JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Modeling and Analyzing Graph Algorithms by Means of Graph Transformation Units

Hans-Jörg Kreowski and Sabine Kuske
University of Bremen, Germany

ABSTRACT Graph transformation units are rule-based devices to specify processes on graphs and the dynamics of information-processing systems with graphs as states. In this article, we propose graph transformation units as a framework to model graph algorithms, to prove their correctness and to analyze their complexity. A specific emphasis is laid on the improvement of the efficiency of graph algorithms by massive parallelism.

KEYWORDS graph algorithms, graph transformation units, massive parallelism.

1. Introduction

Graph problems and their algorithmic solutions are of great theoretical and practical interest. Typical examples are the computations of shortest paths, minimum spanning trees and maximum network flows. A look into the books and papers on graph algorithms reveals that there is no common and uniform way to describe and analyze graph algorithms. On the contrary, the authors use a wide spectrum of informal or semi-formal methods, but rarely formal ones. Most frequently, one encounters a kind of pseudo-code where an intuitive standard interpretation is assumed and a formal mathematical semantics is not explicitly given. Consequently, correctness proofs and the deduction of complexity bounds are somewhat doubtful although they often meet a common agreement. An exception is the LEDA approach which provides a platform for combinatorial and geometric computing including graph algorithms (see (Mehlhorn & Näher 1999)). However, algorithms are modeled on the level of imperative programming that is far away from a more intuitive visual modeling paradigm.

In this article, we propose and advocate graph transformation units as rule-based devices to model and analyze graph algorithms. The framework of graph transformation units offers a formal semantics, a proof schema, a structuring principle and analysis methods (cf., e.g., (Andries et al. 1999; H.-J. Kreowski & Kuske 1999; H.-J. Kreowski et al. 2008)). Since graph transformation units are based on graph transformation rules, the stepwise execution of the algorithms may illustrate basic ideas of these algorithms which in turn may be helpful for their understanding, their verification and their teaching. For this purpuse existing graph transformation tools for testing, simulating, visualizing, and verifying the algorithms could be employed (see for example (Taentzer 2003; Ermler et al. 2012; Jakumeit et al. 2010; Plump 2012; Ghamarian et al. 2012)).

A graph transformation unit is a formal syntactic construct consisting – in its simple form – of a finite set of graph transformation rules, two graph class expressions and a control condition. In the structured form, a unit can also use other units. Semantically, the graph class expressions specify initial and terminal graphs; the rules can be applied to graphs yielding graphs so that rule application can be repeated; and the control condition determines the order in which the rules are applied. A graph transformation unit computes a binary relation between initial and terminal graphs. An initial graph is semantically related to a terminal graph if the latter one can be derived from the former by iterated rule application according to the control condition. In other words, a graph transformation unit models a graph algorithm. The derivation lengths and the graph sizes (and related quantities) provide a base for induction proofs. This allows to prove properties of the modeled graph algorithms and

JOT reference format:
Hans-Jörg Kreowski and Sabine Kuske. *Modeling and Analyzing: Graph Algorithms by Means of :Graph Transformation Units*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2020.19.3.a9

in particular their correctness. Since a rule application can be performed in polynomial time where the order of the polynomial corresponds to the size of the left-hand side of the applied rule, the derivation lengths also yield complexity bounds. For the sake of simplicity and general intelligibility, correctness and complexity results are stated and explained, but not formally proven. Furthermore, the framework also offers methods that analyze which rules can be applied in parallel, thus enabling parallelization of graph algorithms.

The main contribution of this paper is a case study to demonstrate the usefulness of the framework of graph transformation units for the modeling of graph algorithms. As illustrating examples, we have chosen the well-known characterization of Eulerian graphs as connected graphs with only even-degree nodes and the computation of shortest paths. As a consequence, the graph transformation units modeling them can be compared with the elaborations of both topics in nearly all text books on graph algorithms showing that our modeling is quite adequate with respect to conciseness, preciseness, proving, and complexity analysis. With respect to parallelization of the (sequential) algorithms, we open up a new line of investigation.

The area of graph transformation has been developed for the last 50 years with an emphasis on graph language generation and modeling of graph-based information-processing systems in a wide range of potential applications. Hence, it is somewhat surprising that little systematic work has been done on graph algorithms although they are obvious cases of graph transformation. To our knowledge, there are two approaches towards graph-transformational computation of graph algorithms. The first one is the notion of graph relabeling systems (see, e.g., (Litovski et al. 1999a)) that is particularly designed for distributed algorithms on graphs and networks where the underlying structure is preserved and only labels are changed in computation steps. The second one is the concept of graph programs (see, e.g., (Plump 2009, 2012)) which are a kind of graph transformation units with restricted ways of specifying initial and terminals graphs and control conditions.

The paper is organized as follows. After the preliminaries in Section 2, graph transformation units in their simple form are introduced in Section 3 and in Section 4 in the structured form which allows the reuse of units within units. Both concepts are illustrated by classical graph algorithms such as the computation of shortest paths and Euler tours. Parallel rules and their applications are considered in Section 5 and we show that the shortest path algorithm of Section 3 can be parallelized such that all results are computed in a logarithmic number of steps. The paper ends with a related work section and a conclusion.

That the development of graph transformation units into a visual modeling language could be worthwhile is largely inspired by our cooperation on an integrated graph-based semantics for UML with Martin Gogolla and Paul Ziemann (Kuske et al. 2009).

2. Preliminaries

In this section, we recall the basic notions and notations of graphs and rule-based graph transformation as far as needed in this paper to define graph transformation units in Sections 3 and 4. All preliminaries are standard in the area of graph transformation. We organize them by the subsections "graph-tranformational rule bases", "graph class expressions" and "control conditions" as these three components form graph transformation approaches underlying graph transformation units (cf., e.g., (H.-J. Kreowski et al. 2008)).

2.1. Graph-transformational rule bases

There are many classes of graphs such as undirected or directed, labeled or unlabeled, hypergraphs, trees, forests, Petri nets, finite automata, etc. There are also various ways to transform graphs in a rule-based manner (cf. (Rozenberg 1997)). A graph-transformational rule base provides a class of graphs, a class of graph transformation rules and a prescription of how the rules are applied to graphs. Hence, the general concept of graph-transformational rule bases allows for choosing the class of graphs and the rule application approach that are most adequate for modeling the algorithm at hand. We focus our considerations on directed edge-labeled graphs and the type of rule application that is most frequently used in the literature. It is called DPO-approach where DPO is the acronym for *double pushout* (see, e.g., (Ehrig et al. 2006, 2015)), but we define the rule application explicitly without recourse to categorical notations.

Definition 1 (Rule base)**.** A (graph-transformational) *rule base* $B = (\mathcal{G}, \mathcal{R}, \Longrightarrow)$ consists of a class of graphs $\mathcal{G}$, a class of rules $\mathcal{R}$, and a rule application operator $\Longrightarrow$ with $\underset{r}{\Longrightarrow} \subseteq \mathcal{G} \times \mathcal{G}$ for every $r \in \mathcal{R}$.

The rule application operator is used in infix notation, i.e., $(G, H) \in \underset{r}{\Longrightarrow}$ is denoted by $G \underset{r}{\Longrightarrow} H$. In the following, we present a sample rule base that is used throughout this paper.

Definition 2 (Directed edge-labeled graph)**.** Let Σ be a set of labels with $* \in \Sigma$. A (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 $e \in E$, and $l \colon E \to \Sigma$ is a mapping assigning a *label* to every edge $e \in E$.

An edge e with $s(e) = t(e)$ is called a *loop* and if $l(e) = z$ it is also called a z*-loop*. For a node $v \in V$, the number of edges with v as source is denoted by $outdegree(v)$ and the number of edges with v as target by $indegree(v)$. An edge with label $*$ is also called an *unlabeled edge*. In drawings of graphs, the label $*$ is omitted. A symbol next to a node or within a node represents its name. 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 class of all directed edge-labeled graphs is denoted by $\mathcal{G}_\Sigma$. Two graphs in $\mathcal{G}_{\{*\}}$ are given in Fig. 1.

Each directed graph $G = (V, E, s, t, l)$ has an *underlying undirected graph* $U(G) = (V, E, att, l)$ with $att(e) = \{s(e), t(e)\}$ for each $e \in E$. A *path* in $U(G)$ between v and v' of length n is a sequence $v_0 e_1 v_1 \cdots e_n v_n$ with $n \geq 0$, $v_0 = v$, $v_n = v'$, and $att(e_i) = \{v_{i-1}, v_i\}$ for $i = 1, \ldots, n$. The graph $U(G)$ is *connected* if for each $v, v' \in V$, there is a path in $U(G)$ between v and v'. The graph G is *connected*, if $U(G)$

is connected. A *path* in G is a sequence $p = v_0 e_1 v_1 \cdots e_n v_n$ with $n \geq 0$, $v_0 = v$, $v_n = v'$, $s(e_i) = v_{i-1}$ and $t(e_i) = v_i$ for $i = 1, \ldots, n$. If $v_0 = v_n$ and $n > 0$, the path p is a *cycle*.

