a context-free model of graph transformation in Section 7.

Unfortunately, graphs are quite generic structures that can be encountered in many variants in the literature, and there are also many ways to apply rules to graphs. One cannot deal with all possibilities in an introductory survey. Therefore, we focus on directed, edge-labeled graphs (Section 2) and on rule application in the sense of the so-called double-pushout approach (Section 3). The directed, edge-labeled graphs can be specialized into many other types of graphs. And the double-pushout approach (which is introduced here by means of set-theoretic constructions on graphs without reference to categorical concepts) is one of the most frequently used approaches. In Section 4, we define graph grammars as a language-generating device and the more general notion of a transformation unit that models binary relations on graphs.

2 Graphs and the Need to Transform Them

Graphs are well-suited and frequently-used structures to represent complex relations between objects of various kinds. They are the central structures of interest in at least four areas of mathematics and computer science: graph theory (see, e.g., Harary [Har69]), graph algorithms (see, e.g., [Gib85]), Petri nets (see, e.g., [Rei85, GV03]), and graph transformation (see, e.g., the Handbooks on Graph Grammars and Computing by Graph Transformation [Roz97, EEKR99, EKMR99]). But they are also popular and useful in many other disciplines like biology, chemistry, economics, logistics, engineering, and many others.

Maps are typical examples of structures that are often represented by graphs. Already in 1736, Euler formulated the *Königsberger Brückenproblem* concerning the map of Königsberg, which consists of four areas that are separated from each other by the two arms of the river Pregel. There are seven bridges connecting two areas each, and the question is whether one can walk around passing each bridge exactly once. This becomes a graph problem if the areas are considered as nodes and the bridges as edges between the corresponding nodes. A sketch of the map and the respective graph are shown in the left side of Figure 1. For such graphs, the general question (known as the Eulerian Cycle Problem) is whether there is a cycle passing each edge exactly once. Similarly, maps of countries can be represented as graphs by considering

the countries as nodes and by connecting each two nodes with an edge the corresponding countries of which share a borderline. In this way the famous Four-Color-Problem of maps becomes the Four-Color-Problem of graphs (see, e.g., [Gib85, AH89]). Finally, road maps are nicely represented as graphs by considering sites as nodes and a road that connects two sites directly as an edge that may be labeled with the distance. Such graphs are the basic data structures for various transportation and tour planning problems.

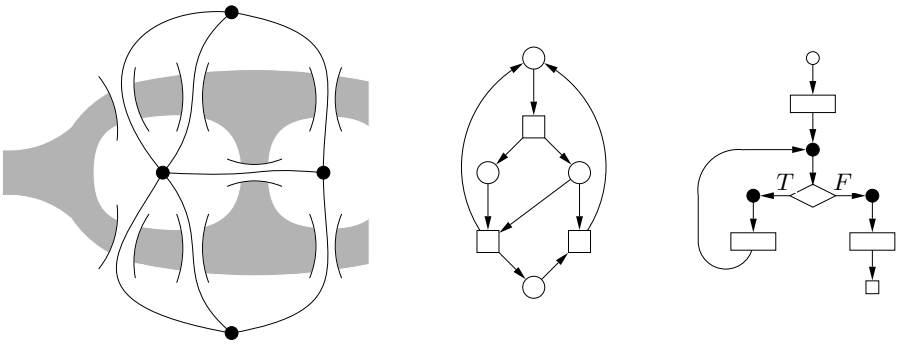

Fig. 1. Various graphs

Another typical example of graphs are Petri nets, which allow one to model concurrent and distributed systems (see, e.g., [Rei98]). A Petri net is a simple bipartite graph meaning that there are two types of nodes, called conditions and events or places and transitions, and a set of edges, called flow relation, which connect nodes of distinct types only. The middle graph of Figure 1 shows a sample Petri net; as usual, round nodes represent conditions or places, and square nodes represent events or transitions.

A further example of graphs are well-structured flow diagrams such as the right graph in Figure 1. Such a graph has an entry node (the circle) and an exit node (the square), boxes representing statements, rhombs representing tests, and auxiliary nodes in between each two linked boxes or box and rhomb. The edges represent the control flow. No edge leaves the exit. Each rhomb is left by two edges representing the test results *TRUE* and *FALSE*, respectively. Each other node is left by a single edge. Each rhomb is the test of a *while*-loop meaning that the *TRUE*-edge starts a path that ends at the node immediately before the rhomb. Like well-structured flow diagrams, many other kinds of diagrams including all the UML diagrams may be represented by and seen as graphs.

Graphs are quite generic structures which can be encountered in the literature in many variants: directed and undirected, labeled and unlabeled, simple and multiple, with binary edges and hyperedges, etc. In this survey, we focus on directed, edge-labeled, and multiple graphs with binary edges.

2.1 Graphs

Let Σ be a set of labels. A (multiple directed edge-labeled) *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a finite set of *nodes*, E is a finite set of *edges*, $s, t\colon E \to V$ are mappings assigning a *source* $s(e)$ and a *target* $t(e)$ to every edge in E, and $l\colon E \to \Sigma$ is a mapping assigning a label to every edge in E. An edge e with $s(e) = t(e)$ is also called a *loop*. The components V, E, s, t, and l of G are also denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs over Σ is denoted by $\mathcal{G}_\Sigma$.

The notion of multiple directed edge-labeled graphs with binary edges is flexible enough to cover other types of graphs. *Simple graphs* form a subclass consisting of those graphs two edges of which are equal if their sources and their targets are equal respectively. A label of a loop can be interpreted as a label of the node to which the loop is attached so that *node-labeled graphs* are covered. On the other hand, we assume a particular label $*$ which is omitted in drawings of graphs. In this way, graphs where all edges are labeled with $*$ may be seen as *unlabeled graphs*. Moreover, undirected graphs can be represented by directed graphs if one replaces each undirected edge by two directed edges attached to the same two nodes, but in opposite directions. Finally, even *hypergraphs* can be handled by the introduced type of graphs as done explicitly in Section 7.

If graphs are the structures of interest, it is rarely the case that just a single static graph is considered. Rather, graphs may be the inputs of algorithms and processes, so that means are needed to search and manipulate graphs. Graphs may represent states of systems, so that means for updates and state transitions are needed. Or graph languages are in the center of consideration like the set of all well-structured flow diagrams or all Petri nets or all connected and planar graphs. Like in the case of string languages, one needs means to generate and recognize graph languages. To meet all these needs, rule-based graph transformation is defined in the next section. This requires some prerequisites to deal with graphs, which are introduced in the following.

2.2 Subgraphs

A graph $G \in \mathcal{G}_\Sigma$ is a *subgraph* of a graph $H \in \mathcal{G}_\Sigma$, denoted by $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$, $s_G(e) = s_H(e)$, $t_G(e) = t_H(e)$, and $l_G(e) = l_H(e)$ for all $e \in E_G$. In drawings of graphs and subgraphs, shapes, colors, and names will be used to indicate the identical nodes and edges.

Given a graph, a subgraph is obtained by removing some nodes and edges subject to the condition that the removal of a node is accompanied by the removal of all its incident edges. More formally, let $G = (V, E, s, t, l)$ be a graph and $X = (V_X, E_X) \subseteq (V, E)$ be a pair of sets of nodes and edges. Then $G - X = (V - V_X, E - E_X, s', t', l')$ with $s'(e) = s(e)$, $t'(e) = t(e)$, and $l'(e) = l(e)$ for all $e \in E - E_X$ is a subgraph of G if and only there is no

$e \in E - E_X$ with $s(e) \in V_X$ or $t(e) \in V_X$. This condition is called *contact condition* of X in G.

