Graph Transformation Units – An Overview*

Hans-Jörg Kreowski[1], Sabine Kuske[1], and Grzegorz Rozenberg[2]

[1] University of Bremen, Department of Computer Science
P.O. Box 33 04 40, 28334 Bremen, Germany
{kreo,kuske}@informatik.uni-bremen.de
[2] Leiden University,
Leiden Institute for Advanced Computer Science (LIACS)
2333 CA Leiden, The Netherlands
rozenber@liacs.nl

Abstract. In this paper, we give an overview of the framework of graph transformation units which provides syntactic and semantic means for analyzing, modeling, and structuring all kinds of graph processing and graph transformation.

1 Introduction

Graphs are used in computer science in many forms and contexts, and for many purposes. Well-known examples are Petri nets, entity-relationship diagrams, UML diagrams, and state graphs of finite automata. In many applications, graphs are not of interest as singular entities, but as members of graph and diagram languages, as states of processes and systems, as structures underlying algorithms, as networks underlying communication, tour planning, production planning, etc. Therefore, there is genuine need to generate, recognize, process, and transform graphs. The development and investigation of means and methods to achieve this goal is the focus of the area of graph transformation. (See the Handbook of Graph Grammars and Computing by Graph Transformation [1–3].)

In this paper, we give an overview of the framework of graph transformation units which provides syntactic and semantic means for analyzing, modeling, and structuring all kinds of graph processing and graph transformation. In particular, graph transformation units can be used to generate graph languages, to specify graph algorithms, to transform models whenever they are represented by graphs, and to define the operational semantics of data processing systems whenever the data and system states are given as graphs.

Graph transformation units encapsulate rules and control conditions that regulate the application of rules to graphs including the specification of initial and

* The first two authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

P. Degano et al. (Eds.): Montanari Festschrift, LNCS 5065, pp. 57–75, 2008.
© Springer-Verlag Berlin Heidelberg 2008

terminal graphs. To support the re-use and the stepwise development of graph transformation units, also a structuring principle is introduced that allows the import of units by units. The operational semantics of a unit transforms initial graphs into terminal graphs by interleaving rule applications and calls of imported units so that the control condition is fulfilled.

An important aspect of this framework is its approach independence, meaning that all introduced concepts work for all kinds of graphs, rules, rule applications, and control conditions. In particular, one can build new approaches from given ones through the product of approaches. In this way, graph transformation units can be combined in such a way that tuples of graphs are transformed componentwise. As a consequence, the interleaving semantics covers computable relations on graphs with m input graphs and n output graphs for arbitrary numbers m, n rather than binary relations between initial and terminal graphs.

This overview is organized in the following way. In Section 2, we recall the basic notions of graphs and of rule-based graph transformation. In Section 3, simple graph transformation units encompassing rules and control mechanisms are introduced, while a structuring principle by means of import is added in Section 4. Section 5 addresses the idea of approach independence and the product type. In the last section we discuss some related work and point out some further aspects such as the iterated interleaving semantics in case of cyclic import, the interlinking semantics that supplements the sequential interleaving semantics by parallel and concurrent elements, and autonomous units which interact in a common graph environment.

2 Graphs and Rule-Based Graph Transformation

In this section, we recall the basic notion of graphs, matches, rules, and rule application as they are needed to model the examples of this paper.

Graphs are quite generic structures which are 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 (see 2.1). As a running example, we consider the generation of simple paths and related problems including the search for Hamiltonian paths. Please note that the search for simple paths of certain lengths and in particular the search for Hamiltonian paths are NP-complete problems.

Graphs are often considered as inputs of algorithms and processes so that methods are needed to search and manipulate graphs. Graphs may also represent states of systems so that methods for updates and state transitions are needed. Also, graphs may often be used to specify the structure of all the data of interest, e.g., the set of all connected and planar graphs. Like in the case of string languages, one needs then mechanisms to generate and recognize graph languages. To meet all these needs, rule-based graph transformation is defined in 2.4 to 2.6.

2.1 Graphs

Let Σ be a set of labels. A *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 $\mathcal{G}_\Sigma$. We reserve a specific label $*$ which is omitted in drawings of graphs. In this way, graphs where all edges are labeled with $*$ may be seen as *unlabeled graphs*.

Example 1. Consider the graphs G_0, G_1, G_{12}, G_{123}, and G_{1234} in Figure 1.

A box □ represents a node with an unlabeled loop. Therefore, G_0 has four nodes, four loops and five additional unlabeled edges. The other graphs are variants of G_0. We use ⦿ to represent a *begin*-node which is a node with a loop labeled with *begin*. Analogously, ○ represents an *end*-node, and ◎ represents a node with a *begin*-loop and an *end*-loop. If one starts in the *begin*-node and follows the p-labeled edges, one reaches the *end*-node in the graphs G_{12}, G_{123}, and G_{1234}. In each case, the sequence of p-edges defines a simple path of G_0, where the intermediate nodes have no loops. In G_1, the *begin*-node and the *end*-node are identical which means that the corresponding simple path has length 0.

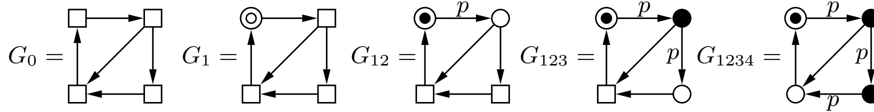


Fig. 1. G_0 with some of its simple paths

If one numbers the nodes of G_0 clockwise by 1 to 4 starting in the upper left-most corner, then the node sequences 1, 12, 123, and 1234 define simple paths of G_0 that correspond to the simple paths in G_1, G_{12}, G_{123}, and G_{1234}, resp. In this way, every simple path s of G_0 can be represented by a graph G_s. The set of all those graphs is denoted by $SP(G_0)$. The next subsection describes paths more formally.

