Graph Transformation for Specification and Programming*

Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann,
Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, Gabriele Taentzer†

Abstract

The framework of graph transformation combines the potentials and advantages of both, graphs and rules, to a single computational paradigm. In this paper we present some recent developments in applying graph transformation as a rule-based framework for the specification and development of systems, languages, and tools. After reviewing the basic features of graph transformation, we discuss a selection of applications, including the evaluation of functional expressions, the specification of an interactive graphical tool, an example specification for abstract data types, and the definition of a visual database query language. The case studies indicate the need for suitable structuring principles which are independent of a particular graph transformation approach. To this end, we present the concept of a transformation unit, which allows systematic and structured specification and programming based on graph transformation.

keywords: Graph transformation, rule-based specification, transformation units, structuring.

1 Introduction

Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples. Areas like language definition, logic and functional programming, algebraic specification, term rewriting, theorem proving, concurrent processes,

*This work was partially supported by the Deutsche Forschungsgemeinschaft, the ESPRIT Basic Research Working Group Computing by Graph Transformation (COMPUGRAPH II), the ESPRIT Working Group Applications of Graph Transformation (APPLIGRAPH), and the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems) through the Universities of Aachen, Berlin, Bremen, Hildesheim, and Paderborn.

†The author's addresses: M. Andries: Tractebel Information Systems (Brussels), e-mail: andries@wi.leidenuniv.nl, G. Engels: Universität-GH Paderborn, e-mail: engels@uni-paderborn.de, A. Habel: Universität Hildesheim, e-mail: habel@informatik.uni-hildesheim.de, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump: Universität Bremen, e-mail: {hof,kreo,kuske,det} @informatik.uni-bremen.de, A. Schürr: Universität der Bundeswehr, München, email: schuerr@informatik.unibw-muenchen.de, G. Taentzer: TU Berlin, e-mail: gabi@cs.tu-berlin.de .

and expert systems witness the prominent role of rules. Graph transformation, also known as graph rewriting or graph reduction, combines the potentials and advantages of both, graphs and rules, into a single computational paradigm.

More than 25 years ago, Rosenfeld et al. [PR69, RM72] in the USA, and Schneider, Ehrig, Pfender, and Wadsworth [Sch70, Sch71, Wad71, EPS73] in Europe introduced graph transformation for the generation, manipulation, recognition, and evaluation of graphs. Since then graph transformation has been studied in a variety of approaches, motivated by application domains such as pattern recognition, semantics of programming languages, compiler description, implementation of functional programming languages, specification of database systems, specification of abstract data types, specification of distributed systems, etc. This development is documented in proceedings and other collections of selected papers, in some monographs as well as in a handbook [CER79, ENR83, ENRR87, EKR91, Hab92, PvE93, SPvE93, SE94, CM95, CEER96, Roz97].

This paper surveys some recent developments in applying graph transformation to the specification and development of systems, languages, and tools. Section 2 recalls the basic concepts of a fairly general approach to graph transformation, and points out possible variations and generalizations. Section 3 reports on four case studies that demonstrate the usefulness of graph transformation in some typical application domains.

1. As known from the implementation of functional programming languages, the efficient evaluation of functional expressions requires to avoid multiple representations of the same subexpression. This leads to the idea of term graph rewriting (also known as jungle evaluation or graph reduction) which represents one of the oldest and most successful applications of graph transformation.
2. The Agg-system is a graphical tool for editing graphs and transforming them by graph transformation rules. Parts of the Agg-system have been specified by graph transformation to exemplify the use of graph transformation concepts in the specification of graphical CASE tools and similar systems.
3. Red/black trees are a kind of balanced binary trees with non-trivial rebalancing operations. The specification of red/black trees as a graph transformation system illustrates that, and how, (abstract) data types can be modelled adequately by means of graph transformation.
4. Visual languages are a promising research topic in the area of programming languages in general and database languages in particular, because the use of visual representations improves the comprehension of programs. As an example of the usefulness of graph transformation in this context, we report on the definition of the hybrid database query language HQL/EER by graph transformation.

The four case studies are a selection out of the wide range of applications for graph transformation one can find in the literature. They are chosen from the authors' recent work to illustrate the potentials of graph transformation for specification and programming within different fields of computer science.

In order to support applications like the four case studies properly and to apply graph transformation in a realistic context, one needs structuring principles for graph transformation. In Section 4, we present the notion of transformation units as a structuring principle which incorporates the experiences reported in the case studies in particular. Moreover, we illustrate how specification and programming with graph transformation is performed with transformation units by reformulating some aspects of the case study of term graph rewriting. Transformation units differ from other graph transformation concepts in at least the following two respects:

1. Graphs exist in many variants: directed, and undirected; unlabelled, labelled, and attributed; as simple, multi-, and hypergraphs. Moreover, there are various ways to apply rules to graphs, and to control the process of iterated rule application. This has led to several competing graph transformation approaches. Each of them has its own merits, and the choice of a particular approach largely depends on the application one has in mind. And, each of them needs structuring concepts. Therefore, transformation units are made approach-independent.

2. The concept of a transformation unit offers a structuring principle for building large specifications and programs from smaller ones, and for reusing existing components. In particular, the concept of a transformation unit encapsulates rules, control conditions, descriptions of initial and terminal graphs, and allows to use other transformation units. Thus the application of local rules may be interleaved with calls to used transformation units. This interleaving semantics supports induction principles and modular proofs.

Specifying and programming with transformation units combines the advantages of graphs and rules and relies on graph transformation as a basis for interpreter semantics and theorem proving facilities.

The paper is addressed to readers who want to know how graph transformation works and how it is applied, independently of a particular graph transformation approach, that is, independently of a certain type of graphs and rules and a specific way of rule application. Hence, the paper may be particularly interesting to readers who are already familiar with a special graph transformation approach and want to know how this fits into the general idea and setting of applied graph transformation.

It should be mentioned that we concentrate in this paper on the sequential mode of graph transformation and do not discuss aspects of parallelism and concurrency although they deserve attention (cf. e.g. [EK76, Kre87, CMR$^+$97, Tae96]).

2 Concepts of Graph Transformation

In this section we recall the basic features of sequential graph transformation where a graph is derived from another graph by applying one rule after another. The presentation is not completely formal; it is oriented at the general approach given by Kreowski and Rozenberg [KR90] that combines many aspects of major approaches.

Based on labelled, directed graphs as underlying structures, we consider a fairly general type of rule application, covering many of the major notions encountered in the literature and, in particular, the features used in Section 3. Ignoring the technical details, the application of a rule to a graph, often called a direct derivation, defines a binary relation on graphs which can be iterated arbitrarily, yielding the derivation process. In this way, a set of rules gets an operational semantics. If one starts in a particular initial graph and collects all derivable graphs with only labels from a terminal label alphabet, one gets the notion of a graph grammar with its generated language. At the end of this section, we discuss some properties of graph transformation that are relevant for using it as a specification and programming paradigm. In Section 4, we will take up the ideas of graph transformation, independently of the particular type of rule application, as the basic elements of a structuring principle for graph transformation.

2.1 Graphs

As basic structures of graph transformation, we choose node- and edge-labelled, directed graphs. Such a graph consists of a set of labelled nodes and a set of labelled directed edges, each of which connects a pair of nodes. More formally, graphs are defined as follows.

Graphs. A *(labelled, directed) graph* $G = (NODES, EDGES, source, target, label)$ consists of a finite set $NODES$ of *nodes*, a finite set $EDGES$ of *edges*, two mappings *source* and *target*, assigning a *source* and a *target* node to each edge, and a mapping *label*, assigning a *labelling* symbol from a given alphabet to each node and each edge. An edge e in G goes from the source $source(e)$ to the target $target(e)$ and is *incident* to $source(e)$ and $target(e)$. The nodes and edges of G are also called *items*. G may contain *parallel edges* e, e' from a node v to a node v', even with the same label. (Such graphs are sometimes called *multigraphs*.)

In graphical representations, nodes are drawn as points (circles, boxes, etc.), and edges are drawn as arrows from their source to their target. Labels are written beside points and arrows, or inside circles, boxes, etc., or indicated by different shapes, colours, fill styles, etc. They are not shown if all nodes and edges have the same labelling. Sometimes, node identifiers are also incorporated in the graphical representations. Figure 1 shows three graphs with node identifiers.

A graph L is a *subgraph* of G, denoted by $L \subseteq G$, if the node and edge sets of L are subsets of the respective sets of G, and the source, target and label mappings of L coincide with the respective mappings of G. In Figure 1, the graph L is a subgraph of G.

L has an *occurrence* in G, denoted by $L \to G$, if there is a mapping *occ* which maps the nodes and the edges of L to the nodes and the edges of G, respectively, and preserves sources, targets, and labellings, that is, for each edge e in L, the source of the image of e coincides with the image of the source of e, the target of the image of e coincides with the image of the target of e, and for each item x in L, the label of the image of x coincides with the label of x. In particular, the occurrence of L in G given by *occ* is the subgraph of G consisting of the images of all nodes and edges. In the following, an occurrence of L in G will be also denoted by $occ\colon L \to G$ or simply by *occ* if L and G are known from the context. If the mapping is bijective, then L and G are *isomorphic*. The graph L in Figure 1 has an occurrence in H in which the nodes 1 and 3 are identified (denoted by 1=3) and node 2 in L corresponds to node 2 in H.

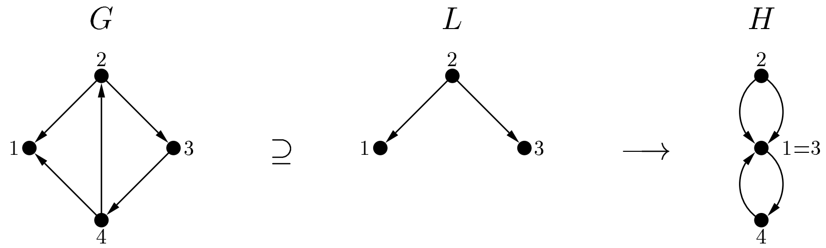


Figure 1: Three graphs where L is a subgraph of G and has an occurrence in H.

Variations and extensions of the graph concept. There are a number of variations and simple extensions of the graph concept. *Directed graphs in the classical sense* (i.e., graphs where the edge set is given by a binary relation on the set of nodes) coincide with directed graphs without parallel edges. *Undirected graphs* may be represented as directed graphs by replacing each undirected edge by two directed edges in opposite direction.

Hypergraphs generalize graphs in the sense that an edge has a sequence of source nodes and a sequence of target nodes. Finally, *hierarchical graphs* are graphs where a subgraph can be abstracted to one node and a bunch of edges between two abstracted subgraphs to one edge.[1] Another concept of hierarchical graphs was introduced by Pratt where nodes are labelled with graphs [Pra69, Pra71].

Typing is an important concept for specification and programming. Transferred to graphs, this means that labels in *typed graphs* are divided into classes, called types, and that edges of a certain type are restricted to be incident only to certain types of source and target nodes. In a language like PROGRES [Sch91b], typed graphs can be specified by so-called *graph schemata*.[2]

Attributed graphs are equipped with attributes. An attribute can be a number, a text, an expression, a list or even a graph. Attributes can be of different types and attribute operations compatible with these types are available to manipulate the attributes.[3]

2.2 Graph Transformation

Graph transformation consists of applying a rule to a graph and iterating this process. Each rule application transforms a graph by replacing a part of it by a graph. To this purpose, each rule r contains a left-hand side L and a right-hand side R. The application of r to a graph G replaces an occurrence of the left-hand side L in G by the right-hand side R. This is done by first removing a part of the occurrence of L from G, second, gluing R and the remaining graph D, and third, connecting R with D via the insertion of new edges between the nodes of R and those of D. The gluing of R and D is specified in the gluing component of a rule and the connection of R with D in the embedding component. Since the replacement shall often be done in a controlled way, rules may also contain application conditions.

The described procedure of graph transformation may look unnecessarily complicated to the reader not familiar with graph transformation. This is due to the generality of the framework presented in this section: It captures the properties of the main approaches in the literature, i.e. it includes them as special cases (see below). Hence, most of the major approaches are indeed simpler than the general framework presented here.

Rules. A *(graph transformation) rule* $r = (L, R, K, glue, emb, appl)$ consists of two graphs L and R, called the *left-hand side* and the *right-hand side* of r, respectively, a subgraph K of L called the *interface graph*, an occurrence *glue* of K in R, relating the interface graph with the right-hand side, an *embedding relation emb*, relating nodes of L to nodes of R, and a set *appl* specifying the *application conditions* for the rule.

Application of rules. An *application* of a rule $r = (L, R, K, glue, emb, appl)$ to a given graph G yields a resulting graph H, provided that H can be obtained from G in the following five steps:

1. CHOOSE an occurrence of the left-hand side L in G.
2. CHECK the application conditions according to *appl*.

[1] Examples of hypergraphs are given in Section 3.1; a hierarchical graph can be found in Section 3.2.
[2] Examples of typed graphs and graph schemata are presented in Sections 3.3 and 3.4.
[3] Examples of attributed graphs are given in Sections 3.2 and 3.4.

3. REMOVE the occurrence of L up to the occurrence of K from G as well as all *dangling edges*, i.e. all edges incident to a removed node. This yields the *context graph* D of L which still contains an occurrence of K.
4. GLUE the context graph D and the right-hand side R according to the occurrences of K in D and R. That is, construct the disjoint union of D and R and, for every item in K, identify the corresponding item in D with the corresponding item in R. This yields the *gluing graph* E.
5. EMBED the right-hand side R into the context graph D according to the embedding relation *emb*: For each removed dangling edge incident with a node v in D and the image of a node v' of L in G, and each node v'' in R, a new edge (with the same label) incident with v and the node v'' is established in E provided that (v', v'') belongs to *emb*.

The application of r to G yielding H is called a *direct derivation* from G to H through r and is denoted by $G \Rightarrow_r H$ or simply by $G \Rightarrow H$. Figure 2 illustrates the steps which have to be performed when applying a rule $(L, R, K, glue, emb, appl)$ where *appl* requires the occurrence of L in G to be isomorphic to L, and *emb* relates the unfilled nodes of L and R.

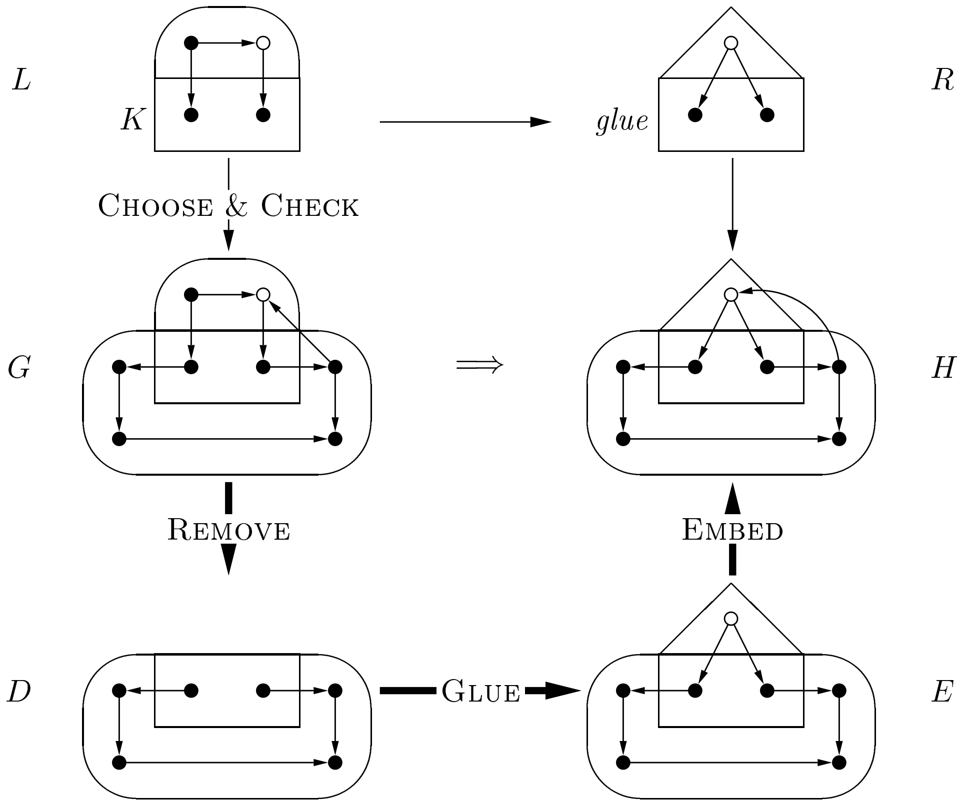


Figure 2: Illustration of a graph transformation step

Given a graph G and a rule $r = (L, R, K, glue, emb, appl)$ together with an occurrence of L in G, there is a variant of constructing the derived graph that is more general than the one above, but less intuitive. In this case, all nodes and edges of the occurrence are removed that are images of corrresponding items of the left-hand side outside the interface graph. The gluing step is adapted accordingly. This means that in general the interface graph is not kept invariant.

While all other special cases of graph transformation we consider in this paper fit into the more special notion of rule application, the so-called single-pushout approach discussed in the next paragraph and used in Section 3.2 applies rules in the more general sense.

Note that in practical applications, the step REMOVE needs not to delete all dangling edges but only those not replaced by a new one in EMBED. If a rule application is implemented in this way, REMOVE does not yield a graph in general. However, from a theoretical point of view (e.g. for correctness proofs) it is useful to obtain a graph after each intermediate step of a rule application.

Graph transformation approaches. For readers familiar with graph transformation, it shall be mentioned that the presented framework combines the gluing aspects of the algebraic approaches [Ehr79, Löw93, CMR+97, EHK+97] and the embedding aspects of the set-theoretic approaches [Nag87, Roz87, ER97] in the 1-context, i.e. the direct neighbourhood of the occurrence, and additional application conditions. The double-pushout approach [Ehr79, CMR+97], the single-pushout approach [Löw93, EHK+97], the node replacement approach [Roz87, ER97], as well as the set-theoretic approach used in PROGRES [Sch90] can be seen as special cases of this framework. To make this more transparent, let us consider some special cases.

