Rule-Based Transformation of Graphs and the Product Type

Renate Klempien-Hinrichs, University of Bremen

Hans-Jörg Kreowski, University of Bremen

Sabine Kuske, University of Bremen

Abstract

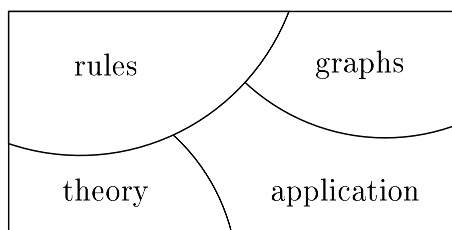
This chapter presents rule-based graph transformation as a framework for modeling data-processing systems. It recalls the structuring concept of graph transformation units which allows for transforming graphs in a rule-based, modularized, and controlled way. In order to get a flexible typing mechanism and a high degree of parallelism, this structuring concept is extended to the product of transformation units. Moreover, it is demonstrated how the product type can be used to transform graph transformation units. The authors advocate rule-based graph transformation for all applications where data, knowledge, and information can be modeled as graphs and their transformation can be specified by means of rules in an adequate way.

P. van Bommel (ed): Transformation of Knowledge, Information and Data: Theory and Applications, pp.29-51, 2005.
© Idea Group Publishing, Hershey, Pennsylvania, USA 2005

Introduction

The area of graph transformation brings together the concepts of rules and graphs with various methods from the theory of formal languages and from the theory of concurrency, and with a spectrum of applications, see Fig. 1.

Figure 1: Main ingredients of graph transformation



Graphs are important structures in computer science and beyond to represent complex system states, networks, and all kinds of diagrams. The application of rules provides graphs with a dynamic dimension yielding a rich methodology of rule-based graph transformation. The three volumes of the *Handbook of Graph Grammars and Computing by Graph Transformation* give a good overview of the state of the art in theory and practice of graph transformation (Rozenberg, 1997; Ehrig & Engels & Kreowski & Rozenberg, 1999; Ehrig & Kreowski & Montanari & Rozenberg, 1999).
Although one encounters quite a large number of different approaches to graph transformation in the literature, nearly all of them are composed out of five basic features.

- *Graphs* to represent complex relations among items in an intuitive but mathematically well-understood way.
- *Rules* to describe possible changes and updates of graphs in a concise way.
- *Rule applications* to perform the possible changes and updates on graphs explicitly as they are embodied in the rules.
- *Graph class expressions* to specify special classes of graphs to be used as initial as well as terminal graphs.
- *Control conditions* to regulate the applications of rules such that the inherent non-determinism of rule application can be cut down.

A particular choice of these five features establishes a graph transformation approach. A selection of rules, initial and terminal graphs, and a control condition is often called a *graph transformation system* or a *graph grammar* if there is a single initial graph as axiom (or a finite set of initial graphs likewise). Following Kreowski and Kuske (1999a), we use the term *graph transformation unit* for such a selection where we also allow importing other graph transformation units for structuring purposes.

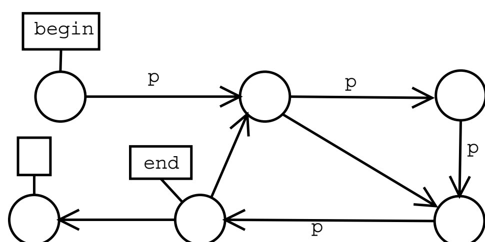
In this paper, we recall the elementary features of graph transformation in Section 2 and – based on it – discuss some new concepts that enhance the usefulness of graph transformation. As graphs are derived from graphs by applying rules, the obvious semantics is a binary relation on graphs or a binary relation between initial and terminal graphs if one provides a subtyping mechanism. To overcome this quite restricted kind of typing, we introduce product types in Section 3. The basic notion is a product of graph transformation units that comprises tuples of graphs to be processed componentwise, but where the transformations of the components run in parallel. The product, together with typical operations on products like embedding and projection, provides a very flexible kind of typing because one can declare a sequence of input components and a sequence of output components independently. To transform input graphs into output graphs, all components are combined into a proper product of graph transformation units. If one controls the parallel transformations of the components suitably, one can get the desired interrelations between input and output graphs. In Section 4, we demonstrate that the product type is also quite useful if one wants to transform graph transformation units.

Graph Transformation

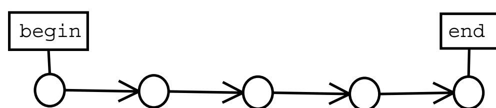
In this section we introduce main concepts of graph transformation like graphs, rules, and transformation units. The concepts are illustrated with a simple example from the area of graph theory. In the literature one can find many more applications of graph transformation which underline the usefulness from the practical point of view. These are for example applications from the area of functional languages (Sleep & Plasmeijer & van Eekelen 1993), visual languages (Bardohl & Minas & Schürr & Taentzer 1999), software engineering (Nagl 1996), and UML (e.g. Bottoni & Koch & Parisi-Presicce & Taentzer 2000; Engels & Hausmann & Heckel & Sauer 2000; Fischer & Niere & Torunski & Zündorf 2000; Petriu & Sun 2000; Engels & Heckel & Küster 2001; Kuske 2001; Kuske & Gogolla & Kollmann & Kreowski 2002).
Graph transformation comprises devices for the rule-based manipulation of graphs. Given a set of graph transformation rules and a set of graphs, one gets a graph transformation system in its simplest form. Such a system transforms a start graph by applying its graph transformation rules. The semantics can be defined as a binary relation on graphs where the first component of every pair is a start graph G and the second component is a graph derived from G by applying a sequence of graph transformation rules. In general, the application of a graph transformation rule to a graph transforms it locally, i.e. it replaces a part of the graph with another graph part. Often one wishes to start a derivation only from certain initial graphs, and accepts as results only those derived graphs that are terminal. Moreover, in some cases the derivation process is regulated in a certain way to cut down the non-determinism of rule applications. For example, one may employ a parallel mode of transformation as in L systems, or one may restrict the order in which rules are applied. Altogether the basic elements of a graph transformation approach are graphs, rules, their application, graph class expressions, and control conditions.