Definition 3 (Graph morphism)**.** 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 $l_G(e) = l_H(g_E(e))$ for all $e \in E_G$.

Fig. 1 shows an example of a graph morphism where the nodes 1 and 2 are mapped to the nodes $1'$ and $2'$, respectively.

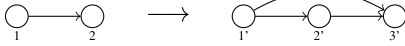


Figure 1 A graph morphism from the left graph to the right one

If the mappings g_V and g_E are bijective, then G and H are *isomorphic*, denoted by $G \cong H$. If they are inclusions, then G is called a *subgraph* of H, denoted by $G \subseteq H$. 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 $g(G) \subseteq H$. If the mappings g_V and g_E are injective, the match $g(G)$ is also called *injective*. In this case, G and $g(G)$ are isomorphic.

Definition 4 (Rule)**.** 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.

A *rule with negative context* $r = (N \supseteq L \supseteq K \subseteq R)$ consists of a rule $(L \supseteq K \subseteq R)$ and a graph $N \in \mathcal{G}_\Sigma$ with $L \subseteq N$ (see, e.g., (Habel et al. 1996)). The components N, L, K, and R are called *negative context*, *left-hand side*, *gluing graph*, and *right-hand side*.

A rule is depicted as $L \to R$ where the nodes belonging to the gluing graph are equally numbered in L and R. The edges of the gluing graph are all edges of $E_L \cap E_R$ connecting the nodes of the gluing graph. A rule with negative context is depicted as $N \to R$ where the items that belong to N but not to L are dashed. Fig. 2 shows a rule with negative context.

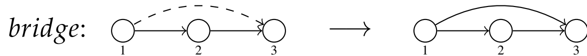


Figure 2 A rule with negative context

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 in such a way that the match of the gluing graph is kept. A rule with negative context can only be applied if the match of L in G has no context that corresponds to the dashed context of the left-hand-side, i.e., to the complement $N - L$.

Definition 5 (Rule application)**.** The application of $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ consists of the following three steps.

1. Choose a match $g(L)$ of L in G subject to the following conditions.
   - *dangling condition*: $v \in g_V(V_L)$ with $s_G(e) = v$ or $t_G(e) = v$ for some $e \in E_G - g_E(E_L)$ implies $v \in g_V(V_K)$.
   - *identification condition*: $g_V(v) = g_V(v')$ for $v, v' \in V_L$ implies $v = v'$ or $v, v' \in V_K$ as well as $g_E(e) = g_E(e')$ for $e, e' \in E_L$ implies $e = e'$ or $e, e' \in E_K$.
2. Now the nodes of $g_V(V_L) - g_V(V_K)$ and the edges of $g_E(E_L) - g_E(E_K)$ are removed yielding the *intermediate graph* $Z \subseteq G$.
3. Add the right-hand side R to Z by gluing Z with R in $g(K)$ yielding the graph H with $V_H = V_Z \uplus (V_R - V_K)$ and $E_H = E_Z \uplus (E_R - E_K)$ where $\uplus$ denotes the disjoint union of sets. The edges of Z keep their labels, sources, and targets so that $Z \subseteq H$. The edges of R keep their labels; they also keep their sources and targets provided that those belong to $V_R - V_K$. Otherwise, they are redirected to the image of their original source or target, i.e., $s_H(e) = g(s_R(e))$ for $e \in E_R - E_K$ with $s_R(e) \in V_K$, and $t_H(e) = g(t_R(e))$ for $e \in E_R - E_K$ with $t_R(e) \in V_K$.

The dangling condition ensures that the removal of $g(L) - g(K)$ does not produce dangling edges (edges without source and/or target) so that Z is a graph. The identification condition requires that those items that are identified via the matching morphism belong to the gluing graph. The identification condition is significant for the application of parallel rules introduced in Section 5.

A rule with negative context $r = (N \supseteq L \supseteq K \subseteq R)$ is applied to G in the same way provided that the morphism $g\colon L \to G$ cannot be extended to N, i.e., there is no graph morphism $h\colon N \to G$ with $h|_L = g$ (where $h|_L$ denotes the restriction of h to L).

The application of r to G w.r.t. the graph morphism g is denoted by $G \underset{r}{\Longrightarrow} H$. It is called a *direct derivation* from G to H. A *derivation* from G to H is a sequence of direct derivations $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ with $G_0 = G$, $G_n = H$ and $n \geq 0$. The sequence $r_1 \cdots r_n$ is called the *application sequence* of the derivation. If $r_1, \ldots, r_n \in P$, the derivation is also denoted by $G \overset{n}{\underset{P}{\Longrightarrow}} H$. If n does not matter, we write $G \overset{*}{\underset{P}{\Longrightarrow}} H$ and if the underlying rule set is known from the context, the subscript P may be omitted.

For example, the rule *bridge* can be applied to the subgraph of the left graph in Fig. 3 depicted with thick lines with the effect that an additional edge is inserted from the first (leftmost) to the third node, also shown as a thick edge.

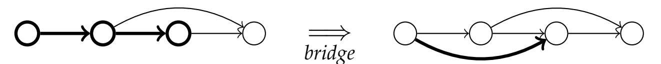


Figure 3 A rule application

It is worth noting that the application of a given (fixed) rule to a graph G with $|V|$ nodes and $|E|$ edges can be performed

in polynomial time provided that the equality of labels can be checked in polynomial time. This is due to the fact that any left-hand side (or negative context) of size k has at most $(|V| + |E|)^k$ matches in G. Moreover, the further steps of the rule application can be done in linear time.

2.2. Graph class expressions

Sometimes it is desirable to restrict the class $\mathcal{G}_\Sigma$ of graphs to some subclass. For example, one may want to start derivations only from specific initial graphs or filter out a subclass of all derived graphs as output. To this aim *graph class expressions* restrict the class $\mathcal{G}_\Sigma$ to subclasses, i.e., each graph class expression e specifies a set $SEM(e) \subseteq \mathcal{G}_\Sigma$. The class of all graph class expressions is denoted by $\mathcal{E}$.

Typical examples of graph class expressions that are used in this paper, are *all*, Δ-*labeled* for $\Delta \subseteq \Sigma$, P-*reduced* for $P \subseteq \mathcal{R}$, *forbidden*(M) for $M \subseteq \mathcal{G}_\Sigma$, *simple*, *strictly-simple* and *loop-free*, where $SEM(all) = \mathcal{G}_\Sigma$, $SEM(\Delta\text{-}labeled) = \mathcal{G}_\Delta$, $SEM(P\text{-}reduced)$ consists of all graphs to which no rule in P can be applied, $SEM(forbidden(M))$ contains all graphs that have no subgraph isomorphic to some $G \in M$, $SEM(simple)$ consists of all graphs the parallel edges of which have different labels, $SEM(strictly\text{-}simple)$ consists of all graphs without parallel edges, and $SEM(loop\text{-}free)$ consists of all graphs without loops.

Graph class expressions e and e' may by combined to $e \,\&\, e'$ with $SEM(e \,\&\, e') = SEM(e) \cap SEM(e')$. Moreover, we employ the graph class operator a-*looped* for $a \in \Sigma$ that can be applied to a graph class expression e such that $a\text{-}looped(e)$ provides each node of each graph in $SEM(e)$ with an extra a-loop.

Given one of the graph class expressions above, it is easy to check whether a graph meets the expression. In the cases of Δ-labeled, P-*reduced*, *forbidden*(M), *simple*, *strictly-simple*, and *loop-free*, one can use the matching algorithm that is part of the rule application procedure because all those expressions require the presence or absence of certain subgraphs. Therefore, all of them can be checked in polynomial time as pointed out at the end of the previous subsection. Moreover, the operator a-looped is obviously linear in the number of nodes. Further possibilities to describe graph class expressions are pointed out in the second paragraph of Section 6.

2.3. Control conditions

Rule application is highly nondeterministic. On one hand a rule may be applied to a graph at several matches. On the other hand, there may be various rules applicable to the current graph. Control conditions can reduce this nondeterminism. In more detail, each control condition is defined over a finite set ID of names with $SEM(id) \subseteq \mathcal{G}_\Sigma \times \mathcal{G}_\Sigma$ for each $id \in ID$. In simple graph transformation units, these names refer to rules and for each rule r the relation $SEM(r)$ is defined as $\underset{r}{\Longrightarrow}$.[1] The class of all control conditions is denoted by $\mathcal{C}$. Every control condition $C \in \mathcal{C}$ specifies a binary relation $SEM(C) \subseteq \mathcal{G}_\Sigma \times \mathcal{G}_\Sigma$.