1. CHOOSE & CHECK. In general, the occurrences of L in G are allowed to identify nodes or edges. In practice, one often uses the *injectivity condition* which requires the occurrence of L in G to be isomorphic to L. Less restrictive special application conditions are the *contact condition* and the *identification condition*. The contact condition assures that no dangling edges will arise in REMOVE. It requires that whenever the image of a node in L contacts some edge not in the image of L, then the node must be in K. The identification condition requires that the occurrence of L in G may only identify nodes and edges in the interface graph K. Other interesting examples of application conditions are context conditions. They require or forbid the existence of nodes, edges, or certain subgraphs in the given graph. A general treatment of application conditions can be found in [EH86, Kre93, HHT96].
2. REMOVE. Only if the contact condition is not required, the application of a rule may remove edges from the context of its occurrence (the dangling ones).
3. GLUE. In the special case where the occurrence of K in R is isomorphic to K, the result of the gluing step may be obtained by adding R (up to K) to D. In the special case where the interface graph K is empty, R is added disjointly to D.
4. EMBED. In the special case where *emb* is the empty relation, no additional edges are inserted. In general, *emb* allows embedding in the 1-context consisting of all nodes with an incident dangling edge: Every dangling edge incident with the image of a node v' of L in G is redirected to the node v'' of R if (v', v'') is in *emb*.

Note that special forms of rules correspond to rules in other graph transformation approaches:

1. Rules without application conditions and with empty embedding relation correspond to single-pushout rules.
2. Rules with contact and identification condition and with empty embedding relation correspond to double-pushout rules.

3. Rules with a single node on their left-hand side and empty interface graph correspond to node replacement rules.
4. Rules where the occurrence *glue* is isomorhic to the interface graph correspond to PROGRES rules (restricted to embedding in the direct neighbourhood of occurrences).

Although the introduced notion of graph transformation is quite general, it does not cover all approaches one encounters in the literature. On one hand, we discuss a sequential mode of graph transformation so that parallel graph transformation is not covered (cf. e.g. [EK76, Kre87, CMR$^+$97, Tae96]). On the other hand, the application of a rule is strictly local with respect to the removal and addition of nodes and edges. Hence, approaches are not covered that allow global effects in a rule application (cf. e.g. [Nag87, FW90, PH96]).

Extensions of the graph transformation concept. When typed graphs are transformed, the application of a rule has to preserve certain typing constraints. The transfomation of attributed graphs may depend on certain attribute values of the occurrence (another kind of application condition), and the application may compute new attributes with the given attribute operations.

2.3 Graph Transformation Systems

Given the notions of a rule and a direct derivation, graph transformation systems, graph grammars, and graph languages can be defined as usual. A set P of rules is the simplest form of a *graph transformation system*; a set P of rules together with an *initial* graph S and a set T of *terminal* labels forms a *graph grammar*. Given a set P of rules and a graph G_0, a sequence of successive direct derivations $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ is a *derivation* from G_0 to G_n by rules of P, provided that all used rules belong to P. The graph G_n is said to be *derived* from G_0 by rules of P. The set of all graphs labelled with symbols of T only that can be derived from the initial graph S by rules of P, is the *language* generated by P, S, and T. Slightly more general, one may consider I and T specifying initial and terminal graphs respectively such that a set P of rules induces a binary relation on graphs relating each initial graph with all terminal graphs that can be derived from the initial graph.

Note the non-determinism occurring in graph transformation: We first have to choose one of the rules applicable to a given graph. Furthermore, the chosen rule may be applicable at several occurrences of its left-hand side. The result of a graph transformation depends on these choices which are still completely arbitrary. This non-determinism may be restricted by control conditions in several ways:

1. by prescribing an order in which rules have to be applied,
2. by determining the next rule in dependence on the previous one(s), or
3. by applying a rule according to its priority.

Prototypes of such transformation systems are matrix systems as considered in formal language theory, programmed graph transformation systems (see e.g. Bunke [Bun79, Bun82] and Schürr [Sch97]), and graph transformation systems with priorities (see Litovsky and Métivier [LM93]).[4]

[4] In Section 4, we introduce the notion of a transformation unit that encapsulates the four discussed components of graph transformation systems, i.e. a set of rules, descriptions of initial and terminal graphs, and control conditions, and allows additionally the use of other transformation units.

2.4 Properties of Graph Transformation

In the following, properties of graph transformation are discussed which are of interest if graph transformation is considered as a specification and programming method. The satisfaction of the discussed properties are not requested in general but may be helpful in certain contexts.

Locality. The rule-based character of graph transformation ensures a certain degree of locality of action as it manipulates mainly the occurrence of a rule. This principle is weakened in some respects. These may be non-local application conditions and non-local computations of new attributes. The deletion of dangling edges and the embedding mechanism as described above do not destroy the locality property since they concern the direct neighbourhood of the occurrence only. Using programmed graph transformation, the principle of locality is no longer obeyed.

Invertability. In many systems, such as databases and graphical tools, undo operations are desirable to reconstruct a previous state. Using graph transformation for specification, undo operations may be described by inverted rule applications. The application of a rule can be inverted if it satisfies the contact condition, the identification condition, if the morphism *glue* is injective, and if the embedding relation is empty (see [Ehr79]). Furthermore, all application conditions given for the left-hand side have to be transformed into equivalent right-hand side conditions. Moreover, in the case of attributed graphs, the transformation concerning the attribute part has to be invertible, too.

Consistency conditions. Consistency conditions concerning the existence and non-existence of subgraphs, cardinality restrictions for the number of in- and outgoing edges, etc. can be proven for all derived graphs. Some of the consistency conditions can be transformed into equivalent application conditions in the sense that, given a consistent graph, a transformation where these application conditions are satisfied produces a consistent graph again [HW95].

Independence and confluence. Two direct derivations $G \Rightarrow_{r_1} G_1$ and $G \Rightarrow_{r_2} G_2$ *commute* if there is a graph H such that $G_1 \Rightarrow_{r_2} H$ and $G_2 \Rightarrow_{r_1} H$. In the double-pushout approach, this is the case if the occurrences of two steps do not overlap in deleted nodes or edges [EK76]. A graph transformation system is *confluent* if for each two derivations $G \Rightarrow^* G_1$ and $G \Rightarrow^* G_2$ there is a graph H such that $G_1 \Rightarrow^* H$ and $G_2 \Rightarrow^* H$. Confluence implies that every graph can be transformed into at most one irreducible graph. To prove that a system is confluent, there is a method (in the double-pushout approach) based on the computation of *critical pairs* which are pairs of overlapping direct derivations [Plu93a].

Termination. A graph transformation system is called *terminating* if infinite derivations $G_1 \Rightarrow G_2 \Rightarrow G_3 \Rightarrow \ldots$ are impossible. Simple sufficient conditions for termination are that each rule reduces the size of a graph or the number of occurrences of a fixed subset of labels. In general, however, termination is undecidable [Plu98]. A sufficient and necessary condition for termination (in the double-pushout approach) is that certain restricted derivations, so-called forward closures, always terminate [Plu95]. Forward closures can be inductively generated from the rules and have the property that each step in the derivation depends on some previous step.

Complexity. One central problem of graph transformation is the efficient matching of the left-hand side of a rule to a subgraph of the current working graph. Similar problems can also occur when testing the application conditions. Many ideas have been developed for an efficient implementation of the matching problem: A variant of OPS-5's RETE matching algorithm may be used to maintain the set of all occurrences of a set of rules across derivation sequences [BGT91]. Other approaches are based on building search plans for matching purposes [RS97]. These search plans start at already known nodes or use indexes over node labels, attribute values, and the like to locate start nodes efficiently. Then they traverse edges — starting at known nodes — to extend a partial match step by step. Well-known techniques from constraint logic programming languages may be and are used to increase the probability of early failures and to reduce the number of candidate nodes to be considered [Zün95].

2.5 Existing Languages and Tools

Until now there are only few languages and tools available which are based on graph transformation. One of the very early visual programming languages based on graph transformation has been PLAN2D [DFS75]. Graph-transformation-based tools are PAGG [Göt88], Graph$^{\mathrm{Ed}}$ [Him91], PLEXUS [Wan91], PROGRES [Sch91b], Dactl [GKS91] and AGG [LB93]. Most of them contain a graph editor for drawing graphs, and an interpreter for graph transformation. The editors usually support graphical representations, mouse/menu-driven interfaces, and different views on graphs, like on different sections of a graph, different layouts or different levels of abstraction. The interpreters mostly are able to apply rules at occurrences which are selected by the user, or automatically. If the language offers programmed graph transformation, some of the interpreters provide a backtracking mechanism to perform the prescribed applications of rules. Moreover, some of them offer a type checker, a checker for graph-theoretic properties, automatic layout, or cross-compilers to Modula-2 and C programs.

3 Some Case Studies in Graph Transformation

In this section, we discuss four case studies that demonstrate typical applications of graph transformation. First, term graph rewriting for the evaluation of functional programs is presented as one of the oldest and most successful application domains. Second, parts of a graphical tool for editing and transforming graphs, the AGG-system, are specified to illustrate the usefulness of graph transformation for tool development. Third, a particular type of balanced trees with sophisticated rebalancing operations is specified as an example of an abstract data type specification. Finally, it is illustrated how the visual database query language HQL/EER is defined, indicating that graph transformation may play a similar role in the area of visual languages as context-free Chomsky grammars do in traditional programming language design. The latter two case studies are based on the PROGRES framework that provides an already far developed tool environment. The first and the second case studies employ algebraic approaches.

Though different approaches are used in the case studies, all of them may be considered as special cases of the graph transformation approach introduced in the previous section. Above all, the basic idea is always the same: If a system performs local manipulations of complex, structured, maybe graphical objects, graph transformation is a good formalism to specify and program it.

3.1 Term Graph Rewriting

Equations over functional expressions are a common concept in specification and programming. They are an essential ingredient in functional programming, algebraic specification of abstract data types, and program verification. In this context, equations are used to compute the value of an expression or to prove the equality of two expressions. In both cases, equations are employed as directed rewrite rules which replace subterms (see [DJ90, BN98, Klo92]).

To implement rewriting efficiently, it is not adequate to represent expressions as strings or trees. This is because for these data structures, certain equations enforce a copying of subexpressions and hence multiply evaluation work. For example, the application of a distributive law

$$\mathtt{x} \times (\mathtt{y} + \mathtt{z}) = (\mathtt{x} \times \mathtt{y}) + (\mathtt{x} \times \mathtt{z})$$

to a term of the form $s \times (t + u)$ duplicates the subterm s which subsequently has to be evaluated twice.

This kind of inefficiency can be overcome when expressions are represented by graph-like structures which allow to *share* common subexpressions. For instance, the term $(s \times t) + (s \times u)$ resulting from the just mentioned rewrite step can be compactly represented by the graph structure in Figure 3 (where the triangles stand for representations of the subterms s, t, and u). With this representation, the subterm s has to be evaluated only once. To put it in another way, an evaluation of s in the graph corresponds to a parallel evaluation of two occurrences of s in the string or tree representation.

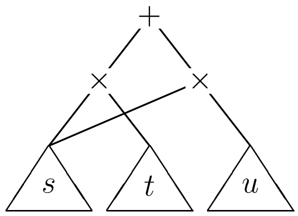


Figure 3: A graph structure representing $(s \times t) + (s \times u)$

The name *term graph rewriting* , introduced in [BvEG+87], has come into use for the evaluation of expressions by graph transformation. In the following, we outline how term graph rewriting works and state fundamental properties of this computational model. We also highlight the relations to the conventional term rewriting model. Our presentation is based on the "jungle evaluation" approach to term graph rewriting [HP88, HKP91, HP91, Plu93b, Plu93c].

3.1.1 Term graphs

Term graphs are directed, acyclic hypergraphs in which hyperedges are labelled with function symbols and variables such that each node represents a term. We consider hyperedges with one source node and a sequence of zero or more target nodes. The number of target nodes (inclusive repetitions) is the fixed number of arguments of the function symbol by which the hyperedge is labelled. Figure 7 shows three term graphs with function symbols +, FIB, SUC, and 0, where + is binary, FIB and SUC are unary, and 0 is a constant. Hyperedges are depicted as boxes with inscribed labels, and filled circles represent nodes. A line connects each hyperedge with its source node, while arrows point to its target nodes. The order in the target sequence is given by the left-to-right order of the arrows leaving the box.

A node v in a term graph G represents the term

$$term_G(v) = \begin{cases} label(e) & \text{if } target(e) \text{ is empty,} \\ label(e)\,(term_G(v_1), \ldots, term_G(v_n)) & \text{if } target(e) = v_1 \ldots v_n, \end{cases}$$

where e is the unique hyperedge with source v, $label(e)$ is the label of e, and $target(e)$ is the target sequence of e. We asssume that every term graph G has a node $root_G$ from which each other node is reachable, and we write $term(G)$ for $term_G(root_G)$. As an example, the left term graph in Figure 7 represents the term $\mathtt{FIB}(\mathtt{SUC}(\mathtt{SUC}(0))) + \mathtt{FIB}(\mathtt{SUC}(\mathtt{SUC}(0)))$.[5] Nodes representing variables (i.e. source nodes of hyperedges labelled with variables) are called *variable nodes*.

To explain term graph rewrite steps, we use hypergraphs of a particular shape. Let $\Diamond t$ be the *variable collapsed tree* for a term t, i.e. a term graph with $term(\Diamond t) = t$ in which only variable nodes are shared and in which each variable is represented by a unique node. For every term graph G, we denote by $\underline{G}$ the hypergraph obtained by removing all hyperedges labelled with variables.

3.1.2 Rewriting

Let $\mathcal{E}$ be a set of equations. We consider equations as left-to-right rewrite rules and require for each equation $l = r$ that the variables in r occur also in l, and that l is not a variable.

Let G and H be term graphs. Then there is an *evaluation step* from G to H, denoted by $G \Rightarrow_{\mathcal{E}} H$, if H is (up to isomorphism) the term graph constructed from G as follows:

(1) Choose an equation $l = r$ in $\mathcal{E}$ and an occurrence $g \colon \underline{\Diamond l} \to G$ of $\underline{\Diamond l}$ in G.

(2) Remove the hyperedge with source $g(root_{\Diamond l})$, yielding a hypergraph D.

(3) Glue together D and $\underline{\Diamond r}$ by
  (i) constructing the disjoint union $D + \underline{\Diamond r}$,
  (ii) identifying $g(root_{\Diamond l})$ with $root_{\Diamond r}$,
  (iii) identifying $g(v)$ with v', for all variable nodes v in $\Diamond l$ and v' in $\Diamond r$ satisfying $term_{\Diamond l}(v) = term_{\Diamond r}(v')$.

(4) Remove all nodes and hyperedges that are not reachable from $root_G$.

Steps (1) to (3) together can alternatively be described as the application of a so-called *evaluation rule* for the equation $l = r$. Evaluation rules are hypergraph transformation rules in the sense of Section 2. The shape of an evaluation rule for an equation $F(t_1, \ldots, t_k) = r$ is shown in Figure 4. The left-hand hypergraph L is $\underline{\Diamond F(t_1, \ldots, t_k)}$ and the middle hypergraph K is obtained from L by removing the topmost hyperedge. The right-hand hypergraph R is constructed by gluing K with $\underline{\Diamond r}$. The embedding relation *emb* and the set *cond* of application conditions are both empty.

Step (4) in the above construction of evaluation steps is known as *garbage collection* in the implementation of functional programming languages. Garbage collection can also be described by hypergraph transformation rules, see Section 4.

[5] We use + as an infix operator for better readability.

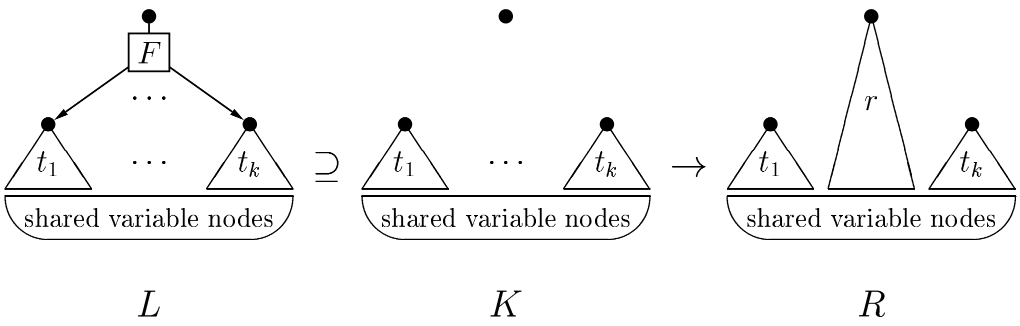


Figure 4: An evaluation rule (schematic)

Besides evaluation steps, we consider so-called collapse steps which enhance the degree of sharing in a term graph without changing the represented term. Given a term graph G with two distinct hyperedges e and e' that have identical labels and target sequences, the identification of e and e' (including their source nodes) realizes a *collapse step* $G \Rightarrow_{\mathcal{C}} H$.

To specify collapse steps by rules, a *collapse rule* as shown in Figure 5 is assigned to each function symbol and each variable. Collapse rules have an empty embedding relation and an application condition requiring that an occurrence of the left-hand side must not identify the two hyperedges. For instance, the collapse rule for the function symbol `FIB` is applied in the first rewrite step of Figure 7.

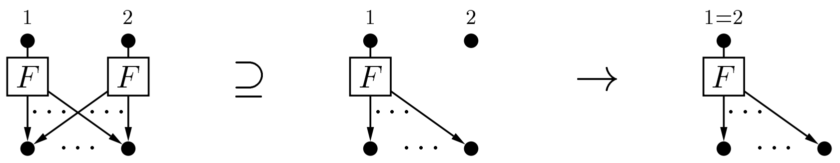


Figure 5: A collapse rule[6]

In standard implementations of functional programming languages, collapsing is usually not employed (but see [Kah92] for an implementation based on fully collapsed term graphs). We allow collapse steps for several reasons. Firstly, they are needed to deal properly with equations having repeated variables in their left-hand sides. Without collapsing, such an equation may not be applicable to a term graph although it is applicable to the represented term. Secondly, collapsing makes term graph rewriting complete for equational proofs in the same sense as term rewriting is. That is, two term graphs are equivalent with respect to the reflexive, symmetric and transitive closure of the rewrite relation whenever their represented terms are equivalent with respect to the given equations. This property—which fails without collapsing, see [Plu93b]—is particularly important when term graph rewriting is used in implementations of theorem provers. Finally, the example of the next subsection will demonstrate that collapsing may speed up the evaluation process drastically in certain cases. In this respect, collapsing is related to the *memoization* technique in functional programming (see [Hug85]) which has been studied for term graph rewriting in [Hof92].

