PRACE IPI PAN • ICS PAS REPORTS

Andrea Maggiolo-Schettini
and
Józef Winkowski

A FORMALISM FOR PROGRAMMED REWRITING OF (HYPER)GRAPHS*

745

INSTYTUT PODSTAW INFORMATYKI
POLSKIEJ AKADEMII NAUK

INSTITUTE OF COMPUTER SCIENCE
POLISH ACADEMY OF SCIENCES

Andrea Maggiolo-Schettini
and
Józef Winkowski

A FORMALISM FOR PROGRAMMED REWRITING OF (HYPER)GRAPHS*

745

PAN

Warszawa, July 1994

*This work has been supported by the Italian National Council for Research (CNR-GNIM), by the Polish Academy of Sciences (IPI PAN), and by COMPUGRAPH Basic Research Esprit Working Group n. 7183

Pracę zgłosił Piotr Dembiński

Adresy autorów: Andrea Maggiolo-Schettini
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
56100-Pisa
Italy

Józef Winkowski
Institute of Computer Science
Polish Academy of Sciences
ul. Ordona 21
01-237 Warszawa
Poland

CR: F.1., H.1.

Printed as a manuscript
Na prawach rękopisu

Nakład 180 egz. Ark. wyd. 1,20; ark. druk. 1,00. Papier kserograficzny kl. III.
Oddano do druku w lipcu 1994 r. Wydawnictwo IPI PAN

ISSN: 0138-0648

Abstract. Streszczenie

The paper presents a formalism for rewriting (hyper)graphs in a controlled manner. This formalism is essentially a simple programming language with productions, that is rewriting rules, playing the role of basic instructions. The programs in this language are built from productions by means of rather standard constructors, including a parallel composition. They may contain parameters to point to specific elements of graphs to which they are supposed to be applied. The programs are intended to describe how to transform a graph and a valuation of parameters in this graph in order to reach a resulting graph and a resulting valuation of parameters.

FORMALIZM DO PROGRAMOWANIA TRANSFORMACJI (HYPER)GRAFÓW

Praca prezentuje formalizm do kontrolowanego przetwarzania (hyper)grafów. Formalizm ten stanowi w istocie prosty język programowania z produkcjami, tzn. regulami przetwarzania, odgrywającymi rolę instrukcji podstawowych. Programy w tym języku są budowane z produkcji za pomocą typowych konstruktorów programotwórczych wśród których jest złożenie równoległe. Programy mogą zawierać parametry. Parametry programu wskazują elementy grafów do których ten program ma być stosowany. Ogólnie, program opisuje jak transformować graf i wartościowanie parametrów w tym grafie ażeby uzyskać graf wynikowy i wynikowe wartościowanie parametrów.

Key words:
(hyper)graph, rewriting rule, production, parametrized production, rewriting step, program of rewriting, structured operational semantics, resulting relation.

1 Introduction

Some models of computing can be formulated in a natural manner in terms of rewriting of appropriate data structures represented as graphs. Take, for instance, the representation of actor systems as in [JaRo 91] or that of logic programs in [CMREL 91].

A theory of graph rewriting systems has been developed which describes how to rewrite graphs according to formal rules called productions, where a rule says that a certain given pattern can be replaced by another pattern if it occurs in a graph, and where graphs may be of very general types, including hypergraphs, coloured hypergraphs, relational structures, etc. (cf. [CER 79], [EKMRW 82], [ENR 83], [ENRR 87], [EKR 91]). This theory in its pure form does not assume anything about where and in what order to apply productions. In this situation at each stage of rewriting an independent search of an applicable rule and of a place of application must be done, which in general is a task of high complexity. On the other hand, in some problems the structure of data and the algorithm to solve the problem allow to organize rewriting in an efficient manner.

This paper presents a formalism with mechanisms which make such an organized rewriting possible. Our formalism is in the framework of the algebraic approach proposed in [EPS 73]. It is essentially a kernel of a simple programming language with productions playing the role of basic instructions. Programs are built in this formalism from productions by means of rather standard constructors which define the order and modalities of rewriting steps. Among the program constructors there is a parallel composition which declares the possibility of executing programs in parallel. The parallelism is understood here as an arbitrary interleaving of atomic (i.e. indivisible) actions of component programs, where atomic actions are either single instructions or complexes of instructions which are specified as atomic with the aid of a special constructor. Productions and programs may contain parameters to point to particular elements of graphs to which they are supposed to be applied. When applied to pairs consisting of a graph and a valuation of parameters in this graph they transform such pairs one into another as long as it follows from their meaning. The mechanism of accessing graphs through valuations of parameters allows to enforce components of a program to operate on the same data and to realize shared variables whose values represent some parts of data.