In other words, two subsets of nodes and edges $Y = (V_Y, E_Y) \subseteq (V, E)$ induce a subgraph $Y^\bullet = (V_Y, E_Y, s', t', l') \subseteq G$ with $s'(e) = s(e), t'(e) = t(e)$, and $l'(e) = l(e)$ for all $e \in E_Y$ if and only if $(V - V_Y, E - E_Y)$ satisfies the contact condition in G, i.e. there is no edge $e \in E_Y$ with $s(e) \in V - V_Y$ or $t(e) \in V - V_Y$.

2.3 Graph Morphisms

For graphs $G, H \in \mathcal{G}_\Sigma$ a *graph morphism* $g\colon G \to H$ is a pair of mappings $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that are structure-preserving, i.e. $g_V(s_G(e)) = s_H(g_E(e))$, $g_V(t_G(e)) = t_H(g_E(e))$, and $l_H(g_E(e)) = l_G(e)$ for all $e \in E_G$. We will usually write $g(v)$ and $g(e)$ for nodes $v \in V_G$ and edges $e \in E_G$ since the indices V and E can be reconstructed easily from the type of the argument.

For a graph morphism $g\colon G \to H$ the image of G in H is called a *match* of G in H, i.e. the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$ which is induced by $(g(V), g(E))$. The corresponding contact condition is satisfied because g preserves the structure of graphs.

Given $F \subseteq G$, then the two inclusions of the sets of nodes and edges define a graph morphism. It is also easy to see that the (componentwise) sequential composition of two graph morphisms $f\colon F \to G$ and $g\colon G \to H$ yields a graph morphism $g \circ f\colon F \to H$. Consequently, if f is the inclusion w.r.t. $F \subseteq G$, $g(F)$ is the match of F in H w.r.t. g restricted to F.

2.4 Extension of Graphs

Instead of removing nodes and edges, one may add some nodes and edges to extend a graph such that the given graph is a subgraph of the extension. The addition of nodes causes no problem at all, whereas the addition of edges requires the specification of their labels, sources, and targets, where the latter two may be given or new nodes. Let $G = (V, E, s, t, l)$ be a graph and (V', E', s', t', l') be a structure consisting of two sets V' and E' and three mappings $s'\colon E' \to V \uplus V'$, $t'\colon E' \to V \uplus V'$, and $l'\colon E' \to \Sigma$ (where $\uplus$ denotes the disjoint union of sets). Then $H = G + (V', E', s', t', l') = (V \uplus V', E \uplus E', s'', t'', l'')$ is a graph with $G \subseteq H$ (which establishes the definition of the three mappings s'', t'', l'' on E) and $s''(e') = s'(e')$, $t''(e') = t'(e')$, and $l''(e') = l'(e')$ for all $e' \in E'$.

2.5 Disjoint Union

If G is extended by a full graph $G' = (V', E', s', t', l')$, the graph $G + G'$ is the *disjoint union* of G and G'. Note that in this case s' and t' map E' to V' rather than $V \uplus V'$, but V' is included in $V \uplus V'$ such that the extension works.

The disjoint union of graphs puts graphs together without any interconnection. If graphs are disjoint, their disjoint union is just the union. If they are not disjoint, the shared nodes and edges must be made different from each other. Because this part of the construction is not made explicit, the disjoint union is only unique up to isomorphism, i.e. up to naming. Nevertheless, the disjoint union of graphs has got some useful properties. It is associative and commutative. If $G_1 + G_2$ is the disjoint union of G_1 and G_2, there are two inclusions $incl_i \colon G_i \to G_1 + G_2$. And whenever one has two graph morphisms $g_i \colon G_i \to G$ into some graph G, there exists a unique graph morphism $g \colon G_1 + G_2 \to G$ with $g \circ incl_i = g_i$ for $i = 1, 2$. This property is the categorical characterization of the disjoint union up to isomorphism.

3 Rule-Based Transformation of Graphs

Graph transformation is a rule-based method that performs local changes on graphs. With graph transformation rules it is possible to specify formally and visually for instance the semantics of rule-based systems (like the firing of transitions in Petri nets or in state charts, or the semantics of functional languages), specific graph languages (like the set of all well-formed flow graphs), graph algorithms (like the search of all Eulerian cycles in a graph), and many more.

3.1 Graph Transformation Rule

The idea of a graph transformation rule is to express which part of a graph is to be replaced by another graph. Unlike strings, a subgraph to be replaced can be linked in many ways (i.e., by many edges) with the surrounding graph. Consequently, a rule also has to specify which kind of links are allowed; this is done with the help of a third graph that is common to the replaced and the replacing graph and requires that the surrounding graph may be linked to the replaced graph only with edges incident to this third graph.

Formally, a *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathcal{G}_\Sigma$ such that K is a subgraph of L and R. The components L, K, and R of r are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively.

Example 1 (flow diagrams). Figure 2 shows two rules representing the replacement of a single statement in a flow diagram by a more complex instruction: the statement is replaced by two consecutive statements with $r_{compound}$, and by a while-loop with $r_{while\text{-}do}$. For both rules, the gluing graph consists of two nodes that can be located in the respective left- and right-hand sides by their shape and color.

Example 2 (shortest paths). Figure 3 shows the two essential rules for the computation of shortest paths in distance graphs, that is graphs labeled with

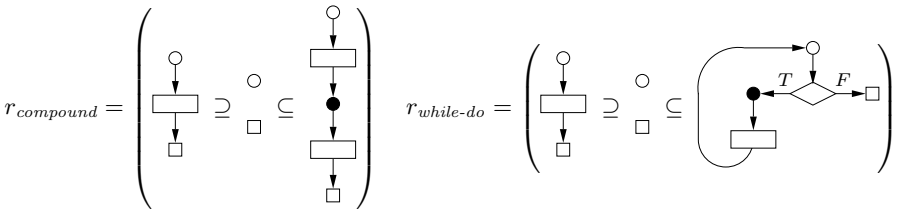

Fig. 2. Graph transformation rules for the construction of flow diagrams

non-negative integers. The first rule adds a direct connection between each two nodes that are connected by a path of length 2 and sums the distances up. Using this rule, one can compute the transitive closure of the given distance graph. If one applies the second rule, which chooses the shortest connection of two direct connections as long as possible, one ends up with shortest connections between each two nodes.

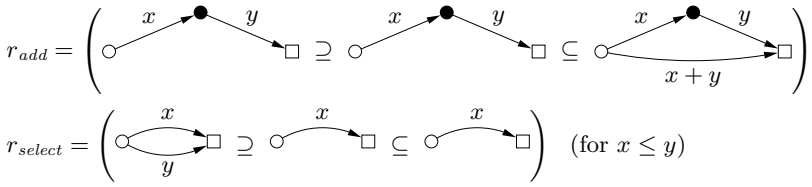

Fig. 3. Graph transformation rules for the computation of shortest paths

In practice one often has the special case where the gluing graph of a rule $r = (L \supseteq K \subseteq R)$ is a set of nodes. In this case the graphical representation of r may omit the gluing graph K by depicting only the graphs L and R, with numbers uniquely identifying the nodes in K. As an example, the rule $r_{while\text{-}do}$ from Figure 2 is drawn in Figure 4 using this alternative representation.

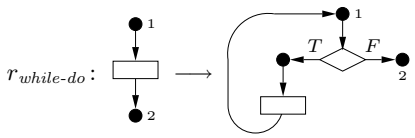

Fig. 4. Alternative representation of the rule $r_{while\text{-}do}$

3.2 Application of a Graph Transformation Rule

The application of a graph transformation rule to a graph G consists of replacing a match of the left-hand side in G by the right-hand side such that the match of the gluing graph is kept. Hence, the application of $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ comprises the following three steps.