3.1.3 An Example: the Fibonacci function

The following equations define the Fibonacci function on natural numbers (where SUC denotes the successor function):

$$
\begin{aligned}
\mathtt{FIB(0)} &= \mathtt{0} \\
\mathtt{FIB(SUC(0))} &= \mathtt{SUC(0)} \\
\mathtt{FIB(SUC(SUC(x)))} &= \mathtt{FIB(SUC(x))} + \mathtt{FIB(x)}
\end{aligned}
$$

The evaluation rule for the third equation is depicted in Figure 6. Figure 7 shows a collapse step and an evaluation step with the rule of Figure 6. Note that the two SUC-hyperedges in the middle term graph are removed by garbage collection.

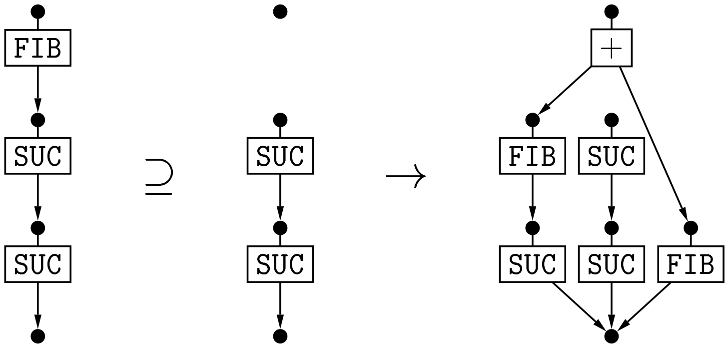


Figure 6: The evaluation rule for the third Fibonacci equation

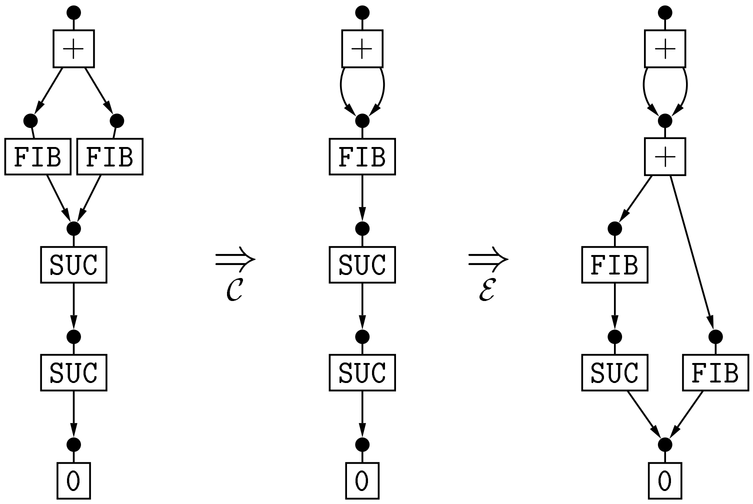


Figure 7: A collapse step and an evaluation step

[6]The notation 1=2 means that node 1 is glued with node 2.

The evaluation step corresponds to two consecutive term rewriting steps on the term represented by the middle term graph:

$$\begin{array}{ll} & \texttt{FIB(SUC(SUC(0)))} + \texttt{FIB(SUC(SUC(0)))} \\ \to & (\texttt{FIB(SUC(0))} + \texttt{FIB(0)}) + \texttt{FIB(SUC(SUC(0)))} \\ \to & (\texttt{FIB(SUC(0))} + \texttt{FIB(0)}) + (\texttt{FIB(SUC(0))} + \texttt{FIB(0)}) \end{array}$$

Suppose that we want to evaluate a term $t = \texttt{FIB(SUC(}\ldots\texttt{SUC(0)}\ldots\texttt{))}$ according to the three Fibonacci equations. Using term graph rewriting, this requires space and a number of evaluation and collapse steps linear in the size of t if a suitable strategy for applying evaluation and collapse steps is chosen (see [HP91]). In contrast, evaluating t by term rewriting (or by term graph rewriting without collapsing) requires space and a number of rewrite steps exponential in the size of t, no matter which rewrite strategy is used.

3.1.4 Properties of term graph rewriting

We consider the transformation of term graphs by evaluation and collapse steps in an arbitrary order. To this end, let $\Rightarrow_{\mathcal{R}}$ be the union of the relations $\Rightarrow_{\mathcal{E}}$ and $\Rightarrow_{\mathcal{C}}$. We write $t \to_{\mathcal{E}} u$ for a term rewrite step with an equation from $\mathcal{E}$. The reflexive-transitive closures of the relations $\Rightarrow_{\mathcal{R}}$ and $\to_{\mathcal{E}}$ are denoted by $\Rightarrow^*_{\mathcal{R}}$ and $\to^*_{\mathcal{E}}$.

Soundness and completeness. Term graph rewriting is sound in the sense that every step realizes a sequence of term rewrite steps: $G \Rightarrow_{\mathcal{R}} H$ implies $term(G) \to^*_{\mathcal{E}} term(H)$ [HP88, HP91] . Moreover, we have completeness with respect to the equivalence generated by the given equations. That is, $G \Leftrightarrow^*_{\mathcal{R}} H$ if and only if $term(G) \leftrightarrow^*_{\mathcal{E}} term(H)$, where $\Leftrightarrow^*_{\mathcal{R}}$ and $\leftrightarrow^*_{\mathcal{E}}$ are the reflexive-symmetric-transitive closures of $\Rightarrow_{\mathcal{R}}$ and $\to_{\mathcal{E}}$. It follows that term graph rewriting is a complete proof method for the equational theory of $\mathcal{E}$ [Plu93b, Plu93c].

Termination. If term rewriting with $\mathcal{E}$ is terminating (meaning that infinite chains of $\to_{\mathcal{E}}$-steps are impossible), then the same holds for $\Rightarrow_{\mathcal{R}}$. The converse, however, need not hold. So term graph rewriting terminates for more equation sets than term rewriting [HP88, HP91]. The reason is that the sharing created by equations with repeated variables in their right-hand sides may exclude certain term rewrite derivations. Termination of term graph rewriting can be proved by a *recursive path order* on term graphs which generalizes the corresponding order for terms (see [Plu97]).

Evaluation strategies. Evaluation strategies for term rewriting restrict the non-determinism of rewriting. Such strategies are called normalizing if they lead to an evaluated term whenever possible. For certain classes of equation sets, normalizing term rewriting strategies can be turned into normalizing term graph rewriting strategies [BvEG+87, Plu91a, Rao95].

Confluence. This property means that arbitrary rewrite sequences starting from a common object can be extended such that they end in a common object. Confluence implies that no object can be evaluated to more than one irreducible object. Term rewriting is confluent whenever term graph rewriting is, but the converse does not hold in general.

To test term graph rewriting for confluence, there is a method based on the analysis of *critical pairs* of term graph rewrite steps [Plu94]. A survey of (positive and negative) confluence results for term graph rewriting over various classes of equation sets is given in [AKP98].

Modularity. It is known that the union of two equation sets $\mathcal{E}$ and $\mathcal{E}'$, which are both terminating under term rewriting and have disjoint function symbols, may yield a non-terminating set $\mathcal{E} \cup \mathcal{E}'$. In contrast, termination of $\mathcal{E}$ and $\mathcal{E}'$ under term graph rewriting carries over to $\mathcal{E} \cup \mathcal{E}'$ [Plu91b]. This is a nice consequence of the fact that term graph rewriting terminates more often than term rewriting. If the two sets are additionally confluent, then so is the resulting set [Plu93b]. For both modularity properties, $\mathcal{E}$ and $\mathcal{E}'$ may have certain function symbols in common. See [KO95, Rao96] for recent extensions of the above modularity results.

3.1.5 Summary

Term graph rewriting is a computational model which is more efficient than the conventional term rewriting model and which is close to real implementations. In fact, term graph rewriting is a common implementation technique for lazy functional programming languages, as witnessed by the books of Peyton Jones [Pey87] and Plasmeijer and van Eekelen [PvE93].

Term graph rewriting is sound and complete for equational computations in the same sense as term rewriting is, but differs from the latter in properties like termination and confluence. With respect to termination and modularity, the graph model behaves even more friendly.

Variations of the here presented rewriting model have been considered by various authors. See, for example, [Sta80, Pad82, BvEG+87, GKM87, AK96]. Extensions deal with cyclic term graphs and infinite computations [FW90, KKSdV94], and with term graph rewriting for logic programming and equation solving [CMR+91, CR93, CW94, HP96]. An area related to term graph rewriting is graph reduction for the lambda calculus. From the numerous literature of this area we mention only [Wad71, Lam90, AL95].

3.2 Specification of an Interactive Graphical Tool

Often, the objects of an interactive graphical tool can be represented as graphs, and its operations can be modelled by graph transformation. Here we consider the AGG-system being developed at TU Berlin, a tool for editing and transforming graphs, and show how parts of it can themselves be specified by graph transformation (see also [LB93, TB94]).

3.2.1 The AGG-system

The AGG-system is a prototype implementation of the algebraic approach to graph transformation. It contains a flexible graph editor with a transformation component that applies user-selected rules at automatically or user-selected occurrences.

The editor supports the mouse/menu-driven creation and modification of graphs and rules. The underlying graphs of AGG are hierarchical graphs, with higher-order edges, i.e. edges between edges. Such graphs, also called AGG-graphs in the following, are useful to

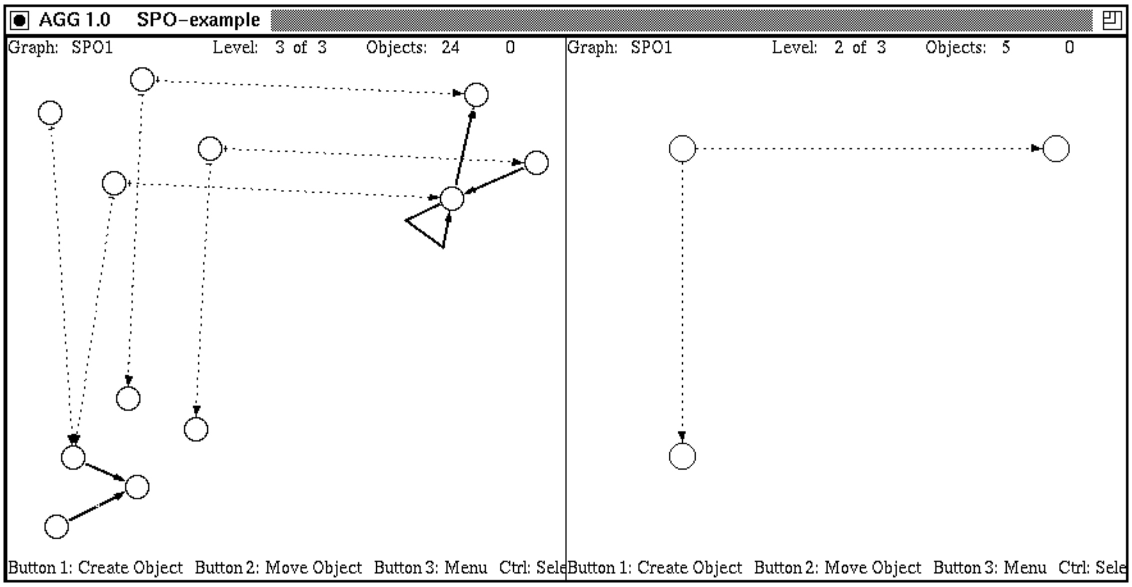


Figure 8: Rule and occurrence: graph level and diagram level

describe and manipulate graphs, rules and occurrences in a uniform way. They include the usual definition of graphs as a special case. Higher-order edges are used to represent relations between graphs as graphs again. Common relations between graphs are rules and occurrences of graphs. Edges between objects of any order, e.g. between an edge and a node, are also possible. Graphs can be abstracted to nodes, bundles of edges can be abstracted to single edges. The formal definition of this concept, which can be found in e.g. [LB93], regards graphs as a set of objects with three partial mappings: *source, target*, and *abstraction*. A node is an object whose *source* and *target* are undefined. Some rather obvious consistency conditions have to be enforced, e.g. the abstraction mapping has to be compatible with the source and target mapping, and it has to be acyclic.

The editor can show different views on a graph in different windows, and allows several graphs to be edited concurrently, each on different abstraction levels. An abstraction level may be shown in different windows. All windows can be used for editing as well as for transformation. Any change in one window immediately propagates to all windows showing that graph.

Some editing operations are tailored to the manipulation of rules, occurrences, and transformations: a *gluing* operation merges two objects to a new object that inherits the incidence relations of the original objects, and a *connecting copy* operation draws edges between all original objects and their copies.

Two consecutive snapshots shall illustrate how the Agg-system allows graphs, rules, and occurrences to be handled in a uniform way.

Figure 8 shows a rule together with an occurrence on two different abstraction levels, i.e. the graph level in the left window and the diagram level in the right window. In the left window, circles and solid arrows display graphs: the left-hand side of the rule (up left), its right-hand side (up right), and the host graph to which the rule shall be applied (down left). Dashed arrows display relations between these graphs: the *glue* component of the rule (horizontal dashed arrows), and its occurrence (vertical dashed arrows). In general, dashed arrows may connect edges to edges; see Figure 9, at the bottom.

The right window presents the same rule and occurrence on a higher abstraction level, the diagram level: The left-hand side of the rule, its right-hand side as well as the host

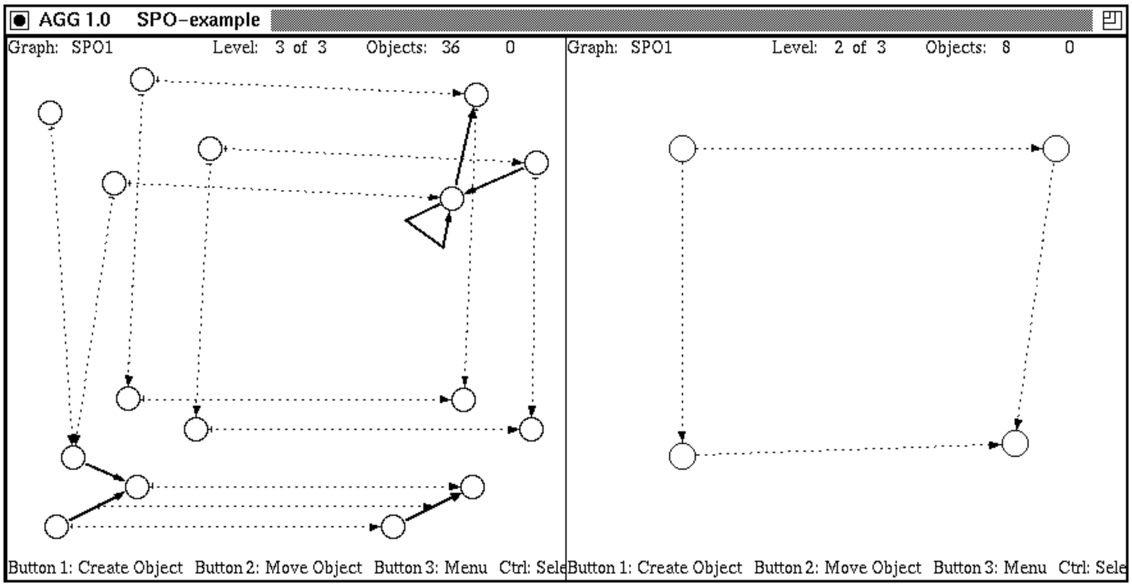


Figure 9: A graph transformation step: graph level and diagram level

graph are abstracted to a single node, respectively. The left upper node is the abstraction of the left-hand side, the right upper one of the right-hand side, and the lower one of the host graph. The *glue* and occurrence relation have been abstracted to single arrows: All dashed arrows pointing from nodes of the left-hand side to those of the right-hand side are represented by a single dashed arrow from the abstraction of the left-hand side to the abstraction of the right-hand side. Analogously, all dashed arrows from the left-hand side to the host graph are abstracted to a single dashed arrow pointing from the abstraction of the left-hand side to the abstraction of the host graph.

Applying the rule at the indicated occurrence leads to the situation shown in Figure 9. The diagram level of Figure 9 contains a rectangle where the derived graph and its relations to the host graph and the right-hand side of the rule are indicated. The graph level in the left window shows this in more detail. (The transformation component implements the single-pushout approach to graph transformation, see [Löw93].) Note that the node of the host graph, the two dashed arrows point to, is deleted because only one of the corresponding nodes in the left-hand side belongs to the interface graph. Moreover, the node in the right-hand side corresponding to this interface node is deleted together with all dangling edges.

In general, graphs may have more than two abstraction levels. Thereby, large graphs can be structured by hiding details in lower levels of abstraction.

3.2.2 Modelling the AGG-system by Graph Transformation

We now discuss some interesting aspects of specifying the AGG-system by graph transformation. The states of the AGG-system are described by attributed graphs in the sense of Section 2.1. We represent AGG-objects like windows, AGG-graphs, abstraction levels, AGG-nodes, AGG-edges, etc. by nodes, and the relations between them by edges. Properties of AGG-objects, like their position, type, state of selection, etc. are reflected by attributes of the corresponding nodes.