2.2 Paths

One of the most important concepts concerning graphs is the notion of a path that is the subject of many research problems and applications of graphs such as connectivity, shortest paths, long simple paths, Eulerian paths, Hamiltonian paths, traveling salesperson problem, etc.

Given a graph $G = (V, E, s, t, l)$, a path from node v to node v' is a sequence of edges $p = e_1 \dots e_n$ with $n \geq 1$, $s(e_1) = v$, $s(e_i) = t(e_{i-1})$ for $i = 2, \dots, n$, and $t(e_n) = v'$. The *length* of p is n. Moreover, the empty sequence λ is considered to be a path from v to v of length 0 for each $v \in V$. A path $p = e_1 \dots e_n$ from v

to v' *visits* the nodes $V(p) = \{v\} \cup \{t(e_i) \mid i = 1, \ldots, n\}$. A path p is *simple* if it visits no node twice, i.e., $\#V(p) = length(p)+1$. [1] A simple path is *Hamiltonian* if it visits all nodes, i.e., $\#V(p) = \#V$.

2.3 Graph Morphisms, Subgraphs, and Matches

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 $e \in E_G$.

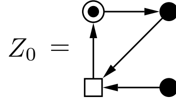
If the mappings g_V and g_E are bijective, then g is an *isomorphism*, and G and H are called *isomorphic*.

If the mappings g_V and g_E 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.

Example 2. There is a graph morphism from the graph L_{run} = ○→□ into some $G_s \in SP(G_0)$ whenever G_s has a subgraph isomorphic to L_{run}. Hence, there are two graph morphisms into G_{12}, there is one graph morphism into G_1 and one into G_{123}, but no graph morphism into G_{1234}.

Consider the injective match of L_{run} in G_{12} given by the right-most vertical edge. The removal of the edges of this match yields the subgraph



2.4 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 rest of the 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.

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 3. Consider the following two rules.

[1] $\#X$ denotes the number of elements of a finite set X.

start = □ ⊇ ● ⊆ ◎

run = ○⟶□ ⊇ ● ● ⊆ ●$\xrightarrow{p}$○

The rule *start* describes the removal of an unlabeled loop and the addition of a *begin*-loop and an *end*-loop at the same node. The rule *run* replaces an unlabeled edge by a p-edge removing the two loops of the left-hand side and adding an *end*-loop at the target node of the right-hand side. The identity of the nodes is chosen in such a way that the direction of the edge is preserved, i.e., the two sources are equal and the two targets are equal.

2.5 Application of a Graph Transformation Rule

The application of a graph transformation rule to a graph G consists of replacing an injective 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. Hence, 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. An injective match $g(L)$ of L in G is chosen.
2. Now the nodes of $g_V(V_L - V_K)$ are removed, and the edges of $g_E(E_L - E_K)$ as well as the edges incident to removed nodes are removed yielding the *intermediate graph* $Z \subseteq G$.
3. Afterwards the right-hand side R is added to Z by gluing Z with R in $g(K)$ yielding the graph $H = Z + (R - K)$ with $V_H = V_Z + (V_R - V_K)$ [2] and $E_H = E_Z + (E_R - E_K)$. The edges of Z keep their labels, sources, and targets so that $Z \subseteq H$. The edges of R keep their labels. They keep their sources and targets provided that those belong to $V_R - V_K$. Otherwise, $s_H(e) = g(s_R(e))$ 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 application of a rule r to a graph G is denoted by $G \underset{r}{\Longrightarrow} H$, where H is the graph resulting from the application of r to G. A rule application is called a *direct derivation*. The subscript r may be omitted if it is clear from the context.

Given a finite set of rules and a finite graph G, the number of injective matches is bounded by a polynomial in the size of G because the sizes of left-hand sides of rules are bounded by a constant. Given an injective match, the construction of the directly derived graph is linear in the size of G. Therefore, it needs polynomial time to find a match and to construct a direct derivation.

Example 4. The rule *run* of Example 3 can be applied to the graph G_{12} in two ways. One injective match of L_{run} is given by the right vertical edge of G_{12}. The intermediate graph is Z_0 as constructed in Example 2. And the derived graph is G_{123}. Consequently, one gets a direct derivation $G_{12} \underset{run}{\Longrightarrow} G_{123}$.

[2] Given sets X and $Y, X + Y$ denotes the disjoint union of X and Y.

Accordingly, the rule *run* can be applied to other graphs in $SP(G_0)$ yielding a graph in $SP(G_0)$ in each case. Here is the complete list of direct derivations by applying *run*:

$$\begin{array}{lllll} G_1 \Longrightarrow G_{12}, & G_2 \Longrightarrow G_{23}, & G_2 \Longrightarrow G_{24}, & G_3 \Longrightarrow G_{34}, & G_4 \Longrightarrow G_{41}, \\ G_{12} \Longrightarrow G_{123}, & G_{12} \Longrightarrow G_{124}, & G_{23} \Longrightarrow G_{234}, & G_{24} \Longrightarrow G_{241}, & G_{34} \Longrightarrow G_{341}, \\ G_{41} \Longrightarrow G_{412}, & G_{123} \Longrightarrow G_{1234}, & G_{234} \Longrightarrow G_{2341}, & G_{341} \Longrightarrow G_{3412}, & G_{412} \Longrightarrow G_{4123}. \end{array}$$

Moreover, the rule *start* can be applied to G_0 in four ways deriving G_1, G_2, G_3 and G_4.