A useful class of control conditions are regular expressions. For a given set of rules P, the set $REG(P)$ of regular expressions over P is recursively defined as follows. $\lambda, \emptyset \in REG(P)$, $P \subseteq REG(P)$, and for $reg_1, reg_2 \in REG(P)$, $(reg_1; reg_2)$, $(reg_1|reg_2)$, and $(reg_1^*) \in REG(P)$ where $*$ has a stronger binding than ; which in turn has a stronger binding than $|$. As usual, the language $L(reg)$ of a regular expression $reg \in REG(P)$ is defined by $L(\lambda) = \{\lambda\}$, $L(\emptyset) = \emptyset$, $L(reg) = \{reg\}$ if $reg \in P$, and $L(reg_1; reg_2) = L(reg_1)L(reg_2)$, $L(reg_1|reg_2) = L(reg_2) \cup L(reg_2)$ and $L(reg^*) = L(reg)^*$. A pair (G, G') is specified by a regular expression reg if there is a derivation $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ such that $G_0 = G$, $G_n = G'$ and $r_1 \cdots r_n \in L(reg)$. For example, the control condition $r_1; r_2^*$ only allows derivations in which rule r_1 is applied first and then rule r_2 is applied arbitrarily often.

The unary operator ! can be applied to regular expressions for expressing that they must be applied as long as possible. Hence, $(G, G') \in SEM(reg!)$ if and only if $(G, G') \in SEM(reg^*)$ and there is no graph G'' such that $(G', G'') \in SEM(reg)$.

One further useful control condition is a priority relation over rules, i.e., a binary relation $<$ on a set P of rules which is transitive, but neither reflexive nor symmetric. It demands that a rule $r \in P$ can only be applied if there is no applicable rule $r' \in P$ with a higher priority, i.e., with $r' > r$.

3. Simple Graph Transformation Units

In this section, we recall the notion of graph transformation units in their simple form which can be seen as basic means to model graph algorithms. A structuring principle is added in the next section. Graph transformation units were introduced in (H.-J. Kreowski & Kuske 1996) and further developed in (H.-J. Kreowski et al. 1997; H.-J.. Kreowski & Kuske 1999; Andries et al. 1999; Janssens et al. 2005) (see (H.-J. Kreowski et al. 2008) for a comprehensive overview).

The illustrating units in this and the next two sections model the well-known characterization of Eulerian graphs by connected graphs with nodes of even degree as well as some variants of shortest-paths algorithms. Some of the shortest-paths units are used as illustrating examples in (H.-J. Kreowski et al. 2018). The Eulerian-graphs example is new.

A simple graph transformation unit provides rules, a control condition, and two graph class expressions.

Definition 6 (Simple graph transformation unit)**.** A (simple) *graph transformation unit* is a system $gtu = (I, P, C, T)$ where $I \in \mathcal{E}$ is the *initial graph class expression*, $P \subseteq \mathcal{R}$ is a finite set of *rules*, $C \in \mathcal{C}$ is a control condition over P and $T \in \mathcal{E}$ is the *terminal graph class expression*. The *semantics* of gtu is the binary relation

$$SEM(gtu) = (SEM(I) \times SEM(T)) \cap \overset{*}{\underset{P}{\Longrightarrow}} \cap SEM(C).$$

The components I, P, C, and T are also denoted by I_{gtu}, P_{gtu}, C_{gtu}, T_{gtu}, respectively. In examples, a graph transformation unit is presented schematically where the components I, P, C, and T are listed after respective keywords "initial", "rules", "cond", and "terminal". We omit the control condition if it does not impose any restriction on the order of rule applications. We also omit the graph class expression *all*.

[1] For technical simplicity, we do not distinguish between rules and their names.

It is worth noting that each graph transformation unit gtu may serve as a graph class expression with $SEM(gtu) = pr_1(SEM(gtu))$ where pr_1 denotes the projection to the first component.

The following three examples illustrate the use and usefulness of this concept for the modeling of graph algorithms.

3.1. Connected graphs

Consider the simple graph transformation unit in Figure 4. Its initial graph class expression specifies the class of all unlabeled graphs. Its rule set consists of the three rules *opp*, *bridge*, and *test*. According to the control condition, the rules *opp* and *bridge* can be applied arbitrarily often in arbitrary order whereas the rule *test* is not allowed. Instead, it occurs in the terminal graph class expression permitting only graphs to which the rule *test* is not applicable via an injective matching morphism.

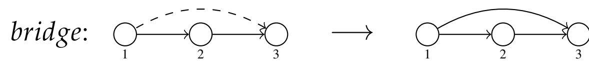
connected

initial: $\{*\}$-*labeled*

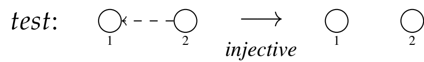
rules:

opp:

bridge:



test:



cond: $(opp \mid bridge)^*$

terminal: $\{test\}$-*reduced*

Figure 4 The graph transformation unit *connected*

The rule *opp* inserts an edge $\bar{e}$ in the opposite direction of an existing edge e provided that $\bar{e}$ is not present, yet. The rule *bridge*, already known from Section 2, inserts an edge from the start node to the end node of each path of length 2 provided that such an edge does not yet exist. In particular, none of the two rules can be applied to the same match twice. Both rules applied as long as possible produce the symmetric and transitive closure of each initial graph. In particular, they cannot connect disconnected graph components. The rule *test* can only be applied if there are two nodes that are not connected by an edge. It is applicable as long as *opp* or *bridge* is applicable. It is still applicable to $\{opp, bridge\}$-reduced graphs if they are not connected. In other words, an initial graph is connected if and only if the application of *opp* and *bridge* as long as possible in arbitrary order derives a graph to which *test* is not applicable.

Altogether, we get the following results concerning correctness and complexity of *connected*.

Proposition 1 (Correctness and complexity)**.** Let $P = \{opp, bridge\}$. Then the following holds:

1. Let $G \in \mathcal{G}_\Sigma$. Then $(G, H) \in SEM(connected)$ for some $H \in \mathcal{G}_\Sigma$ if and only if G is unlabeled and connected and H ist the symmetric and transitive closure of G.
2. Let $G_0 \underset{P}{\Longrightarrow} G_1 \underset{P}{\Longrightarrow} \cdots \underset{P}{\Longrightarrow} G_k$ be a derivation such that G_k is P-*reduced*. Let n be the number of nodes of G_0. Then $k \leq n^2$.
3. The membership problems for $SEM(I_{connected})$ and $SEM(T_{connected})$ are decidable in polynomial time.

Hence, the unit *connected* can be effectively used as a graph class expression specifying in polynomial time the class of connected unlabeled graphs.

3.2. indegree = outdegree

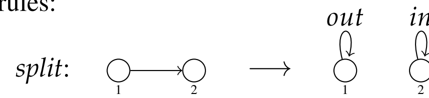
It is well known that in each connected directed graph G an Euler tour can be constructed if and only if for each node v in G the condition $indegree(v) = outdegree(v)$ is satisfied. This condition can be checked for unlabeled graphs with the graph transformation unit *indeg=outdeg* in Figure 5. Each derivation of *indeg=outdeg* starts with an unlabeled graph. The control condition allows all derivations over $\{split, rem\}$ and is therefore omitted. Each application of the rule *split* removes an unlabeled edge and adds an *out*-loop to the source of the edge and an *in*-loop to the target. The rule *rem* removes an *in*-loop as well as an *out*-loop from a node v. As the set of nodes is not changed during derivations, a derived graph that is terminal is discrete consisting of the nodes of the initial graph and, therefore, it is uniquely determined independent of the order of rule applications. Moreover, one may consider, for each node, the difference between the number of outgoing unlabeled edges plus the number of out-loops and the number of incoming unlabeled edges plus the number of in-loops. It turns out by a simple induction on the lengths of derivations that these differences are invariant. This means that $indegree = outdegree$ holds for an initial graph if and only if the application of rules as long as possible yields a discrete graph.

indeg=outdeg

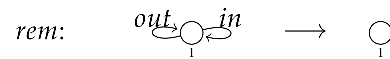
initial: $\{*\}$-*labeled*

rules:

split:



rem:



terminal: $\{split\}$-*reduced* & $\{*\}$-*labeled*

Figure 5 The graph transformation unit *indeg=outdeg*

Summarizing, we get the following results concerning correctness and complexity of *indeg=outdeg*.

Proposition 2 (correctness and complexity)**.** Let $P = \{split, rem\}$. Then the following holds:

1. Let $G \in \mathcal{G}_\Sigma$. Then $(G, H) \in SEM(indeg{=}outdeg)$ for some $H \in \mathcal{G}_\Sigma$ if and only if G is unlabeled, $indegree_G(v) = outdegree_G(v)$ for each $v \in V_G$ and H is the discrete graph consisting of the node set V_G and the empty edge set.