Graphs. First of all, there is a class of graphs $\mathcal{G}$, may they be directed or undirected, typed or untyped, labelled or unlabelled, simple or multiple. Examples for graph classes are labelled

directed graphs, hypergraphs, trees, forests, finite automata, Petri nets, etc. The choice of graphs depends on the kind of applications one has in mind and is a matter of taste.
In this paper, we consider directed, edge-labelled graphs with individual, multiple edges. A *graph* is a construct $G = (V, E, s, t, l)$ where V is a set of *vertices*, E is a set of *edges*, $s, t: E \to V$ are two mappings assigning each edge $e \in E$ a *source* $s(e)$ and a *target* $t(e)$, and $l: E \to C$ is a mapping *labelling* each edge in a given label alphabet C. A graph may be represented in a graphical way with circles as nodes and arrows as edges that connect source and target each, with the arrowhead pointing to the target. The labels are placed next to the arrows. In the case of a loop, i.e. an edge with the same node as source and target, we may draw a flag that is posted on its node with the label inside the box. To cover unlabelled graphs as a special case, we assume a particular label * that is invisible in the drawings. This means a graph G is unlabelled if $l(e) = *$ for all $e \in E$. For instance the graph



consists of six nodes, one of them with a *begin*-flag, another with an *end*-flag, and a third one with an unlabelled flag. Moreover, it consists of seven directed edges where some of them are labelled with *p*. The *p*-edges form a simple path (i.e. a path without cycles) from the *begin*-flagged node to the *end*-flagged node. If one takes the subgraph induced by the edges of the simple path and the *begin*- and *end*-flag and removes all occurrences of the label *p*, one gets the following string graph (i.e. a graph that is a simple path from a *begin*-flagged node to an *end*-flagged node).



String graphs can be used to represent natural numbers. The string graph above represents the number 4 because it has four unlabelled edges between its *begin*-flagged and its *end*-flagged node. Whenever a string graph represents a natural number *k* in this way, we say that it is the *k*-string graph.

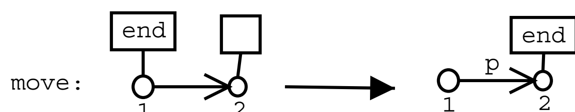
Rules and rule applications. To be able to transform graphs, rules are applied to graphs yielding graphs. Given some class $\mathcal{R}$ of graph transformation rules, each rule $r \in \mathcal{R}$ defines a binary relation $\Rightarrow_r \subseteq \mathcal{G} \times \mathcal{G}$ on graphs. If $G \Rightarrow_r H$, one says that *G directly derives H* by applying *r*.

There are many possibilities to choose rules and their applications. Rule classes may vary from the more restrictive ones, like edge replacement (Drewes & Kreowski & Habel, 1997) or node replacement (Engelfriet, & Rozenberg, 1997), to the more general ones, like double-pushout rules (Corradini et al., 1997), single-pushout rules (Ehrig et al., 1997), or PROGRES

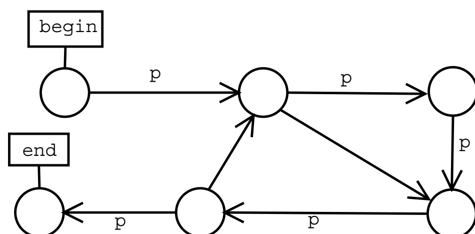
rules (Schürr, 1997).
In this paper, we use *rules* of the form $r = (L \rightarrow_K R)$ where L and R are graphs (the *left*- and *right-hand side* of r, respectively) and K is a set of nodes shared by L and R. In a graphical representation of r, L and R are drawn as usual, with numbers uniquely identifying the nodes in K. Its application means to replace an occurrence of L with R such that the common part K is kept. In particular, we will use rules that add or delete flags, label edges, and add or delete a node together with an edge.
The rule $r = (L \rightarrow_K R)$ can be applied to some graph G directly deriving the graph H if it can be constructed up to isomorphism (i.e. up to renaming of nodes and edges) in the following way.

(i) Find an isomorphic copy of L in G, i.e. a subgraph that coincides with L up to the naming of nodes and edges.
(ii) Remove all nodes and edges of this copy except the nodes corresponding to K, provided that the remainder is a graph (which holds if the removal of a node is accompanied by the removal of all its incident edges).
(iii) Add R by merging K with its corresponding copy.

For example, the following rule *move* has as left-hand side a graph consisting of an *end*-flagged node 1, a node 2 with unlabelled flag, and an unlabelled edge from node 1 to node 2. The right-hand side consists of the same two nodes where node 1 has no flag and node 2 has an *end*-flag. Moreover, there is a *p*-labelled edge from node 1 to node 2. The common part of the rule *move* consists of the nodes 1 and 2.



The application of *move* labels an unlabelled edge with *p* if the edge connects an *end*-flagged node and a node with an unlabelled flag, moves the *end*-flag from the source of the edge to its target, and removes the unlabelled flag. For example, the application of *move* to the graph above yields the following graph. Note that this rule cannot be applied to the former graph in any other way; for instance, its left-hand side requires the presence of an unlabelled flag.



Graph class expressions. The aim of graph class expressions is to restrict the class of graphs to which certain rules may be applied or to filter out a subclass of all the graphs that can be derived by a set of rules. Typically, a graph class expression may be some logic formula describing a graph property like connectivity, or acyclicity, or the occurrence or absence of certain labels. In this sense, every graph class expression e specifies a set $SEM(e)$ of graphs in

$\mathcal{G}$. For instance, *all* refers to all directed, edge-labelled graphs, whereas *empty* and *bool* designate a class of exactly one graph each (the empty graph *EMPTY* for *empty*, and the graph *TRUE* consisting of one *true*-flagged node for *bool*). Moreover, *graph* specifies all unlabelled graphs each node of which carries a unique flag (that is unlabelled, too). Also, a particular form of the graphs may be requested, e.g. the expression *nat* defines all *k*-string graphs.