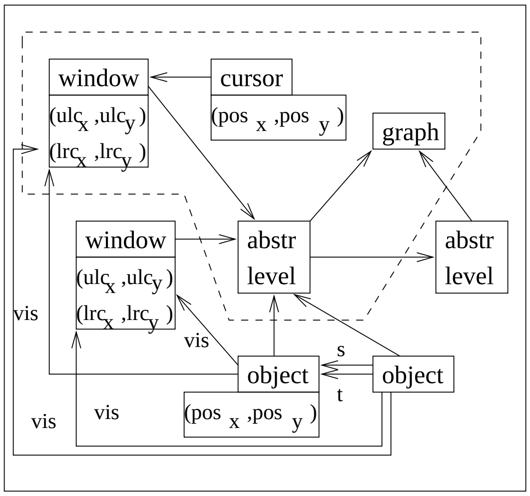


Figure 10: Section of an AGG-state modelled by a graph

The representation has three facets:

1. The *logical structure* of AGG-graphs is represented by nodes of type[7] `object` that are connected by source (`s`), target (`t`) and abstraction (`a`) edges. (`object`-nodes representing AGG-nodes do not have outgoing `s`- or `t`-edges.) In this way, higher-order edges and the abstraction mechanism can be modelled easily. In addition, `abstrlevel`-nodes represent a chain of abstraction levels for each graph. Every object points (by an untyped edge) to the `abstrlevel`-node it belongs to.
2. The *system state* of the AGG-system is specified by further nodes representing windows, the cursor etc. Selection and hiding of AGG-objects is described by special attributes of `object`-nodes.
3. The *layout of graphs* is described by different types of attributes, e.g. the position and type of AGG-nodes, the size and position of windows, the bending of AGG-edges, the visibility of AGG-objects, etc.

Figure 10 shows the representation of a state of the AGG-system with one graph, consisting of two abstraction levels: the lower one contains one node with a loop, and is displayed in two windows; the upper level is empty, and not displayed at all; the cursor indicates the current window. Only the position attributes of cursor, windows, AGG-objects are shown. (The position of an AGG-edge is given by the positions of its source and target, and its bending points, if any.) The initial state of the AGG-system is bordered with dashed lines; its initial graph is given by one abstraction level which is empty.

An editor operation or a transformation step in the AGG-system is modelled by a graph transformation according to the single-pushout approach mentioned in Section 2, extended with application conditions.

As an example, we consider how the AGG-operation `open new window` is modelled. This operation opens a new window which becomes the current window. Its size is adjusted by parameter values for the position of its upper left corner (*ulc*) and lower right corner (*lrc*). All objects of the previous current window are displayed in this new window as long as they fit into it; for each object it has to be decided whether it is visible in the new window, or not.

This operation is modelled by two co-operating graph transformation rules. The rule

[7] In this case study, nodes are labelled by their type.

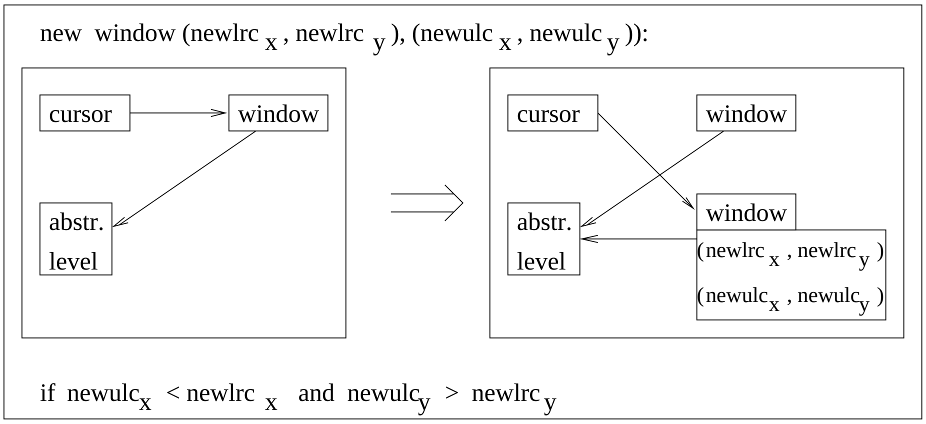


Figure 11: Rule for opening a new window

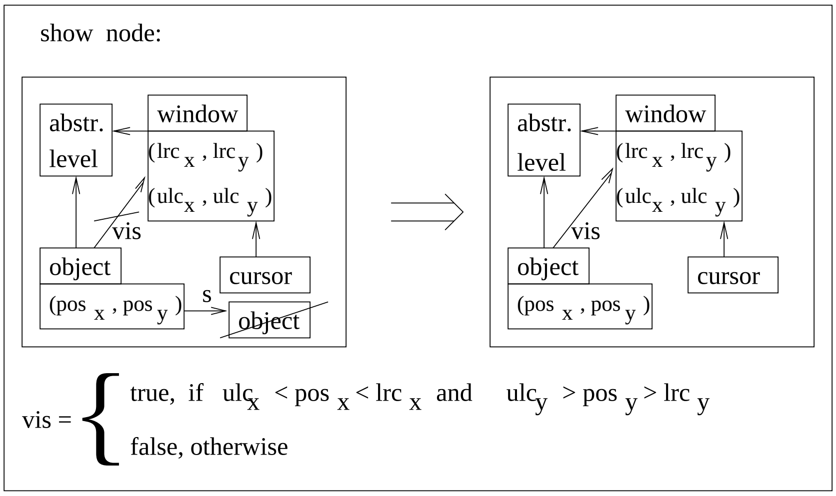


Figure 12: Rule for computing the visibility of nodes

`new window` is depicted in Figure 11. Its input parameters set the size attributes of the new `window`-node (if they are consistent). After its application the rule `show node` is applied as long as it is applicable. This rule determines the visible graph objects in the current window.

The application of `new window` works as presented in Section 2: CHOOSE an occurrence of the left-hand side in some AGG-state graph. CHECK the application condition (shown down left). REMOVE the edge issuing from the occurrence of the `cursor`-node. GLUE the right-hand side to this graph, i.e. insert a new `window`-node, and connect it by two edges, coming from the `cursor`-node, and leading to the `abstrlevel`-node, resp., and set its attribute values according to the rule's parameters. (There is no additional embedding step.)

The rule `show node` depicted in Figure 12 determines whether a graph object belonging to the current abstraction level is visible, or not. On the left-hand side of the rule, an `object`-node and a `vis`-edge are struck through to indicate a negative context condition.[8] (Implicitely, the `s`-edge is negative context, too.) To apply this rule, an occurrence of the left-hand side *without the negative context* has to be found, such that it cannot be extended by the "forbidden" nodes and edges. This asserts that the object represents

[8]As pointed out in Section 2, negative context conditions are application conditions. Hence, they fit into the general framework of graph transformation presented in Section 2.

an AGG-node (it must not issue an `s`-edge), and that its visibility edge has not been computed already. A rule `show edge` similar to `show node` is needed to compute the visibility of AGG-edges, depending on the visibility of their source and target nodes.

In order to perform the AGG-operation `open new window`, the rules `show node` and `show edge` have to be applied as long as possible after every application of `new window`.

Other AGG-operations, like insertion and deletion of nodes, edges and abstraction levels, gluing and copying, have been specified similarly by such a kind of controlled graph transformation. Together with the initial AGG-state (shown as part of Figure 10), we obtain a graph grammar describing all reachable AGG-states.

Consistency conditions. AGG-states satisfy consistency conditions, such as the following:

(1) There is exactly one current window.

(2) Every window shows objects of one abstraction level.

(3) Every object is either visible or not.

(4) The *ulc*- and *lrc*-values of a window are consistent, i.e. $ulc_x < lrc_x$ and $ulc_y > lrc_y$.

It has to be shown that the modelling of every AGG-operation preserves these consistency conditions. For `open new window`, it can be stated that it does not violate condition (1) since the cursor is moved to the new window. Rule `show node` (and similarly `show edge`) is applied to each object of the current abstraction level. Thus, condition (2) is preserved. According to their negative context conditions, these rules are applied to each object exactly once so that condition (3) is satisfied. Condition (4) is enforced by the application condition of rule `new window`.

Moreover, the consistency check can be done automatically by transforming all consistency conditions into equivalent appliciation conditions for the rule in the sense that, given a consistent graph, an application of rule `new window` where the additional application conditions are satisfied yields a consistent graph again. (Compare Section 2.4 (Consistency conditions).)

Further properties. The applications of rule `show node` to each AGG-node on the current abstraction level are independent of each other since they modify different parts of the AGG-state graph. Thus, they can also be applied in parallel, in order to speed up the computation process for a new AGG-state graph.

An undo/redo-mechanism is very helpful for editors. Modelling it here with graph transformation the given rules have to be inverted.[9] Looking at the AGG-operation `open new window` this is possible by defining an AGG-operation `close window` modelled by the application of the inverted rule of `show node`, now without application conditions, to each node of the current abstraction level (similarly for edges). Moreover, the inverted rule of `new window` has to be applied here also without its application condition.

Although this undo/redo-mechanism looks complicated the invertibility of rules offers a more or less automatic way to provide undo/redo-functionality.

3.2.3 Summary

The above description of AGG-states by graphs and AGG-operations by graph transformations is imperative, and rather close to object-based description methods. Its graphical

[9]Note that a single-pushout rule can be inverted if it is injective and satisfies the contact condition.

presentation demonstrates that graph transformation leads to intuitive, constructive specifications that can be checked thoroughly for consistency. Similar work has been done in [Zün95] where a development environment for PROGRES has been specified in PROGRES itself.

The biggest problem in developing a graph specification of the AGG-system has been its complexity. AGG allows different views of graphs to be displayed in several windows, and offers complex editor operations. *Refinement* possibilities for a stepwise incorporation of more (structural) information would help to manage this complexity. Starting with the logical description of AGG-graphs, the specification may be refined by incorporating the system states of AGG and later on tackling also the graphical layout of AGG-graphs. (Compare Subsection 3.2.2.) Tools for *consistency checking* would support the development of correct specifications.

It is planned to extend the AGG-system by a more comprehensive transformation component, a system manager, a consistency checker, a tool for checking the independence of transformations, and a tool for defining the layout of graphs. Such a system would offer a comprehensive set of tools for the specification of software by graph transformation. It is a challenge to specify the kernel of this extended system in itself, which is possible only if a *modularity* concept for graph transformation is available, and to show the consistency of this specification.

3.3 Specification of an Abstract Data Type

This and the following section report on the usage of the graph transformation-based language PROGRES for specifying and prototyping abstract data types as well as visual (programming) languages. The name PROGRES is a short-hand for **PRO**gramming with **G**raph **RE**writing **S**ystems.

PROGRES and its integrated programming environment have been developed by the graph grammar group of M. Nagl at RWTH Aachen. They are the first attempt to define and implement a graph transformation-based, strongly typed programming language with well-defined syntax, and static and dynamic semantics [Sch90, Sch91b]. Being a mixed textual and diagrammatic language, it permits quite different styles of programming, and supports

1. graphical as well as textual definition of graph (database) schemata with declaration of derived graph properties,
2. rule-oriented and diagrammatic specification of atomic graph transformation steps by means of parametrized graph transformation rules with complex preconditions, and
3. imperative programming of composite graph transformation processes by means of deterministic and non-deterministic control structures.

In some respect, PROGRES is rather similar to so-called visual database languages, as for instance HY+ [CM93], QBD* [ACS90], or GOOD [GPTvdB93]. But note that these languages are either pure query languages, or have only very limited support for the definition of complex transformations. In contrast, PROGRES and its programming environment support definition and manipulation of data to the same extent [Zün95].

3.3.1 Red/black trees and graph schemata

The language PROGRES was mainly designed for the specification of CASE tools [ELN+92]. But gradually it became obvious that it is also a kind of visual (database) programming language, and that it might even be used as a very high-level language for implementing graph-like data types.

The running example of this section are red/black trees (cf. [CLR90]), a special kind of almost height-balanced binary search trees. Red/black trees have edges (or sometimes nodes) of two different colours (black and red). They compute the heights of their subtrees to be balanced with respect to black edges only. As a consequence, the height of the left and the right subtree of a node may differ up to a factor of 2. This increases the time complexity of search operations, but reduces the necessary rebalancing overhead of insert and remove operations considerably. This example has been selected in order to demonstrate the advantages of programming with graph transformation rules in contrast to the usage of conventional programming languages. Red/black trees have rather complex tree restructuring operations, but their data model is just a binary tree with coloured nodes. Therefore, we have to emphasize that PROGRES allows for the definition of arbitrary directed (attributed) graphs and is not restricted to trees. Nevertheless, we do believe that it makes sense to specify even tree-like data structures with PROGRES, thereby profiting from the language's elaborated type system and its visual rule-oriented character.

The language's graph data model are attributed graphs (cf. Section 2.1) with typed nodes and typed directed edges. Node attributes, and binary relations between nodes are either extensionally defined and manipulated by graph transformation rules (as intrinsic attributes and edges) or they are intensionally defined and maintained by the PROGRES runtime system (as derived attributes and materialized path expressions). A lazy and incremental evaluation process keeps derived data in a consistent state [KSW95].

The PROGRES graph schema for red/black trees can be defined using either a graphical (Entity-Relationship diagram like) representation or/and a more detailed textual representation. The latter one is displayed in Figure 13 and defines the following invariants:

1. Different nodes in a tree have different integer values as key attributes (cf. declaration of the intrinsic, i.e. not derived, attribute `Value` in `(1)`).
2. A red/black tree contains only one type of nodes, but different types of edges (cf. declaration of node type `RBNode` in `(11)` and edge types `b(lack)l(eft)` in `(2)`, etc.).
3. Any parent/child edge is either red or black and leads either to a left or a right child (cf. declaration of derived relations/paths `left` in `(6)`, `right` in `(7)`, and `parent` in `(8)`).
4. Any node has at most one left and right child and at most one parent node (cf. cardinality constraints of the form `[0:1]` in `(6)`, `(7)` and `(8)`).
5. The height of a subtree is the maximum number of black edges on any path to one of its leaves (cf. declaration of the derived attributes `height`, `lheight`, and `rheight` in `(1)`).
6. The height of the left subtree of a node is always equal to the height of its right subtree (cf. declaration of restriction `balanced` in `(9)`).
7. The target of a red coloured edge is never the source of another red coloured edge (cf. declaration of restriction `correctColored` in `(10)`).

```
(1)  node class RBNODE
       intrinsic
         key Value : integer;
       derived
         Height : integer = max ( self.lHeight, self.rHeight );
         lHeight = [ self.-bl->.Height + 1 | self.-rl->.Height | 0 ];
         rHeight = [ self.-br->.Height + 1 | self.-rr->.Height | 0 ];
     end;
(2)  edge type bl : RBNODE [0:1] -> RBNODE [0:1];
(3)  edge type br : RBNODE [0:1] -> RBNODE [0:1];
(4)  edge type rl : RBNODE [0:1] -> RBNODE [0:1];
(5)  edge type rr : RBNODE [0:1] -> RBNODE [0:1];
(6)  path left : RBNODE [0:1] -> RBNODE [0:1] =
       [ -bl-> | -rl-> ]
     end;
(7)  path right : RBNODE [0:1] -> RBNODE [0:1] =
       [ -br-> | -rr-> ]
     end;
(8)  path parent : RBNODE [0:1] -> RBNODE [0:1] =
       [ <=left= | <=right= ]
     end;
(9)  restriction balanced : RBNODE =
       valid (self.lHeight = self.rHeight)
     end;
(10) restriction correctColoured : RBNODE =
       not with (<-rl- or <-rr-) or not with (-rl-> or -rr->)
     end;
(11) node type RBNode : RBNODE end;
(12)       ...
```

Figure 13: Cutout of red/black tree graph schema

Figure 14 contains two examples of binary search trees with coloured edges. The first one in the figure's top left corner contains a red path of length two from node 2 to node 6, thus violating the integrity constraint `correctColored` in `(10)`. The operation `RotateRR` on the right of Figure 14 — explained within the following subsection — restructures the given tree into a legal red/black tree, presented in the figure's bottom left corner. Please note that restrictions may be used to specify the invariants or static integrity constraints of a given data structure. Intermediate graph transformation results may violate these constraints, but automatically inserted runtime checks ensure that specified top-level transactions either abort or return consistent results only (like transaction `Insert` of Figure 16). For ongoing work concerning the verification — instead of runtime validation — of (graphically expressed) integrity constraints the reader is referred to [HW95].

The distinction between node classes and node types in Figure 13 needs a more detailed explanation. `RBNode` is a concrete node type, which belongs to the abstract node class `RBNODE`. PROGRES has a stratified type system, where types of nodes are "first class objects". Node classes within multiple inheritance hierarchies are types of node types and determine all common properties of their node type instances. For instance, any node of type `RBNode` of class `RBNODE` has the above mentioned attributes and is a legal source or target for the introduced edge types and paths (Section 3.4 gives an example of a more interesting class hierarchy).