1. A graph morphism $g\colon L \to G$ is chosen to establish a match of L in G subject to the following two *application conditions*:
   a) *Contact condition* of $g(L) - g(K) = (g(V_L) - g(V_K), g(E_L) - g(E_K))$ in G; and
   b) *Identification condition.* If two nodes or edges of L are identified in the match of L they must be in K.
2. Now the match of L up to $g(K)$ is removed from G, resulting in a new intermediate graph $Z = G - (g(L) - g(K))$.
3. Afterwards the right-hand side R is added to Z by gluing Z with R in $g(K)$ yielding the graph $H = Z + (R - K, g)$ where $(R - K, g) = (V_R - V_K, E_R - E_K, s', t', l')$ with $s'(e') = s_R(e')$ if $s_R(e') \in V_R - V_K$ and $s'(e') = g(s_R(e'))$ otherwise, $t'(e') = t_R(e')$ if $t_R(e') \in V_R - V_K$ and $t'(e') = g(t_R(e'))$ otherwise, and $l'(e') = l_R(e')$ for all $e' \in E_R - E_K$.

The contact condition guarantees that the removal of $g(L) - g(K)$, yields a subgraph of G. The identification condition is not needed for the construction of a direct derivation, but will be helpful in dealing with parallel rules as considered in Section 6. The extension of Z to H is properly defined because s' and t' map the edges of $E_R - E_K$ into nodes of $V_R - V_K$ or $g(V_K)$ which is part of V_Z.

Example 3 (flow diagrams). Figure 5 shows an application of the rule $r_{while\text{-}do}$ to a flow graph representing a sequence of three statements. The gray areas indicate the match (left), its parts belonging to the image of the gluing graph (middle), and the right-hand side (right).

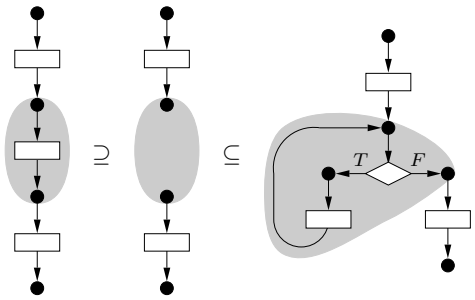

Fig. 5. An application of the rule $r_{while\text{-}do}$

A rule application of $r = (L \supseteq K \subseteq R)$ can be depicted by the following diagram where the graph morphisms $d\colon K \to Z$ and $h\colon R \to H$ are given by $d(v) = g(v)$ for all $v \in V_K$, $d(e) = g(e)$ for all $e \in E_K$, $h(v) = d(v)$ if $v \in V_K$, $h(v) = v$ if $v \in V_R - V_K$, $h(e) = d(e)$ if $e \in E_K$, and $h(e) = e$ if $e \in E_R - E_K$.

$$\begin{array}{ccccc} L & \supseteq & K & \subseteq & R \\ \downarrow g & & \downarrow d & & \downarrow h \\ G & \supseteq & Z & \subseteq & H \end{array}$$

It is worth noting that if the subgraph relations in the diagram are interpreted as inclusion morphisms, both squares of the diagram are pushouts in the category of graphs. This is why the presented approach is also called double-pushout approach (cf. [CEH+97]). Here the identification condition is significant because the left diagram is not a pushout if g does not obey the identification condition.

3.3 Derivation and Application Sequence

The application of a rule r to a graph G is denoted by $G \underset{r}{\Longrightarrow} H$ where H is a graph resulting from an application of r to G. A rule application is called a *direct derivation*, and the iteration of direct derivations $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ $(n \in \mathbb{N})$ is called a *derivation* from G_0 to G_n. As usual, the derivation from G_0 to G_n can also be denoted by $G_0 \overset{n}{\underset{P}{\Longrightarrow}} G_n$ where $\{r_1, \ldots, r_n\} \subseteq P$, or by $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$ if the number of direct derivations is not of interest.

The string $r_1 \cdots r_n$ is called an *application sequence* of the derivation $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$.

Example 4 (flow diagrams). Figure 6 contains a derivation using the rules from Figure 2. Its last direct derivation is the one detailed in Figure 5, and the application sequence of the whole derivation is $r_{compound}\, r_{compound}\, r_{while\text{-}do}$.

In the literature one encounters various approaches to graph transformation, among them specific ones, like edge replacement [DHK97] or node replacement [ER97], and general ones, like the double-pushout approach [CEH+97], the single-pushout approach [EHK+97], or the PROGRES approach [Sch97].

4 Graph Grammars and Graph Transformation Units

Analogously to Chomsky grammars in formal language theory, graph transformation can be used to generate graph languages. A graph grammar consists

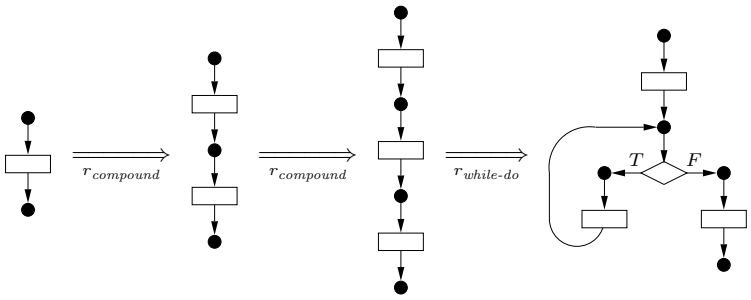

Fig. 6. A derivation of a flow diagram

of a set of rules, a start graph, and a terminal expression fixing the set of terminal graphs. Such a terminal expression may consist of a set $\Delta \subseteq \Sigma$ of terminal labels admitting all graphs that are labeled over Δ.

4.1 Graph Grammar

A *graph grammar* is a system $GG = (S, P, \Delta)$ where $S \in \mathcal{G}_\Sigma$ is the *initial graph* of GG, P is a finite set of graph transformation rules, and $\Delta \subseteq \Sigma$ is a set of *terminal symbols*. The *generated language* of GG consists of all graphs $G \in \mathcal{G}_\Sigma$ that are labeled over Δ and that are derivable from the initial graph S via successive application of the rules in P, i.e. $L(GG) = \{G \in \mathcal{G}_\Delta \mid S \overset{*}{\underset{P}{\Longrightarrow}} G\}$.

Example 5 (connected graphs). As an example of a graph grammar consider *connected* $= (\bullet, P, \{*\})$ where the start graph consists of a single node and the terminal expression allows all graphs labeled only with $*$. Recall that the symbol $*$ denotes a special label in Σ standing for *unlabeled* and being invisible in displayed graphs. The rules in $P = \{p_1, p_2, p_3\}$ are depicted in Figure 7. The rule p_1 adds a node v and an edge e such that v is the target of e, and takes as source of e an already existing node. The rule p_2 is similar, the only difference being that the direction of the new edge e is inverted. The third rule p_3 generates a new edge between two existing nodes. The new edge can also be a loop if the two nodes in the left-hand side of p_3 are identified, i.e. if they are one and the same node in the match of the left-hand side. It can be shown that the generated language of *connected*, $L(\textit{connected})$, consists of all non-empty connected unlabeled graphs.

Example 6 (Petri nets). A place/transition system (N, m_0) consists of a Petri net $N = (S, T, F)$ with a set of places S, a set of transitions T, and a flow relation $F \subseteq (S \times T) \cup (T \times S)$, and an initial marking $m_0 \colon S \to \mathbb{N}$. To model the firing of transitions by graph transformation in the introduced way, one may represent a net $N = (S, T, F)$ with a marking $m \colon S \to \mathbb{N}$ by the graph $G(N, m)$ as indicated in Figure 8 where all tokens become nodes. Moreover,

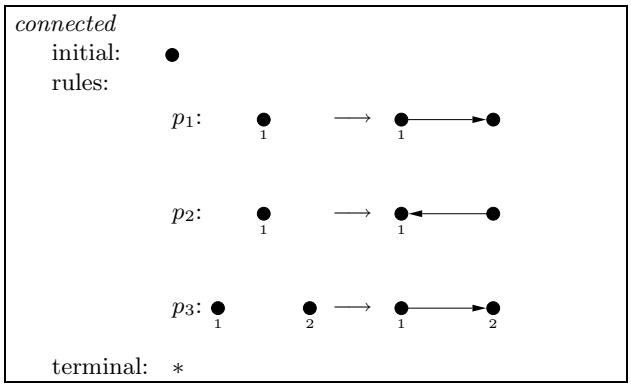

Fig. 7. A graph grammar generating connected graphs

all places and transitions of N are labeled with their respective names (i.e., a loop – that is not drawn here – carrying the respective label is attached).