Control conditions. A control condition is an expression that determines, for example, the order in which rules may be applied. Semantically, it relates start graphs with graphs that result from an admitted transformation process. In this sense, every control condition *c* specifies a binary relation $SEM(c)$ on $\mathcal{G}$. As control condition we use in particular the expression *true* that allows all transformations (i.e. all pairs of graphs). Moreover, we use regular expressions as control conditions. They describe in which order and how often the rules and imported units are to be applied. In particular, the Kleene star states that an arbitrary number of iterations may be executed. The precise meaning of a regular expression is explained where it is used. More about control conditions can be found in (Kuske, 2000).

Altogether, a class of graphs, a class of rules, a rule application operator, a class of graph class expressions, and a class of control conditions form a graph transformation approach based on which graph transformation units as a unifying formalization of graph grammars and graph transformation systems can be defined. To transform graphs, a unit has got local rules, but may also import other graph transformation units. Therefore, the semantic relation of a unit is given by the interleaving of rule applications and calls of imported units. Transformation units were presented in Andries et al. (1999) and Kreowski and Kuske (1999b) as a modularization concept for graph transformation systems (cf. also Heckel & Engels & Ehrig & Taentzer, 1999). In the literature there exist also some case studies where transformation units are employed to model the semantics of functional programming languages (Andries et al., 1999), UML state machines (Kuske, 2001), and logistic processes (Klempien-Hinrichs & Knirsch & Kuske, 2002).

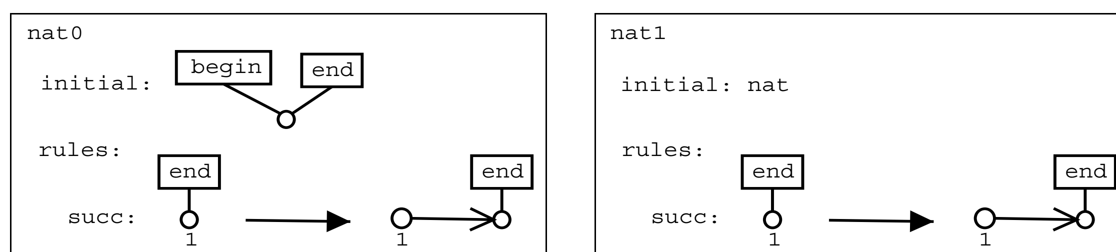
Transformation units. In general, a graph transformation system may consist of a huge set of rules that by its size alone is difficult to manage. Transformation units provide a means to structure the transformation process. The main structuring principle of transformation units relies on the import of other transformation units or – on the semantic level – on binary relations on graphs. The input and the output of a transformation unit each consist of a class of graphs that is specified by a graph class expression. The input graphs are called initial graphs and the output graphs terminal graphs. A transformation unit transforms initial graphs to terminal graphs by applying graph transformation rules and imported transformation units in a successive and sequential way. Since rule application is non-deterministic in general, a transformation unit contains a control condition that may regulate the graph transformation process.

A *graph transformation unit* is a system $tu = (I, U, R, C, T)$ where I and T are graph class expressions, U is a (possibly empty) set of imported graph transformation units, R is a set of rules, and C is a control condition.

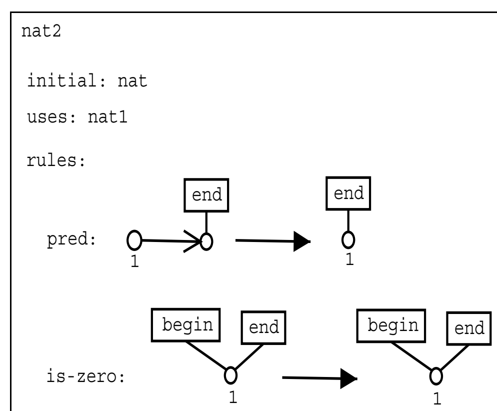
To simplify technicalities, we assume that the import structure is acyclic (for a study of cyclic imports see (Kreowski & Kuske & Schürr, 1997)). Initially, one builds units of level 0 with empty import. Then units of level 1 are those that import only units of level 0, and units of level n+1 import only units of level 0 to level n, but at least one from level n.

In graphical representations of transformation units we omit the import component if it is empty, the initial or terminal component if it is set to *all*, and the control condition if it is equal to *true*.

In the following we present some examples of transformation units. We start with very simple specifications of natural numbers and truth values because they are auxiliary data types to be used later to model the more interesting examples of simple paths, long simple paths, and Hamiltonian paths.
The first transformation unit *nat0* constructs all string graphs that represent natural numbers by starting from its initial graph, which represents 0, and transforming the *n*-string graph into the *n*+1-string graph by applying the rule *succ*.
The second transformation unit *nat1* is a variant of *nat0*, but now with all *n*-string graphs as initial graphs. Consequently, it describes arbitrary additions to arbitrary *n*-string graphs by sequentially increasing the represented numbers by 1.

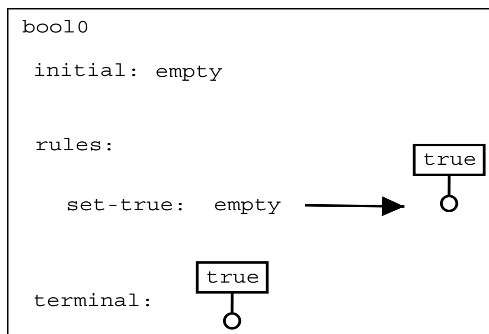


The third transformation unit *nat2* also transforms string graphs into string graphs. It has two rules *pred* and *is-zero*. The application of the rule *pred* to the *n*-string graph (with $n \geq 1$ since otherwise the rule cannot be applied) converts it into the *n*–1-string graph. The second rule *is-zero* can be applied only to the 0-string graph but does not transform it, which means that this rule can be used as a test for 0. Moreover, the transformation unit *nat2* imports *nat1* so that arbitrary additions can be performed, too. The rules of *nat2* and the imported unit *nat1* can be applied in arbitrary order and arbitrarily often. Hence *nat2* converts *n*-string graphs into *m*-string graphs for natural numbers *m*, *n*. Therefore *nat2* can be considered as a data type representing natural numbers with a simple set of operations.