2.6 Derivation and Application Sequence

The sequential composition of direct derivations $d = G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ ($n \in \mathbb{N}$) 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 just by $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$. The string $r_1 \cdots r_n$ is the *application sequence* of the derivation d.

Example 5. The direct derivations in Example 3 can be composed into the following derivations:

$$\begin{array}{l} G_0 \Longrightarrow G_1 \Longrightarrow G_{12} \Longrightarrow G_{123} \Longrightarrow G_{1234}, \\ G_0 \Longrightarrow G_1 \Longrightarrow G_{12} \Longrightarrow G_{124}, \\ G_0 \Longrightarrow G_2 \Longrightarrow G_{23} \Longrightarrow G_{234} \Longrightarrow G_{2341}, \\ G_0 \Longrightarrow G_2 \Longrightarrow G_{24} \Longrightarrow G_{241}, \\ G_0 \Longrightarrow G_3 \Longrightarrow G_{34} \Longrightarrow G_{341} \Longrightarrow G_{3412}, \\ G_0 \Longrightarrow G_4 \Longrightarrow G_{41} \Longrightarrow G_{412} \Longrightarrow G_{4123}. \end{array}$$

Altogether, these derivations show that exactly the graphs in $SP(G_0)$ can be derived from G_0 by applying the rule *start* once and then the rule *run* k-times for $k \in \{0, 1, 2, 3\}$.

It is not difficult to see that one can generate the set of all simple paths of every unlabeled graph if each of its node has a simple unlabeled loop and if *start* is only applied in the first derivation step. Moreover, the length of each such derivation equals the number of nodes visited by the derived path. In particular, the length is bounded by the size of the initial graph. The proof can be done by induction on the lengths of derivations on one hand and on the lengths of simple paths on the other hand.

The six derived graphs above correspond to the dead-ended simple paths of G_0 being those that cannot be prolonged by a further edge. The four graphs $G_{1234}, G_{2341}, G_{3421}$, and G_{4123} represent the Hamiltonian paths of G_0.

3 Simple Graph Transformation Units

A rule yields a binary relation on graphs and a set of rules a set of derivations. The example of simple paths shows (like many other examples would show)

that more features are needed to model processes on graphs in a proper way, in particular one needs initial graphs to start the derivation process, terminal graphs to stop it, and some control conditions to regulate it. This leads to the concept of a simple graph transformation unit. The well-known notion of a graph grammar is an important special case.

Analogously to Chomsky grammars in formal language theory, graph transformation can be used to generate graph languages. A graph grammar consists of a set of rules, a start graph, and a terminal expression fixing the set of terminal graphs. This terminal expression is a set $\Delta \subseteq \Sigma$ of terminal labels admitting all graphs that are labeled over Δ.

3.1 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 6. The following graph grammar

unlabeled graphs
initial: *empty*
rules: *new-node* = *empty* $\supseteq$ *empty* $\subseteq$ □
new-edge = ● ● $\supseteq$ ● ● $\subseteq$ ●⟶●
terminal: $\{*\}$

generates the unlabeled graphs with a single unlabeled loop at each node. The start graph is the empty graph. The rule *new-node* adds a node with an unlabeled loop in each application. The rule *new-edge* adds an edge between two nodes. Due to the fact that we consider only injective matches, no two nodes can be identified in a match of the left-hand side of the rule *new-edge*. This guarantees that no new loops are generated. The terminal expression $\{*\}$ specifies all unlabeled graphs. Since the given rules transform unlabeled graphs into unlabeled ones, all derived graphs belong to the generated language. It should be noted that the grammar *unlabeled graphs* generates exactly the graphs that are used as initial graphs to obtain their simple paths in Example 5.

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 through the concept of transformation units (see, e.g., [4–6]), which generalize standard graph grammars in the following ways:

– 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.

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.

3.2 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 $SEM(\Delta) = \mathcal{G}_\Delta \subseteq \mathcal{G}_\Sigma$. Forbidden structures are also frequently used. Let F be a graph, then $SEM(\mathit{forbidden}(F))$ contains all graphs G such that there is no graph morphism $f : F \to G$. Another useful type of graph class expressions is given by sets of rules. More precisely, for a set P of rules, $SEM(\mathit{reduced}(P))$ 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 $SEM(GG) = L(GG)$.

3.3 Control Conditions

A *control condition* is any syntactic entity that cuts down the non-determinism of 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 $SEM(\mathit{as\ long\ as\ possible})$ 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 $\mathit{reduced}(P)$ introduced above. Also similar to *as long as possible* are *priorities* on rules, which are partial orders on rules such that if $p_1 > p_2$, then 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 [7].

By now, we have collected all components for defining simple graph transformation units.

3.4 Simple Graph Transformation Units

A *simple graph transformation unit* 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.

Each such transformation unit tu 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. The semantic relation $SEM(tu)$ may also be denoted by $RA(tu)$ if the aspect is stressed that it is based on rule application.

Example 7. The constructions from Examples 1 through 6 can be summarized by the following simple graph transformation unit:

simple paths
initial: *unlabeled graphs*
rules: *start, run*
control: *start ; run**
terminal: *all*

The initial graphs are unlabeled graphs with a single loop at each node as generated by the graph grammar *unlabeled graphs* in Example 6. The rules *start* and *run* are given in Example 3, and the control condition is a regular expression over the set of rules with the sequential composition ; and the Kleene star * (specifying that a single application of *start* can be followed by an arbitrary sequence of applications of *run*). All graphs derived in this way from initial graphs are accepted as terminal. This is expressed by the graph class expression *all*. As discussed in Example 5, this unit generates all simple paths of each initial graph.