The presented formalism is endowed with a structured operational semantics in the style of [Plo 81]. This semantics defines the possible executions of a program. Consequently, it determines the corresponding relations between the data and results of performed executions.

The formalism we define may be useful whenever a problem can naturally be reduced to graph rewriting and the process of rewriting is too complex to be represented as a result of a free application of a system of productions. We shall illustrate it on example of a concurrent execution of a program in a simple concurrent logic language (called FCP after [Sh 89]).

1.1. Example. Consider the logic program:

$$\begin{aligned}
&sum(Y,S) \leftarrow sum'(Y,0,S)\\
&sum'([],P,S) \leftarrow P = S\\
&sum'([X|Y],P,S) \leftarrow plus(X,P,Q), sum'(Y,Q,S)\\
&plus(0,0,X) \leftarrow X = 0\\
&plus(0,1,X) \leftarrow X = 1\\
&\ldots
\end{aligned}$$

When applied to the goal $sum([1,2],S)$ this program computes the sum of elements of the list $Y = [1,2]$ and assigns the result to S.

If such a program is regarded as a concurrent logic program in FCP then the atomic formula of the goal and those which are obtained by applying the clauses of this program to the goal can be viewed as processes which communicate via their variables (in [Sh 89] such variables are called logical ones). Each process of this type keeps trying to match its formula (that is the formula it corresponds to) with a clause head by a suitable substitution of terms for variables, and, if successful, it creates processes corresponding to the atomic formulas of the right hand side of the clause. This procedure may imply an instantiation of variables the process shares with other existing processes. Due to this, the processes awaiting for such an instantiation may advance.

For our program and goal we obtain the following computation:

$$\begin{aligned}
&sum([1,2],S)\\
&sum'([1,2],0,S)\\
&plus(1,0,P), sum'([2],P,S)\\
&plus(1,0,P), plus(2,P,Q), sum'([],Q,S)\\
&P = 1, plus(2,P,Q), sum'([],Q,S)\\
&P = 1, plus(2,P,Q), Q = S\\
&P = 1, plus(2,P,S)\\
&plus(2,1,S)\\
&S = 3.
\end{aligned}$$

In this computation S, P, Q are variables. In order to find the required sum and assign it to S, the process $sum([1,2],S)$ matches its formula with the head of the first clause and creates the process $sum'([1,2],0,S)$. This process matches its formula with the third clause and creates two parallel processes $plus(1,0,P)$ and $sum'([2],P,S)$ which contain a new variable P. Now $plus(1,0,P)$ instantiates P to 1 and $sum'([2],P,S)$ evolves into $plus(2,P,Q)$ and $sum'([],Q,S)$. The processes $plus(1,0,P)$ and $plus(2,P,Q)$ synchronize in the sense that $plus(2,P,Q)$ waits for instantiation of P to 1 in order to instantiate Q. As simultaneously S is instantiated to Q due to the second clause, we obtain finally the required result $S = 3$.

In our approach each state of a computation of this type is represented by a jungle as in [CR 93]. For instance, the state $plus(1,0,P), sum'([2],P,S)$ is represented as shown in figure 1.1. In this representation hyperedges correspond to concrete occurrences of predicate and function symbols and nodes represent terms (more precisely, nodes are roots of subjungles which represent terms). For instance, the node σ represents the term 2|[], that is the list [2].

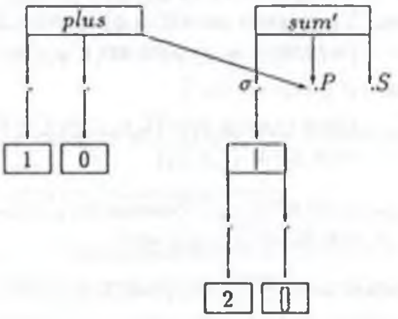


Figure 1.1

Processes which take part in a computation are present in it as subjungles which represent the respective atomic formulas. They are realized by calling with suitable values of parameters and executing, possibly many times, programs which specify how a process of a given class performs its step. Such a realization leads usually to parallel processes and then it appears as an interleaving of actions of the existing processes which is synchronized solely by instantiations of variables shared by processes.