3.3.2 Graph transformation rules and transactions

To ensure all the afore-mentioned properties, red/black trees must be reorganized after insertion or deletion of certain nodes. The "backbone" for the reorganization process

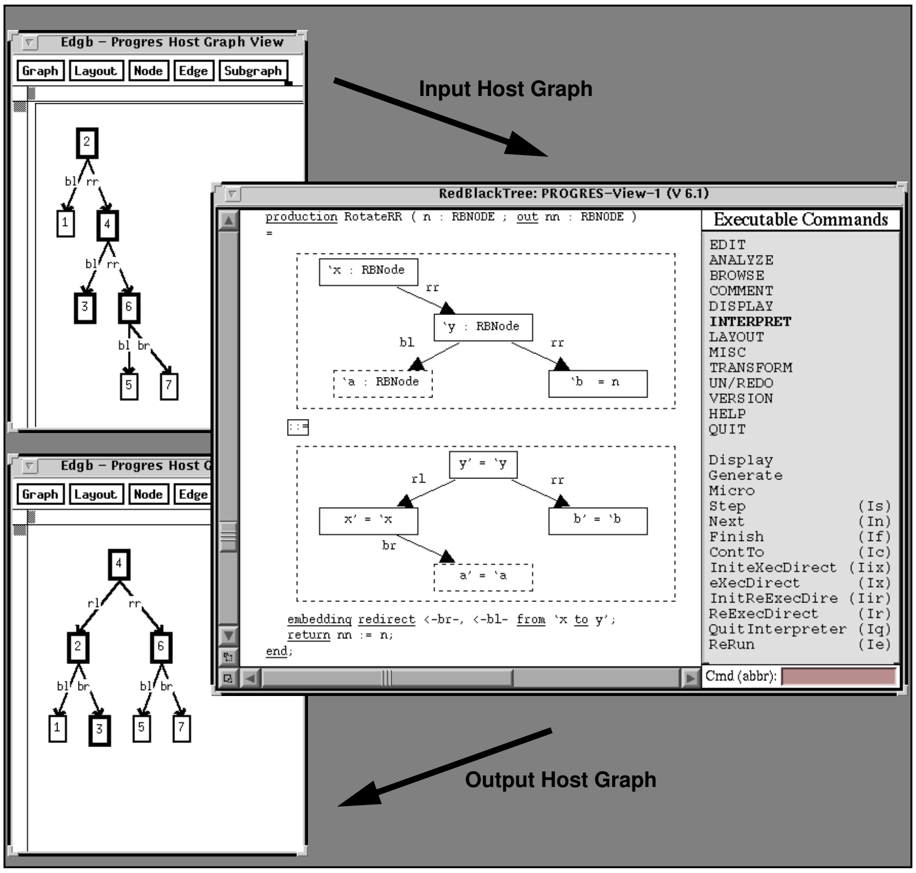


Figure 14: Red/black tree rotating rule with two graph instances

are four subtree-rotation operations. Figure 15 presents the pseudocode for one of the tree rotating operations (slightly simplified). It is called `RotateRR`, since it deals with two `r(ight)r(ed)` edges which meet each other. It is a slightly modified version of the operation `LeftRotate` in [CLR90] which is rather difficult to understand. In contrast, the Progres specification of `RotateRR`, shown in Figure 14, consists of a rule which is very similar to the graphical explanation of `LeftRotate` given in [CLR90]. It clearly states that `RotateRR` reorganizes a subtree with root `x` such that its red right child `y` becomes the root of the reorganized subtree. Its application to a given (red/black) tree graph is divided in the five basic steps Choose, Check, Remove, Glue, and Embed of Section 2.2.

The rule's left-hand side L (above the separator `::=`) defines the graph pattern which has to be transformed. It matches a subgraph of a given host graph which consists of the given node `b = n`, its parent `y`, and the parent `x` of `y`. Both nodes `y` and `b` have to be red right children of their parents. Furthermore, the node `y` may or may not have a black left child `a`. Its optional existence is visualized by using a dashed box as its representation. Please note that the node inscription `b = n` requires that `b` is always bound to the actual value of the rule's input parameter `n`. In general, the symbol "=" in the left-hand side links node identifiers to input parameters, while a "=" in the right-hand side links node identifiers from the right-hand side to those of the left-hand side. The rule application steps Choose and Check are just responsible for finding appropriate matches for the remaining three node variables `x`, `y`, and `a`.

The rule's right-hand side R defines the subgraph which replaces the selected match of its left-hand side. It consists of four nodes with inscriptions of the form v`' = '`v where v`'`

```
procedure RotateRR(n) =
  (* Compute match of variables y and x.  *)
  y := parent[n]; x := parent[y];
  (* Make black left child of y a black right child of x.  *)
  right[x] := left[y]; rightColour[x] := black;
  if left[y] ≠ nil then parent[left[y]] := x;
  (* Redirect black left or right edge and make parent of x a parent of y.  *)
  parent[y] := parent[x];
  if x = left[parent[x]] then left[parent[x]] := y else right[parent[x]] := y;
  (* Make x a red left child of y.  *)
  left[y] := x; leftColour[y] := red;
  parent[x] := y;
end
```

Figure 15: Pseudocode for operation `RotateRR`

is a node identifier of the right-hand side and $'v$ is a node identifier of the left-hand side. This is the PROGRES notation for defining the interface graph K and its occurrence *glue* in R. The presented rule preserves all matched nodes, but deletes and creates a number of edges. It has the effect that `x` becomes the red left child of `y`, `b` becomes the red right child of `y`, and `a` becomes the black, right child of `x` (REMOVE and GLUE). Furthermore, the line starting with `embedding` redirects any incoming black left or right edge from the old root node `x` to the new root node `y` (EMBED).

The remaining basic operations for tree rotation are defined in a similar way. All needed red/black tree interface operations are then programmed by means of imperative control structures and calls to graph transformation rules or graph-traversing paths. Figure 16 contains the specification of the operation `Insert` which first selects the correct place for inserting a new value (simultaneously reorganizing the tree where needed), then inserts a new leaf, and finally reorganizes again the tree by means of rotating rules. For further details see [Sch94], which contains a complete specification of red/black trees.

Finally, we should mention that the presented example did not make use of all available PROGRES features. For instance, node types are first-class objects which may be used as parameter values of graph transformation rules and have their own meta-attributes (cf. Section 3.4). Furthermore, PROGRES offers means to deal with non-determinism, an inherent property of our graph transformation paradigm. They vary from collecting and manipulating sets of all possible outcomes (of atomic values) to a Prolog-like style of programming with built-in backtracking capabilities. Finally, its graph transformation rules support

1. the selection of injective as well as non-injective occurrences of their left-hand sides,
2. the definition of application conditions in the form of predicate logic formulas over (derived) node attributes and path expressions, and
3. the construction of edges between new nodes and the context graph by means of textual as well as graphical embedding rules.

3.3.3 Summary

We have presented the graph transformation language PROGRES, and its application to the specification of a well-known data structure. The language provides its users with diagrammatic and textual constructs for the definition of graph schemata, derived graph

```
transaction Insert ( Val: integer ) =
   use n: RBNODE do
         GetRoot ( out n )
      & loop
            choose
                  Split ( n, out n )
               & Rotate ( n, out n )
            else
               n := n.down ( Val )
            end
         end
      & (   InsertRight ( n, Val, out n )
          or InsertLeft ( n, Val, out n ) )
      & Rotate ( n, out n )
   end
end;

path down ( Val: integer ) : RBNODE [0:1] -> RBNODE [0:1] =
   [ valid (Val < self.Value) ? =left=>
   | valid (Val > self.Value) ? =right=> ]
end;
```

Figure 16: Complex graph transformation for insertion of red/black tree

properties, graph transformation rules, and complex graph transformation processes. The examples have been produced with the PROGRES environment, which offers means to create, analyze, and execute specifications. Furthermore, two compiler backends were recently added, which translate a given PROGRES specification into equivalent Modula-2 or C code.

The latest version of the PROGRES environment is available as free software for Sun workstations (see WWW page http://www-i3.informatik.rwth-aachen.de/research/progres/index.html). It contains a rapid prototyping tool which translates any given specification into a stand-alone C program with a Tk/Tcl-based user interface. Generated prototypes store their graphs in the graph-oriented database system GRAS [KSW95]. In such a way, our graph transformation environment supports the specification and realization of abstract data types or, more generally, of tools which manipulate complex data structures (as in Section 3.2). It is subject of current research activities to develop new compiler backends for generating C++ programs, which store their graphs either in main memory or in an object-oriented database system.

Experience with PROGRES at various sites (cf. Section 3.4) has shown that there are needs for further enhancement:

1. The data model should be generalized to support (attributed) hyperedges and hierarchical graphs (as used in Sections 3.1 and 3.2).
2. Concepts for modularization should be developed to structure large specifications into small, reusable parts with well-defined interfaces.
3. It should be possible to define language subsets that can be executed more efficiently and allow to use PROGRES for specific applications, e.g. term graph rewriting in Section 3.1.

Serious efforts must still be made to develop a new graph-transformation-based language and environment with a suitable module concept and mechanisms to adapt its graph data model and graph transformation approach to the specific needs of selected application areas. This is partly the topic of Section 4.

3.4 Definition of a Visual Language

Over the years, graphical (or visual) software tools and environments are taking over the role of purely text-oriented terminal display. An illustrative example can be found in the field of CASE environments, where, for instance, class diagrams support the modelling of the structural aspects of a software system, while data flow diagrams, state transition diagrams, or Petri nets support the modelling of its dynamic aspects. The user of such CASE environments deals with documents written in a visual language instead of a textual language. In this section we illustrate with an example the potentials of graph transformation for the formal definition of diagram languages. The chosen example is the hybrid database query language HQL/EER for an Extended Entity-Relationship model (EER) [AE94, AE96], which allows the specification of database queries partly graphically and partly textually.

In this section we present a short informal introduction to HQL/EER, and briefly illustrate with a simple example how syntax and semantics of its graphical part may be defined by a graph grammar specification. For the latter purpose, we use the PROGRES language and environment, which has already been introduced in Section 3.3. See [And96] for the complete specification of HQL/EER.

3.4.1 The hybrid database query language HQL/EER

HQL/EER allows the formulation of queries on a database, the structure of which has been defined by a schema based on an Extended Entity-Relationship model [EGH+92]. A small example EER database schema, visually represented as an EER diagram, is shown in Figure 17. It defines the structure of the entity `PERSON`, which has a single-valued attribute `Name` of type `string` and a list-valued attribute `Addr` with `address` as (predefined) type for the list members. Thus, a database instance consists of `PERSON` entities, where each `PERSON` entity has a name and a list of addresses as attributes.

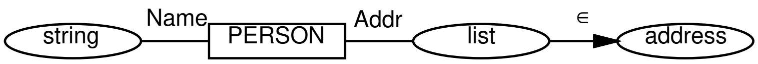


Figure 17: Sample EER diagram

The EER model has been equipped with a textual SQL-like query language called SQL/EER [HE92]. By extending SQL/EER with graphical alternatives for its language constructs, we have obtained the Hybrid Query Language HQL/EER. In this language the database user can choose which part of a query he or she wants to express graphically or textually, respectively.

Figure 18 gives a purely graphical query in HQL/EER, where the node drawn in bold indicates which information shall be retrieved: the names of all persons who share an address with `John`.

Figure 18: Sample HQL/EER query

The example shows that an HQL/EER query consists of nodes and edges, labelled with names from the corresponding EER diagram. Hence, the graphical part of an HQL/EER query can be viewed as a graph (cf. Section 2.1), and its syntactical structure can be defined by a graph grammar.

3.4.2 The syntax definition of HQL/EER

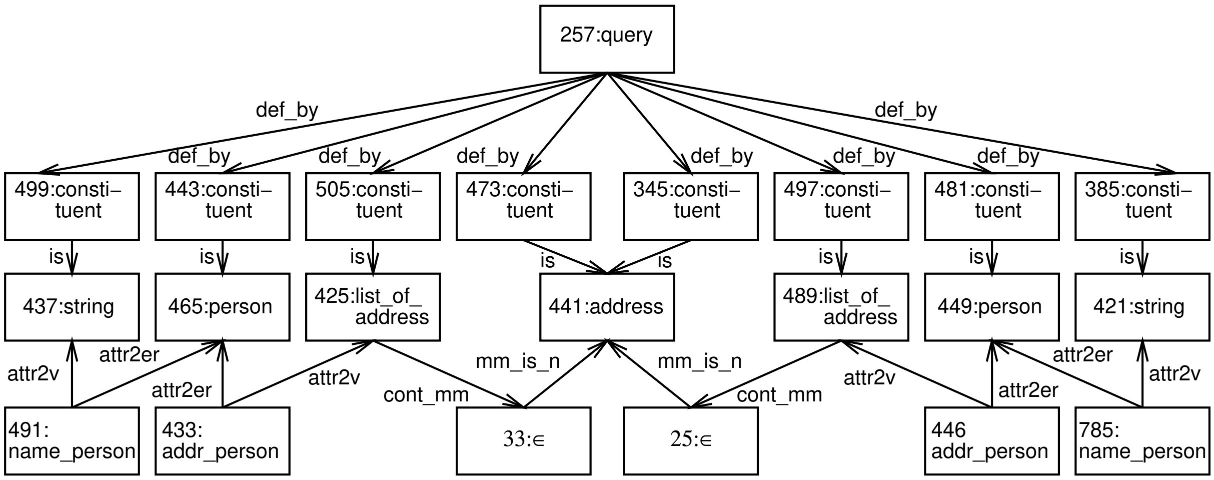


Figure 19: Graph Representation of an HQL/EER Query

The syntax of HQL/EER is defined by a graph grammar in two steps: an appropriate graph representation of words of the visual language is chosen, and then the language of those graphs is defined by a graph grammar. A graph in this language may be associated to the graphical part of a hybrid query as follows (cf. Figure 19, which shows the nodes and edges of the graph corresponding to the hybrid query of Figure 18):

1. All graphical elements of the query (i.e., both nodes and edges) are represented by nodes. Nodes corresponding to edges in the query of Figure 18 are shown at the bottom, while nodes corresponding to nodes in the query are shown directly above. A node representing an edge of the query is linked to the nodes representing its source and target by means of suitably labelled edges. The design decision to represent edges of a query by nodes, too, is due to the (technical) restriction of Progres that node types may be refined in a graph schema, while edge types may not. The use of nodes to represent edges allows to specify the transformation rules in terms of language constructs ("entity", "relationship", ...) and keep them independent from any particular EER database schema. EER database schema-dependent elements now only occur in a clearly identifiable part of the graph schema (i.e. subclasses and node types added to the EER schema- independent part of this graph schema).

2. The graph representation also contains a node for each (sub)query in the graphical part of the query. Since the query of Figure 18 has no subqueries, its graph representation contains a single `query`-node, linked (indirectly) to those nodes that correspond to nodes in the query. However, one node can play different "roles" in a query, as illustrated by the `address`-node in Figure 18. Since it is impossible to have parallel edges (with identical label) in Progres, intermediate `constituent`-nodes were introduced.

3. The labels of nodes in the graph representation are either dependent on and derived from a concrete EER diagram (like `person` or `address`), or are independent of any concrete EER diagram (like `query`, `constituent` or $\in$).

4. Non-structural characteristics of the query are stored in node-attributes. The table below shows the "syntactic" attributes of two nodes of the graph of Figure 19. The fact that a `string`-node is selected for output (respectively has the value `John`) is represented by setting the boolean flag `Output` of the corresponding `constituent`-node to `TRUE` (respectively by setting the string-valued attribute `Value` of the `string`-node to `John`).

Id.	Label	Att. Name	Attribute Value
499	constituent	Output	TRUE
421	string	Value	John

Next, we formally define the language of graphs representing (the graphical part of) HQL/EER queries by means of a Progres specification, consisting of

1. a graph schema, declaring the types of nodes and edges, as well as attributes, and
2. a set of graph transformation rules which obey the type restrictions imposed by the graph schema, and specify how instances of the components defined in the graph schema may be composed.

Such a specification defines a graph language exactly as described in Section 2.3: a syntactically correct graph is yielded by applying a sequence of rules to an initial (empty) graph.

```
node class ENTITY is a ENT_REL end;
node class ATOMIC_VALUE is a VALUE
    intrinsic Value : string := ``'';
end;
node class CONSTITUENT is a ...
    intrinsic Output : boolean := false;
end;
node type constituent : CONSTITUENT end;
node class ATTRIBUTE
    meta entrel : type in ENT_REL [0:N];
         val : type in VALUE [1:1];
end;
edge type attr2er : ATTRIBUTE → ENT_REL [1:1];
edge type attr2v : ATTRIBUTE → VALUE [1:1];
node class PERSON is a ENTITY end;
node type person : PERSON end;
node type string : ATOMIC_VALUE end;
node type name_person : ATTRIBUTE
    redef meta entrel := PERSON;
               val := string;
end;
```

Figure 20: Graph schema (partial)

Figure 20 shows a cutout of the graph schema for the specification of HQL/EER queries. It declares several node classes, some as a specialization of other node classes, like `ENTITY` which is a specialization of `ENT_REL` (which has `RELSHIP` as its other subclass). Note that some declarations in the graph schema are independent from the considered EER diagram (such as `ENTITY`), while others use the labels of the EER diagram (such as `PERSON`).

Attributes are declared and initialized in node class declarations, and may be redefined in node type declarations. Intrinsic attributes, such as the `string`-attribute `Value` declared for node class `ATOMIC_VALUE`, apply to nodes of the considered class. In contrast, meta-attributes apply to node types. In this specification, meta-attributes always have a node type or set of node types (represented by a class) as value. For instance, the meta-attribute `entrel` applies to types of class `ATTRIBUTE`, and is initialized to `PERSON` (representing the set of all node types of this class) in the `name_person` node type.[10] We use meta-attributes furtheron to enforce type correctness.

Nodes are linked by typed edges, like the `attr2er`-edges, which connect an `ATTRIBUTE` node to an `ENT_REL` node, expressing the fact that attributes may be assigned to entity or relationship types. The cardinality `[1:1]` expresses that this is a one-to-one correspondence: an attribute applies to exactly one entity or relationship.

The collection of declarations contained in the graph schema defines a graph class that contains as a subset those graphs that represent syntactically correct HQL/EER queries. This means that further restrictions have to be imposed on how subgraphs are linked. These restrictions are expressed by means of graph transformation rules.

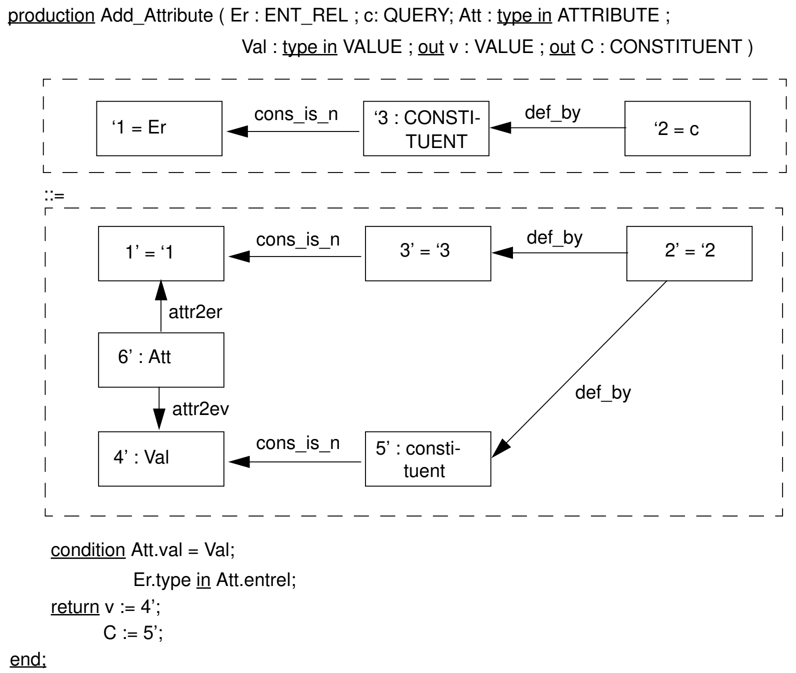


Figure 21: Rule for inserting an attribute

Figure 21 shows the rule that adds an (EER) attribute to an entity or relationship. Such an attribute is represented by means of two nodes `6'` and `4'`, inserted and linked by a new constituent `5'` to existing nodes. The parameter `Att` may be replaced by any node type which is a descendant of `ATTRIBUTE` in the node class hierarchy of the graph schema. Additionally, the condition clause of the rule enforces type compatibility on the input parameters. For instance, it is checked whether the value type `Val` is the value type expected by the attribute type `Att`, as stored in this type's meta-attribute `val`.