Analoguously, one can model *dead-ended simple paths* and *Hamiltonian paths* by replacing the terminal graph expression *all* by the expressions *reduced*$(\{run\})$ and *forbidden*$(\Box)$, resp.

4 Graph Transformation Units with Structuring

Simple graph transformation units allow one to model computational processes on graphs in the small. For modeling in the large, structuring concepts are needed for several reasons:

(1) To describe and solve a practical problem, one may need hundreds, thousands, or even millions of rules. Whereas a single set of rules of such a size would be hard to understand, the division into small components could help.
(2) Many problems can be solved by using the solutions of other problems. For example, most kinds of tour planning require a shortest path algorithm. Therefore, it would be unreasonable to model everything from scratch rather than to re-use available and working components.
(3) The modeling of a large system requires the subdivision into smaller pieces and the distribution of subtasks. But then it is necessary to know how components interact with each other.

Graph transformation units can be provided with a structuring principle by the *import* and *use* concept. The basic observation behind this concept is that the

semantics of a simple graph transformation unit is a binary relation on graphs, like the rule application relation. Therefore, a unit (maybe with many rules) can play the role of a single rule. Units may use or import entities that describe binary relations on graphs. In particular, units may be used. This allows re-use as well as the distribution of tasks. See, e.g., [6, 8] for more details.

4.1 Units with Import and Interleaving Semantics

A *graph transformation unit with import* is a system $tu = (I, U, P, C, T)$, where (I, P, C, T) is a simple graph transformation unit, and the *use* component U is a set of identifiers (which is disjoint of P).

If each $u \in U$ defines a relation $SEM(u) \in \mathcal{G} \times \mathcal{G}$, then tu specifies a binary relation on graphs $INTER_{SEM}(tu)$ defined as follows:

$$(G, G') \in INTER_{SEM}(tu) \quad \text{if} \quad G \in SEM(I), G' \in SEM(T),$$

and there is a sequence $G_0, \ldots, G_n$ with $G = G_0, G_n = G'$, and, for $i = 1, \ldots, n, G_{i-1} \underset{r}{\Longrightarrow} G_i$ for some $r \in P$ or $(G_{i-1}, G_i) \in SEM(u)$ for some $u \in U$. Moreover, (G, G') must be accepted by the control condition C.

This relation is called *interleaving semantics* because the computation interleaves rule applications and calls of the imported relations. It should be noted that each choice of used relations defines an interleaving relation.

4.2 Networks of Units

The definition of structuring by import allows the use of any relation whenever it has an identifier. There may be some library that offers standard relations. Or one may use another framework that supports the modeling of relations of graphs. But the most obvious choice is the import of units as use component. This leads to sets of units that are closed under import:

Let V be a set of identifiers and $tu(v) = (I(v), U(v), P(v), C(v), T(v))$ a graph transformation unit with import for each $v \in V$. Then the set $\{tu(v) \mid v \in V\}$ is *closed under import* if $U(v) \subseteq V$ for all $v \in V$.

If one considers V as a set of nodes, the pairs (v, v') for $v' \in U(v)$ as edges with the projections as source and target, and the mapping tu as labeling, then one gets a *network* $(V, \{(v, v') \mid v' \in U(v), v \in V\}, tu)$.

If this network is finite and acyclic, then each of its units can be assigned with an import level that is used to define the interleaving semantics of networks in the next subsection:

$$level(v) = \begin{cases} 0 & \text{if } U(v) = \emptyset, \\ n + 1 & \text{if } n = \max\{level(v') \mid v' \in U(v)\}. \end{cases}$$

4.3 Interleaving Semantics of Acyclic Networks

The interleaving semantics of a unit with import requires that the used relations are predefined. This can be guaranteed in finite and acyclic networks of units

level-wise so that the semantics can be defined inductively. The units of level 0 are simple graph transformation units such that the rule application semantics is defined. If the semantic relations up to level n are defined by induction hypothesis, then the interleaving semantics of units on level $n+1$ is defined.

Let (V, E, tu) be a finite and acyclic network of units. Then the interleaving semantics $INTER(tu(v))$ for each $v \in V$ is inductively defined by:

– $INTER(tu(v)) = RA(tu(v))$ for $v \in V$ with $level(v) = 0$;
– assume that $INTER(tu(v)) \subseteq \mathcal{G} \times \mathcal{G}$ is defined for all $v \in V$ with $level(v) \leq n$ for some $n \in \mathbb{N}$; then $INTER(tu(v))$ is equal to $INTER_{SEM}(tu(v))$ with $SEM(v') = INTER(tu(v'))$ for $v' \in U_{tu(v)}$ for $v \in V$ with $level(v) = n+1$.

Example 8. As shown in Example 7, the generation of dead-ended paths and Hamiltonian paths coincides with the generation of simple paths except for stronger terminal graph expressions. Hence, they can be modeled by the import and re-use of *simple paths*.

dead-ended simple paths	*Hamiltonian paths*
uses: *simple paths*	uses: *simple paths*
control : *simple paths*	control : *simple paths*
terminal $reduced(run)$	terminal $forbidden(\square)$

5 Approach Independence and Product Type

Graph class expressions and control conditions are introduced as generic concepts that can be chosen out of a spectrum of possibilities. On the other hand, the graphs, the rules and their application are fixed in a specific way in the preceding sections. This is done to be able to illustrate the concepts by explicit examples. The notions of graph transformation units and the interleaving semantics are independent of the particular kind of graphs and rules one assumes. This is made precise by the generic notion of a graph transformation approach.