For an illustration of this way of representing and realizing of processes let us consider the process $plus(1,0,P)$. Each step of this process can be realized by calling with ξ representing 1 and η representing 0, and with $\zeta = P$, and executing a program $PLUS(\xi,\eta,\zeta)$ whose body can be defined as

```
Σx Σy   if     ξ represents x and η represents y
        then   replace plus(ξ,η,ζ) by ζ = x + y
        else   PLUS(ξ,η,ζ)
```

where *replace* $plus(\xi,\eta,\zeta)$ *by* $\zeta = x + y$ denotes a reduction of the jungle which represents

$plus(\xi,\eta,\zeta)$ (see the leftmost jungle in figure 2.4) to a single edge without target nodes, with a single source node ζ, and with the colour equal to the sum of x and y (see the rightmost jungle in figure 2.4), and where Σx and Σy denote an indeterministic choice of some instances of x and y in the jungle to which the program is applied. Each call of $PLUS(\xi,\eta,\zeta)$ is an attempt of instantiating variables of terms represented by ξ, η, ζ. If successful, it terminates the process $plus(1,0,P)$ with $P = x + y$. Otherwise it causes a subsequent call of $PLUS(\xi,\eta,\zeta)$ which may be viewed as a subsequent step the process $plus(1,0,P)$ to be realized (possibly in parallel with other processes) after completing the current step.

Similarly, each step of the process $sum'([1,2],0,S)$ can be realized by calling with ξ representing $[1,2]$, η representing 0, and with $\zeta = S$, and executing a program $SUM'(\xi,\eta,\zeta)$ whose body can be defined as

```
$\Sigma X\ \Sigma Y$   if    $\xi$ represents $[X|Y]$
            then  $\Sigma\varrho$ (replace $sum'(\xi,\eta,\zeta)$ by $plus(X,\eta,\varrho)$ and $sum'(Y,\varrho,\zeta)$;
                  $(PLUS(X,\eta,\varrho) \parallel SUM'(Y,\varrho,\zeta)))$
            else  if $\xi$ represents []
                  then replace $sum'(\xi,\eta,\zeta)$ by $\zeta = 0$
                  else $SUM'(\xi,\eta,\zeta))$
```

where $(PLUS(X,\eta,\varrho) \parallel SUM'(Y,\varrho,\zeta)$ denotes the parallel interleaving execution of programs $PLUS(X,\eta,\varrho)$ and $SUM'(Y,\varrho,\zeta)$, and

$$replace\ sum'(\xi,\eta,\zeta)\ by\ plus(X,\eta,\varrho)\ and\ sum'(Y,\varrho,\zeta),$$

$$replace\ sum'(\xi,\eta,\zeta)\ by\ \zeta = 0$$

are the operations of replacing the leftmost jungle by the rithtmost one in figures 2.2 and 2.4, respectively. The call of $SUM'(\xi,\eta,\zeta)$ if ξ represents the empty list [] may be viewed as a subsequent step of the realized process $sum'([1,2],0,S)$ whereas $PLUS(X,\eta,\varrho)$ and $SUM'(Y,\varrho,\zeta)$ start two new processes $plus(1,0,P)$ and $sum'([2],P,S)$.

The present paper extends and improves a previous work in [MW 83], [MW 91], and [MW 92]. It is organized as follows. In section 2 we recall and modify for our purposes the basic notions related to rewriting graphs. In section 3 we define programs of rewriting graphs. In section 4 we present a structured operational semantics of these programs. In section 5 we describe input-output relations of programs.

2 Graphs, productions, and derivations

Let Λ be a fixed many-sorted first-order language with equality which has sorts *nodes*, *edges*, *colours*, operation symbols
$none :\rightarrow nodes$,
$1\_source, ..., m\_source, 1\_target, ..., n\_target : edges \rightarrow nodes$,
$edgecolour : edges \rightarrow colours$,

$\omega_1 : colours \times \ldots \times colours \rightarrow colours$, $\omega_2 : colours \times \ldots \times colours \rightarrow colours$, etc., and infinite, mutually disjoint sets *nodevariables*, *edgevariables*, *colourvariables* of node-, edge-, and colour variables, respectively.

Let Ω denote the signature consisting of the sorts and operation symbols of Λ and Ω_0 the part of Ω consisting of the sort *colours* and operation symbols $\omega_1, \ldots$.

By an *Ω-graph* (or a *graph*) we mean an Ω-algebra G such that $i\_source_G(x) = none_G$ implies $j\_source_G(x) = none_G$ for $j \geq i$ and $i\_target_G(x) = none_G$ implies $j\_target_G(x) = none_G$ for $j \geq i$. By $source_G(x)$ (resp.: by $target_G(x)$) we denote the string of subsequent different from $none_G$ nodes $i\_source_G(x)$ (resp.: the string of subsequent different from $none_G$ nodes $i\_target_G(x)$). Thus we obtain functions

$source_G, target_G : edges_G \rightarrow (nodes_G)^*$.