2. Let $G_0 \underset{P}{\Longrightarrow} G_1 \underset{P}{\Longrightarrow} \cdots \underset{P}{\Longrightarrow} G_k$ be a derivation such that G_k is P-*reduced*. Let m be the number of edges in G_0. Then $k \leq 2m$.

3. For $SEM(I_{indeg=outdeg})$ and $SEM(T_{indeg=outdeg})$, the membership problems are decidable in polynomial time.

Hence, the unit $indegree = outdegree$ can be effectively used as a graph class expression specifying in polynomial time the class of all directed unlabeled graphs satisfying the condition $indegree = outdegree$.

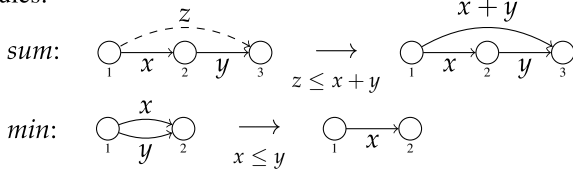
3.3. Shortest paths

Most shortest-path-algorithms like the ones by Floyd/Warshall (Floyd 1962; Warshall 1962) and by Dijkstra (Dijkstra 1959) are based on two elementary operations: the sequential composition of paths summing up the distances and keeping the path with minimum distance out of some parallel paths (i.e. paths with the same source and target nodes). The algorithms differ from each other by the order in which the two basic operations are applied. The graph transformation unit in Figure 6 models the computation of shortest paths (or more precisely their distances) by these two operations in arbitrary order.

shortest_paths

initial: 0-*looped*(*loop-free* & *strictly-simple* & $\mathbb{N}$-*labeled*)

rules:



terminal: $\{sum, min\}$-*reduced*

Figure 6 The graph transformation unit *shortest_paths*

The edges of the initial graphs are labeled with natural numbers representing the distances of direct connections between nodes. Each node is provided with a 0-loop representing the shortest paths of length and distance 0. Moreover, a pair of different nodes is connected by at most one edge so that this is a shortest path of length 1. The rules *sum* and *min* are applied to the initial graphs as long as possible in arbitrary order. An application of the *sum* rule matches a path of length 2 and creates a bridging edge with the sum of the distances of the two edges as distance provided that there is not yet a bridging edge with an equal or smaller distance. An application of the *min*-rule matches two parallel edges where the one with the minimum distance is kept, the other edge is removed. As nodes are neither created nor removed, the node set does not change during derivations.

It may be noted that the unit is infinite because we allow arbitrary large distances. But this causes no problems. For each initial graph, the number of potentially applicable rules is finite as the distances x, y, z in the rules can be bounded by the sum of distances of all edges.

The unit computes shortest distances of paths between each two nodes in a graph G where the distance of a path p is the sum of the distances of the edges on p and is denoted by $dist_G(p)$. This is stated in the following result.

Proposition 3 (Correctness)**.** Let $(G, H) \in SEM(shortest_paths)$. Then the following holds:

1. For every shortest path p from v to v' in G, there is some $e \in E_H$ with $s_H(e) = v$, $t_H(e) = v'$, and $l_H(e) = dist_G(p)$.

2. For every $e \in E_H$, there is a shortest path p from $s_H(e)$ to $t_H(e)$ in G with $l_H(e) = dist_G(p)$.

The first statement can be proved by induction on the lengths of shortest paths and the second one by an induction on the length of derivations.

What about complexity? The graph transformation unit *shortest_paths* is highly nondeterministic because, for every initial graph, the order of rule applications is not restricted. But to reach a terminal graph, a derivation must be prolonged as long as rules are applicable. In particular, a computation cannot get stuck before a result is obtained. Moreover, the correctness guarantees that the resulting terminal graph is unique independent of the order of rule applications. But the various derivations may have quite different lengths. Therefore, one may get the shortest distance more efficiently if one restricts the computation to special derivations by adding appropriate control conditions. Two specializations may illustrate the idea.

First, we mimic the well-known Floyd/Warshall algorithm ((Floyd 1962)). Assume that the nodes of some initial graph are numbered from 1 to n. Then, for $i = 1, \cdots, n$, we apply *sum* as long as possible with i as intermediate node of the path of length 2 in the left-hand side followed by *min* as long as possible. If the application of *sum* with intermediate node i is denoted by $sum(i)$, then the sketched control can be formally expressed by

$$(sum(i)!; min!)_{i=1}^{n}.$$

The length of a derivation according to this control is bounded by $2n^3$ because, for $i = 1, \ldots, n$, *sum* can be applied at most $(n-1)(n-2)$ times generating at most as many parallel edges so that *min* must be applied at most $(n-1)(n-2)$ times. In other words, the bound of the derivation length reflects exactly the known complexity of the cubic Floyd/Warshall algorithm.

Second, we model the shortest-paths algorithm by Mahr (Mahr 1982). The control condition looks similar:

$$(sum[ine]!; min!)^*$$

It means that, in an arbitrary number of rounds, one must apply *sum* as long as possible in the *ine*-mode followed by *min* as

long as possible. Here *ine* is the shortcut of *ignore-new-edges* and requires that *sum* is not allowed to match edges that are newly generated in the same round. Therefore, there are at most $n(n-1)(n-2)$ matches of *sum* in one round producing at most as many parallel edges, so that the number of following *min* steps is bounded by n^3, too. Consequently, the length of a derivation according to this control is bounded by $2 \cdot n^3 \cdot k$ where k is the number of rounds one needs to reach a reduced graph. By a proof quite similar to the proof of Proposition 3, one can show that the edges after l rounds represent the shortest paths of the initial graph of lengths up to 2^l. As there is always a shortest path the length of which is smaller than n if there is a shortest path at all, the result of computations is reduced after k steps latest with $2^k \leq n-1 < 2^{k+1}$. Therefore, the length bound coincides with the complexity bound of Mahr's algorithm which is shown to be of the order $n^3 \cdot \log n$.

4. Structured Graph Transformation Units

Simple graph transformation units allow the modeling of computational processes on graphs in the small. In order to divide large rule sets into smaller parts or to reuse already defined units within others, structuring concepts are needed. This gives rise to the concept of structured graph transformation units which import a set of graph transformation units so that an imported unit (maybe with many rules) can play the role of a single rule. In the following, we assume that the import structure is acyclic (cf. (H.-J. Kreowski et al. 1997) for graph transformation units with a more general import structure).

Definition 7 (Structured graph transformation unit)**.** The set *STRUCT* of *structured graph transformation units* is recursively defined as follows:

- Each simple graph transformation unit *gtu* is in *STRUCT*. The *import depth* of *gtu* is 0 and is denoted by $depth(gtu)$.
- Let $U \subseteq STRUCT$ with $\max\{depth(t) \mid t \in U\} = n$. Let $I, T \in \mathcal{E}$, $P \subseteq \mathcal{R}$, and let $C \in \mathcal{C}$ be a control condition over $P \cup U$. Then $(I, U, P, C, T) \in STRUCT$ with $depth(I, U, P, C, T) = n + 1$.

Structured graph transformation units transform initial graphs into terminal graphs by applying rules or imported units sequentially such that the control condition is satisfied. This means that structured graph transformation units have a binary relation as interleaving semantics.

Definition 8 (Semantics of structured graph transformation units)**.** The semantics of each $gtu \in STRUCT$ is given by $SEM(gtu) = (SEM(I) \times SEM(T)) \cap (\bigcup_{u \in U} SEM(u) \cup \underset{P}{\Longrightarrow})^* \cap SEM(C)$.

The following example of the construction of Euler tours illustrates how the structuring principle can be used. While the existence of an Euler tour implies the connectedness and the *indegree* = *outdegree* condition obviously, the converse is more difficult to see. One usual proof is mainly based on two algorithmic constructions: First, one gets a set of edge-disjoint simple cycles which cover all edges. Second, a particular traversal of the simple cycles yields an Euler tour.

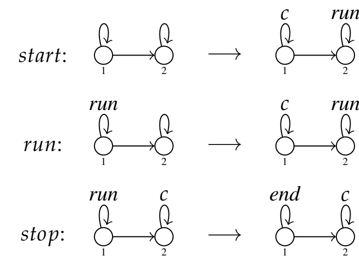
4.1. Constructing Euler tours

The explicit construction of Euler tours is useful in some applications such as the planning of delivering tours or for constructing tours with the approximation algorithm of Christofides for the Travelling Salesperson Problem (Christofides 1976). In order to construct an Euler tour in a directed Eulerian graph, we use the ideas of (Hierholzer 1873) and cover the graph with simple cycles which are then traversed in a special order, where a cycle $v_0 e_1 v_1 \cdots e_n v_n$ is *simple*, if $v_i \neq v_j$ for all $i, j \in \{1, \ldots n\}$ with $i \neq j$. The covering of the graph can be obtained by constructing successively simple cycles and labeling them with distinct numbers. The search of a simple cycle is performed by the graph transformation unit *simple_cycle* which uses the simple units *search*, *extract* and *rel* presented in the following.