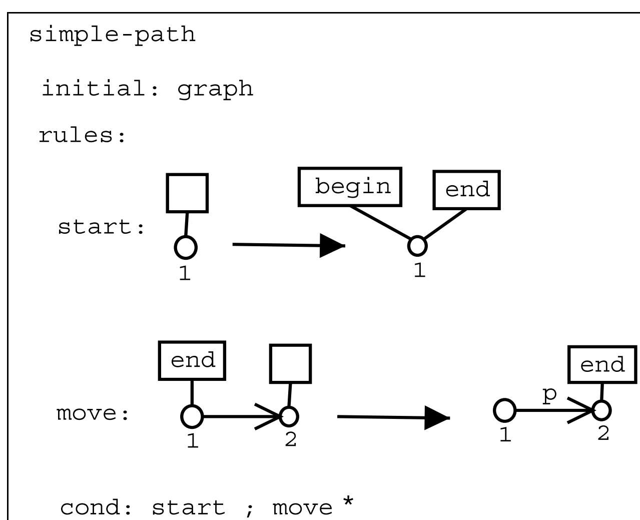


The forth transformation unit, *bool0* = (*empty*, ∅, *set-true, true, bool*), has a single initial graph, the empty graph *EMPTY*. It does not import other transformation units and it has one

rule *set-true* which turns *EMPTY* to the graph *TRUE*. The control condition allows all transformations, meaning that *TRUE* may be added arbitrarily often to *EMPTY*. However, the terminal graph class expression specifies the set consisting of *TRUE,* which ensures that the rule *set-true* is applied exactly once to the initial graph.
One can consider *bool0* as a unit that describes the type *Boolean* in its most simple form. At first sight, this may look a bit strange. But it is quite useful if one wants to specify predicates on graphs by non-deterministic graph transformation: If one succeeds to transform an input graph into the graph *TRUE*, the predicate holds; otherwise it fails. In other words, if the predicate does not hold for the input graph, none of its transformations yields *TRUE*.



The following transformation unit *simple-path* constitutes an example of another kind. As initial graphs it admits all unlabelled graphs with exactly one flag on every node. It chooses an arbitrary simple path in an initial graph by labelling the edges of the path with *p* and adding a *begin*-flag and an *end*-flag to the beginning and the end of the path, respectively. This is done with the help of two rules *start* and *move*. The rule *start* turns an unlabelled flag of an arbitrary node into two flags respectively labelled with *begin* and *end*, and the rule *move* is the same as above, i.e. it labels with *p* an edge from an *end*-flagged node to a node with an unlabelled flag, moves the *end*-flag to the other node, and removes the unlabelled flag. The control condition is a regular expression which is satisfied if first the rule *start* is applied, followed by *move* applied arbitrarily often. The terminal graph class expression admits all graphs, which is why it is not explicitly shown.



Interleaving semantics of transformation units. Transformation units transform initial graphs to terminal graphs by applying graph transformation rules and imported transformation units so that the control condition is satisfied. Hence, the semantics of a transformation unit can be defined as a binary relation between initial and terminal graphs. For example, the interleaving semantics of the transformation unit *simple-path* consists of all pairs (G,G') such that G is an unlabelled graph with exactly one flag on every node and G' is obtained from G by labelling the edges of a simple path with p, setting a *begin*-flag at the source of the path and an *end*-flag at the target of the path, and removing the flags from the intermediate nodes on the path.

In general, for a transformation unit *tu* without import, the semantics of *tu* consists of all pairs (G,G') of graphs such that

1. G is an initial graph and G' is a terminal graph;
2. G' is obtained from G via a sequence of rule applications, i.e. (G,G') is in the reflexive and transitive closure of the binary relation obtained from the union of all relations $\Rightarrow_r$ where r is some rule of *tu*; and
3. the pair (G,G') is allowed by the control condition.

If the transformation unit *tu* has a non-empty import, the interleaving semantics of *tu* consists of all pairs (G,G') of graphs which satisfy the preceding items 1 and 3, and where, in addition to rules, imported transformation units can be applied in the transformation process of *tu*, i.e. the second item above is extended to:

2'. G' is obtained from G via a sequence of rule applications and applications of imported units. This means that (G,G') is in the reflexive and transitive closure of the binary relation obtained from the union of all relations $\Rightarrow_r$ and $SEM(u)$ where r is some rule of *tu* and u is some imported transformation unit of *tu*.

More formally, the interleaving semantics of *tu* is defined as follows. Let $tu = (I, U, R, C, T)$ be a transformation unit. Then the interleaving semantics $SEM(tu)$ is recursively defined as

$$SEM(tu) = SEM((I,U,R,C,T)) = SEM(I) \times SEM(T) \cap (\bigcup_{r \in R} \Rightarrow_r \cup \bigcup_{u \in U} SEM(u))^* \cap SEM(C).$$

If the transformation unit *tu* is of level 0, the semantic relation is well-defined because the union over U is the empty set. If *tu* is of level n+1, we can inductively assume that $SEM(u)$ of each imported unit u is already well-defined, so that $SEM(tu)$ is also well-defined as a union and intersection of defined relations.