[10] The fact that the node types of Figure 20 only redefine attribute values is just a coincidence: PROGRES allows the attributes themselves to be redefined.

The rule `Add_Attribute` can be applied to an entity of type `person` with the node types `name_person` and `string` for the parameters `Att` and `Val`, respectively.

3.4.3 The semantics definition of HQL/EER

HQL/EER offers graphical alternatives for some of the (textual) language constructs of SQL/EER. The most natural way for defining the semantics of HQL/EER is therefore the translation of its graphical elements to SQL/EER. An additional motivation for choosing this approach is the fact that the PROGRES formalism allows a seamless integration of such a semantics definition with the syntax definition into one and the same graph grammar specification.

The translation is incorporated into the graph representation of HQL/EER queries by attributing nodes in this graph with corresponding SQL/EER expressions. Remember that `constituent`-nodes are used precisely for representing the fact that some query element plays a part in some query. Hence they are the ideal place for storing these SQL/EER expressions.

The table below shows some semantic attributes from the graph of Figure 19. The `SFW_Term`-attribute of the `query`-node contains precisely the SQL/EER-query represented by the HQL/EER query of Figure 18. All other semantic attributes are associated to `constituent`-nodes, and contain SQL/EER-declarations, -terms or -formulas.

Id.	Label	Att.Name	Attribute Value
257	query	SFW_Term	select p1.name_person from a1 in p1.addr_person, a2 in p2.addr_person, p1 in person, p2 in person where a2 = a1 and p2.name_person = John
481	constituent	Declaration Term	p2 in person p2
497	constituent	Term	p2.addr_person
385	constituent	Term	p2.name_person
		Formula	p2.name_person = John

From the above example, the extension to be made to the graph schema of the specification of HQL/EER follows directly: the declarations of node classes `QUERY` and `CONSTITUENT` have to be extended with declarations for the attributes mentioned above, as well as appropriate derivation rules for the `SFW_Term`-attribute. The semantic attributes of `CONSTITUENT`-nodes, however, have to be initialized by the rules that create them. As an example, the clause `transfer 5'.Term := '3.Term & "." & string( Att );` has to be added to the rule `Add_Attribute`, shown in Figure 21. Here, `&` stands for string concatenation, while the `string`-function converts `Att`-values into string representations.

3.4.4 Summary

From the act of specifying HQL/EER by means of a graph grammar, some important lessons may be drawn. First, it shows that the use of an attributed graph grammar allows the seamless integration of the definition of both syntax and semantics of a graph-based language into one single specification, in the same way as attributed string grammars do for textual languages.

Second, when defining graph-based languages, the ability to work with graphs as basic primitives in the graph rules surely has its advantages over having to work with some general-purpose specification language in which these graphs first have to be (cryptically) encoded themselves. It allows one to concentrate on the essentials of the graph-based language itself, rather than on the details of the applied specification language.

Third, the specification of HQL/EER illustrates once more that as soon as the size of a (graph grammar) specification exceeds a certain limit, it becomes very hard to manage and understand. While the PROGRES language already offers a variety of mechanisms for capturing the detailed aspects of a language of graphs, it clearly lacks modularization concepts. The availability of such a module concept, including well-defined notions of interfaces and module interrelationships (such as "use" and inheritance), would substantially improve both understandability and reusability of specifications.

4 A Structuring Principle for Graph Transformation

The case studies in Section 3 have illustrated that graph transformation can be applied to system specification and programming in a natural way. However, the case studies are quite small compared to "real world" applications which may consist of very large and complex graphs with hundreds and thousands of rules. Thus graph transformation systems must be extended by structuring mechanisms that allow to compose them in a modular way. Furthermore, the case studies have indicated that different graph transformation approaches are suitable for different applications. Hence, structuring principles are needed which are approach-independent in the sense that they allow specification and programming in different approaches. One of the first steps in this direction is the concept of a transformation unit (see [KK96, KKS98]).

In this section, we outline basic features of transformation units. Their use is demonstrated by reformulating some aspects of the case study of term graph rewriting given in Section 3.1. Similarly, our other examples could be redesigned in a structured way. This could be interesting in various respects, but would not add much to the illustration how the concept of graph transformation works. Hence, we concentrate here on only one of the examples.

4.1 Main Features

In the following we informally discuss the main characteristics of transformation units; a more formal treatment is given in Section 4.2.

Rule-based graph transformation. As motivated in Section 1 and illustrated in Section 3, it is quite natural to combine graphs as a means for describing complex structures with rules as a means to manipulate these structures. On the most elementary level, a transformation unit consists of a set of graph transformation rules which are applied to graphs. More precisely, a transformation unit transforms a graph by applying one of its rules to it and iterating this process. Hence, the key operation performed by a transformation unit is the application of a rule to a graph.

Initial and terminal graphs. Often, it is desirable to start or stop a graph transformation with a special graph from the underlying graph class. For example, to generate a graph language, transformation may start with a particular initial graph and stop with graphs of a particular shape (cf. Section 2.3). Or, to reduce graphs, the initial graphs may be arbitrary, and are transformed until no rule is applicable. The concept of a transformation unit generalizes such notions by providing graph class expressions that specify classes of initial and terminal graphs. Consequently, "valid" transformations must start with initial graphs and stop with terminal graphs. The description of initial graphs can be viewed as a precondition for the graph transformations performed by the transformation unit. Analogously, the terminal description serves as a postcondition.

Control conditions. In general, graph transformation is non-deterministic because of two reasons: First, there may be more than one rule which can be applied to a certain graph. Second, applying a graph transformation rule to a graph means to transform it locally so that there may be various parts in the graph which can be manipulated by a rule application. To regulate the graph transformation process, for example by choosing rules according to a priority, or by prescribing a certain sequence of steps, suitable control conditions are needed. Such conditions are treated, for example, in [Bun79, Sch91a, SZ91, MW96] (cf. also [DP89] for regulation concepts in string grammars and [PvE93] for evaluation strategies in functional programming). With transformation units the graph transformation process can be regulated by using control conditions.

Approach independence. The mentioned components, i.e. graphs, rules, graph class expressions, and control conditions, form a so-called graph transformation approach. In the literature, there exist a variety of graph transformation approaches like PROGRES [SWZ95], algebraic ones like the double-pushout approach [Ehr79], the single-pushout approach [Löw93] or AGG [TB94], and more restricted approaches like node replacement [JR80], edge replacement [HK87], or hyperedge replacement [Hab92]. See for an overview [Roz97]. They differ mainly in the underlying graph data model and the type of transformation rules. For example, in PROGRES attributed graphs are transformed, whereas AGG manipulates hierarchical ones with single-pushout rules, and hyperedge-replacement applies so-called hyperedge-replacement rules to hypergraphs. Some of the approaches provide specific kinds of control conditions or graph class expressions like PROGRES or AGG. The case studies underline that several approaches are convenient for specification and programming. In order to be widely applicable, one main feature of transfomation units is that they are independent of a specific graph transformation approach. This means that they are suitable to manipulate graphs in all existing or user-defined graph transformation approaches.

Structuring. In practice, rule-based systems may consist of an immense set of rules. To design such systems in a reasonable way, one needs structuring principles that allow to compose them from small re-usable units in a systematic way. Therefore, transformation units provide – apart from rules, specifications of initial and terminal graphs, and a control condition – an import component which allows for using other transformation units. In this way, the set of local rules of a transformation unit can be kept small and managable. The import part of a transformation unit consists of a set of (names refering to) transformation units. For reasons of simplicity, we consider in this section transformation units with hierarchical import structure.

Interleaving semantics. A graph transformation is a sequence of rule applications starting with some initial graph. The operational semantics of a graph transformation system is usually defined as the set of all transformations. By abstracting from the intermediate steps of a transformation, one obtains an input-output relation on graphs where each pair contains the initial graph of a transformation and its transformed graph. This concept of operational semantics is the basis for the semantics of transformation units which is defined as a binary relation on graphs. In particular, transformation units transform initial graphs to terminal ones by interleaving rule applications with calls of imported transformation units such that the control condition is obeyed. Hence, transformation units provide a functional abstraction meaning that a complex graph transformation is encapsulated as a binary relation of graphs. Imported transformation units are treated as atomic in the sense that they are applied as a whole to graphs by the importing unit. This encapsulation property can be also found in other well-known structuring methods in computer science like transactions in database systems or procedures in programming languages.

Note that in general, the concept of transformation units can be used to transform arbitrary objects in a rule-based manner, but since graph transformation is the basic subject of this paper, we restrict ourselves to the data models of graphs.

4.2 Transformation Units

With transformation units, graph transformation systems can be built up from small pieces. To make them independent of a particular graph data model, a specific type of rules, etc. they are defined over some graph transformation approach.

4.2.1 Graph transformation approaches

A graph transformation approach comprises a class of graphs, a class of rules, a rule application operator, a class of graph class expressions, and a class of control conditions. The purpose of control conditions is to restrict the non-determinism of graph transformation; its semantics is a binary relation on graphs. Since control conditions may contain identifiers (usually for imported transformation units or local rules), their semantics depends on their environment, a mapping which associates each identifier with a binary relation on graphs. In this way, the semantic effect of control conditions can be defined without forward reference to transformation units. Intuitively, one may think of an environment as names of rules (or of transformation units) with their corresponding direct derivation relations (or their corresponding interleaving semantics).

More formally, a *graph transformation approach* $\mathcal{A}$ consists of a class $\mathcal{G}$ of *graphs*, a class $\mathcal{R}$ of *rules*, a *rule application operator* $\Rightarrow$ yielding a binary relation $\Rightarrow_r \subseteq \mathcal{G} \times \mathcal{G}$ for each $r \in \mathcal{R}$, a class $\mathcal{X}$ of *graph class expressions* such that each $e \in \mathcal{X}$ specifies a subclass $SEM(e) \subseteq \mathcal{G}$, and a class $\mathcal{C}$ of *control conditions* over some set ID of identifiers such that each $c \in \mathcal{C}$ specifies a binary relation $SEM_E(c) \subseteq \mathcal{G} \times \mathcal{G}$ for each *environment* $E\colon ID \to \mathcal{P}(\mathcal{G} \times \mathcal{G})$.[11]

In Section 2, we have already introduced different classes of graphs and rules. Here, we give some typical examples for the remaining components of a graph transformation approach, that is, for graph class expressions and control conditions.

[11]Note that we use the overloaded operator *SEM* here and elsewhere to denote various semantic operations. They are distinguished by the types of arguments or – as above – by the presence or absence of the additional index (refering to the environment).

4.2.2 Graph class expressions

There are various standard ways to choose graph class expressions that can be used for arbitrary graph classes $\mathcal{G}$, and hence used in many graph transformation approaches:

1. In most cases, one deals with finite graphs with some explicit representations. Then single graphs (or finite enumerations of graphs) may serve as graph class expressions. Semantically, each graph G represents itself, i.e. $SEM(G) = \{G\}$. The initial graph of a graph grammar is a typical example of this type.
2. As a kind of default graph class expression (without effect), we may use the term *all* specifying the class of all graphs, i.e. $SEM(all) = \mathcal{G}$.
3. A graph G is *reduced* with respect to a set of rules $P \subseteq \mathcal{R}$ if no rule from P is applicable to G. In this way, P can be considered as a graph class expression where $SEM(P)$ is the set of all reduced graphs with respect to P. Reducedness is often used in term rewriting and term graph rewriting as a halting condition.
4. If $\mathcal{G}$ is a class of graphs labelled over an alphabet Σ, then a set $T \subseteq \Sigma$ is a suitable graph class expression, specifying the class of all graphs labelled in T only. This way of distinguishing terminal objects is quite popular in formal language theory.
5. A graph grammar can be considered as a graph class expression which specifies its generated language.
6. Choosing some of the examples above as atomic expressions, one may build up more complicated expressions recursively by using set-theoretic operators like union, intersection and complement.
7. Graph-theoretic properties can be used as graph class expressions. In particular, monadic second-order formulas for directed graphs or hypergraphs are suitable candidates (see e.g. [Cou90]).
8. Graph schemata, as used in PROGRES, and typed graphs are graph class expressions that allow to specify generic graph classes (see Sections 3.3 and 3.4).

4.2.3 Control conditions

Every description of a binary relation on graphs may be used as a control condition. Some typical examples are the following:

1. Every grammar, every automaton and every expression specifying a string language L over *ID* can serve as a control condition. For an environment E each string $x_1 \cdots x_n$ in L specifies the binary relation obtained by the sequential composition of the meaning of the x_i in E, i.e. $SEM_E(x_1 \cdots x_n) = E(x_1) \circ \cdots \circ E(x_n)$;[12] the empty string λ specifies the empty set. For example, if $x_1, \ldots, x_n$ $(n \geq 1)$ are graph transformation rules and the meaning of each x_i in E consists of all pairs (G_i, G_i') of graphs such that G_i' results from applying x_i to G_i, then $x_1 \cdots x_n$ specifies all pairs (G, G') of graphs where G' is obtained from G by applying the rules $x_1, \ldots, x_n$ in this order. The whole language L specifies the union of the semantic relations given by the elements of L.
2. The control conditions of the language type in point 1 can be extended from the set *ID* to an arbitrary set C of (elementary) control conditions. Hence, each mechanism that

[12]Given binary relations $\rho, \rho' \in \mathcal{G} \times \mathcal{G}$, the sequential composition of ρ and ρ' is defined by $\rho \circ \rho' = \{(G, G'') \mid (G, G') \in \rho \text{ and } (G', G'') \in \rho' \text{ for some } G' \in \mathcal{G}\}$.

specifies a language over C can also serve as a control condition the semantics of which is defined as in point 1, i.e. for a string $c_1 \cdots c_n$ over C we have $SEM_E(c_1 \cdots c_n) = E(c_1) \circ \cdots \circ E(c_n)$ for each environment E, and $SEM_E(\lambda) = \emptyset$.

3. In particular, given some set $C \subseteq \mathcal{C}$ of (elementary) control conditions, the class of regular expressions over C can be used for this purpose. For explicit use below, $REG(C)$ is recursively given by $C \subseteq REG(C)$, and $(e_1 ; e_2)$, $(e_1 \mid e_2)$, $(e^*) \in REG(C)$ if $e, e_1, e_2 \in REG(C)$. In order to omit parentheses we assume that * has a stronger binding than ; and $\mid$, and that ; has a stronger binding than $\mid$. Moreover, the regular expression $e ; e^*$ (with $e \in REG(C)$) will be abbreviated by e^+. For an environment E, the semantics of such regular expressions is obtained as follows: $SEM_E(e_1 ; e_2) = SEM_E(e_1) \circ SEM_E(e_2)$, $SEM_E(e_1 \mid e_2) = SEM_E(e_1) \cup SEM_E(e_2)$, and $SEM_E(e^*) = SEM_E(e)^*$.[13]
4. As a kind of default control condition (without effect) we may use the term *true*, which specifies the all relation, i.e. $SEM_E(true) = \mathcal{G} \times \mathcal{G}$, independently of the environment.
5. For each control condition $c \in \mathcal{C}$ the expression $c!$ serves as an induced control condition which applies c as long as possible. This means that for each environment E a pair (G, G') is in $SEM_E(c!)$ if $(G, G') \in SEM_E(c)^*$ and there is no graph G'' with $(G', G'') \in SEM_E(c)$.
6. Each pair (e_1, e_2) of graph class expressions in $\mathcal{X}$ defines a binary relation on graphs by $SEM((e_1, e_2)) = SEM(e_1) \times SEM(e_2)$ and, therefore, it can be used as a control condition which is independent of the choice of an environment, i.e. $SEM_E((e_1, e_2)) = SEM((e_1, e_2))$ for all environments E.
7. The deterministic and non-deterministic control structures of PROGRES serve as control conditions. They allow to define imperative commands over control conditions.

4.2.4 Transformation units

Each transformation unit encapsulates a specification of initial graphs, a set of transformation units to be imported, a set of rules, a control condition, and a specification of terminal graphs. Its import structure is hierarchical, i.e. on the lowest level, there are only unstructured transformation units with no import; moreover, each transformation unit may only import transformation units of a lower level. Transformation units are defined over some graph transformation approach.

More formally, for a graph transformation approach $\mathcal{A}$, the set of all transformation units over $\mathcal{A}$ is recursively defined as follows: Let U be a set of transformation units over $\mathcal{A}$ that is empty, initially. Then a transformation unit over $\mathcal{A}$ is a system $trut = (I, U, R, C, T)$ where I and T are graph class expressions, R is a finite set of rules, and C is a control condition.

Note that if I specifies a single graph, U is empty and C is the constant *true*, one gets the usual notion of a graph grammar (in which approach ever) as a special case of transformation units.

4.2.5 Interleaving semantics

The operational semantics of a transformation unit is a binary relation on graphs containing a pair (G, G') of graphs if, first, G is an initial graph and G' is a terminal graph,

[13]For a binary relation ρ, ρ^* denotes its reflexive and transitive closure.

second, G' can be obtained from G by interleaving derivations with the graph transformations specified by the imported transformation units, and third, the pair is allowed by the control condition.