5.1 Graph Transformation Approach

A *graph transformation approach* $\mathcal{A} = (\mathcal{G}, \mathcal{R} \Longrightarrow, \mathcal{X}, \mathcal{C})$ consists of a class of graphs $\mathcal{G}$, a class of rules $\mathcal{R}$, a rule application operator $\Longrightarrow$ that provides a binary relation on graphs $\underset{r}{\Longrightarrow} \subseteq \mathcal{G} \times \mathcal{G}$ for every $r \in \mathcal{R}$, a class of graph class expressions $\mathcal{X}$ with $SEM(x) \subseteq \mathcal{G}$ for every $x \in \mathcal{X}$, and a class of control conditions $\mathcal{C}$ with $SEM(c) \subseteq \mathcal{G} \times \mathcal{G}$ for every $c \in \mathcal{C}$.

All the notions of Sections 3 and 4 remain valid over such an approach $\mathcal{A}$ if one replaces the class of graphs $\mathcal{G}_{\Sigma}$ by $\mathcal{G}$, the class of rules by $\mathcal{R}$, and each rule application by its abstract counterpart. In this sense, the modeling by graph transformation units is approach-independent because it works independently of a particular approach.

The approach independence is meaningful in at least two ways. On the one hand, it means that everybody can use and choose his or her favorite kinds of graphs, rules, rule applications, graph class expressions, and control conditions. On the other hand, it allows one to adapt given approaches to particular applications or to build new approaches out of given ones without the need to change the modeling concepts.

5.2 Restriction

Given a graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Longrightarrow, \mathcal{X}, \mathcal{C})$, each subclass $\mathcal{G}' \subseteq \mathcal{G}$ induces a restricted approach $\mathit{restrict}(\mathcal{A}, \mathcal{G}') = (\mathcal{G}', \mathcal{R}, \Longrightarrow, \mathcal{X}, \mathcal{C})$, where the semantics of $\Longrightarrow, \mathcal{X}$, and $\mathcal{C}$ are taken from $\mathcal{A}$, but restricted to the graphs of $\mathcal{G}'$.

Similarly, all other components of $\mathcal{A}$ may be restricted – while this can be done for $\mathcal{X}$ and $\mathcal{C}$ without side effects, the restriction of rules and rule application may influence the semantics of $\mathcal{X}$ and $\mathcal{C}$, because graph class expressions and control conditions may refer to rules.

With respect to the class of graphs $\mathcal{G}_\Sigma$ as considered in the preceding sections, various restrictions are possible: unlabeled graphs, connected graphs, planar graphs, etc. A rule application to such a sort of graph would only be accepted if the derived graph is of the same type. An undirected graph can be represented by a directed graph if each undirected edge is replaced by two directed edges in opposite direction. In this way, undirected graphs can be handled in the given framework as a restriction of the introduced approach.

An example of a restricted kind of rules is a rule the left-hand side of which coincides with the gluing graph so that no removal takes place if the rule is applied.

A typical restriction concerning the rule application is the requirement of injective matching as used in our sample approach. This is requested in many graph transformation approaches in the literature.

5.3 Product Type

Another approach-building operator is the product of approaches. The idea is to consider tuples of graphs and rules where a tuple of rules is applied to a tuple of graphs by applying the component rules to the component graphs simultaneously. It is not always convenient to apply a rule to each component graph. Therefore, we add the symbol $-$ to each class of rules. Whenever $-$ is a component of a tuple of rules, the corresponding component graph remains unchanged.

Let $\mathcal{A}_i = (\mathcal{G}_i, \mathcal{R}_i, \Longrightarrow_i, \mathcal{X}_i, \mathcal{C}_i)$ for $i = 1, \ldots, n$ for some $n \in \mathbb{N}$ be a graph transformation approach. Then the product approach is defined by

$$\prod_{i=1}^{n} \mathcal{A}_i = (\prod_{i=1}^{n} \mathcal{G}_i, \prod_{i=1}^{n} (\mathcal{R}_i \cup \{-\}), \Longrightarrow, \prod_{i=1}^{n} \mathcal{X}_i, \prod_{i=1}^{n} \mathcal{C}_i)$$

where

- $(G_1, \ldots, G_n) \Longrightarrow (G'_1, \ldots, G'_n)$ for $(r_1, \ldots, r_n) \in \prod_{i=1}^{n} (\mathcal{R}_i \cup \{-\})$ if, for $i = 1, \ldots, n, G_i \underset{r_i}{\Longrightarrow} G'_i$ for $r_i \in \mathcal{R}_i$ and $G_i = G'_i$ for $r_i = -$,
- $(G_1, \ldots, G_n) \in SEM(x_1, \ldots, x_n)$ for $(x_1, \ldots, x_n) \in \prod_{i=1}^{n} \mathcal{X}_i$ if $G_i \in SEM(x_i)$ for $i = 1, \ldots, n$,
- $((G_1, \ldots, G_n), (G'_1, \ldots, G'_n)) \in SEM(c_1, \ldots, c_n)$ for $(c_1, \ldots, c_n) \in \prod_{i=1}^{n} \mathcal{C}_i$ if $(G_i, G'_i) \in SEM(c_i)$ for $i = 1, \ldots, n$.