Product Type

As the iterated application of rules transforms graphs into graphs yielding an input-output relation, the natural type declaration of a graph transformation unit $tu = (I, U, R, C, T)$ is tu: $I \rightarrow T$ where moreover the initial and terminal graphs are subtypes of the type of graphs that are transformed by the unit. But in many applications one would like to have a typing that allows one to consider several inputs and maybe even several outputs, or at least an output of a type different from all inputs. For instance, a test whether a given graph has a simple path of a certain length would be suitably declared by *long-simple-path*: $graph \times nat \rightarrow bool$ (or something like this) asking for a graph and a non-negative integer as inputs and a truth value

as output.
Such an extra flexibility in the typing of graph transformations can be provided by products of graph transformation units together with some concepts based on the products. In more detail, we introduce the following new features.

1. The product of graph transformation units providing tuples of graphs to be processed and particularly tuples of initial and terminal graphs as well as tuples of rules and calls of imported units, called action tuples, that can be executed on graph tuples in parallel.
2. The embedding and projection of a product into resp. onto another product that allow one to choose some components of a product as inputs or outputs and to copy some components into others.
3. The semantics of a product of graph transformation units is the product of the component semantics such that – intuitively seen – all components run independently from each other. If one wants to impose some iteration and interrelation between the components, one can use control conditions for action tuples like for rules and imported units.

The product type generalizes the notion of pair grammars and triple grammars as introduced by Pratt (1971) and Schürr (1994), respectively.

Product of Graph Transformation Units

Let $tu_1,\ldots,tu_m$ for $m \geq 1$ be a sequence of graph transformation units with $tu_j = (I_j, U_j, R_j, C_j, T_j)$ for $j = 1,...,m$. Then the *product* $prod = tu_1 \times\ldots\times tu_m = \prod_{i=1}^{m} tu_i$ transforms m-tuples of graphs $(G_1,\ldots,G_m)$ by means of componentwise transformation
The global semantic relation of the product is just the product of the semantic relations of the components, i.e. $((G_1,\ldots,G_m), (H_1,\ldots,H_m)) \in SEM(prod)$ if and only if $(G_i,H_i) \in SEM(tu_i)$ for $i = 1,...,m$.
But there is also a notion of a single computation step that transforms graph tuples by applying action tuples. An *action tuple* $(a_1,\ldots,a_m)$ consists of rules, imported units and an extra void action –, i.e. $a_i \in R_i$ or $a_i \in U_i$ or $a_i = -$ for $i = 1,...,m$. It transforms a graph tuple $(G_1,\ldots,G_m)$ into a graph tuple $(G_1',\ldots,G_m')$ if $G_i \longrightarrow_{a_i} G_i'$ for $a_i \in R_i$ and $(G_i,G_i') \in SEM(a_i)$ for $a_i \in U_i$ and $G_i = G_i'$ for $a_i = -$.
In other words, a computation step applies simultaneously rules to some components and performs calls of import units in other components while the remaining components of the graph tuple are kept unchanged.
Let a single computation step be denoted by $(G_1,\ldots,G_m) \to (G_1',\ldots,G_m')$, and let $\to^*$ be the reflexive and transitive closure of $\to$. Then one can say that $(G_1,\ldots,G_m) \to^* (H_1,\ldots,H_m)$ *satisfies the control condition tuple* $(C_1,\ldots,C_m)$ if $(G_i,H_i) \in SEM(C_i)$ for $i = 1,...,m$. Similarly, $(G_1,\ldots,G_m)$ is an *initial graph tuple* if $G_i \in SEM(I_i)$, and $(H_1,\ldots,H_m)$ is a *terminal graph tuple* if $H_i \in SEM(T_i)$ for $i = 1,...,m$. If all this holds, the pair of tuples belongs to the *step semantics* of the product, which is denoted by *STEPSEM*(*prod*). It is easy to see that the global semantics and the step semantics coincide, i.e. $SEM(prod) = STEPSEM(prod)$.
For example, consider the product *simple-path* × *nat2* of the transformation units *simple-path*

and *nat2*. Its semantics consists of all pairs $((G_1,G_2), (H_1,H_2))$ where (G_1,G_2) is in the semantics of *simple-path* and (H_1,H_2) is in the semantics of *nat2*. This product combines two units in a free way like the well-known Cartesian product. In order to model interrelation between the components, e.g. to test if a path in a graph is of a certain length, we would like to have control conditions for the computation steps and a boolean value as output. This can be achieved with the concepts introduced in the following two subsections.

Embedding and Projection

If not all initial graph class expressions of a product are meant as inputs, but some of them are just of an auxiliary nature for intermediate computations or to be used as outputs, one may choose the input types and embed their product into the actual product that provides the graph tuples to be transformed. This is possible whenever the auxiliary components have got unique initial graphs and if every chosen input type is a subtype of the corresponding initial graphs.

Let $prod = tu_1 \times \ldots \times tu_m$ be a product of transformation units and let X be a set of graph class expressions that is associated with the product components by an injective mapping *ass*: $X \to \{1,\ldots,m\}$ such that $SEM(x) \subseteq SEM(I_{ass(x)})$ for all $x \in X$. Assume, moreover, for all $j \in \{1,\ldots,m\} \setminus ass(X)$ that either $SEM(I_j) = \{G_j\}$ for some graph G_j or $SEM(x) \subseteq SEM(I_j)$ for some chosen $x \in X$, which will be denoted by $copy: x \to j$. Then we get an embedding of the product of the graphs in $SEM(x)$ for $x \in X$ into the product of initial graphs of the product *prod*,

$$embed: \prod_{x\in X} SEM(x) \to \prod_{j=1}^{m} SEM(I_j)$$

defined by $embed((G_x)_{x\in X}) = (G_1,\ldots,G_m)$ with $G_i = G_x$ for $ass(x) = i$ and $copy: x \to i$, and $G_i \in SEM(I_i) = \{G_i\}$ otherwise.