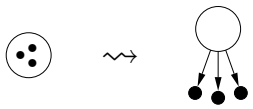

Fig. 8. Turning tokens into nodes

Then we can transform a place/transition system (N, m_0) into a graph grammar $GG(N, m_0) = (G(N, m_0), P(N), P \cup T \cup \{*\})$ where $P(N)$ contains a rule $r(t)$ for each transition t as depicted in Figure 9. Note that the labels attached to the nodes are needed to make sure that distinct pre- or postplaces cannot be identified in a match.

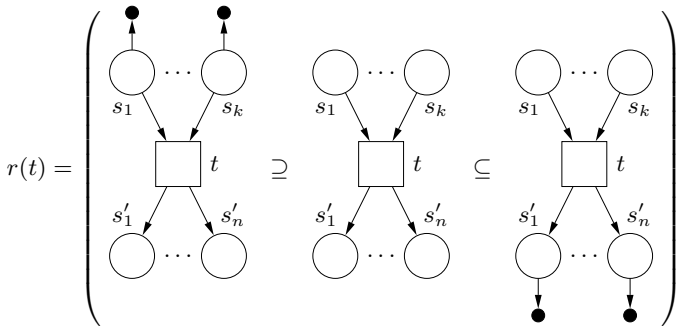

Fig. 9. Firing rule for a transition t with ${}^\bullet t = \{s_1, \ldots, s_k\}$ and $t^\bullet = \{s'_1, \ldots, s'_n\}$

It is not difficult to show that a marking m is reachable from m_0 if and only if there is a derivation $G(N, m_0) \stackrel{*}{\Longrightarrow} G(N, m)$. Thus, $L(GG(N, m_0))$ consists of all net representations $G(N, m)$ where m is a marking reachable from m_0 in N.

One may also model Petri nets with weights assigned to the edges of the flow relation in this way. The left-hand side of the firing rule for a transition t has as many token nodes attached to each preplace s of t as the weight assigned to the edge (s, t) of the flow relation specifies, and analogously for the right-hand side and the postplaces. Then the identification condition ensures that token nodes cannot be identified in a match, so that the application of such a rule works correctly.

As for formal string languages, one does not only want to generate languages, but also to recognize them or to verify certain properties. Moreover, for modeling and specification aspects one wants to have additional features like the possibility to cut down the non-determinism inherent in rule-based graph transformation. This can be achieved with the concept of transformation units (see, e.g., [AEH$^+$99, KK99a, KK99b]), which generalize graph grammars in the following aspects:

- Transformation units allow a set of initial graphs instead of a single one.
- The class of terminal graphs can be specified in a more general way.
- The derivation process can be controlled.
- Transformation units provide a structuring concept for graph transformation.
- Transformation units do not depend on a specific graph transformation approach.

The first two points are achieved by replacing the initial graph and the terminal alphabet of a graph grammar by a graph class expression specifying sets of initial and terminal graphs. The regulation of rule application is obtained by means of so-called control conditions.

4.2 Graph Class Expressions

A *graph class expression* may be any syntactic entity X that specifies a class of graphs $SEM(X) \subseteq \mathcal{G}_\Sigma$. A typical example is the above-mentioned subset $\Delta \subseteq \Sigma$ with $SEM(\Delta) = \mathcal{G}_\Delta \subseteq \mathcal{G}_\Sigma$. Similarly, the expression *all edges* x for $x \in \Sigma$ specifies the class of all graphs G with $l_G(e) = x$ for every $e \in E_G$, i.e. all edges are labeled with x. Another useful type of graph class expressions is given by sets of rules. More precisely, for a set P of rules $SEM(P)$ contains all *P-reduced* graphs, i.e. graphs to which none of the rules in P can be applied. Finally, it is worth noting that a graph grammar GG itself may serve as a graph class expression with $SEM(GG) = L(GG)$.

4.3 Control Conditions

A *control condition* may be any syntactic entity that cuts down the derivation process. A typical example is a regular expression over a set of rules (or any other string-language-defining device). Let C be a regular expression specifying the language $L(C)$. Then a derivation with application sequence s is *permitted* by C if $s \in L(C)$. As a special case of this type of control condition, the condition *true* allows every application sequence, i.e. $L(C) = P^*$ where P is the set of underlying graph transformation rules. Another useful control condition is *as long as possible*, which requires that all rules be applied as long as possible. More precisely, let P be the set of underlying rules. Then $SEM(\textit{as long as possible})$ allows all derivations $G \underset{P}{\Longrightarrow} G'$ such that no rule of P is applicable to G'. Hence, this control condition is similar to the graph class expression P introduced above. Also similar to *as long as possible* are *priorities* on rules being partial orders on rules such that if $p_1 > p_2$, p_1 must be applied as long as possible before any application of p_2. More details on control conditions for transformation units can be found in [Kus00a].

Now we have collected all components for defining unstructured transformation units.

4.4 Transformation Units

A *transformation unit* (without import) is a system $tu = (I, P, C, T)$ where I and T are graph class expressions to specify the *initial* and the *terminal* graphs respectively, P is a set of rules, and C is a control condition.

Such a transformation unit specifies a binary relation $SEM(tu) \subseteq SEM(I) \times SEM(T)$ that contains a pair (G, H) of graphs if and only if there is a derivation $G \overset{*}{\underset{P}{\Longrightarrow}} H$ permitted by C.

Example 7 (Eulerian graphs). As an example consider the transformation unit *Eulerian* shown in Figure 10. It takes as initial graphs all those generated by the graph grammar *connected* introduced in Example 5, i.e. all connected unlabeled graphs. The terminal graphs are all graphs whose edges are labeled only with *ok*. The five rules label edges in a specific way, and the control condition requires that the rules $p_2, \ldots, p_5$ can only be applied if p_1 is not applicable. In more detail, the transformation unit *Eulerian* checks whether every node in a connected graph has the same number of incoming and outgoing edges, where loops count as one incoming and one outgoing edge. Hence it checks whether the input graphs are Eulerian. The rule p_1 labels loops with *ok*. The control condition requires that first all loops are labeled. This is necessary because otherwise a rule out of $p_2, \ldots, p_5$ could also be applied to a loop (by identifying the target and the source node of an edge in a match of a left-hand side), which would also admit non-Eulerian graphs as output. The label s in the rules indicates that the edge has already been counted as outgoing edge of its source node. Analogously, t means that it has been counted as incoming

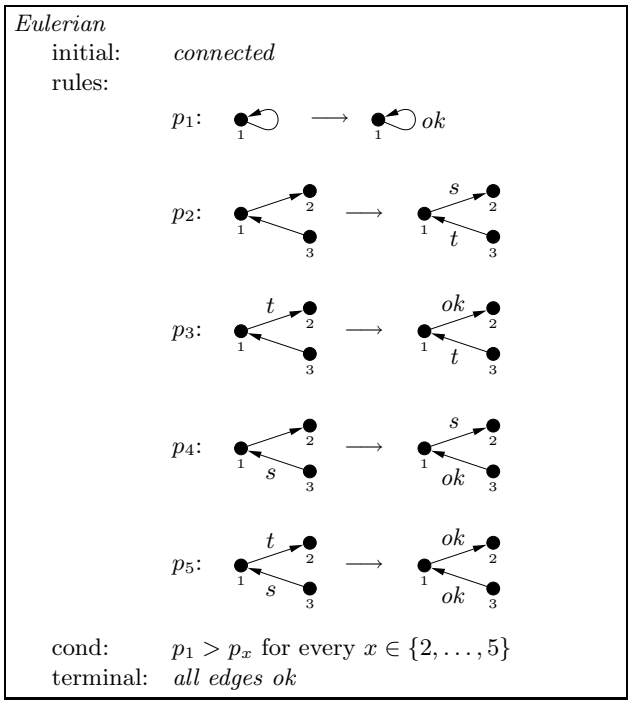

Fig. 10. A transformation unit that specifies Eulerian graphs

edge of its target. The label *ok* means that the edge has been counted once for its source and once for its target.