5.4 Tuples of Graph Transformation Units

Analoguously to the tupling of graphs, rules, rule applications, graph class expressions and control conditions in the product of graph transformation approaches, graph transformation units over the approaches can be tupled.

Let $tu_i = (I_i, U_i, P_i, C_i, T_i)$, for each $i = 1, \ldots, n$ for some $n \in \mathbb{N}$, be a graph transformation unit over the graph transformation approach $\mathcal{A}_i = (\mathcal{G}_i, \mathcal{R}_i, \Longrightarrow_i, \mathcal{X}_i, \mathcal{C}_i)$. Then the tuple

$$(tu_1, \ldots, tu_n) = ((I_1, \ldots, I_n), \prod_{i=1}^{n} U_i, \prod_{i=1}^{n} P_i, (C_1, \ldots, C_n), (T_1, \ldots, T_n))$$

is a graph transformation unit over the product approach $\prod_{i=1}^{n} \mathcal{A}_i$.

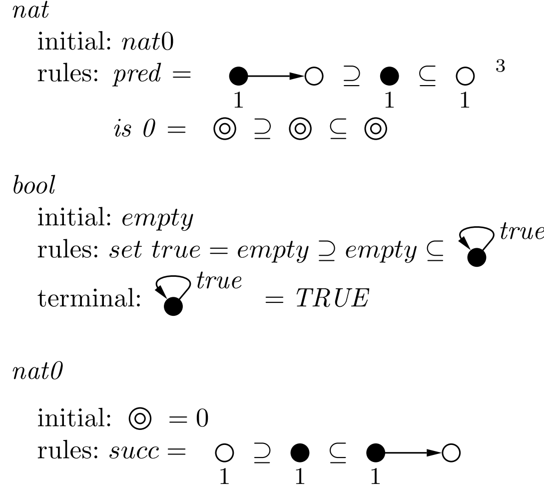
The important aspect of the tupling of units is that the choice of the components of the single units induces automatically all the components of the tuple of units by tupling the graph class expressions for initial and terminal graphs resp. and the control conditions as well as by the products of the import and rule sets.

As the tuple of units is defined component-wise, its semantics is the product of the semantic relations of the components up to a reordering of components. Let $SEM_i(u) \subseteq \mathcal{G}_i \times \mathcal{G}_i$ be a binary relation on graphs for each $u \in U_i, i = 1, \ldots, n$. Let $SEM_i(u_1, \ldots, u_n) \subseteq \prod_{i=1}^{n} \mathcal{G}_i \times \prod_{i=1}^{n} \mathcal{G}_i$ be defined for $(u_1, \ldots u_n) \in \prod_{i=1}^{n} U_i$ by $((G_1, \ldots, G_n), (G'_1, \ldots, G'_n)) \in SEM((u_1, \ldots, u_n))$ if and only if $(G_i, G'_i) \in SEM_i(u_i)$ for $i = 1, \ldots, n$. Then the following holds:

$((G_1, \ldots, G_n), (G'_1, \ldots, G'_n)) \in INTER_{SEM}(tu)$ if and only if $(G_i, G'_i) \in INTER_{SEM_i}(tu_i)$ for $i = 1, \ldots, n$.

As the interleaving semantics of a tuple of units is given by the product of the interleaving semantics, the tuple of units tu may also be denoted by $tu_1 \times \cdots \times tu_n$ to stress the meaning already on the syntactic level.

Example 9. Consider the tuple of units $simple\ paths \times nat \times bool$ where the first component is given in Example 7 and nat and $bool$ are modeled as follows:



The start graph of *nat0* is a node with a *begin*-loop and an *end*-loop which may be seen as the number 0. The application of *succ* adds an edge to the *end*-node while the added target becomes the new *end*. Therefore, the derived graphs are simple paths of the form

with n edges, for some $n \in \mathbb{N}$, representing the number n. There is no control, and all graphs are terminal. In other words, *nat0* generates the natural numbers being the initial graphs of *nat*. The rule *pred* is inverse to *succ* so that its application transforms the graph $n+1$ into the graph n. The rule *is 0* is applicable to a graph n if and only if $n = 0$ such that the rule provides a 0-test. Altogether, the unit *nat* can count down a given number and test whether 0 is reached.

The unit *bool* is extremely simple. The start graph is empty. The only rule adds a node with a *true*-loop whenever applied. But after one application, the terminal graph is reached already. This graph can be seen as the truth value *TRUE*.

According to the definitions of the components, an initial graph of the tuple of units *simple paths* × *nat* × *bool* has the form (G, n, empty), where G is an unlabeled graph with an unlabeled loop at each node. Consider, in particular, $(G_0, 3, \mathit{empty})$. To such an initial graph, one may apply the triple rule $(\mathit{start}, -, -)$ replacing an unlabeled loop of G by a *begin*- and an *end*-loop and keeping n and *empty* unchanged. For example, one can derive $(G_1, 3, \mathit{empty})$ from $(G_0, 3, \mathit{empty})$. Now one may apply the triple rule $(\mathit{run}, \mathit{pred}, -)$ repeatedly. This builds a simple path in G edge by edge while n is decreased 1 by 1. If G has a simple path of length n, one can derive $(G', 0, \mathit{empty})$ in this way. For example, one can derive $(G_{1234}, 0, \mathit{empty})$ from $(G_1, 3, \mathit{empty})$. Finally, the triple rule $(-, \mathit{is}\ 0, \mathit{set\ true})$ becomes applicable deriving $(G', 0, \mathit{TRUE})$ and $(G_{1234}, 0, \mathit{TRUE})$ in particular.