This means that each input component is embedded into its corresponding component of the product of units with respect to *ass* and into all other components given by the *copy* relation. All remaining components of the product of units are completed by the single initial graphs of these components.
As a simple example, let $prod = simple\text{-}path \times nat2 \times bool0$ and let $X = \{graph, nat\}$. Consider the initial graph class expressions *graph*, *nat* and *empty* of the transformation units *simple-path*, *nat2*, and *bool0*, respectively. Every pair $(G_1,G_2) \in SEM(graph) \times SEM(nat)$ can be embedded into $SEM(graph) \times SEM(nat) \times SEM(empty)$ by choosing $ass(graph) = 1$ and $ass(nat) = 2$, i.e. we get $embed((G_1,G_2)) = (G_1,G_2,EMPTY)$ for every pair $(G_1,G_2) \in SEM(graph) \times SEM(nat)$.

Conversely, if one wants to get rid of some component graphs, the well-known projection may be employed. The same mechanism can be used to multiply components, which allows one, in particular, to copy a component graph into another component.

Let Y be a set which is associated with the product *prod* by $ass: Y \to \{1,\dots,m\}$. Then one can consider the product of the terminal graphs in $SEM(T_{ass(y)})$ for all $y \in Y$ as the semantics of the association *ass*, i.e. $SEM(ass) = \prod_{y \in Y} SEM(T_{ass(y)})$. The product of terminal graphs of the product *prod* can be projected to *SEM*(*ass*), i.e. $proj: \prod_{i=1}^{m} SEM(T_i) \to SEM(ass)$ defined by $proj(H_1,\dots,H_m) = (H_{ass(y)})_{y \in Y}$. For example, consider the terminal graph class expressions *all* and *bool* of the transformation units *simple-path*, *nat2*, and *bool0*. Let $Y = \{3\}$ and let $ass(3) = 3$. The semantics of *ass* is equal to the terminal graph *TRUE* of *bool0* and every triple $(H_1,H_2,H_3) \in SEM(all) \times SEM(all) \times SEM(bool)$ is projected to H_3, i.e. to *TRUE*.
In general, there are two cases of interest. Firstly, if $Y \subseteq \{1,\dots,m\}$ and *ass* is the corresponding inclusion, then *proj* is the ordinary projection of a product to some of its components. (This is the case in the described example.) Secondly, if several elements of Y are mapped to the same index i, this results in the multiplication of the i-th component.

Embedding and projection may be used to realize transformations on graphs with type declarations of the form $trans: in_1 \times \dots \times in_k \to out_1 \times \dots \times out_l$ where the in_i and the out_j are graph class expressions. The intention is that *trans* relates the product of inputs $SEM(in_1) \times \dots \times SEM(in_k)$ with the product of outputs $SEM(out_1) \times \dots \times SEM(out_l)$. This is obtained by using a product *prod* of graph transformation units $tu_1,\dots,tu_{k+l}$ such that $SEM(in_i) \subseteq SEM(I_i)$ for $i = 1,\dots,k$ and $SEM(T_j) \subseteq SEM(out_j)$ for $j = k+1,\dots,k+l$. The first k inclusions allow one to embed the inputs into the initial graph tuples of the product *prod* if, for $j = k+1,\dots,k+l$, we can choose some i with $copy: i \to j$ or $SEM(I_j) = \{G_j\}$ for some graph G_j. The last l inclusions allow one to project the terminal graph tuples of *prod* onto outputs. Therefore, the semantic relation of *trans* has the proper form, but the output tuples are totally independent of the input tuples due to the product semantics. To overcome this problem, we generalize the notion of control conditions in such a way that it applies not only to the control of rule applications and calls of imported units, but also to action tuples.

Control Conditions for Action Tuples

A control condition regulates the use of rules and imported units formally by intersecting the interleaving semantics with the semantic relation given by the control condition. This is easily generalized to action tuples if one replaces the interleaving semantics by the step semantics of the product of graph transformation units.
In concrete cases, the control condition may refer to action tuples like it can refer to rules and imported units. To make this more convenient, action tuples may get identifiers.

As an example how the features based on the product may be used, we specify the test *long-simple-path* that transforms graphs and non-negative integers as inputs into truth values as output.

long-simple-path: *graph* × *nat* → *bool*
prod: *simple-path* × *nat2* × *bool0*

$actions$: $a_0 = (start, -, -)$

$a_1 = (move, pred, -)$

$a_2 = (-, is\text{-}zero, set\text{-}true)$

$cond$: a_0 ; a_1^* ; a_2

It is modelled on top of the product of the units *simple-path*, *nat2* and *bool0*. The typing is appropriate as *graph* and *nat* specify the initial graphs of *simple-path* and *nat2* respectively, and *bool* refers to the terminal graph of *bool0*.
Hence, a computation in *long-simple-path* starts with an unlabelled graph and a non-negative integer completed to a triple by the initial graph of *bool0*. Then the control condition requires to perform a_0 that chooses a start node in the graph without changing the other two components. This is followed by the iteration of a_1 which in each step synchronously prolongs a simple path in the first component by one edge and decreases the integer in the second component by 1. Hence we get a graph with a path of the input length if the second component becomes zero. This is tested by the second component of a_2. In the positive case, a_2 is performed yielding *TRUE* as output in the third component. In other words, *long-simple-path* computes *TRUE* if and only if the input graph G has got a simple path of the input length n.

Transformation of Graph Transformations

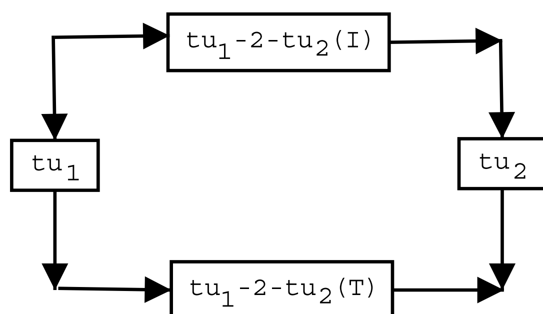
Two graph transformations may be related with each other in various significant ways.