In every rule application there is a node for which exactly one incoming and one outgoing edge are counted. Hence, for each node the difference between the number of uncounted in- and outgoing edges is invariant under a transformation step. Conversely, it can be shown by induction on the number of simple cycles that every Eulerian graph is recognized by the unit *Eulerian*. Hence, the semantics of *Eulerian* consists of all pairs (G, G') where G is connected, unlabeled, and Eulerian, and G' is obtained from G by labeling every edge with *ok*.

It is worth noting that in general transformation units have an import component that allows to use the semantic relations specified by other units as transformation steps [KK99b]. Moreover, transformation units have been generalized to arbitrary m, n-relations on graphs in [KKK04a, KKK04b], to distributed graph transformation in [KK02], to other data structures than graphs in [KK03], and to parameterized transformation units in [Kus02]. A thorough study of transformation units can also be found in [Kus00b].

5 Transformation of Chomsky Grammars into Graph Grammars

Intuitively, it may be clear that graph transformation is computationally complete. This claim is made precise in this section by translating Chomsky grammars into graph grammars. The translation is based on the observation that a string $x = a_1 \cdots a_k$ with $a_i \in \Sigma$ for $i = 1, \ldots, k$ can be represented by a so-called string graph $x^\bullet$ that consists of $k + 1$ nodes and k edges, where for $i = 1, \ldots, k$ the source of the ith edge is the ith node, the target is the $(i+1)$th node, and the label is a_i (see Figure 11). The first and the last nodes are denoted by $b(x^\bullet)$ and $e(x^\bullet)$, respectively.

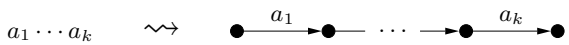

Fig. 11. Translating a string into a string graph

Let $CG = (N, T, S, P)$ be a Chomsky grammar. For the sake of convenience, we assume that the right-hand side of every production is not empty, i.e. for all productions $u \to v$ in P we have $v \neq \lambda$. Such a production p is translated into a graph transformation rule r_p as follows. Let $u^\bullet$ and $v^\bullet$ be string graphs associated with u and v, respectively, such that $b(u^\bullet) = b(v^\bullet)$ and $e(u^\bullet) = e(v^\bullet)$. Then let $r_p = (u^\bullet \supseteq be \subseteq v^\bullet)$ with be the graph consisting of the two nodes $b(u^\bullet)$ and $e(u^\bullet)$ be the graph transformation rule associated with p.

Since the edges in a string graph are directed, there exists a match of $u^\bullet$ in a string graph $x^\bullet$ if and only if u is a substring of x. In other words, the rule r_p can be applied to $x^\bullet$ if and only if the production p can be applied to x. The results of the applications correspond, too, so that we have the following theorem.

Theorem 8 (Correct Translation). *Let $CG = (N, T, S, P)$ be a Chomsky grammar with $v \neq \lambda$ for all $u \to v$ in P.*

1. *Let $x, y \in (N \cup T)^*$ and $p \in P$. Then $x \Longrightarrow_p y$ if and only if $x^\bullet \underset{r_p}{\Longrightarrow} y^\bullet$.*
2. *Let $CG^\bullet = (S^\bullet, P^\bullet, T)$ with $P^\bullet = \{r_p \mid p \in P\}$ be the graph grammar associated with CG. Then $L(CG^\bullet) = L(CG)^\bullet = \{x^\bullet \mid x \in L(CG)\}$.*

The reason for excluding productions of the form $u \to \lambda$ from the construction given above is that $\lambda^\bullet$ is a single-node graph with $b(\lambda^\bullet) = e(\lambda^\bullet)$. Since $|u| > 0$, this implies that there is no string graph $u^\bullet$ with $b(u^\bullet) = b(\lambda^\bullet)$ and $e(u^\bullet) = e(\lambda^\bullet)$. We can deal with this problem as follows.

In the case of a Chomsky grammar of type 1 or higher, we may assume w.l.o.g. that there is only the production $S \to \lambda$ with empty right-hand side, and this only if S does not occur in the right-hand side of any other production.

Thus, we may use the graph transformation rule $S^\bullet \supseteq EMPTY \subseteq \lambda^\bullet$ where $EMPTY$ denotes the empty graph.

In the case of a Chomsky grammar of type 0, we may eliminate each production $u \to \lambda$ by replacing it with all productions of the form $ua \to a$ and $au \to a$, where $a \in N \cup T$. If the original grammar generates the empty word, a new axiom S' and productions $S' \to S \mid \lambda$ must be added. Then the construction(s) given above can be used.

As a consequence of Theorem 8, all undecidability results known for Chomsky grammars transfer to graph grammars. In particular, one gets the following results.

Corollary 9 (Undecidability Results). *For graph grammars, the emptiness, finiteness, membership, inclusion, and equivalence problems are undecidable.*

6 Parallelism and Concurrency

Parallelism is one of the key concepts of computer science. On the one hand, parallel computing may speed up computational processes such that, for example, data processing problems with exponential time complexity become solvable in polynomial time. On the other hand, parallelism may allow one to model certain applications in a realistic way, like the growing of plants or the transportation of goods. Graph transformation provides a framework in which parallelism can be studied in various respects.

The parallel application of rules is easily introduced into the double-pushout approach because the disjoint unions of rules are rules, which are called parallel rules. Consequently, simultaneous applications of rules are just ordinary direct derivations using parallel rules. Moreover, these parallel derivations have some nice properties with respect to sequentialization and parallelization. Given a direct derivation through a parallel rule, the component rules can be applied in arbitrary order yielding the same result. Conversely, a derivation the steps of which are independent of each other in a certain sense can be composed into a single derivation step. Together, this yields pure concurrency, meaning that independent derivation steps in arbitrary order and the parallel application of the same rules perform the same computation.

6.1 Parallel Rules

1. Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ for $i = 1, 2$ be two rules. Then

$$r_1 + r_2 = (L_1 + L_2 \supseteq K_1 + K_2 \subseteq R_1 + R_2)$$

   is the *parallel rule* of r_1 and r_2.

2. Let P be a set of rules. Then P_+ denotes the set of parallel rules over P which is recursively defined to be the smallest set with
   (i) $P \subseteq P_+$ and
   (ii) $r_1 + r_2 \in P_+$ for $r_1, r_2 \in P_+$.

The definition of parallel rules makes use of the disjoint union of graphs as defined and discussed in Section 2.5. Since the disjoint union of graphs is associative and commutative, the disjoint union of rules is associative and commutative, too. Hence, P_+ can be considered as the free commutative semigroup over P.

Theorem 10 (Sequentialization). *Let $G \underset{r_1+r_2}{\Longrightarrow} H$ be a direct derivation. Then there are two direct derivations $G \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} H$ for some graph G_1.*

Because $r_1 + r_2 = r_2 + r_1$, Theorem 10 means that there are also two direct derivations $G \underset{r_2}{\Longrightarrow} G_2 \underset{r_1}{\Longrightarrow} H$ for some G_2.

Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ for $i = 1, 2$ be two rules and $incl_i \colon L_i \to L_1 + L_2$ be the inclusions of the left-hand sides into the disjoint union. Let $g \colon L_1 + L_2 \to G$ be the graph morphism underlying the direct derivation $G \underset{r_1+r_2}{\Longrightarrow} H$. Then $g \circ incl_i \colon L_i \to G$ are the graph morphisms inducing the direct derivations $G \underset{r_i}{\Longrightarrow} G_i$. Using the gluing condition of g one can show $g(L_2) \subseteq G_1$ and $g(L_1) \subseteq G_2$. This allows to define graph morphisms $g_2' \colon L_2 \to G_1$ and $g_1' \colon L_1 \to G_2$ which induce the direct derivations $G_1 \underset{r_2}{\Longrightarrow} H_1$ and $G_2 \underset{r_1}{\Longrightarrow} H_2$. Finally, it is not difficult to show that H, H_1 and H_2 are equal up to isomorphism.

Corollary 11. *Let P be a set of rules. Then $\overset{*}{\underset{P}{\Longrightarrow}} = \overset{*}{\underset{P_+}{\Longrightarrow}}$.*

Theorem 10 implies $\underset{P_+}{\Longrightarrow} \subseteq \overset{*}{\underset{P}{\Longrightarrow}}$, and consequently $\overset{*}{\underset{P_+}{\Longrightarrow}} \subseteq \overset{*}{\underset{P}{\Longrightarrow}}$. The converse inclusion follows from $P \subseteq P_+$.

6.2 Independence

1. Let $G \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} H$ be two direct derivations. Let $h_1 \colon R_1 \to G_1$ be the right graph morphism of the first step and $g_2' \colon L_2 \to G_1$ be the (left) graph morphism of the second step. Then the two derivation steps are *sequentially independent* if

$$h_1(R_1) \cap g_2'(L_2) \subseteq h_1(K_1) \cap g_2'(K_2),$$

   i.e. the matches overlap in G_1 in gluing parts only.

2. Let $G \underset{r_i}{\Longrightarrow} G_i$ be two direct derivations and $g_i \colon L_i \to G$ their underlying graph morphisms. Then the two derivation steps are *parallel independent* if

$$g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2),$$

i.e. the matches overlap in G in gluing parts only.

Sequential independence is equivalently described by the property

$$h_1(R_1) \cap (g_2'(L_2) \setminus (g_2'(K_2)) = \emptyset = (h_1(R_1) \setminus (h_1(K_1)) \cap g_2'(L_2).$$

In other words, the second step does not remove anything of the right match of the first step and the first step does not add anything of the match of the second step.

Parallel independence is equivalently described by the property

$$g_1(L_1) \cap (g_2(L_2) \setminus g_2(K_2)) = \emptyset = (g_1(L_1) \setminus (g_1(K_1)) \cap (g_2(L_2),$$

meaning that none of the derivation steps removes anything of the match of the other one.

The sequentializations $G \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} H$ and $G \underset{r_2}{\Longrightarrow} G_2 \underset{r_1}{\Longrightarrow} H$ of a parallel derivation step $G \underset{r_1+r_2}{\Longrightarrow} H$ are sequentially independent, and the two first steps are parallel independent.

Theorem 12 (Parallelization).

1. *Let $G \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} H$ be two sequentially independent direct derivations. Then there is a direct derivation $G \underset{r_1+r_2}{\Longrightarrow} H$ such that the given derivation is one of its sequentializations.*
2. *Let $G \underset{r_i}{\Longrightarrow} G_i$ for $i = 1, 2$ be two parallel independent direct derivations. Then there is a direct derivation $G \underset{r_1+r_2}{\Longrightarrow} H$ such that the given direct derivations are the first steps of its sequentializations.*

Let $g_1 \colon L_1 \to G$ be the graph morphism of the first step and $g_2' \colon L_2 \to G_1$ be the graph morphism of the second step. Then the sequential independence implies that $g_2'(L_2) \subseteq G$ (up to some renaming of nodes and edges). Therefore the graph morphism $g \colon L_1 + L_2 \to G$ that induces $G \underset{r_1+r_2}{\Longrightarrow} H$ can be defined by $g(x) = g_1(x)$ for x of L_1 and $g(x) = g_2'(x)$ for x of L_2.

Let $g_i \colon L_i \to G$ for $i = 1, 2$ be the graph morphism of the direct derivation $G \underset{r_i}{\Longrightarrow} G_i$. Then the graph morphism $g \colon L_1 + L_2 \to G$ that induces $G \underset{r_1+r_2}{\Longrightarrow} H$ is given by $g(x) = g_i(x)$ for x of L_i. The parallel independence and the application conditions of g_1 and g_2 imply the application conditions of g.

Altogether, the sequentialization and parallelization theorems show that a parallel derivation step, each of two sequentially independent derivations, and two parallel independent direct derivations imply each other.

The situation is illustrated in Figure 12, where || indicates independence. This phenomenon is known as *pure concurrency*.

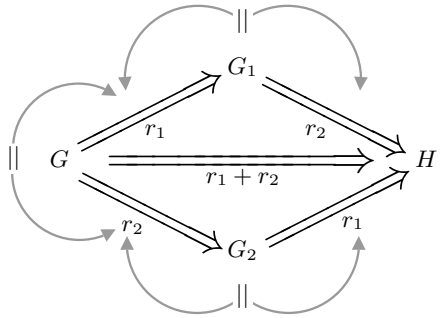

Fig. 12. Parallel and sequential independence

Example 13 (Petri nets). In the case of place/transition systems, the only non-gluing items are the token nodes and their incident edges. Hence, the applications of two firing rules for two transitions are independent if and only if they do not share tokens on some common preplace, i.e. if they are concurrent in the sense of net theory. Therefore, the application of the parallel firing rule of the involved transitions – see for an example the parallel rule in Figure 13 and its application in Figure 14 – corresponds to the parallel firing of these transitions.

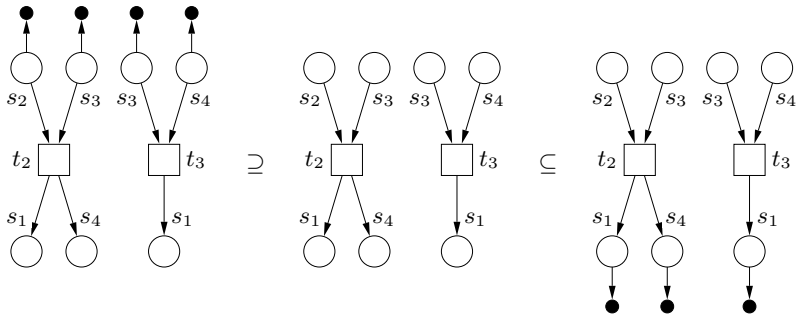

Fig. 13. A parallel rule $r(t_2) + r(t_3)$

More about parallelism and concurrency in the line of this section can be found in Corradini et al. [CEH+97]. The counterpart for the so-called single-pushout approach is surveyed by Ehrig et al. in [EHK+97]. And the third volume of the Handbook of Graph Grammars and Computing by Graph Transformation [EKMR99] provides a collection of seven chapters on various graph-transformational approaches to parallelism, concurrency, distribution, and coordination.