By an *Ω-homomorphism* from an Ω-graph G to an Ω-graph G' we mean any homomorphism $h : G \rightarrow G'$ from the Ω-algebra G to the Ω-algebra G', and by h_{nodes}, h_{edges}, $h_{colours}$, we denote the components of h corresponding to the sorts *nodes*, *edges*, *colours*, respectively.

By *Ω_graphs* we denote the category of Ω-graphs and Ω-homomorphisms.

The category *Ω_graphs* enjoys the following property.

2.1. Proposition. Each pair of Ω-homomorphisms of the form $(L \xleftarrow{l} K \xrightarrow{d} D)$ such that the pair $(\Omega_0\_reduct(L) \xleftarrow{l_{colours}} \Omega_0\_reduct(K) \xrightarrow{d_{colours}} \Omega_0\_reduct(D))$ has a pushout $(\Omega_0\_reduct(L) \xrightarrow{g'} X \xleftarrow{b'} \Omega_0\_reduct(D))$ in the category of Ω-algebras has also a pushout $(L \xrightarrow{g} G \xleftarrow{b} D)$ in *Ω_graphs*, where $\Omega_0\_reduct(G) = X$, $g_{colours} = g'$, $b_{colours} = b'$, $nodes_G$, $edges_G$, g_{nodes}, g_{edges}, b_{nodes}, b_{edges} are obtained from the respective pushouts in the category of sets, and i_source_G, i_target_G, $edgecolour_G$ are determined by the properties of Ω-graphs. $\square$

In particular, if we are not interested in operations on colours, and so in Ω there are no operation symbols, then $(\Omega_0\_reduct(L) \xleftarrow{l_{colours}} \Omega_0\_reduct(K) \xrightarrow{d_{colours}} \Omega_0\_reduct(D))$ is a diagram in the category of sets. Consequently, the corresponding category *Ω_graphs* has pushouts.

The fact that $nodes_G$, $edges_G$, g_{nodes}, g_{edges}, b_{nodes}, b_{edges} can be taken from the respective pushouts in the category of sets is straightforward.

As far as the definitions and uniqueness of i_source_G, i_target_G, $edgecolour_G$ are concerned, they follow from the fact that, if an edge $x \in edges_L$ has $i\_source_L(x) = y$, $j\_target_L(x) = z$, and $edgecolour_L(x) = u$, then the only candidate for $i\_source_G(g_{edges}(x))$ is $g_{nodes}(y)$, the only candidate for $j\_target_G(g_{edges}(x))$ is $g_{nodes}(z)$, and the only candidate for $edgecolour_G(g_{edges}(x))$ is $g_{colours}(u)$.

In this paper we restrict ourselves to special pushouts.

Given a pair of Ω-homomorphisms $(L \xleftarrow{l} K \xrightarrow{d} D)$ and a pushout $(L \xrightarrow{g} G \xleftarrow{b} D)$ of this pair, we call this pushout *natural*, and we call $(K \xrightarrow{d} D \xrightarrow{b} G)$ the *natural pushout complement* of $(K \xrightarrow{l} L \xrightarrow{g} G)$, if the components of $b : D \to G$ are inclusions.

For our purposes we shall have to do only with cases in which natural pushouts and pushout complements can be obtained in a particularly simple way.

Let Ω_0_terms be the Ω_0-algebra of terms with colour variables. From the universal property of free algebras we obtain the following proposition.

2.2. Proposition. For each Ω_0-algebra A and each Ω_0-homomorphism $a : \Omega_0\ terms \to A$, the diagram $(\Omega_0\_terms \xleftarrow{identity} \Omega_0\_terms) \xrightarrow{a} A)$ has a pushout and the diagram $(\Omega_0\_terms \xrightarrow{a} A \xleftarrow{identity} A)$ is such a pushout. □

From this proposition we obtain immediately the following one.