1. They may be semantically equivalent, meaning that their semantic relations coincide or, seen from another perspective, that a semantic relation is modelled in two different ways.
2. One graph transformation may be the refinement of the other one, meaning that each computational step of the one can be accomplished by an interleaving sequence of the other.
3. One graph transformation may be reduced to the other, meaning that the semantic relation of the one can be translated into the semantic relation of the other.

Such situations are nicely modelled by transformations of graph transformations. In the case of two graph transformation units $tu_i = (I_i, U_i, R_i, C_i, T_i)$ with $SEM(tu_i) \subseteq SEM(I_i) \times SEM(T_i)$ for $i = 1,2$, a *transformation of the translational type* (or a *translation* for short) of tu_1 into tu_2 is defined by two graph transformation units $tu_1\text{-}2\text{-}tu_2(I) = (I_1, U_I, R_I, C_I, I_2)$ and $tu_1\text{-}2\text{-}tu_2(T) = (T_1, U_T, R_T, C_T, T_2)$ where the former transforms initial graphs of tu_1 into initial graphs of tu_2 and the latter does the same with respect to terminal graphs.

How a translation relates graph transformations units is depicted in Figure 2.

Figure 2: Translation of graph transformation units



Clearly, such a translation is only meaningful if it preserves the semantics, which is covered by the notion of correctness. A translation of tu_1 into tu_2 is *correct* if the diagram in Figure 2 commutes, i.e. if the sequential compositions of the semantic relations of $SEM(tu_1)$ with $SEM(tu_1\text{-}2\text{-}tu_2(T))$ on one hand and of $SEM(tu_1\text{-}2\text{-}tu_2(I))$ with $SEM(tu_2)$ on the other hand coincide.

Correct translations can be very helpful because they carry over certain properties from the source unit to the target unit and the other way round. For example, if some question is undecidable for the source unit, the corresponding question must be undecidable for the target unit provided that the translating units have computable semantic relations (which holds in all reasonable cases). To demonstrate the usefulness of translations more explicitly, we restrict the notion of translations to the notion of reductions as used in the study of the complexity class NP of all decision problems that can be computed non-deterministically in polynomial time. A *reduction* of tu_1 to tu_2 is a correct translation of tu_1 into tu_2 subject to the following further conditions.

(i) tu_1 and tu_2 model predicates, i.e. their output domain is *bool*,
(ii) $tu_1\text{-}2\text{-}tu_2(T)$ is the identity on *bool*, and
(iii) $tu_1\text{-}2\text{-}tu_2(I)$ has no import and runs in polynomial time, i.e. each derivation starting in an initial graph of tu_1 has a length polynomial in the size of its start graph and can be prolonged such that it derives an initial graph of tu_2.

If tu_2 models an NP-problem, i.e. it has no import and each derivation starting in an initial graph has a length that is polynomial in the size of the start graph, then the composition of the reduction and the semantic relation of tu_2 is in NP, too. While the reduction yields an output for every input in a polynomial number of steps, the following computation in tu_2 runs also in polynomial time, but it is nondeterministic because it may compute *TRUE* for some of its inputs while other computations for the same input end in deadlocks. Hence the sequential composition, which is the semantic relation of tu_1 due to the correctness of the translation, is nondeterministic, too, with polynomial runtime. By a similar reasoning, it turns out that tu_2 models an NP-complete problem if tu_1 does, i.e. if each NP-problem can be reduced to the

semantic relation of tu_1. So the graph-transformational variants of reductions may be used to investigate the class NP in the same way as ordinary reductions are useful. But as many interesting problems in NP are graph problems, graph-transformational reductions may be quite suitable.

As an illustrating example, we specify a reduction from the Hamiltonian-path problem *HP* into the unit *long-simple-path*. We assume that *HP* is a predicate with the typing *HP*: *graph* → *bool* that yields *TRUE* for an input graph *G* if and only if *G* has a simple path that visits all nodes. An explicit specification by graph transformation is not needed, but it would look similar to *simple-path*, only making sure that all nodes are involved. Due to the typing, the reduction must consist of a graph transformation unit of the type *HP-2-lsp*: *graph* → *graph* × *nat* that copies the input graph as output graph and computes the number of nodes minus one of the input graph as second output. For this purpose, the product of the units *mark-all-nodes*, *graph* and *nat0* will be used. The unit *graph* = (*graph*, ∅, ∅, *true*, *graph*) takes unlabelled graphs as initial and terminal graphs without import and rules such that its semantics is the identity relation on *SEM*(*graph*), i.e. the input graph becomes the output graph. The unit *mark-all-nodes* consists of unlabelled graphs as initial graphs, of one rule *mark* that replaces the unlabelled flag by another flag (*ok*-labelled for example), and of graphs without unlabelled flags as terminal graphs. This is an auxiliary unit the meaning of which is that each derivation from an initial to a terminal graph has the number of nodes as length. Hence an action tuple that applies the rule *mark* in the first component allows one to count the number of nodes.
Summarizing, we get the following specification.

HP-2-lsp: *graph* → *graph* × *nat0*
prod: *mark-all-nodes* × *graph* × *nat0*
copy: 1 → 2
actions: b_0 = (*mark*, −, −)
b_1 = (*mark* , −, *succ*)
cond: b_0 ; b_1*

Note that the length of all computations is bounded by the number of nodes of the input graph and that each computation can be prolonged until all nodes are marked. As one always marks the first node without increasing the initial integer 0 and as all other nodes are marked while the integer is increased by 1 in each step, one ends up with the number of nodes minus 1 as integer output. And the runtime of *HP-2-lsp* is linear. If one composes the semantic relation of *HP-2-lsp* with that of *long-simple-path*, it returns *TRUE* if and only if the original input graph has got a simple path of a length that is the number of nodes minus 1 such that it visits all nodes. In other words, the translation is correct. And as the Hamilton-path problem is NP-complete, our reduction shows that *long-simple-path* is also NP-complete (which is already well known in this case).