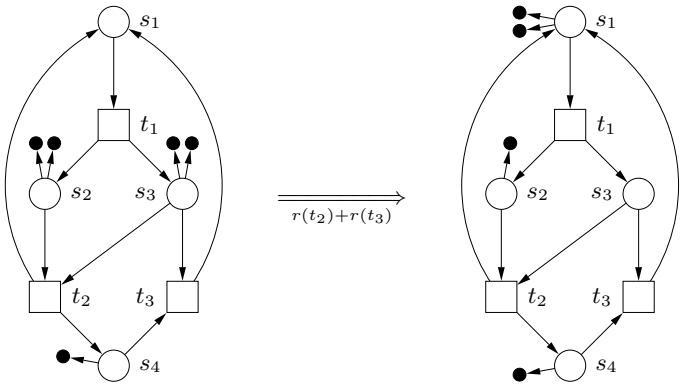

Fig. 14. Modeling the parallel firing of transitions t_2 and t_3

7 Context-Freeness of Hyperedge Replacement

Graph transformation is a general modeling framework that allows one to specify arbitrary computable relations on graphs. Consequently, any non-trivial semantic property of transformation units and graph grammars is undecidable. If one looks for subclasses with decidable properties (and other nice properties), graph-transformational counterparts of context-freeness are candidates. One of these is hyperedge replacement (see, e.g., [Hab92, DHK97]), which is usually formulated for hypergraphs the edges of which may be incident to more than two nodes. But hyperedge replacement can also be seen as a special case of graph grammars as introduced in Section 4.

To make this precise, we assume some subset $N \subseteq \Sigma$ of *nonterminals* which are typed, i.e. there is an integer $k(A) \in \mathbb{N}$ for each $A \in N$. Moreover we assume that Σ contains the numbers $1, \ldots, max$ for some $max \in \mathbb{N}$ with $k(A) \leq max$ for all $A \in N$. A hyperedge with label $A \in N$ is meant to be an atomic item which is attached to a sequence of nodes $v_1 \cdots v_{k(A)}$. It can be represented by a node with an A-labeled loop and $k(A)$ edges the labels of which are $1, \ldots, k(A)$, respectively, and the targets of which are $v_1, \ldots, v_{k(A)}$, respectively, as depicted in Figure 15. Accordingly, we call such a node with its incident edges an A-hyperedge. A graph is said to be *N-proper* if each occurring nonterminal and each number between 1 and max belongs to some hyperedge.

Each $A \in N$ induces a particular N-proper graph $A^\bullet$ with the nodes $\{0, \ldots, k(A)\}$ and a single hyperedge where the A-loop is attached to 0 and $i \in \{1, \ldots, k(A)\}$ is the target of the edge labeled with i for $i = 1, \ldots, k(A)$. Let $[k(A)]$ denote the discrete graph with the nodes $\{1, \ldots, k(A)\}$. Thus a rule of the form $A^\bullet \supseteq [k(A)] \subseteq R$ for some N-proper graph R is a *hyperedge replacement rule*, which can be denoted by $A ::= R$ for short.

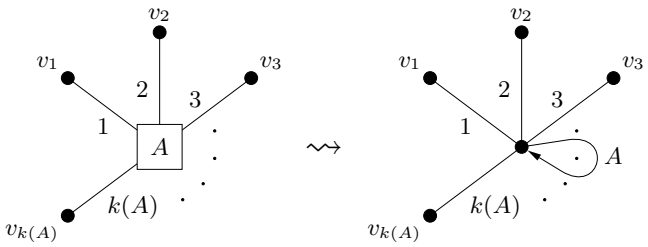

Fig. 15. Graph representation of a hyperedge

A *hyperedge replacement grammar* is a system $HRG = (N, T, P, S)$ with $S \in N$, $T \subseteq \Sigma$ with $T \cap N = \emptyset$, and a set of hyperedge replacement rules P. Its generated language $L(HRG)$ is defined as the graph language generated by the graph grammar $(S^\bullet, P, T)$.

Example 14 (flow diagrams). The graph transformation rules $r_{compound}$ and $r_{while\text{-}do}$ in Section 3.1 become proper examples of hyperedge replacement rules with hyperedges of type 2 if one replaces every statement (i.e. every rectangle node) with its two incident edges by a node with a box-labeled node and two outgoing numbered edges. This conversion is shown in Figure 16.

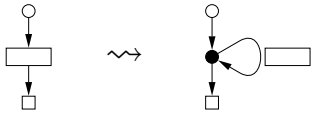

Fig. 16. Conversion of statements into the graph representation of a hypergraph

Example 15 (Sierpiński triangles). Another example, this time with hyperedges of type 3, is the hyperedge replacement grammar depicted in Figure 17 that generates Sierpiński triangles. A sample derivation is depicted in Figure 18, where the second and third step each are replacing three hyperedges in parallel.

Further examples can be found in the literature (see, e.g. [Hab92, DHK97]).

In this way, hyperedge replacement is just a special case of graph transformation, but with some very nice properties. Some simple observations are the following, giving first indications to the context-freeness of hyperedge replacement.

1. Let $r = (A ::= R)$ be a hyperedge replacement rule and G an N-proper graph with an A-hyperedge y. Then there is a unique graph morphism $g\colon A^\bullet \to G$ mapping $A^\bullet$ to the A-hyperedge y such that the contact condition is satisfied and therefore r is applicable to G.

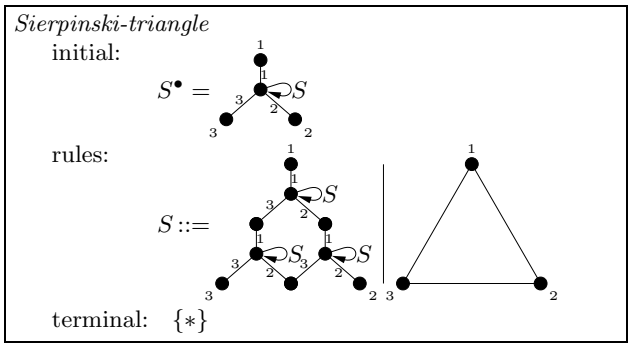

Fig. 17. A hyperedge replacement grammar generating Sierpiński triangles

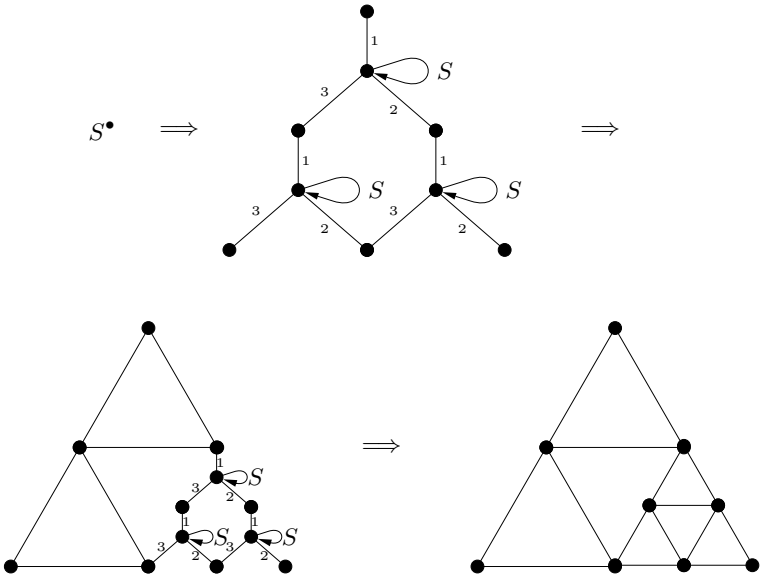

Fig. 18. A derivation generating a Sierpiński triangle

2. The directly derived graph H is N-proper and is obtained by removing y, i.e. by removing the node with the A-loop and all other incident edges, and by adding R up to the nodes $1, \ldots, k(A)$ where edges of R incident to $1, \ldots, k(A)$ are redirected to $g(1), \ldots, g(k(A))$, respectively. Due to this construction, H may be denoted by $G[y/R]$.
3. Two direct derivations $G \underset{r_1}{\Longrightarrow} H_1$ and $G \underset{r_2}{\Longrightarrow} H_2$ are parallel independent if and only if they replace distinct hyperedges.
4. A parallel rule $r = \sum_{i \in I} r_i$ of hyperedge replacement rules $r_i = (A ::= R_i)$ for $i \in I$ is applicable to G if and only if there are pairwise distinct A_i-

hyperedges y_i for all $i \in I$. In analogy to the application of a single rule, the resulting graph may be denoted by $G[y_i/R_i \mid i \in I]$.