The unit *search* is given in Figure 7. The initial and terminal graph class expressions allow all graphs. The rule *start* chooses an edge between two nodes with a $*$-loop and labels the loop of its source with c and the loop of its target with *run*. The rule *run* chooses a next edge from the node with the *run*-loop to a node with a $*$-loop, replaces the *run*-loop by a c-loop and the $*$-loop by a *run*-loop. The application of *stop* replaces the *run*-loop by an *end*-loop, if there is an edge from the node with the *run*-loop to some node with a c-loop. The rule *start* must be applied exactly once, then *run* arbitrarily often and finally *stop* exactly once.

search

rules:



cond: $start; run^*; stop$

Figure 7 The graph transformation unit search

Let G be an unlabelled graph with exactly one loop at each node, at least one edge between two distinct nodes and let G satisfy the *indegree* = *outdegree* condition. Let $G \overset{*}{\Longrightarrow} G'$ be a derivation the application sequence of which is a prefix of some word in $L(start; run^*; stop)$. Then there is a derivation $G \overset{*}{\Longrightarrow} G' \overset{*}{\Longrightarrow} H$ such that $(G, H) \in SEM(C_{search})$ which implies that $(G, H) \in SEM(search)$. The graph H is the result of labeling the loops of a path $v_0 e_1 v_e \cdots e_n v_n$, where $v_0 e_1 \cdots e_{n-1} v_{n-1}$ is a simple path (i.e., $v_i = v_j$ implies $i = j$ for all $i, j \in \{0, \ldots, n-1\}$) and $v_n = v_k$ for some $k \in \{0, \ldots, n-1\}$. The unit labels the loop of v_{n-1} with *end* and the loop of each other node in the path with c.

For example, the unit *search* can be applied to the upper graph of Figure 8 with the lower graph as a possible result

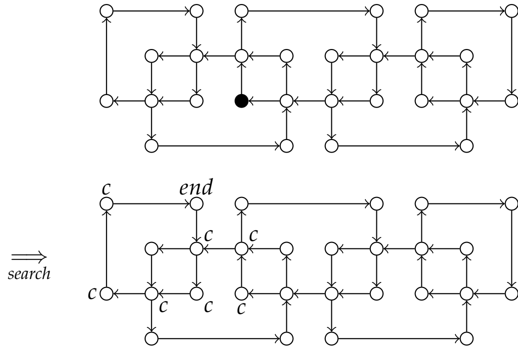


Figure 8 An execution of *search*

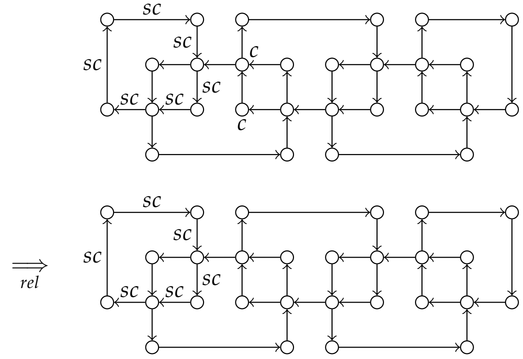


Figure 10 An execution of *rel*

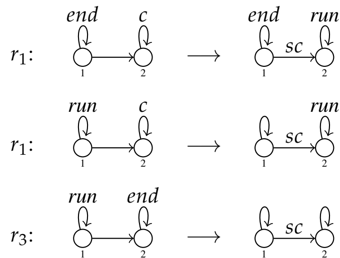
where for a better readability the loops are omitted and the loop labels (apart from $*$) are written next to the corresponding nodes. The filled node represents the match of node 1 in the application of the rule *start*.

Since every rule application reduces the (finite) number of unlabeled loops, the maximum length of every derivation is linear in the number of nodes of the initial graph if each node of the initial graph carries at most one unlabeled loop.

The unit *extract* given in Figure 9 is similar to *search*.

extract

rules:

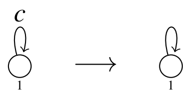


cond: $r_1; r_2^*; r_3$

Figure 9 The graph transformation unit extract

If it is applied to the lower graph of Figure 8, it replaces the edges of the simple cycle found by the unit *search* by *sc*-edges and the *c*-loops at the nodes of the cycle by $*$-loops. This yields the upper graph in Figure 10.

The simple unit *rel* is defined as $(all, \{r\}, r!, all)$ where r is the rule



Its application to the upper graph of Figure 10 yields the lower graph in the figure.

The unit *simple_cycle* applies the three units *search*, *extract* and *rel* in this order, i.e.,

$$simple_cycle = (all, \{search, extract, rel\}, \emptyset, search; extract; rel, all).$$

The covering of the graph with simple cycles is performed by the unit *cycle_cover* given in Figure 11. The initial graph class expression requires that the initial graph is unlabeled, connected and satisfies the $indegree = outdegree$ property. Moreover, each node is equipped with exactly one unlabeled loop. At first, a counter with the value 0 is generated by the rule *gen_counter*. The counter is a node with a *counter*-loop depicted simply as a square node. Afterwards, the following procedure is repeated as long as possible. The counter is increased by 1 if there is an unlabeled edge left (rule *check*); then the unit *simple_cycle* labels an unlabeled simple cycle with *sc* and afterwards the unit *num* (defined below) labels each edge of this cycle with the value of the counter. When the rule *check* cannot be applied anymore the counter is removed by the rule *rem_counter*.

cycle_cover

initial: $*$-*looped*$(\{*\})$-*labeled* & *loop-free* & *connected* & *indeg=outdeg*$)$

uses: *simple_cycle*, *num*

rules:

gen_counter: $\emptyset \longrightarrow$ □ 0

check: ○→○ (1, 2), □ k (3) $\longrightarrow$ ○→○ (1, 2), □ $k+1$ (3)

rem_counter: □ k $\longrightarrow \emptyset$

cond: *gen_counter*; (*check*; *simple_cycle*; *num*)!; *rem_counter*

Figure 11 The graph transformation unit *cycle_cover*

The simple unit *num* is defined as $(all, \{r\}, r!, all)$ with

r: $\bigcirc_1 \xrightarrow{sc} \bigcirc_2 \quad \square_3\ k \quad \longrightarrow \quad \bigcirc_1 \xrightarrow{k} \bigcirc_2 \quad \square_3\ k$

By induction one can show that the unit *cycle_cover* covers the initial graph with edge-disjoint simple cycles. If *cycle_cover* is applied to the upper graph of Figure 8, the graph of Figure 12 is a possible output graph.

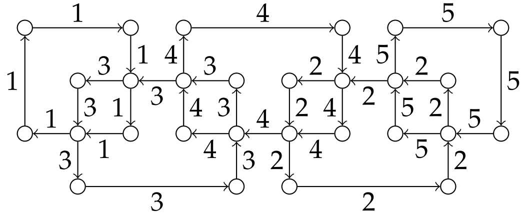


Figure 12 A result of *cycle_cover*

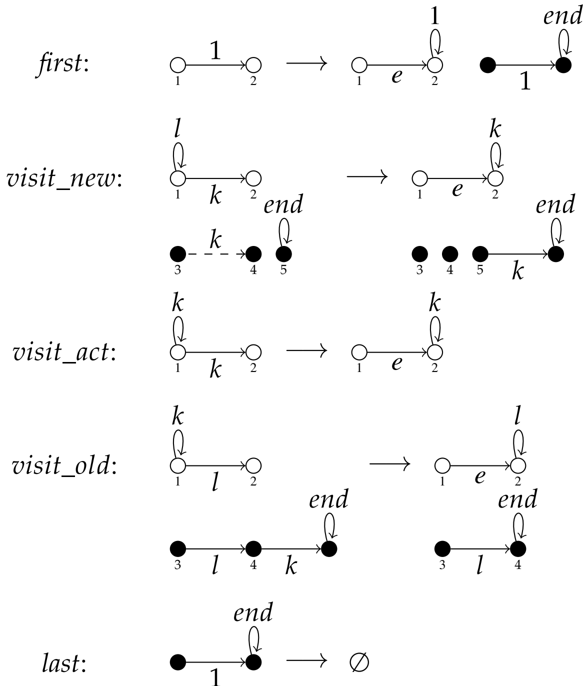
The structured transformation unit *EulerTour* is given in Fig. 13.