Conclusion

In this paper, we have given an introductory survey of graph transformation with graphs, rules, rule application, graph class expressions, and control conditions as basic features. As all the concepts are handled in a generic, parametric way, this covers nearly all the graph transformation approaches one encounters in the literature (see, e.g., Rozenberg (1997) for an overview). Readers who are interested to see a spectrum of applications of graph transformation and its relation to the theory of concurrency are referred to the *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2 and 3* (Ehrig & Engels & Kreowski & Rozenberg, 1999; Ehrig & Kreowski & Montanari & Rozenberg, 1999).

In addition, we have proposed the new concept of product types that allow one to transform a tuple of graphs by the synchronous transformation of the components. This is quite helpful to specify transformations with a flexible typing, i.e. with an arbitrary sequence of input graphs and an arbitrary sequence of output graphs. Moreover, the types of the input and output graphs need not be subtypes of the same type of graphs anymore. As a consequence, the product type is particularly useful if one wants to transform graph transformations into each other. Further investigation of the product type may concern the following aspects.

As we used graph-transformational versions of the truth values and the natural numbers in our illustrating examples, one may like to combine graph types with arbitrary abstract data types. In the presented definition, we consider the product of graph transformation units. But one may like to import products in units and to use components that are again products. Whether such a composite use of products works must be investigated.
The transformation of graph transformation units is only tentatively sketched in Section 4. It must be worked out how it helps to study refinement and semantic equivalence and other interesting relationships between graph transformation systems.

Acknowledgements

The research presented here was partially supported by the EC Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modelling Techniques).

References

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

Bardohl, R., Minas, M., Schürr, A., & Taentzer, G. (1999). Application of Graph Transformation to Visual Languages. 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. 105–180). Singapore: World Scientific.

Bottoni, P., Koch, M., Parisi-Presicce, F., & Taentzer, G. (2000). Consistency Checking and Visualization of OCL Constraints. In A. Evans, S. Kent, & B. Selic (Eds.), *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*, Lecture Notes in Computer Science Vol. 1939 (pp. 294–308). Springer.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., & Löwe, M. (1997). Algebraic Approaches to Graph Transformation – Part I : Basic Concepts and Double Pushout Approach. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations* (pp. 163–245). Singapore: World Scientific.

Drewes, F., Kreowski, H.-J., & Habel, A. (1997). Hyperedge Replacement Graph Grammars. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations* (pp. 95–162). Singapore: World Scientific.

Ehrig, H., Engels, G., Kreowski, H.-J., & Rozenberg, G. (Eds.) (1999). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. Singapore: World Scientific.

Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., & Corradini, A. (1997). Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations* (pp. 247–312). Singapore: World Scientific.

Ehrig, H., Kreowski, H.-J., Montanari, U., & Rozenberg, G. (Eds.) (1999). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. Singapore: World Scientific.

Engelfriet, J., & Rozenberg, G. (1997). Node Replacement Graph Grammars. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations* (pp. 1–94). Singapore: World Scientific.

Engels, G., Hausmann, J.H., Heckel, R., & Sauer, S. (2000). Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, & B. Selic (Eds.), *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*, Lecture Notes in Computer Science Vol. 1939 (pp. 323–337). Springer.

Engels, G., Heckel, R., & Küster, J.M. (2001). Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In M. Gogolla, & C. Kobryn (Eds.), *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, Lecture Notes in Computer Science Vol. 2185 (pp. 272–286). Springer.

Fischer, T., Niere, J., Torunski, L., & Zündorf, A. (2000). Story Diagrams: A new Graph Transformation Language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Proc. Theory and Application of Graph Transformations*, Lecture Notes in Computer Science Vol. 1764 (pp. 296–309). Springer.

Heckel, R., Engels, G., Ehrig, H., & Taentzer, G. (1999). Classification and Comparison of Module Concepts for Graph Transformation Systems. 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. 669–689). Singapore: World Scientific.

Klempien-Hinrichs, R., Knirsch, P., & Kuske, S. (2002). Modeling the Pickup-and-Delivery Problem with Structured Graph Transformation. In H.-J. Kreowski, P. Knirsch, *Proc.*

APPLIGRAPH Workshop on Applied Graph Transformation, Satellite Event of ETAPS 2002 (pp. 119–130).

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

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

Kuske, S. (2000). More about control conditions for transformation units. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Proc. Theory and Application of Graph Transformations*, Lecture Notes in Computer Science Vol. 1764 (pp. 323–337). Springer.

Kuske, S. (2001). A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In M. Gogolla, & C. Kobryn (Eds.), *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, Lecture Notes in Computer Science Vol. 2185 (pp. 241–256). Springer.

Kuske, S., Gogolla, M., Kollmann, R., & Kreowski, H.-J. (2002). An Integrated Semantics for UML Class, Object, and State Diagrams based on Graph Transformation. In M. Butler, & K. Sere (Eds.), *3rd Int. Conf. Integrated Formal Methods (IFM'02)*, Lecture Notes in Computer Science Vol. 2335 (pp. 11–28). Springer.

Nagl, M. (Ed.) (1996). *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Lecture Notes in Computer Science Vol. 1170. Springer.

Petriu, D.C., & Sun, Y. (2000). Consistent Behaviour Representation in Activity and Sequence Diagrams. In A. Evans, S. Kent, & B. Selic (Eds.), *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*, Lecture Notes in Computer Science Vol. 1939 (pp. 359–368). Springer.

Pratt, T.W. (1971). Pair grammars, graph languages, and string-to-graph translations. Journal of Computer and System Sciences, 5(6), 560–595.

Rozenberg, G. (Ed.) (1997). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. Singapore: World Scientific.

Schürr, A. (1994). Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer (Ed.), *Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science Vol. 903 (pp. 151–163). Springer.

Schürr, A. (1997). Programmed Graph Replacement Systems. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations* (pp. 479–546). Singapore: World Scientific.

Sleep, R., Plasmeijer R., & van Eekelen, M. (Eds.) (1993). *Term Graph Rewriting: Theory and Practice*. John Wiley.