[3] The number 1 identifies the identical nodes to make the inclusions unambiguous.

Altogether, this models a test whether a graph has a simple path of a certain length.

simple paths of some lengths
initial: : *simple paths* $\times$ *nat*
uses : *simple paths* $\times$ *nat* $\times$ *bool*
control : $(start, -, -); (run, pred, -)^*; (-, is\ 0, set\ true)$
terminal : *bool*

Due to the import, this unit is based on the tuple of units considered above. This provides the tuple rules in the control condition, which is just a regular expression over the set of rules. But the graph class expressions are of a new type related to the product. The initial expression means that the first two components can be chosen freely as inputs of the modeled test. The third component is always the empty graph so that it can be added by default. The terminal expression means the projection to the third component as output of the test.

The semantic analysis shows that the unit *simple paths of some lengths* relates a graph G and a number n to the truth value *TRUE* if and only if G has a simple path of length n. Moreover, the length of every derivation is bounded by $n + 2$ because n can be decreased by 1 n times at most. It is also bounded by the number m of nodes of G plus 1 because simple paths are shorter than m. Because the number of matches for the rules is also bounded polynomially, the unit proves that the test for simple paths of certain lengths is in the class NP. This is a well-known fact in this case, but illustrates that the introduced framework supports proofs like this.

If the input length is chosen as the number of nodes of the input graph minus 1, then the unit yields *TRUE* if and only if the input graph has a Hamiltonian path. In this way, the test for simple paths of certain lengths turns out to be NP-complete because the Hamiltonian-path problem is NP-complete and a special case.

5.5 Typing of Units

A graph transformation unit models a relation between initial and terminal graphs. Hence one may say that the type of a unit $tu = (I, P, C, T)$ is $I \to T$. The introduced product type allows a more sophisticated typing of the form $I_1 \times \cdots \times I_m \to T_1 \times \cdots \times T_n$. This works as follows. Let $tu_1 \times \cdots \times tu_p$ be a tuple of units, let $input : \{1, \ldots, m\} \to \{1, \ldots, p\}$ be an injective mapping with $I_i = I(tu_{input(i)})$ for $i = 1, \ldots, m$, and let $output\colon \{1, \ldots, n, \} \to \{1, \ldots, p\}$ be a mapping with $T_j = T(tu_{output(j)})$. Moreover, there may be some extra control condition c for the tuple of units. Then this defines a unit of type $I_1 \times \cdots \times I_m \to T_1 \times \cdots \times T_n$ by

typed unit
initial: *input*
uses: $tu_1 \times \cdots \times tu_p$
control: c
terminal: *output*

This unit relates graph tuple $(G_1, \ldots, G_m) \in \prod_{i=1}^{m} SEM(I_i)$ with the graph tuple $(H_1, \ldots, H_n) \in \prod_{i=1}^{n} SEM(T_j)$ if there are graphs $((G'_1, \ldots, G'_p), (H'_1, \ldots, H'_p))$ belonging to the interleaving semantics of $tu_1 \times \cdots \times tu_p$, fulfilling in addition the control condition c, and $G_i = G'_{input(i)}$ for $i = 1, \ldots, m$ as well as $H_j = H'_{output(j)}$ for $j = 1, \ldots, n$, where the graphs G_i with $i \notin input(\{1, \ldots, m\})$ can be chosen arbitrarily. Some of the components of the product are inputs, some outputs, some may be auxiliary. Initially, the input components are given. The other components must be chosen which is meaningful if there are unique initial graphs that can serve as defaults. Then the product is running component-wise according to the component rules and control conditions reflecting its extra control condition. If all components are terminal, the output components are taken as results.

A more detailed investigation of the product type can be found in [9, 10].

6 Further Research and Related Work

In the preceding sections, we have given an overview of graph transformation units as devices to model algorithms, processes, and relations on graphs. Such units consist of rules together with specifications of initial and terminal graphs as well as control conditions to cut down the nondeterminism of rule applications. Moreover, units can import other units (or other relations on graphs) providing in this way possibilities of re-use and of structuring. The operational semantics of graph transformation units is given by the interleavings of rule applications and calls of imported relations; it yields a relation between initial and terminal graphs. This interleaving semantics is well-defined if the import structure is acyclic. All the considered concepts work for arbitrary graph transformation approaches, where an approach is the computational base underlying graph transformation units. Such an approach consists of classes of graphs, rules, graph class expressions, and of control conditions as well as a notion of rule application. Every component of an approach can be chosen out of a wide spectrum of graph transformation concepts one encounters in the literature. An interesting aspect of this kind of approach independence is the possibility to construct new approaches from given ones. For example, this allows one to transform undirected graphs as a restriction of an approach for directed graphs. This also includes the product of approaches which handles tuples of graphs and provides a quite general typing of semantic relations.

With respect to the modeling of graph algorithms, Plump's and Steinert's concept of graph programs [11] is closely related to transformation units (cf. also Mosbah and Ossamy [12]). The major difference is that graph programs are based on a particular graph transformation approach. In contrast to this, approach independence allows the modelers to choose their favorite approaches of which the area of graph transformation offers quite a spectrum (see, e.g., [1] for the most frequently used approaches and Corradini et al. [13], Drewes et al. [14],

and Klempien-Hinrichs [15] as examples of newer approaches). In this context, one may note that the categorial framework of adhesive categories provides a kind of approach independence. While rules and rule application are fixed by the use of pushouts, one can enjoy quite a variety of classes of graphs due to the possible choices of the underlying category (see, e.g., [16, 17]).