EulerTour

initial: $*$-*looped*($\{*\}$-*labeled* & *loop-free* & *connected* & *indeg*=*outdeg*)

uses: *cycle_cover*

rules:



cond: *cycle_cover*;
first; (*visit_new* > *visit_act* > *visit_old*)!; *last*

Figure 13 The structured transformation unit *EulerTour*

The rule *first* selects some edge from the 1-cycle and labels it with *e*. This *e*-edge is the first edge of the Euler tour to be

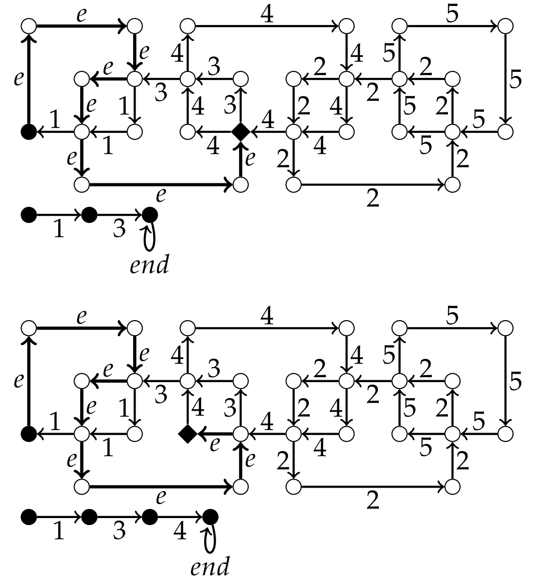


Figure 14 An execution of *visit_new*

constructed. Simultaneously, the rule generates a string graph which represents the sequence 1 meaning that cycle 1 is the first visited. Technically, the filled round nodes represent nodes with a *string*-loop. The rule *visit_new* is applied when the current tour meets a cycle not yet visited. In this case the tour continues on this new cycle and the number of the new cycle is appended to the string graph. Please note that the dashed edge means that no *k*-edge must occur in the string graph. According to the priority control condition, the rule *visit_act* can be applied when *visit_new* is not applicable. It continues the tour on the actual cycle. If neither *visit_new* nor *visit_act* is applicable, the *visit_old* rule may be applied continuing on the last visited circle and deleting the last edge from the string graph (together with its target). When the tour is complete, the rule *last* is applied in order to remove the remaining of the string graph.

Fig. 14 shows a rule application of *EulerTour*. The uppermost filled round node indicates the beginning of the already constructed tour and diamond represents its end.

From the diamond in the upper graph the tour must proceed on cycle 4 which is modeled by the rule application of *visit_new* leading to the lower graph of Fig. 14. After four further steps on cycle 4, the tour continues on cycle 2, enters cycle 5 and after going along the whole cycle 5 it reenters cycles 2, 4, 3, and 1 in this order and ends at the starting point.

Altogether, the execution of the unit *EulerTour* visualizes the construction of an Euler tour in an unlabeled and loop-free connected graph that satisfies *indegree* = *outdegree*.

5. Parallel Rule Application

The framework of graph transformation offers concepts and results concerning the parallel application of rules. From the point of view of graph algorithms, this is of great interest because it provides a machinery for the parallelization of graph algorithms. As far as needed in this section, some basic notions and facts on parallel rule application are recalled in the following (see,

e.g., (Ehrig & Kreowski 1976; Ehrig et al. 2006; H. Kreowski et al. 2018)). To demonstrate the potentials, we show how by means of massive parallelism the *indegree* = *outdegree* condition can be checked in a constant number of steps and shortest paths can be computed in a logarithmic number of steps. The example units are the parallel versions of *indeg=outdeg* and *shortest_paths* in Section 3.

Definition 9 (Parallel rule application)**.** 1. Let $P \subseteq \mathcal{R}$ and let $r_i = (L_i \supseteq K_i \subseteq R_i) \in P$ for $i = 1, \ldots, n$. Then the *parallel rule* $p = r_1 + \cdots + r_n = \sum_{i=1}^{n} r_i$ is given by the disjoint unions of the components $(\sum_{i=1}^{n} L_i \supseteq \sum_{i=1}^{n} K_i \subseteq \sum_{i=1}^{n} R_i)$.

2. Let $r = (L \supseteq K \subseteq R)$ and $r' = (L' \supseteq K' \subseteq R')$ be two rules and let $G \underset{r}{\Longrightarrow} H$ and $G \underset{r'}{\Longrightarrow} H'$ be two direct derivations w.r.t. $g\colon L \to G$ and $g'\colon L' \to G$. Then the direct derivations are *parallel independent* if the corresponding matches intersect in gluing items only, i.e., $g(L) \cap g'(L) \subseteq g(K) \cap g'(K)$.

Facts. Let $r_i = (L_i \supseteq K_i \subseteq R_i) \in P$ for $i = 1, \ldots, n$ and $p = (L \supseteq K \subseteq R) = \sum_{i=1}^{n} r_i$ be the corresponding parallel rule. Then the following holds.

1. Let $G \underset{p}{\Longrightarrow} X$ be a direct derivation w.r.t. $g\colon L \to X$. Then there is a sequential derivation $G = G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n = X$ where the morphism of the first step is $g_1 = g|L_1$ (where $g|L_1$ is the restriction of g to L_1).
2. Let $G \underset{r_i}{\Longrightarrow} H_i$ for $i = 1, \ldots, n$ be direct derivations w.r.t. $g_i\colon L_i \to G$. Let each two of them be parallel independent. Then there is a direct derivation $G \underset{p}{\Longrightarrow} X$ w.r.t. $g\colon L \to G$ defined by $g|L_i = g_i$ for $i = 1, \ldots, n$.

The first fact states that the graph obtained by the application of a parallel rule can also be derived by applying the component rules sequentially. As the disjoint union of graphs is commutative, the fact holds for every order of the atomic rules. The second fact is the key for the parallelization of graph algorithms. It states that the direct derivation of a parallel rule can be constructed from matches of the atomic component rules, and these matches may be found in parallel. If one can make sure that these direct derivations are pairwise independent, the complexity of the parallel step is of the same order as the complexity of the ordinary steps. The following example illustrates how a graph algorithm may be improved by parallelization. It uses some new control conditions for parallel derivations defined as follows.

Definition 10 (Control conditions for parallel derivations)**.** Let $r_i = (L_i \supseteq K_i \subseteq R_i) \in P$ for $i = 1, \ldots, n$ and let $p = (L_p \supseteq K_p \subseteq R_p) = \sum_{i=1}^{n} r_i$ be the corresponding parallel rule. Let $G \underset{p}{\Longrightarrow} X$ be a direct derivation w.r.t. $g\colon L_p \to G$.

1. $G \underset{p}{\Longrightarrow} X$ is *maximum parallel* if there is no $r = (L \supseteq K \subseteq R) \in P$ such that $p + r$ is applicable to G w.r.t. $\overline{g}\colon (L_p + L) \to G$ with $\overline{g}|L_p = g$.
2. $G \underset{p}{\Longrightarrow} X$ is *double-free* if there is no pair $i \neq j$ with $r_i = r_j$ and $g|L_i = g|L_j$.
3. $G \underset{p}{\Longrightarrow} X$ is *double-free maximum parallel* if it is double-free and there is no $r = (L \supseteq K \subseteq R) \in P$ such that a double-free derivation $G \underset{p+r}{\Longrightarrow} Y$ exists w.r.t. $\overline{g}\colon L_p + L \to G$ with $\overline{g}|L_p = g$.
4. $G \underset{p}{\Longrightarrow} X$ is *larger* than another application of a parallel rule $G \underset{q}{\Longrightarrow} Y$ if p consists of more atomic rules than q.

5.1. indegree = outdegree in parallel

The following graph transformation unit checks the *indegree* = *outdegree* condition by means of massive parallelism. The control condition requires that at first the rule *split* and then the rule *rem* be applied with maximum parallelism.

indeg=outdeg_in_parallel
initial: $\{*\}$-*labeled*
rules: *split*, *rem*
cond: $split[maxpar]; rem[maxpar]$
terminal: $\{split, rem\}$-*reduced* & $\{*\}$-*labeled*

Obviously, all derivations in this unit consist of two applications of parallel rules whereas the lengths of derivations in its sequential counterpart are between $|E_G|$ and $2 \cdot |E_G|$ where $|E_G|$ denotes the number of edges in the initial graph.

5.2. Shortest paths in parallel

The following graph transformation unit computes shortest paths by means of massive parallelism. The rules *sum* and *min* are already used in the *shortest_paths* unit in Section 3. The control condition requires that, repeatedly, the *sum* rule be applied with double-free maximum parallelism followed by the largest maximum parallel application of the *min* rule.

shortest_paths_in_parallel
initial: 0-*looped*(*loop-free* & *strictly-simple* & $\mathbb{N}$-*labeled*)
rules: *sum*, *min*
cond: $(sum[\textit{double-free maxpar}]; min[\textit{largest maxpar}])^*$
terminal: $\{sum, min\}$-*reduced*

The initial graphs have a 0-loop at each node and no parallel edges so that the *min* rule cannot be applied and the present edges represent the shortest paths of length 0 and 1. As the left-hand side of the *sum* rule coincides with the gluing graph, each two applications of *sum* are parallel independent. The negative application condition prevents that one edge of the left-hand side of *sum* is matching with a 0-loop. Therefore, there are

at most $n \cdot (n-1) \cdot (n-2)$ double-free applications of *sum* where n is the number of nodes in the initial graph. And a *sum* step is double-free maximum parallel if each path of length 2 from v to v' is matched once provided that there is no edge from v to v' with a distance shorter than the sum of distances of the edges on the path. Each application of *sum* may create a parallel edge. The following largest maximum parallel *min* step makes sure that no two parallel edges are left. More precisely, two *min* applications are parallel independent if they match four different edges or intersect in the edge that is kept. Therefore, whenever there are m parallel edges between two nodes, the largest parallel step removes $m-1$ of them, and this happens if all applications of *min* choose the same edge to be kept.