5. If $I = I_1 \uplus I_2$, then we have in addition

$$G[y_i/R_i \mid i \in I] = (G[y_i/R_i \mid i \in I_1])[y_i/R_i \mid i \in I_2].$$

6. Two successive direct derivations $G \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} H$ are sequentially independent if and only if the hyperedge replaced by the second step is not created by the first one.

Observation 1 holds because the A-hyperedge of $A^\bullet$ and y have the same structure. The only identifications that are possible concern the targets of numbered edges. But they are gluing nodes such that the identification condition holds. The only node to be removed is the one with the A-loop. But all its incident edges are removed, too, so that the contact condition holds. Observation 2 rephrases the definition of a direct derivation for the special case of a hyperedge replacement rule. Observation 3 holds because the matches of two direct derivations are either equal and then dependent on each other because the non-gluing node is removed by both of them, or they share only target nodes of numbered edges and then they are parallel independent. If a parallel rule is applicable, the identification condition is satisfied in particular. Therefore, as observation 4 states, no two rules can remove the same hyperedge as they would share non-gluing items. Then, observation 5 is a consequence of the sequentialization theorem as $\sum_{i \in I} r_i = \sum_{i \in I_1} r_i + \sum_{i \in I_2} r_i$. Finally, observation 6 holds by definition and the special form of the hyperedge replacement rules. Altogether, the direct derivations through hyperedge replacement rules can be ordered arbitrarily as long as they deal with different hyperedges. This observation leads to the following result.

Theorem 16 (Context-Freeness Lemma). *Let $HRG = (N, T, P, S)$ be a hyperedge replacement grammar and let $A^\bullet \overset{n+1}{\underset{P}{\Longrightarrow}} H$ be a derivation. Then there are some rule $A ::= R$ and a derivation $A(y)^\bullet \overset{n(y)}{\underset{P}{\Longrightarrow}} H(y)$ for each hyperedge y of R with label $A(y)$ such that $H = R[y/H(y) \mid y \in Y_R]$ and $\sum_{y \in Y_R} n(y) = n$, where Y_R is the set of hyperedges of R.*

If one varies the start symbol of HRG through all nonterminals, one gets a family of hyperedge replacement grammars $(HRG(A))_{A \in N}$ with $HRG(A) = (N, T, P, A)$. The Context-Freeness Lemma relates the graphs derived by this family to each other. Reformulated for the generated languages, hyperedge replacement languages turn out to be fixed points.

Theorem 17 (Fixed-Point Theorem). *Let $(HRG(A))_{A \in N}$ with $HRG(A) = (N, T, P, A)$ be a family of hyperedge replacement grammars (which share rules*

as well as terminals and nonterminals). Then, for each $A \in N$, the following equality holds:

$$L(HRG(A)) = \bigcup_{(A ::= R) \in P} \{(R[y/H(y) \mid y \in Y_R] \mid H(y) \in L(HRG(A(y)))\}.$$

The Context-Freeness Lemma and Fixed-Point Theorem characterize generated graphs as covered each by a right-hand side of a rule without the hyperedges and smaller derived graphs. This provides a recursive way to prove and decide properties of the generated languages and their members if the properties are compatible with the composition of generated graphs as subsitution of hyperedges by derived graphs in right-hand sides. Many graph-theoretic properties like connectedness, planarity, Hamiltonicity, k-colorability, and many more are compatible. The explicit decidability results of hyperedge replacement grammars can be found in [Hab92, DHK97] where also further structured results are surveyed. For other context-free graph-transformational approaches which are mainly based on node replacement, the reader may consult Engelfriet and Courcelle [Eng97, Cou97].

8 Conclusion

In this chapter, we have given an introductory survey on some essentials of graph transformation focussing on theoretical aspects. For further reading, we recommend the three volumes of the Handbook of Graph Grammars and Computing by Graph Transformation [Roz97, EEKR99, EKMR99]. Language-theoretic topics with respect to node and hyperedge replacement are addressed in Chapters 1, 2 and 5 of Volume 1. Collage grammars as a picture-generating device based on hyperedge replacement are the subject of Chapter 11 of Volume 2. Graph transformation as a general computational framework is presented in Chapters 3 and 4 of Volume 1 with respect to the double and single pushout approaches while an alternative approach can be found in Chapter 7. Chapters 3 and 4 discuss aspects of parallelism and concurrency in particular, which are also studied in the whole of Volume 3. Finally, Volume 2 is devoted to potential applications of graph transformation relating it to term rewriting and functional programming (Part 1), to visual and object-oriented languages (Part 2), to software engineering (Part 3), to other engineering disciplines (Part 4), to picture processing (Part 5), to the implementation of graph-transformational specification languages and tools (Part 6), and to structuring and modularization (Part 7). In recent years, main topics of interest have become the syntactic and semantic foundation of visual languages in the broadest sense and of model transformation, as graphs seem to be obvious candidates to represent visual models, diagrams, and all kinds of complex structures in a precise way (see, e.g., [CEKR02, EEPPR04, PNB04] and the SeGraVis webpage `www.segravis.org`).

References

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[AH89] Kenneth Appel and Wolfgang Haken. *Every Planar Map is Four Colorable*, volume 98 of *Contemporary Mathematics*. Amer. Mathematical Society, 1989.

[CEH+97] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation Part I: Basic concepts and double pushout approach. In Rozenberg [Roz97].

[CEKR02] Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proc. 1st Int. Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*. Springer, 2002.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [Roz97], pages 313–400.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [Roz97], pages 95–162.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, Singapore, 1999.

[EEPPR04] Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors. *Proc. 2nd Int. Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[EHK+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation Part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [Roz97], pages 247–312.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, Singapore, 1999.

[Eng97] Joost Engelfriet. Context-free graph grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 125–213. Springer, 1997.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [Roz97], pages 1–94.

[Gib85] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[GV03] Claude Girault and Rüdiger Valk. *Petri Nets for Systems Engineering*. Springer, 2003.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.

[Har69] Frank Harary. *Graph Theory.* Addison Wesley, 1969.

[KK99a] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig et al. [EEKR99], pages 607–638.

[KK99b] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[KK02] Peter Knirsch and Sabine Kuske. Distributed graph transformation units. In Corradini et al. [CEKR02], pages 207–222.

[KK03] Hans-Jörg Kreowski and Sabine Kuske. Approach-independent structuring concepts for rule-based systems. In Martin Wirsing, Dirk Pattison, and Rolf Hennicker, editors, *Proc. 16th Int. Workshop on Algebraic Development Techniques (WADT 2002)*, volume 2755 of *Lecture Notes in Computer Science*, pages 299–311. Springer, 2003.

[KKK04a] Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske. Rule-based transformation of graphs and the product type. In Patrick van Bommel, editor, *Transformation of Knowledge, Information, and Data: Theory and Applications*, pages 29–51. Idea Group Publishing, Hershey, Pennsylvania, USA, 2004.

[KKK04b] Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske. Typing of graph transformation units. In Ehrig et al. [EEPPR04], pages 112–127.

[Kus00a] Sabine Kuske. More about control conditions for transformation units. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[Kus00b] Sabine Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems.* PhD thesis, University of Bremen, 2000.

[Kus02] Sabine Kuske. Parameterized transformation units. In *Proc. GETGRATS Closing Workshop*, volume 51 of *Electronic Notes in Theoretical Computer Science*, 2002.

[PNB04] John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors. *Proc. 2nd Int. Workshop and Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science.* Springer, 2004.

[Rei85] Wolfgang Reisig. *Petri Nets. An Introduction.* Springer, 1985.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets.* Springer, 1998.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, Singapore, 1997.

[Sch97] Andy Schürr. Programmed graph replacement systems. In Rozenberg [Roz97], pages 479–546.