As a structuring principle, transformation units are closely related to other module concepts for graph transformation systems like the ones introduced by Ehrig and Engels [18], by Taentzer and Schürr [19], by Große-Rhode, Parisi-Presicce and Simeoni [20, 21] as well as Schürr's and Winter's package concept [22]. Heckel et al. [23] classify and compare all these concepts including transformation units in some detail.

We point out now some further possible directions of the investigation of graph transformation units:

1. If one permits cyclic import, meaning that units can use and help each other in a recursive way, then the interleaving semantics as defined in Section 4 is no longer meaningful because the imported relations cannot be assumed to be defined already. In this case, the infinite iteration of the interleaving construction works. (see, e.g.,[24]).
2. The interleaving semantics is based on the iterated sequential composition of the relations given by rule application and the imported relations. Therefore, it is a purely sequential semantics. But one may replace the sequential composition by other operations on relations like, for example, parallel composition. Every choice of such operations defines an interlinking semantics of graph transformation units that, in particular, covers modes of parallel and concurrent processing (see, e.g.,[25]).
3. The semantic relations considered so far are associated to single units which control the computations and call the service of other units. In this sense, a graph transformation unit is a centralized computational entity. The concept of autonomuous units (see, e.g.,[26–28]) is a generalization to a decentralized processing of graphs. The autonomuous units in a community act, interact, and communicate in a common environment, with each of them controling its own activities autonomuously.

Acknowledgement. We are grateful to Andrea Corradini for valuable comments on an earlier version of the overview.

References

1. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)
2. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
3. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. 3. World Scientific, Singapore (1999)

4. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph transformation for specification and programming. Science of Computer Programming 34(1), 1–54 (1999)
5. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. In: [2], pp. 607–638.
6. Kreowski, H.J., Kuske, S.: Graph transformation units with interleaving semantics. Formal Aspects of Computing 11(6), 690–723 (1999)
7. Kuske, S.: More about control conditions for transformation units. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 323–337. Springer, Heidelberg (2000)
8. Kuske, S.: Transformation Units—A Structuring Principle for Graph Transformation Systems. PhD thesis, University of Bremen (2000)
9. Klempien-Hinrichs, R., Kreowski, H.J., Kuske, S.: Rule-based transformation of graphs and the product type. In: van Bommel, P. (ed.) Transformation of Knowledge, Information, and Data: Theory and Applications, pp. 29–51. Idea Group Publishing, Hershey, Pennsylvania (2005)
10. Klempien-Hinrichs, R., Kreowski, H.J., Kuske, S.: Typing of graph transformation units. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 112–127. Springer, Heidelberg (2004)
11. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004)
12. Mosbah, M., Ossamy, R.: A programming language for local computations in graphs: Computational completeness. In: 5th Mexican International Conference on Computer Science (ENC 2004), pp. 12–19. IEEE Computer Society, Los Alamitos (2004)
13. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006)
14. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Eetvelde, N.V.: Adaptive star grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 77–91. Springer, Heidelberg (2006)
15. Klempien-Hinrichs, R.: Hyperedge substitution in basic atom-replacement languages. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 192–206. Springer, Heidelberg (2002)
16. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement systems: A new categorical framework for graph transformation. Fundamenta Informaticae 74(1), 1–29 (2006)
17. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. (eds.): Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
18. Ehrig, H., Engels, G.: Pragmatic and semantic aspects of a module concept for graph transformation systems. In: Cuny, J., Engels, G., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 137–154. Springer, Heidelberg (1996)
19. Taentzer, G., Schürr, A.: DIEGO, another step towards a module concept for graph transformation systems. In: Corradini, A., Montanari, U. (eds.) SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation. Electronic Notes in Theoretical Computer Science, vol. 2, Elsevier, Amsterdam (1995)

20. Grosse-Rhode, M., Parisi Presicce, F., Simeoni, M.: Refinements and modules for typed graph transformation systems. In: Fiadeiro, J.L. (ed.) WADT 1998. LNCS, vol. 1589, pp. 137–151. Springer, Heidelberg (1999)
21. Große-Rhode, M., Parisi-Presicce, F., Simeoni, M.: Formal software specification with refinements and modules of typed graph transformation systems. Journal of Computer and System Sciences 64(2), 171–218 (2002)
22. Schürr, A., Winter, A.J.: UML packages for PROgrammed Graph REwriting Systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 396–409. Springer, Heidelberg (2000)
23. Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and comparison of module concepts for graph transformation systems. In: [2], pp. 639–689.
24. Kreowski, H.J., Kuske, S., Schürr, A.: Nested graph transformation units. International Journal on Software Engineering and Knowledge Engineering 7(4), 479–502 (1997)
25. Janssens, D., Kreowski, H.J., Rozenberg, G.: Main concepts of networks of transformation units with interlinking semantics. In: Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 325–342. Springer, Heidelberg (2005)
26. Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous units and their semantics — the sequential case. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 245–259. Springer, Heidelberg (2006)
27. Hölscher, K., Klempien-Hinrichs, R., Knirsch, P., Kreowski, H.J., Kuske, S.: Autonomous units: Basic concepts and semantic foundation. In: Hülsmann, M., Windt, K. (eds.) Understanding Autonomous Cooperation and Control in Logistics. The Impact on Management, Information and Communication and Material Flow, Springer, Berlin, Heidelberg, New York (2007)
28. Kreowski, H.J., Kuske, S.: Autonomous units and their semantics - the parallel case. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 56–73. Springer, Heidelberg (2007)