To illustrate how this parallel algorithm works, Fig. 15 shows the results after each of three rounds starting with the left upper graph. For a better readability, we omit all 0-loops and represent newly inserted edges dashed.

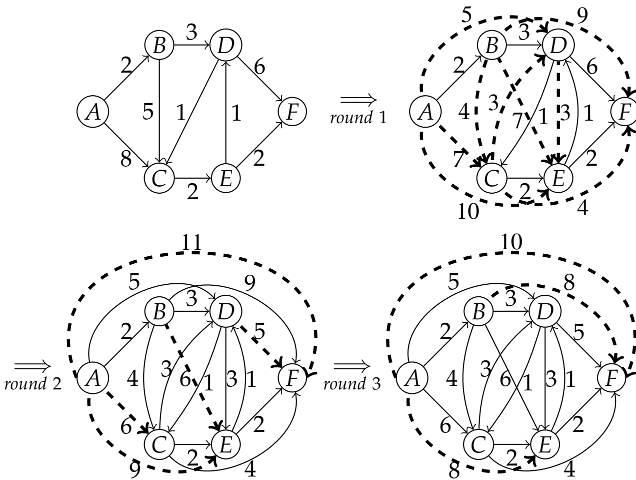


Figure 15 An execution of *shortest_paths_in_parallel*

That the unit computes the shortest distances between each two nodes can be seen as follows. The initial and terminal graphs are the same as in the unit *shortest_paths* in Section 3. A parallel derivation from an initial to a terminal graph can be sequentialized due to the fact above. In this sequential derivation, a *sum* application may occur that does not obey the negative application condition. But then there is already an edge as good as or better than the edge that would be generated by *sum*. Hence this step can be omitted together with the *min* step later on that removes this superfluous edge without changing the result. If the sequential derivation is modified in this way as long as possible, then we end up with a derivation from an initial to a terminal graph in *shortest_paths*. Conversely, consider a derivation of *shortest_paths* from an initial graph G to a terminal graph H. Without loss of generality, one can assume that it follows the second case of control $(sum[ine]!; min!)^*$ defined in Section 3. It is easy to see that all the applications of the *sum* rule in one round according to the *ine* mode are sequentially independent. Therefore, they can be applied in parallel. Moreover, this is maximum parrallel as the *sum* rule is applied as long as possible in one round. The following applications of the *min* rule as long as possible end up with the minimal edge between each two nodes so that one gets the same result as a largest parallel application of the *min* rule. This means that each round according to the *ine* mode can be replaced by two derivation steps in *shortest_paths_in_parallel*. This implies also that after a logarithmic number of parallel steps the terminal graph is reached.

Summarizing, this proves the following result about the correctness and complexity of the unit *shortest_paths_in_parallel*.

Proposition 4 (Correctness and complexity)**.** Let

$$G_0 \underset{parsum}{\Longrightarrow} G_1 \underset{parmin}{\Longrightarrow} \cdots \underset{parsum}{\Longrightarrow} G_{2k-1} \underset{parmin}{\Longrightarrow} G_{2k}$$

be a derivation in *shortest_paths_in_parallel* from an initial graph to a terminal graph with alternating parallel *sum* and *min* steps according to the control condition. Then the following holds.

1. For every shortest path p from v to v' in G_0, there is some $e \in E_{G_{2k}}$ with $s_{G_{2k}}(e) = v$, $t_{G_{2k}}(e) = v'$, $l_{G_{2k}}(e) = dist_{G_0}(p)$.
2. For every $e \in E_{2k}$, there is a shortest path p from $s_{G_{2k}}(e)$ to $t_{G_{2k}}(e)$ in G_0 with $l_{G_{2k}}(e) = dist_{G_0}(p)$.
3. Let n be the number of nodes in G_0. Then the length $2k$ of the given derivation has a logarithmic bound, i.e., $2^k \leq n-1$.

The same parallelization principle may be employed to the unit *shortest_paths* with respect to the control that mimics the Floyd/Warshall algorithm. Again, all the *sum*-applications (where the intermediate node is fixed) can be applied in parallel and all the following *min*-applications also. But this must be repeated for every node such that the derivation length becomes $2n$.

To our knowledge, this kind of parallelization technique is not yet used within the area of graph transformation, and we are not aware of much work in this direction on the level of modeling outside graph transformation.

6. Related Work

In the literature, there are some other graph-transformational approaches to the modeling of graph algorithms. One of them are graph relabeling systems presented in, e.g., (Litovski et al. 1999b). They are well suited to model, visualize and verify distributed algorithms in communication networks. A particular feature of graph relabeling systems is that the application of a rule preserves the structure of the networks, but changes the labels of nodes and edges. In (Plump 2012; Campbell et al. 2019) it is shown how the graph programing language GP can be used to model graph algorithms. Graph programs are defined inductively, where basic graph programs are rule schemata specifying derivation relations and composed graph programs are constructed via while-loops, sequential composition and the

operator as long as possible. To our knowledge, GP is the only other graph transformation approach that is used systematically in the context of graph algorithms but without parallelism.

In this paper, we have considered some typical examples of graph class expressions. Further well-studied frameworks that may be used to provide graph class expressions in transformation units are for example typed graphs (Ehrig et al. 2006), monadic second order logic formulas (Courcelle & Engelfriet 2012), or nested graph conditions (Habel & Pennemann 2009).

There are various sophisticated graph transformation tools that provide control conditions such as AGG (Taentzer 2003), GrGen.NET (Jakumeit et al. 2010) or GROOVE (Ghamarian et al. 2012) (see also (Jakumeit et al. 2014) for a comparison of graph transformation tools). Hence, the tools are promising candidates for implementing graph transformation units. In (Luderer 2016) stepwise control conditions are studied. They guarantee that no derivation step violates the control condition so that the degree of nondeterminism can be reduced dramatically.

Graph transformation units can be considered as a declarative conception of model-to-model transformation like, e.g., triple graph grammars (see, e.g., (Anjorin et al. 2015)). The latter ones are one of the modeling languages that are compared which each other with respect to rule inheritance in (Wimmer et al. 2012). As graph transformation units and triple graph grammars have much in common, the results for triple graph grammars may carry over to graph transformation units.

In the literature one finds quite a variety of graph transformation approaches (cf. (Rozenberg 1997) for some typical examples). All of them employ special types of graphs and rule application principles and use – if at all – particular kinds of initial and terminal graphs as well as of control conditions to regulate the derivation process. In contrast to that, the framework of graph transformation units is approach-independent so that all particular approaches can be used as underlying rule bases in combination with the generic concepts of graph class expressions and control conditions. Moreover, a graph transformation unit embraces a rule set, a set of used units, specifications of input and output graphs, and a control condition while such components are kept and studied separately in most other graph-transformational frameworks (or some of them are not considered at all).

Besides the tools mentioned above, there are several further graph transformation tools like Atom3[2], eMoflon[3], Fujaba[4], Henshin[5], and ViaTra[6]. While they are mainly tailored to software development and model transformation, it may be worthwhile to examine whether they can be employed for the implementation of graph transformation units.

7. Conclusion

In this article, we have proposed the framework of graph transformation units as a general approach to the modeling and analysis of graph algorithms. The framework provides a common syntactic description of graph algorithms in a visual, rule-based, and structured manner, and a precise computational semantics given by iterated rule applications, which are also called derivations. Moreover, it provides an inductive proof schema to prove properties including correctness based on the lengths of derivations (in addition to other induction variables like the sizes of graphs or the lengths of paths), and a common complexity measure given by the lengths of derivations, and finally, it provides the prospect of tool support for testing, simulating, visualizing, and verifying graph algorithms that are modeled as graph transformation units by adapting graph-transformational tools to the processing of graph transformation units.

To shed more light on the significance of our proposal, future research may include the following topics.