In more detail, consider a transformation unit $trut = (I, U, R, C, T)$ and assume that every imported transformation unit $t \in U$ defines already a binary relation on graphs $SEM(t) \subseteq \mathcal{G} \times \mathcal{G}$. Then $trut$ defines a binary relation $SEM(trut)$ in the following way: A pair of graphs (G, G') belongs to $SEM(trut)$ if $G \in SEM(I)$, $G' \in SEM(T)$ and there is a sequence of graphs $G_0, \ldots, G_n \in \mathcal{G}$ such that $G_0 = G$, $G_n = G'$ and, for $i = 1, \ldots, n$, the pair (G_{i-1}, G_i) either belongs to the semantic relation $SEM(t)$ of some imported transformation unit $t \in U$, or G_i directly derives G_{i-1}, i.e. $G_{i-1} \Rightarrow_r G_i$ for some $r \in R$. In addition, the pair (G_{i-1}, G_i) must be accepted by the control condition C in the specific environment given by $trut$, i.e. $(G, G') \in SEM_{E(trut)}(C)$. This environment is given by $SEM(t)$ for $t \in U$ and $\Rightarrow_r$ for $r \in R$, assuming the identifiers occurring in C are elements of U and R (or their names). The resulting semantic relation $SEM(trut)$ is called *interleaving semantics*, and the sequences of graphs it is based on are called *interleaving sequences*. The definition of the interleaving semantics follows the recursive definition of transformation units. Hence, its well-definedness follows easily by an induction on the hierarchical structure of transformation units.

If U is empty, an interleaving sequence consists of derivations only, that is, the interleaving semantics generalizes the ordinary operational semantics of sets of rules given by derivations.

If, in particular, the control condition can be ignored, i.e. $C = true$, and if there is a single initial graph, only the second components of all pairs in the interleaving semantics are significant so that a transformation unit of this kind can be seen as a graph grammar and the interleaving relation corresponds to its generated language. Such a particular transformation unit is called *generating unit*; it can be used as a graph class expression as it is done in the following example.

4.3 An Example

The following example picks up the first case study of Section 3. It illustrates how term graphs can be specified and how term graph rewriting can be modelled by using transformation units.

4.3.1 The approach

Let $\mathcal{F}$ be a set of function symbols, each of which has a fixed arity, that is, a fixed number $n \geq 0$ of arguments. Let $\mathcal{V}$ be a set of variables. The graph transformation approach used in this example consists of the following components.

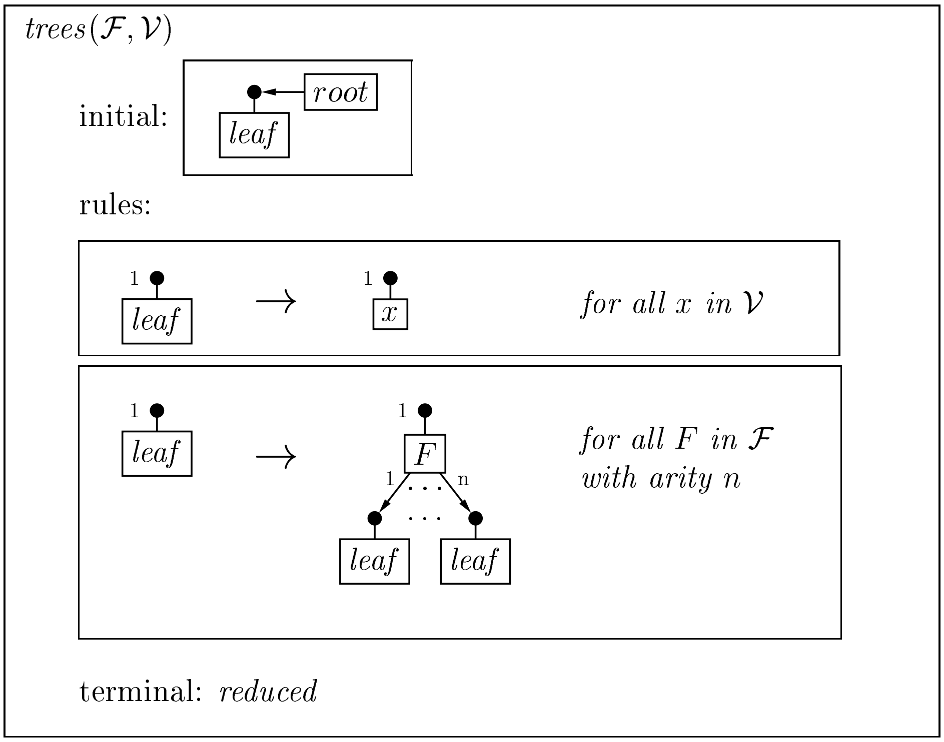
- Each *graph* is a hyperedge-labelled directed hypergraph, with labels in $\mathcal{F} \cup \mathcal{V} \cup \{root, leaf\}$. The label *root* is a special symbol used for indicating the root node of a term graph, i.e. the unique node from which each other node is reachable, and is needed for specifying garbage collection (see below). The label *leaf* is only used in the generation of trees.
- In each *rule* $r = (L, R, K, glue, emb, appl)$, the component *emb* is empty, and *appl* is a subset of $\{contact, ident\}$ where *contact* stands for the contact condition and *ident* for the identification condition (see Section 2.2).

- As graph class expressions, we use a particular initial graph, a particular generating unit, the constant *all*, a set of rules (to specify reduced graphs).
- The class of control conditions contains the set $REG(C)$ of regular expressions over C where C consists of a set ID of identifiers (containing at least the names of all transformation units given below) plus the condition $c!$ for each c in $REG(C)$.

Transformation units are presented by indicating the components with respective keywords. Trivial components (i.e. no imported transformation units, no rules, the graph class expression *all*, and the control condition *true*) are omitted. In a transformation unit *trut* with rule set $\{p_1, \dots p_n\}$, the graph class expression *reduced* is an abbreviation for $\{p_1, \dots, p_n\}$ and the control condition *once* abbreviates the regular expression $p_1|p_2|\cdots|p_n$. Each rule which is not already introduced in Section 3.1 is depicted so that an arrow points from its left-hand side L to its right-hand side R. The subgraph K of L and its occurrence *glue* in R are indicated by numbering the corresponding nodes in L and R. Application conditions are written under the left-hand side L.

4.3.2 Term graphs

To specify the class of all term graphs over $\mathcal{F} \cup \mathcal{V}$ we use the collapse rules introduced in Section 3.1. Since term graphs are “collapsed trees”, each term graph can be constructed by first generating a tree and then collapsing it. The transformation unit $trees(\mathcal{F}, \mathcal{V})$ is a graph grammar generating the class of all trees over $\mathcal{F} \cup \mathcal{V}$ (with a *root*-hyperedge attached to the root) in a top-down manner. The rule set of $trees(\mathcal{F}, \mathcal{V})$ contains one rule for each variable and each function symbol. The rules do not have application conditions and can be applied in an arbitrary order. The terminal expression ensures that generated graphs have labels in $\mathcal{F} \cup \mathcal{V} \cup \{root\}$ only.



The semantics of $trees(\mathcal{F}, \mathcal{V})$ consists of all pairs (G, G') where G is the initial graph, and G' is a tree over $\mathcal{F} \cup \mathcal{V}$ to the root of which a *root*-labelled hyperedge is attached. Note that $trees(\mathcal{F}, \mathcal{V})$ is a generating unit.

The collapse steps are performed by the transformation unit *collapse*$(\mathcal{F}, \mathcal{V})$ that applies a collapse rule *collapse*(x) for some $x \in \mathcal{F} \cup \mathcal{V}$ exactly once to its input graph (see Section 3.1.2 for the definition of collapse rules).

collapse$(\mathcal{F}, \mathcal{V})$		
rules:	*collapse*(x)	*for all* $x \in \mathcal{F} \cup \mathcal{V}$
conds:	*once*	

The semantics of *collapse*$(\mathcal{F}, \mathcal{V})$ consists of all pairs (G, G') where G is a graph and G' is obtained by applying a collapse rule (once) to G.

The transformation unit *term_graphs*$(\mathcal{F}, \mathcal{V})$ generates the set of all term graphs over $\mathcal{F} \cup \mathcal{V}$ from the initial graph of *trees*$(\mathcal{F}, \mathcal{V})$. It calls *trees*$(\mathcal{F}, \mathcal{V})$ exactly once and then *collapse*$(\mathcal{F}, \mathcal{V})$ arbitrarily often.

term_graphs$(\mathcal{F}, \mathcal{V})$	
uses:	*trees*$(\mathcal{F}, \mathcal{V})$, *collapse*$(\mathcal{F}, \mathcal{V})$
conds:	*trees*$(\mathcal{F}, \mathcal{V})$; *collapse*$(\mathcal{F}, \mathcal{V})^*$

The semantics of *term_graphs*$(\mathcal{F}, \mathcal{V})$ depends on the semantics of the imported transformation units and consists of all pairs (G, G') where G is the initial graph in *trees*$(\mathcal{F}, \mathcal{V})$ and G' is a term graph over $\mathcal{F} \cup \mathcal{V}$ the root of which has an incident *root*-labelled hyperedge. Apart from the *root*-labelled hyperedge, the term graphs are the same as in Section 3.1. The *root*-hyperedge is (implicitly) used in the specification of garbage collection given below.

4.3.3 Term graph rewriting

Term graph rewriting, as explained in Section 3.1, comprises evaluation and collapse steps on term graphs. Hence, the transformation unit *collapse*$(\mathcal{F}, \mathcal{V})$ can be reused. Let $\mathcal{E}$ be a set of equations over $\mathcal{F}$ and $\mathcal{V}$ with the properties described in Section 3.1.2. An evaluation step consists of applying an evaluation rule *eval*(e) for some $e \in \mathcal{E}$, followed by garbage collection (see Section 3.1.2 for the definition of evaluation rules). The transformation unit *eval*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ applies a rule *eval*(e) exactly once to the input graph for some $e \in \mathcal{E}$.

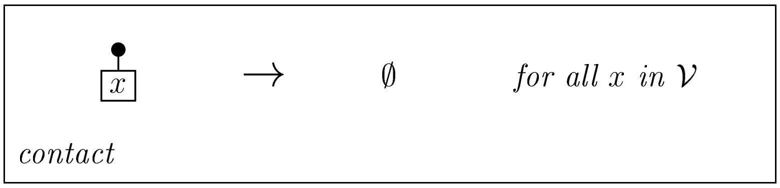
eval$(\mathcal{F}, \mathcal{V}, \mathcal{E})$		
rules:	*eval*(e)	*for all* e *in* $\mathcal{E}$
conds:	*once*	

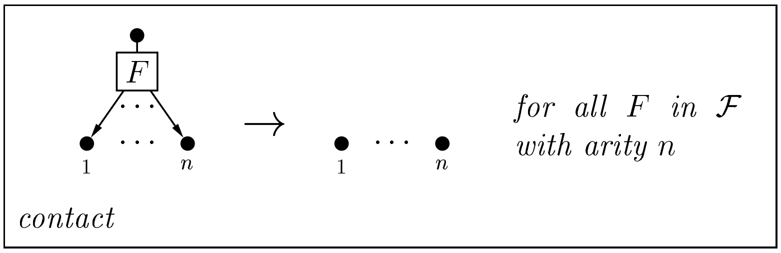
The semantics of *eval*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ consists of all pairs (G, G') where G is a graph and G' is obtained from G by applying an evaluation rule.

The transformation unit *garbage*$(\mathcal{F}, \mathcal{V})$ performs garbage collection. Each rule of *garbage*$(\mathcal{F}, \mathcal{V})$ deletes a node v together with its outgoing hyperedge if the contact condition is satisfied, i.e. if v is not incident to another hyperedge. Due to the contact condition, nodes and hyperedges reachable from the *root*-hyperedge cannot be deleted if the initial graph of *garbage*$(\mathcal{F}, \mathcal{V})$ results from applying an evaluation rule to some term graph.

garbage$(\mathcal{F}, \mathcal{V})$

rules:





terminal: *reduced*

The semantics of *garbage*$(\mathcal{F}, \mathcal{V})$ consists of all pairs (G, G') of graphs where G is a graph and G' is obtained from G by applying the rules of *garbage*$(\mathcal{F}, \mathcal{V})$ in arbitrary order and as long as possible.

Based on *eval*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ and *garbage*$(\mathcal{F}, \mathcal{V})$ the transformation unit *evaluation*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ is constructed. Since evaluation steps shall be performed on term graphs, the initial graphs of *evaluation*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ are those generated by the transformation unit *term_graphs*$(\mathcal{F}, \mathcal{V})$.

evaluation$(\mathcal{F}, \mathcal{V}, \mathcal{E})$

initial: *term_graphs*$(\mathcal{F}, \mathcal{V})$

uses: *eval*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$, *garbage*$(\mathcal{F}, \mathcal{V})$

conds: *eval*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$; *garbage*$(\mathcal{F}, \mathcal{V})$

The semantics of *evaluation*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ consists of all pairs (G, G') of term graphs where G' is obtained from G by applying first *eval*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ and then *garbage*$(\mathcal{F}, \mathcal{V})$. Hence, the semantics of *evaluation*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ coincides with the relation $\Rightarrow_{\mathcal{E}}$ of Section 3.1.2.

The transformation unit *rewrite*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ imports *evaluation*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ and *collapse*$(\mathcal{F}, \mathcal{V})$ which can be called arbitrarily often and in arbitrary order with the restriction that one of both is applied at least once.

rewrite$(\mathcal{F}, \mathcal{V}, \mathcal{E})$

initial: *term_graphs*$(\mathcal{F}, \mathcal{V})$

uses: *evaluation*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$, *collapse*$(\mathcal{F}, \mathcal{V})$

conds: $(\mathit{evaluation}(\mathcal{F}, \mathcal{V}, \mathcal{E}) \mid \mathit{collapse}(\mathcal{F}, \mathcal{V}))^+$

The semantics of *rewrite*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ consists of all pairs (G, G') of term graphs where G' is obtained from G by performing an arbitrary non-empty sequence of evaluation and collapse steps. Hence, this semantics coincides with the transitive closure of the relation $\Rightarrow_{\mathcal{R}}$ of Section 3.1.4.

Now we can construct a transformation unit *TGR*$(\mathcal{F}, \mathcal{V}, \mathcal{E})$ which rewrites term graphs as long as possible.

$TGR(\mathcal{F}, \mathcal{V}, \mathcal{E})$
    initial: $term\_graphs(\mathcal{F}, \mathcal{V})$
    uses: $rewrite(\mathcal{F}, \mathcal{V}, \mathcal{E})$
    conds: $rewrite(\mathcal{F}, \mathcal{V}, \mathcal{E})\,!$

The semantics of $TGR(\mathcal{F}, \mathcal{V}, \mathcal{E})$ contains all pairs (G, G') of term graphs where G' is obtained from G by performing an arbitrary sequence of evaluation and collapse steps and G' is reduced with respect to evaluation and collapse steps.

4.3.4 Lazy and eager collapsing

Depending on the given set $\mathcal{E}$ of equations, it may be desirable to rewrite term graphs according to some strategies in order to improve the efficiency. The control condition $rewrite(\mathcal{F}, \mathcal{V}, \mathcal{E})\,!$ in the above transformation unit $TGR(\mathcal{F}, \mathcal{V}, \mathcal{E})$ can be viewed as a simple, but general strategy for computing reduced term graphs. Two more specialized strategies, namely *lazy* and *eager collapsing*, are presented in the following.

Lazy collapsing. Lazy collapsing attempts collapsing steps only if no evaluation rule is applicable. If there are equations in $\mathcal{E}$ with repeated variables in their left-hand sides, collapsing may enable subsequent evaluation steps.

$lazy\_coll(\mathcal{F}, \mathcal{V}, \mathcal{E})$
    initial: $term\_graphs(\mathcal{F}, \mathcal{V})$
    uses: $evaluation(\mathcal{F}, \mathcal{V}, \mathcal{E}),\ collapse(\mathcal{F}, \mathcal{V})$
    conds: $(evaluation(\mathcal{F}, \mathcal{V}, \mathcal{E})\,!\ ;\ collapse(\mathcal{F}, \mathcal{V}))\,!$

Eager collapsing. Eager collapsing applies collapse rules to a term graph as long as possible before performing an evaluation step.

$eager\_coll(\mathcal{F}, \mathcal{V}, \mathcal{E})$
    initial: $term\_graphs(\mathcal{F}, \mathcal{V})$
    uses: $evaluation(\mathcal{F}, \mathcal{V}, \mathcal{E}),\ collapse(\mathcal{F}, \mathcal{V})$
    conds: $(collapse(\mathcal{F}, \mathcal{V})\,!\,;\ evaluation(\mathcal{F}, \mathcal{V}, \mathcal{E}))\,!$

The control conditions in these units express a special priority schema for transformation units. In general, these units need not have the same semantics as $TGR(\mathcal{F}, \mathcal{V}, \mathcal{E})$. An interesting question is for which classes of equation sets $\mathcal{E}$ the semantics coincide with the semantics of $TGR(\mathcal{F}, \mathcal{V}, \mathcal{E})$. In this case the strategies defined by the units are called normalizing (see also [Plu91a]).

The import structure of the presented transformation units is depicted in Figure 22 where solid edges represent the dependence relation given by the uses components and dashed edges show the use of the generating unit $term\_graphs(\mathcal{F}, \mathcal{V})$ in initial parts.

If we consider all presented transformation units as parameterized units, where each component inside parentheses is a formal parameter, we obtain a family of structured sets of transformation units, one for each list $(\mathcal{F}, \mathcal{V}, \mathcal{E})$ of actual parameters.