2.3. Proposition. Each pair of morphisms $(L \xleftarrow{l} K \xrightarrow{d} D)$ with $\Omega_0\_reduct(L)$ and $\Omega_0\_reduct(K)$ equal to Ω_0_terms, and with $K \xleftarrow{l_{colours}} L$ being the identity homomorphism, has a natural pushout $(L \xrightarrow{g} G \xleftarrow{b} D)$. □

2.4. Proposition. Each pair of Ω-homomorphisms $(K \xrightarrow{l} L \xrightarrow{g} G)$ such that $\Omega_0\_reduct(L)$ and $\Omega_0\_reduct(K)$ are equal to Ω_0_terms, $K \xleftarrow{l_{colours}} L$ is the identity homomorphism, g_{nodes} is one-to-one in $nodes_L - l_{nodes}(nodes_K)$, g_{edges} is one-to-one in $edges_L - l_{edges}(edges_K)$, and all $i\_source_G(x)$ and $j\_target_G(x)$ with $x \in edges_G - g_{edges}(edges_L)$ are in

$$(nodes_G - g_{nodes}(nodes_L)) \cup g_{nodes}(l_{nodes}(nodes_K))),$$

has a natural pushout complement $(K \xrightarrow{d} D \xrightarrow{b} G)$. Moreover,

$$nodes_D = (nodes_G - g_{nodes}(nodes_L)) \cup g_{nodes}(l_{nodes}(nodes_K)),$$

$$edges_D = (edges_G - g_{edges}(edges_L)) \cup g_{edges}(l_{edges}(edges_K)),$$

$$\Omega_0\_reduct(D) = \Omega_0\_reduct(G) = \Omega_0\_terms,$$

d is defined by $d_{nodes}(x) = g_{nodes}(l_{nodes}(x))$, $d_{edges}(x) = g_{edges}(l_{edges}(x))$, $d_{colours}(x) = g_{colours}(l_{colours}(x))$, and b consists of inclusions b_{nodes}, b_{edges} and of the identity of colours. □

A proof of this proposition for the graph part is essentially as the proof of a similar theorem for relational structures in [EKMRW 82]. For the colour part it follows from 2.3.

The category Ω_graphs may be too large for some applications. For example, for logic programming the full subcategory of this category with graphs being *jungles* is more suitable, where a jungle is a (hyper)graph without cycles and without edges having a common node in sources (cf. [CMREL 91] and [CRP 91]). For other applications we may need other subcategories in which pushouts and pushout complements, arbitrary or natural, may be different from those in Ω_graphs, or even may not exist. So, in the sequel we shall relate all our formalism to an arbitrary but fixed full subcategory C of Ω_graphs.

Productions representing rewriting rules for Ω-graphs can be defined as follows.

By a *production* we mean any $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, where L, K, R are Ω-graphs in the subcategory C with finite sets of nodes and edges and the Ω_0-reducts coinciding with Ω_0_terms, the Ω_0-algebra of terms, and $l : K \to L$, $r : K \to R$ are Ω-homomorphisms with $l_{colours}$ and $r_{colours}$ being identities.

We call K the *gluing graph* of p, and we call L and R the *left side* and the *right side* of p, respectively.

In order to be able to point to some elements of productions, we introduce suitable concepts of parametrized productions and parameters.

By a *parametrized* production we mean any $p = (L \xleftarrow{l} K \xrightarrow{r} R, m, n)$, where $pr_p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is a production and m and n are triples $m = (m_{nodes}, m_{edges}, m_{colours})$ and $n = (n_{nodes}, n_{edges}, n_{colours})$ of partial mappings

$$\begin{aligned}
&m_{nodes} : nodes_L \supseteq\!\to nodevariables\\
&m_{edges} : edges_L \supseteq\!\to edgevariables\\
&m_{colours} : colours_L \supseteq\!\to colourvariables\\
&n_{nodes} : nodes_R \supseteq\!\to nodevariables\\
&n_{edges} : edges_R \supseteq\!\to edgevariables\\
&n_{colours} : colours_R \supseteq\!\to colourvariables
\end{aligned}$$

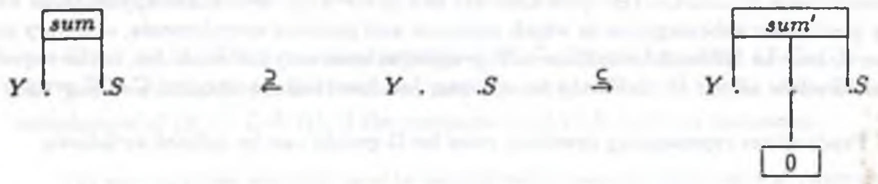
such that

$$\begin{aligned}
&m_{nodes}(l_{nodes}(x)) = n_{nodes}(r_{nodes}(x)) \text{ for all } x \in nodes_K,\\
&m_{edges}(l_{edges}(y)) = n_{edges}(r_{edges}(y)) \text{ for all } y \in edges_K,
\end{aligned}$$

and $m_{colours}$, $n_{colours}$ are respectively an inclusion of a subset of colour-variables occurring in Ω_0-terms assigned to edges of L and an inclusion of a subset of colour-variables occurring in Ω_0-terms assigned to edges of R.