1. Further case studies to get a better feeling and understanding how the modeling, structuring, and analyzing of graph algorithms can be done in a convenient and adequate way,
2. expansion of our considerations to NP-complete graph problems and their heuristic and approximating solutions,
3. further exploration of the parallelization technique as it offers a quite general way to improve the efficiency of algorithms,
4. standardization of the features of graph transformation units including the underlying rule bases, the graph class expressions and the control conditions to turn the framework into a proper modeling language like UML, and
5. the development of translators from graph transformation units into the input formats of graph-transformational tools so that they can be used systematically.

Acknowledgments

We are grateful to Aaron Lye and Aljoscha Windhorst for their comments on various aspects of this paper. Moreover, we would like to thank the anonymous reviewers for their valuable hints that led to several improvements.

References

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., ... Taentzer, G. (1999). Graph transformation for specification and programming. *Science of Computer Programming*, *34*(1), 1–54.

Anjorin, A., Leblebici, E., & Schürr, A. (2015). 20 years of triple graph grammars: A roadmap for future research. *ECEASST*, *73*. Retrieved from https://doi.org/10.14279/tuj.eceasst.73.1031 doi: 10.14279/tuj.eceasst.73.1031

Campbell, G., Courtehoute, B., & Plump, D. (2019). Linear-Time Graph Algorithms in GP 2. In M. Roggenbach & A. Sokolova (Eds.), *8th conference on algebra and coalgebra in computer science (calco 2019)* (Vol. 139, pp. 16:1–16:23). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. Retrieved from http://drops.dagstuhl.de/opus/volltexte/2019/11444 doi: 10.4230/LIPIcs.CALCO.2019.16

2 http://atom3.cs.mcgill.ca
3 https://emoflon.org
4 https://web.cs.upb.de/archive/fujaba/projects.html
5 https://www.eclipse.org/henshin/
6 https://www.eclipse.org/viatra/

Christofides, N. (1976). *Worst-case analysis of a new heuristic for the travelling salesman problem* (Tech. Rep. No. 388). Graduate School of Industrial Administration, Carnegie Mellon University (CMU).

Courcelle, B., & Engelfriet, J. (2012). *Graph structure and monadic second-order logic - A language-theoretic approach* (Vol. 138). Cambridge University Press.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of algebraic graph transformation*. Springer.

Ehrig, H., Ermel, C., Golas, U., & Hermann, F. (2015). *Graph and model transformation - general framework and applications*. Springer.

Ehrig, H., & Kreowski, H.-J. (1976). Parallelism of manipulations in multidimensional information structures. In *Proc. mathematical foundations of computer science* (Vol. 45, pp. 284–293).

Ermler, M., Kreowski, H.-J., Kuske, S., & von Totth, C. (2012). From graph transformation units via MiniSat to GrGen.NET. In A. Schürr, D. Varró, & G. Varró (Eds.), *Proc. international symposium applications of graph transformations with industrial relevance (agtive 2011)* (Vol. 7233, p. 153-168).

Floyd, R. W. (1962). Algorithm 97 (shortest path). *Communications of the ACM*, *5*(6), 345.

Ghamarian, A. H., de Mol, M., Rensink, A., Zambon, E., & Zimakova, M. (2012). Modelling and analysis using GROOVE. *International Journal of Software Tools for technoloqy Transfer*, *14*(1), 15–40. doi: 10.1007/s10009-011-0186-x

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

Habel, A., & Pennemann, K. (2009). Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, *19*(2), 245–296. doi: 10.1017/S0960129508007202

Hierholzer, C. (1873). Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Ann.*, *6*, 30–32.

Jakumeit, E., Buchwald, S., & Kroll, M. (2010). GrGen.Net - the expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer*, *12*(3-4), 263–271.

Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Hegedüs, Á., Herrmannsdörfer, M., … Mazanek, S. (2014). A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, *85*, 41–99. doi: 10.1016/j.scico.2013.10.009

Janssens, D., Kreowski, H.-J., & Rozenberg, G. (2005). Main concepts of networks of transformation units with interlinking semantics. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, & G. Taentzer (Eds.), *Formal methods in software and system modeling* (Vol. 3393, pp. 325–342). Springer Verlag.

Kreowski, H., Kuske, S., & Lye, A. (2018). A simple notion of parallel graph transformation and its perspectives. In R. Heckel & G. Taentzer (Eds.), *Graph transformation, specifications, and nets - in memory of hartmut ehrig* (Vol. 10800, pp. 61–82). Springer.

Kreowski, H.-J., & Kuske, S. (1996). On the interleaving semantics of transformation units — a step into GRACE. In J. E. Cuny, H. Ehrig, G. Engels, & G. Rozenberg (Eds.), *Proc. graph grammars and their application to computer science* (Vol. 1073, pp. 89–108).

Kreowski, H.-J.., & Kuske, S. (1999). Graph transformation units and modules. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Handbook of graph grammars and computing by graph transformation, vol. 2: Applications, languages and tools* (pp. 607–638). Singapore: World Scientific.

Kreowski, H.-J., & Kuske, S. (1999). Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, *11*(6), 690–723.

Kreowski, H.-J., Kuske, S., & Lye, A. (2018). A simple notion of parallel graph transformation and its perspectives. In R. Heckel & G. Taentzer (Eds.), *Graph transformation, specifications, and nets (festschrift in memory of hartmut ehrig)* (Vol. 10800).

Kreowski, H.-J., Kuske, S., & Rozenberg, G. (2008). Graph transformation units – an overview. In P. Degano, R. D. Nicola, & J. Meseguer (Eds.), *Concurrency, graphs and models* (Vol. 5065, pp. 57–75). Springer.

Kreowski, H.-J., Kuske, S., & Schürr, A. (1997). Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, *7*(4), 479–502.

Kuske, S., Gogolla, M., Kreowski, H.-J., & Ziemann, P. (2009). Towards an integrated graph-based semantics for UML. *Software and Systems Modeling*, *8*(3), 385–401.

Litovski, I., Métivier, Y., & Sopena, É. (1999a). Graph relabelling systems and distributed algorithms. In H. Ehrig, H.-J. Kreowski, U. Montanari, & G. Rozenberg (Eds.), *Handbook of graph grammars and computing by graph transformation, vol. 3: Concurrency, parallelism, and distribution* (pp. 1–56). Singapore: World Scientific.

Litovski, I., Métivier, Y., & Sopena, É. (1999b). Graph relabelling systems and distributed algorithms. In H. Ehrig, H.-J. Kreowski, U. Montanari, & G. Rozenberg (Eds.), (pp. 1–56). Singapore: World Scientific.

Luderer, M. (2016). *Control conditions for transformation units : Parallelism, as-long-as-possible, and stepwise control.* Bremen. (PhD thesis)

Mahr, B. (1982). Algebraic complexity of path problems. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, *16*(3), 263-292.

Mehlhorn, K., & Näher, S. (1999). *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press.

Plump, D. (2009). The graph programming language GP. In S. Bozapalidis & G. Rahonis (Eds.), *Algebraic informatics, third international conference (cai 2009), proceedings* (Vol. 5725, pp. 99–122). Springer-Verlag.

Plump, D. (2012). The design of GP 2. In *Proc. international workshop on reduction strategies in rewriting and programming (wrs 2011)* (Vol. 82, pp. 1–16).

Rozenberg, G. (Ed.). (1997). *Handbook of graph grammars and computing by graph transformation, vol. 1: Foundations*. Singapore: World Scientific.

Taentzer, G. (2003). AGG: A graph transformation environment for modeling and validation of software. In *Applications of graph transformations with industrial relevance* (pp. 446–453). Springer.

Warshall, S. (1962). A theorem on boolean matrices. *Journal of the ACM*, *9*(1), 11–12.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., ... Wagelaar, D. (2012). Surveying rule inheritance in model-to-model transformation languages. *J. Object Technol.*, *11*(2), 3: 1–46. Retrieved from https://doi.org/10.5381/jot.2012.11.2.a3 doi: 10.5381/jot.2012.11.2.a3

About the authors

Hans-Jörg Kreowski Hans-Jörg Kreowski is professor emiritus for Theoretical Computer Science at the University of Bremen, Germany. He is a member of the Centre for Computing and Communication Technologies (TZI) and of the interdisciplinary Bremen Research Cluster for Dynamics in Logistics (LogDynamics). His main research topic is rule-based modeling including graph transformation, algebraic specification, syntactic methods in picture generation. and occassional contributions to formal language theory and the theory. Besides the systematic research in theoretical computer science, some research activities have been directed towards the potential applications of rule-based methods in software engineering, data base systems, computer graphics, artificial intelligence, DNA computing, and – most important – in logistics. Moreover, he has been engaged in various aspects of computer and society over the last 40 years. You can contact the author at kreo@uni-bremen.de or visit www.informatik.uni-bremen.de/theorie.

Sabine Kuske Sabine Kuske is a lecturer and research assistant at the university of Bremen in Germany. She is interested in rule based graph transformation, Petri nets and graph algorithms. You can contact the author at kuske@uni-bremen.de or visit www.informatik.uni-bremen.de/~kuske.