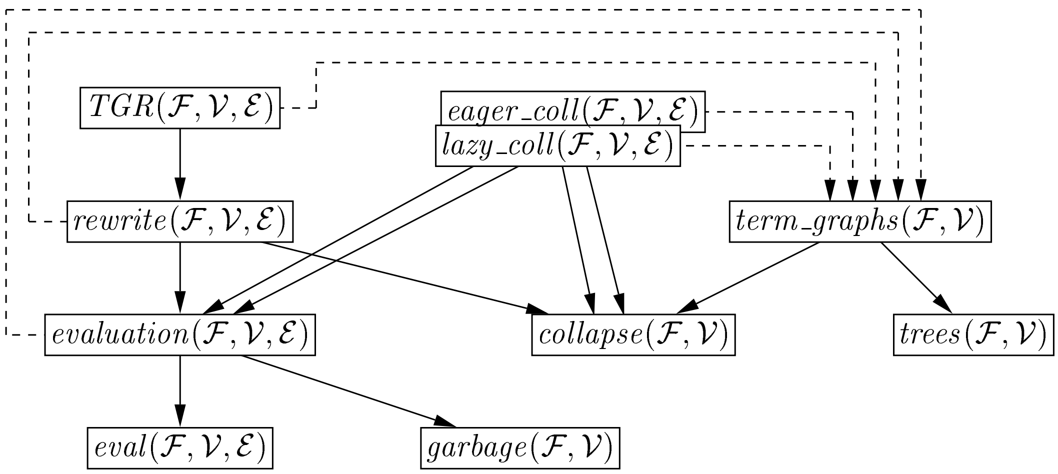


Figure 22: Import structure of a set of transformation units

5 Conclusion

In this paper, we have demonstrated the usefulness of graph transformation for specification and programming. The four case studies in Section 3 indicate the spectrum of potential applications; several more of similar kinds can be found in the literature. They all follow the same simple idea: There are complex objects that can be advantageously represented by graphs, and rules are used to specify transformation, generation or recognition of these objects. To emphasize the common philosophy we have recalled the main features of graph transformation in Section 2. While the case studies prove the principal applicability of graph transformation concepts in various areas, they also reveal some shortcomings and troublesome aspects. There are various types of graphs and rules, as well as various ways of rule applications around leading to many different graph transformation approaches which are not easily combined and among which one cannot easily switch. So the proper choice of an approach may cause problems. Moreover, structuring principles are missing so that it is hard to manage large collections of rules. To overcome these drawbacks of graph transformation, we have introduced the notion of transformation units in Section 4, which provides means for structuring purposes and is approach independent by integrating various approaches among which one may choose.

It should be mentioned that transformation units are the basic structuring concept of the graph- and rule-centered language GRACE. The development of GRACE is an ongoing activity involving researchers from Aachen, Berlin, Bremen, Erlangen, Hildesheim, München and Paderborn. The actual emphasis is laid on the following points:

1. The semantics of a transformation unit is a binary relation on graphs, or a graph language. Hence, more sophisticated module concepts are being developed to cover n-ary relations with n greater than 2, and sets of relations rather than single relations. Moreover, an explicit notion of parameterization, which appears in our example implicitly, is being introduced. (Confer some recent suggestions concerning modularization, refinement and abstraction [EE96, CH95, ES95, TS95].)

2. GRACE is planned to provide an efficient specification and programming environment that includes an editor, an interpreter, a database and a graphical interface. In particular, the latter will play an important role in that it allows to use GRACE as a visual specification and programming language. Since there are already some im-

plementations of graph transformation systems (see [Sch90, Sch91a, Sch91b, GKS91, Him91, LB93, TB94]), we will reuse their components as far as possible to build up the GRACE environment.

3. GRACE offers the choice among various graph transformation approaches. Furthermore, we will investigate the conditions under which approaches can be combined: As a final goal, we want to design a simple and easy-to-use language for defining new approaches by the users.
4. Another very important aim is to provide a proof (support) system for GRACE. The recursion depth of transformation units and the length of interleaving sequences induce induction principles. The use structure of a transformation unit may correspond to a modular proof structure. Both together establish a proof theory for GRACE on which a proof system can be built. (Confer the recent attempt towards consistency checking [HW95].)
5. Further case studies will particularly stress the application of the GRACE language and environment in the specification of parallel and distributed systems [Tae96], including appropriate semantic considerations like process semantics [CMR96].

Acknowledgement. We would like to thank the anonymous referees for their valuable comments which led to various improvements of this survey.

References

[ACS90] Michele Angelaccio, Tiziana Catarci, and Guiseppe Santucci. QBD*: A graphical query language with recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163, 1990.

[AE94] Marc Andries and Gregor Engels. Syntax and semantics of hybrid database languages. In Schneider and Ehrig [SE94], 19–36.

[AE96] Marc Andries and Gregor Engels. A hybrid query language for the extended entity relationship model. *Journal of Visual Languages and Computing*, 7(3):321–352, 1996.

[AK96] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26:207–240, 1996.

[AKP98] Zena M. Ariola, Jan Willem Klop, and Detlef Plump. Bisimilarity in term graph rewriting. *Information and Computation*, 1998. To appear.

[AL95] Andrea Asperti and Cosimo Laneve. Comparing λ-calculus translations in sharing graphs. In M. Dezani-Ciancaglini and Gordon Plotkin, editors, *Proc. Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 902, 1–15, 1995.

[And96] Marc Andries. *Graph Rewrite Systems and Visual Database Languages.* PhD thesis, Rijksuniversiteit te Leiden, the Netherlands, 1996. Available via ftp as ftp://ftp.wi.leidenuniv.nl/pub/CS/PhDTheses/andries-96.ps.gz.

[BGT91] Horst Bunke, T. Glauser, and T-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In Ehrig et al. [EKR91], 174–189.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Bun79] Horst Bunke. Programmed graph grammars. In Claus et al. [CER79], 155–166.

[Bun82] Horst Bunke. On the generative power of sequential and parallel programmed graph grammars. *Computing*, 29:89–112, 1982.

[BvEG+87] Hendrik Barendregt, Marko van Eekelen, John R. W. Glauert, J. Richard Kennaway, Rinus Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In *Proc. Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 259, 141–158, 1987.

[CEER96] Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science 1073, 1996.

[CER79] Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science 73, 1979.

[CH95] Andrea Corradini and Reiko Heckel. A compositional approach to structuring and refinement of typed graph grammars. In Corradini and Montanari [CM95].

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[CM93] Mariano Consens and Alberto Mendelzon. Hy+: A hygraph-based query and visualization system. *SIGMOD RECORD*, 22(2):511–516, 1993.

[CM95] Andrea Corradini and Ugo Montanari, editors. *Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, Electronic Notes in Theoretical Computer Science 2, http://www.elsevier.nl/locate/entcs, 1995.

[CMR+91] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, and Michael Löwe. Graph grammars and logic programming. In Ehrig et al. [EKR91], 221–237.

[CMR96] Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.

[CMR+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation – Part I: Basic concepts and double pushout approach. In Rozenberg [Roz97], 163–245.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logical approach. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, 193–242. Elsevier, Amsterdam, 1990.

[CR93] Andrea Corradini and Francesca Rossi. Hyperedge replacement jungle rewriting for term rewriting systems and logic programming. *Theoretical Computer Science*, 109:7–48, 1993.

[CW94] Andrea Corradini and Dietmar Wolz. Jungle rewriting: an abstract description of a lazy narrowing machine. In Schneider and Ehrig [SE94], 119–137.

[DFS75] Ernst Dehnert, Reinhold Franck, and Wolfgang Streng. PLAN2D — towards a two-dimensional programming language. In Dirk Siefkes, editor, *4. Jahrestagung der Gesellschaft für Informatik*, Lecture Notes in Computer Science 26, 202–213, 1975.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, 243–320. Elsevier, 1990.

[DP89] Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory.* EATCS Monographs on Theoretical Computer Science 18. Springer-Verlag, 1989.

[EE96] Hartmut Ehrig and Gregor Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In Cuny et al. [CEER96], 137–154.

[EGH+92] Gregor Engels, Martin Gogolla, Uwe Hohenstein, Klaus Hülsmann, Perdita Löhr-Richter, Gunter Saake, and Hans-Dieter Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9(2):157–204, 1992.

[EH86] Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In Grzegorz Rozenberg and Arto Salomaa, editors, *The Book of L*, 87–100. Springer-Verlag, 1986.

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

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In Claus et al. [CER79], 1–69.

[EK76] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of manipulations in multidimensional information structures. In *Proc. Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 45, 284–293, 1976.

[EKR91] Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science 532, 1991.

[ELN+92] Gregor Engels, Claus Lewerentz, Manfred Nagl, Wilhelm Schäfer, and Andy Schürr. Building integrated software developments, part I: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1:135–167, 1992.

[ENR83] Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science 153, 1983.

[ENRR87] Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors. *Proc. Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science 291, 1987.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, 167–180, Iowa City, 1973.

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

[ES95] Gregor Engels and Andy Schürr. Hierarchical graphs, graph types and meta types. In Corradini and Montanari [CM95].

[FW90] William M. Farmer and Ronald J. Watro. Redex capturing in term graph rewriting. *International Journal of Foundations of Computer Science*, 1(4):369–386, 1990.

[GKM87] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In *Proc. Graph Reduction*, Lecture Notes in Computer Science 279, 53–93, 1987.

[GKS91] John R. W. Glauert, J. Richard Kennaway, and M. Ronan Sleep. DACTL: An experimental graph rewriting language. In Ehrig et al. [EKR91], 378–395.

[Göt88] Herbert Göttler. *Graphgrammatiken in der Softwaretechnik*. Informatik-Fachberichte 178. Springer-Verlag, Berlin, 1988.

[GPTvdB93] Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan van den Bussche. GOOD: A graph-oriented object database system. *SIGMOD RECORD*, 22(2):505–510, 1993.

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

[HE92] Uwe Hohenstein and Gregor Engels. SQL/EER – syntax and semantics of an entity-relationship-based query language. *Information Systems*, 17(3):209–242, 1992.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.

[Him91] Michael Himsolt. Graph^{ed}: An interactive tool for developing graph grammars. In Ehrig et al. [EKR91], 61–65.

[HK87] Annegret Habel and Hans-Jörg Kreowski. Characteristics of graph languages generated by edge replacement. *Theoretical Computer Science*, 51:81–115, 1987.

[HKP91] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. *Fundamenta Informaticae*, 15:37–60, 1991.

[Hof92] B. Hoffmann. Term Rewriting with Sharing and Memoïzation. In H. Kirchner and G. Levi, editors, *Proc. 3rd Int Conf. on Algebraic and Logic Programming (ALP'92)*, Lecture Notes in Computer Science 632, 128–142, 1992.

[HP88] Berthold Hoffmann and Detlef Plump. Jungle evaluation for efficient term rewriting. In *Proc. Algebraic and Logic Programming.* Mathematical Research 49, 191–203, Berlin, 1988. Akademie-Verlag. Also in Lecture Notes in Computer Science 343, 191–203, 1989.

[HP91] Berthold Hoffmann and Detlef Plump. Implementing term rewriting by jungle evaluation. *RAIRO Theoretical Informatics and Applications*, 25(5):445–472, 1991.

[HP96] Annegret Habel and Detlef Plump. Term graph narrowing. *Mathematical Structures in Computer Science*, 6:649–676, 1996.

[Hug85] R. John M. Hughes. Lazy Memo Functions. In J.-P. Jouannaud, editor, *IFIP Conf. on Funct. Lang. and Comp. Arch.*, Lecture Notes in Computer Science 201, 129–146, 1985.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In Corradini and Montanari [CM95].

[JR80] Dirk Janssens and Grzegorz Rozenberg. On the structure of node-label-controlled graph languages. *Information Sciences*, 20:191–216, 1980.

[Kah92] Stefan Kahrs. Unlimp: Uniqueness as a leitmotiv for implementation. In M. Bruynooghe and M. Wirsing, editors, *Proc. Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 631, 115–129, 1992.

[KK96] Hans-Jörg Kreowski and Sabine Kuske. On the interleaving semantics of transformation units — a step into GRACE. In Cuny et al. [CEER96], 89–108.

[KKS98] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1998.

[KKSdV94] Richard Kennaway, Jan Willem Klop, Ronan Sleep, and Fer-Jan de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994.

[Klo92] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, 1–116. Oxford University Press, 1992.

[KO95] Masahito Kurihara and Azuma Ohuchi. Modularity in noncopying term rewriting. *Theoretical Computer Science*, 152(1):139–169, 1995.

[KR90] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars, I and II. *Information Sciences*, 52:185–210 and 221–246, 1990.

[Kre87] Hans-Jörg Kreowski. Is parallelism already concurrency? Part 1: Derivations in graph grammars. In Ehrig et al. [ENRR87], 343–360.

[Kre93] Hans-Jörg Kreowski. Five facets of hyperedge replacement beyond context-freeness. In Z. Ésik, editor, *Proc. Fundamentals of Computation Theory*, Lecture Notes in Computer Science 710, 69–86, 1993.

[KSW95] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Sciences*, 20:21–51, 1995.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. 17th Symposium on Principles of Programming Languages*, 16–30. ACM Press, Addison-Wesley, 1990.

[LB93] Michael Löwe and Martin Beyer. AGG — an implementation of algebraic graph rewriting. In Claude Kirchner, editor, *Proc. Rewriting Techniques and Applications*, Lecture Notes in Computer Science 690, 451–456, 1993.

[LM93] Igor Litovsky and Yves Métivier. Computing with graph rewriting systems with priorities. *Theoretical Computer Science*, 115:191–224, 1993.

[Löw93] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[MW96] Andrea Maggiolo-Schettini and Józef Winkowski. A kernel language for programmed rewriting of (hyper)graphs. *Acta Informatica*, 33:523–546, 1996.

[Nag87] Manfred Nagl. Set theoretic approaches to graph grammars. In Ehrig et al. [ENRR87], 41–54.

[Pad82] Peter Padawitz. Graph grammars and operational semantics. *Theoretical Computer Science*, 19:117–141, 1982.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PH96] Detlef Plump and Annegret Habel. Graph unification and matching. In Cuny et al. [CEER96], 75–89.

[Plu91a] Detlef Plump. Graph-reducible term rewriting systems. In Ehrig et al. [EKR91], 622–636.

[Plu91b] Detlef Plump. Implementing term rewriting by graph reduction: Termination of combined systems. In Stephane Kaplan and M. Okada, editors, *Proc. Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science 516, 307–317, 1991.

[Plu93a] D. Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In Sleep et al. [SPvE93], 201–213.

[Plu93b] Detlef Plump. Collapsed tree rewriting: Completeness, confluence, and modularity. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Proc. Conditional Term Rewriting Systems*, Lecture Notes in Computer Science 656, 97–112, 1993.

[Plu93c] Detlef Plump. *Evaluation of Functional Expressions by Hypergraph Rewriting*. PhD thesis, Universität Bremen, 1993.

[Plu94] Detlef Plump. Critical pairs in term graph rewriting. In Igor Prívara, Branislav Rovan, and Peter Ružička, editors, *Proc. Mathematical Foundations of Computer Science 1994*, Lecture Notes in Computer Science 841, 556–566, 1994.

[Plu95] Detlef Plump. On termination of graph rewriting. In Manfred Nagl, editor, *Proc. Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science 1017, 88–100, 1995.

[Plu97] Detlef Plump. Simplification orders for term graph rewriting. In *Proc. Mathematical Foundations of Computer Science 1997*, Lecture Notes in Computer Science 1295, 458–467, 1997.

[Plu98] Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[PR69] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *Proc. Int. Joint Confenence on Artificial Intelligence*, 609–619, 1969.

[Pra69] Terrence W. Pratt. Semantic modeling by hierarchical graphs. *ACM SIGPLAN Symp. on Prog. Lang. Def.*, 1969.

[Pra71] Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Rao95] M.R.K. Krishna Rao. Graph reducibility of term rewriting systems. In *Proc. Mathematical Foundations of Computer Science 1995*, Lecture Notes in Computer Science 969, 371–381, 1995.

[Rao96] M.R.K. Krishna Rao. Modularity of termination in term graph rewriting. In *Proc. Rewriting Techniques and Applications*, Lecture Notes in Computer Science 1103, 230–244, 1996.

[RM72] Azriel Rosenfeld and D. Milgram. Web automata and web grammars. *Machine Intelligence*, 7:307–324, 1972.

[Roz87] Grzegorz Rozenberg. An introduction to the NLC way of rewriting graphs. In Ehrig et al. [ENRR87], 55–66.

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

[RS97] Jan Rekers and Andy Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[Sch70] Hans-Jürgen Schneider. Chomsky-Systeme für partielle Ordnungen. Technical Report 3-3, Universität Erlangen, 1970.

[Sch71] Hans-Jürgen Schneider. Formal systems for structure manipulations. Technical Report 3/1/71, TU Berlin, 1971.

[Sch90] Andy Schürr. Introduction to PROGRES, an attribute graph grammar based specification language. In Manfred Nagl, editor, *Proc. Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science 411, 151–165, 1990.

[Sch91a] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung.* PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 1991.

[Sch91b] Andy Schürr. PROGRES: A VHL-language based on graph grammars. In Ehrig et al. [EKR91], 641–659.

[Sch94] Andy Schürr. Rapid programming with graph rewrite rules. In *Proc. USENIX Symp. Very High Level Languages (VHLL), Santa Fé, New Mexico*, 83–100, 1994.

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

[SE94] Hans-Jürgen Schneider and Hartmut Ehrig, editors. *Proc. Graph Transformations in Computer Science*, Lecture Notes in Computer Science 776, 1994.

[SPvE93] M. Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors. *Term Graph Rewriting, Theory and Practice.* Wiley & Sons, Chichester, 1993.

[Sta80] John Staples. Computation on graph-like expressions. *Theoretical Computer Science*, 10:171–185, 1980.

[SWZ95] Andy Schürr, Andreas Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In *Proc. 5th European Software Engineering Conf. (ESEC'95)*, Lecture Notes in Computer Science 989, 219–234. Springer Verlag, 1995.

[SZ91] Andy Schürr and Albert Zündorf. Nondeterministic control structures for graph rewriting systems. In G. Schmidt and Rudolf Berghammer, editors, *Proc. Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science 570, 48–62, 1991.

[Tae96] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, Shaker Verlag, 1996.

[TB94] Gabriele Taentzer and M. Beyer. Amalgamated graph transformation systems and their use for specifying AGG – an algebraic graph grammar system. In Schneider and Ehrig [SE94], 380–394.

[TS95] Gabriele Taentzer and Andy Schürr. DIEGO, another step towards a module concept for graph transformation systems. In Corradini and Montanari [CM95].

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, University of Oxford, 1971.

[Wan91] Egon Wanke. PLEXUS: Tools for analyzing graph grammars. In Ehrig et al. [EKR91], 68–69.

[Zün95] Albert Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaph-ErsetzungsSysteme*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 1995.