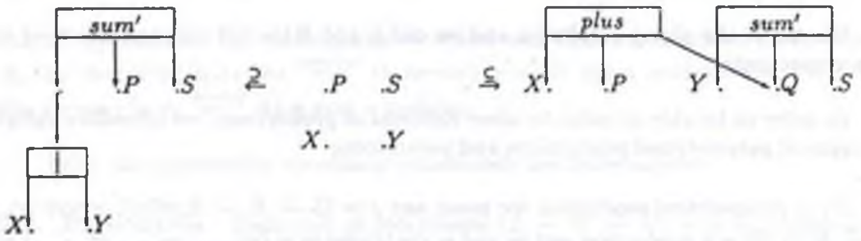
Values of mappings m_{nodes}, n_{nodes}, m_{edges}, n_{edges}, $m_{colours}$, $n_{colours}$ are called respectively *node-*, *edge-*, and *colour-parameters* of p.

2.5. Example. In the case of the logic program in 1.1 atomic formulas corresponding to processes can be rewritten according to the parametrized productions in figures 2.1 - 2.4, where the labels play the role of parameters. ▢



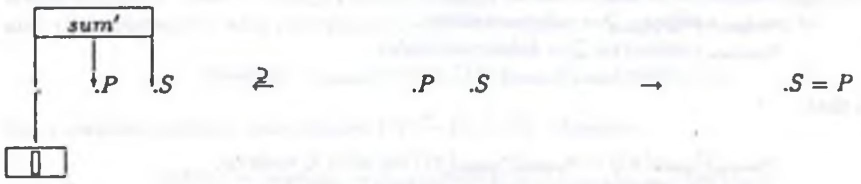
$sum(Y,S) \Rightarrow sum'(Y,0,S)$

Figure 2.1



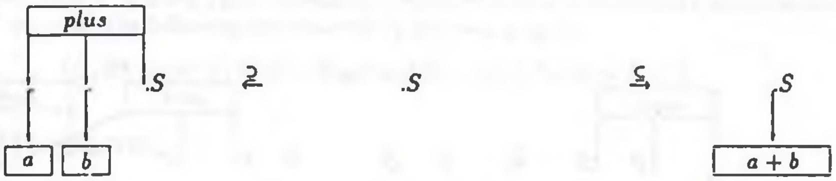
$sum'([X|Y],P,S) \Rightarrow plus(X,P,Q), sum'(Y,Q,S)$

Figure 2.2



$sum'([],P,S) \Rightarrow S := P$

Figure 2.3



$plus(a, b, S) \Rightarrow S := a + b$

Figure 2.4

Applications of usual productions to Ω-graphs can be defined following the standard algebraic approach originated in [EPS 73]. An application of a parametrized production p can be defined as an application of the usual production pr_p in which all occurrences of each parameter are instantiated in the same way.

Let A be a fixed Ω_0-algebra, equipped possibly with some relations whose symbols belong to the language Λ.

By a *rewriting step* (or a *direct derivation*) over A via a parametrized production $p = (L \xleftarrow{l} K \xrightarrow{r} R, m, n)$, we mean a pair $\sigma = (p, i)$ which consists of p and of a diagram i as in figure 2.5 in the subcategory C of Ω*_graphs* such that (1) and (2) for natural pushouts, the Ω_0-reducts of G, D, H coincide with A, $b_{colours}$ and $c_{colours}$ are identities, and, for each sort s, we have:

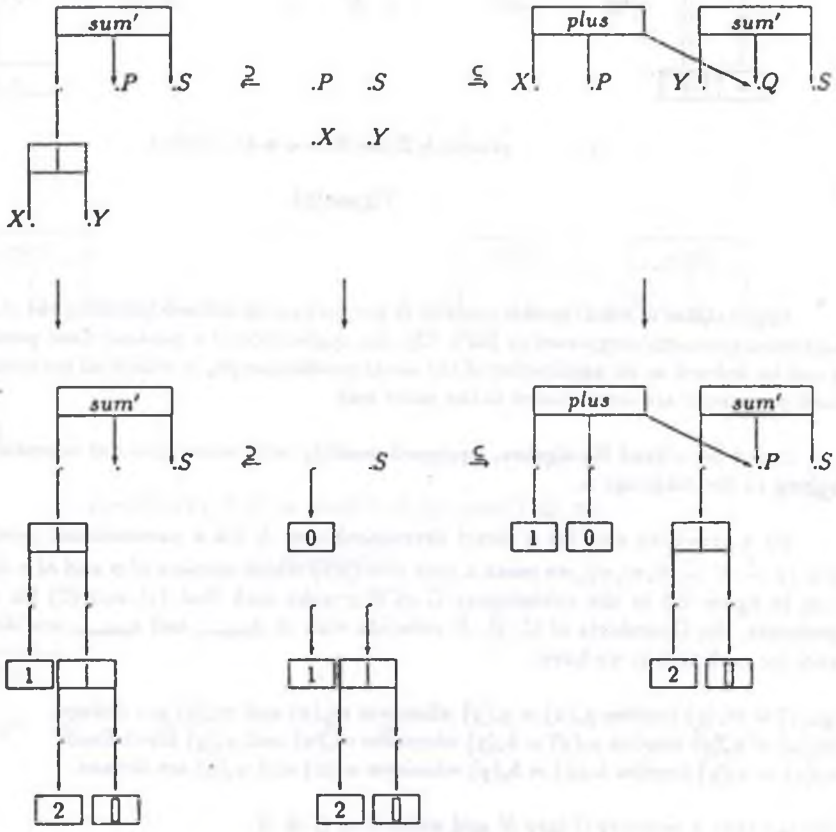
$m_s(x) = m_s(y)$ implies $g_s(x) = g_s(y)$ whenever $m_s(x)$ and $m_s(y)$ are defined,
$m_s(x) = n_s(y)$ implies $g_s(x) = h_s(y)$ whenever $m_s(x)$ and $n_s(y)$ are defined,
$n_s(x) = n_s(y)$ implies $h_s(x) = h_s(y)$ whenever $n_s(x)$ and $n_s(y)$ are defined.

We say that σ *rewrites* G into H and write it as $G \overset{\sigma}{\Rightarrow} H$.

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "h"] \\
G & D \arrow[l, "b"] \arrow[r, "c"'] & H
\end{tikzcd}$$

Figure 2.5

2.6. Example. The replacement of $sum'([1|2], 0, S)$ by $plus(1, 0, P)$ and $sum'([2], P, S)$ can be represented as a rewriting step as in figure 2.6. □



$sum'([1|2],0,S) \Rightarrow plus(1,0,P), sum'([2],P,S)$

Figure 2.6

The concept of a direct derivation can be easily generalized.

Given a set Π of parametrized productions, by a *derivation* over A via productions from Π we mean a finite sequence $\sigma = (G_0 \Rrightarrow G_1 \Rrightarrow ... \Rrightarrow G_i)$ of rewriting steps over A via productions from Π. Given such a sequence σ, we say that it *rewrites* G_i from G_0, write $G_0 \Rrightarrow^* G_i$, and denote G_0 and G_i by $\partial_0(\sigma)$ and $\partial_1(\sigma)$, respectively. By $Der_{C,A,\Pi}$ we denote the set of derivations of this kind with $G_0, G_1, ..., G_i$ having finite sets of nodes and edges.

Given a set $X \subseteq Der_{C,\mathcal{A},\Pi}$ of derivations, by the *relation of derivability* via derivations from X we mean the following relation $rel(X)$ between graphs:

$$(G,H) \in rel(X) \text{ iff } G = \partial_0(\sigma) \text{ and } H = \partial_1(\sigma) \text{ for some } \sigma \in X.$$

3 Programs

We are interested in rewriting graphs according to some programs.

Intuitively, a program p we have in mind is a description, possibly with some parameters, of an algorithm of rewriting graphs by applying productions. In particular, it describes how a given graph G and a given, possibly partial, valuation v of variables in this graph, which is defined for parameters of p, are transformed into subsequent graphs and valuations of variables until reaching a final result.

In order to facilitate a sort of busy waiting of processes as mentioned in section 1, we admit a recursion such that programs may call themselves without executing any real action (an unguarded recursion). Theoretically it leads to infinite idle loops, but in practice such loops do not happen due to a sort of fairness which is usually ensured.

Programs are defined presupposing a set *programidentifiers* of program identifiers, each identifier with an arity which specifies a number of node-, edge-, and colour-parameters. They are given by *program expressions* p, q, r,... which are of the following kinds:

(1) A constant nil. This program expression represents doing nothing.

(2) A parametrized production p. This program expression represents a possible rewriting step $\sigma = (p,i)$ as in figure 2.5 which transforms a graph G and a valuation v of parameters of p in G into a graph H and a valuation w of parameters of p in H, where $v_s(m_s(x)) = g_s(x)$ and $w_s(n_s(y)) = h_s(y)$ for each sort s and all x,y of this sort.

(3) A result $p\gamma$ of a substitution γ of new variables for parameters in a program expression p. This program expression represents an activity which transforms a graph G and a valuation v of parameters of $p\gamma$ in the way in which the activity represented by p transforms G and the valuation $\gamma \circ v$, i.e. the superposition of γ and v.

(4) A conditional if α then p else q, where α is a formula in the language Λ and p, q are program expressions. This program expression represents the choice and execution of p or q depending on the satisfaction of α for the given graph G and the given valuation v of free variables of α and parameters of p and q.

(5) A sequential composition $p;q$ of program expressions p and q. This program expression represents an execution of p followed by an execution of q.

(6) A parallel composition $p \parallel q$ of program expressions p and q. This program expression represents a parallel execution of p and q which can be viewed as an arbitrary interleaving of actions of p and q.

(7) An indeterministic sum $p+q$ of program expressions p and q. This program expression represents an indeterministic choice and execution of p or q.

(8) An indeterministic sum $\Sigma x\ p$, where x is a variable and p is a program expression. This program expression represents an activity which transforms a graph G and a valuation v of parameters of p in G in the way in which the activity represented by p transforms G and a valuation v' obtained from v by an indeterministic choice of a suitable value of x. If there is no such a value for x then the represented activity reduces to doing nothing.

(9) A defined program expression

$$\varphi_k(y_{k1}, y_{k2}, ...) \text{ where } (\varphi_1(y_{11}, y_{12}, ...) = \psi_1, ..., \varphi_n(y_{n1}, y_{n2}, ...) = \psi_n),$$

where $\varphi_1, ..., \varphi_n$ are program identifiers and $y_{11}, y_{12}, ..., y_{n1}, y_{n2}, ...$ are parameters as specified by the respective arities and each ψ_i is a program expression which may contain expressions of the form $\varphi_1(z_{i11}, z_{i12}, ...), ..., \varphi_n(z_{in1}, z_{in2}, ...)$ and is such that all $z_{i11}, z_{i12}, ..., z_{in1}, z_{in2}, ...$ and other parameters of ψ_i occur among $y_{i1}, y_{i2}, ...$ This program expression represents an activity whose execution for $y_{k1}, y_{k2}, ...$ is defined by ψ_k. As in ψ_k there may occur $\varphi_1(z_{k11}, z_{k12}, ...), ..., \varphi_n(z_{kn1}, z_{kn2}, ...)$, one has to define the respective activities by the equations $\varphi_1(y_{11}, y_{12}, ...) = \psi_1, ..., \varphi_n(y_{n1}, y_{n2}, ...) = \psi_n$, and consider an occurrence of $\varphi_j(z_{ij1}, z_{ij2}, ...)$ in ψ_i as a call of the respective ψ_j. In particular, each program expression of the form

φ where ($\varphi =$ if α then $p; \varphi$ else nil)

is equivalent to the standard iteration construct while α do p.

(10) An atomic program expression atom p, where p is a program expression. This program expression represents the activity of successfully executing p as one indivisible step.

Thus we have the following syntax of program expressions:

$$\begin{aligned} p ::= \ & \text{nil} \\ & | < \textit{parametrized production} > \\ & | p\gamma \\ & | \text{if } \alpha \text{ then } p \text{ else } q \\ & | p; q \\ & | p \parallel q \\ & | p + q \\ & | \Sigma x\ p \\ & | \varphi_k(y_{k1}, y_{k2}, ...) \text{ where } (\varphi_1 = \psi_1, ..., \varphi_n = \psi_n) \\ & | \text{atom } p \end{aligned}$$

To each program expression p there correspond a set $FP(p)$ of node-, edge-, and colour-variables called *free parameters* of p, which can be defined as follows:

(1) If p is a parametrized production then $FP(p)$ is the set of parameters of p.

(2) $FP(\text{nil}) = \emptyset$.

(3) $FP(p\ \gamma) = \gamma(FP(p))$.

(4) $FP(\text{if } \alpha \text{ then } p \text{ else } q)$ is the union of $FP(p) \cup FP(q)$ and the set of free variables of α.

(5) $FP(p; q) = FP(p \parallel q) = FP(p + q) = FP(p) \cup FP(q)$.

(6) $FP(\Sigma x\, p) = FP(p) - \{x\}$.

(7) $FP(\varphi_k(y_{k1}, y_{k2}, \ldots) \text{ where } (\varphi_1 = \psi_1, \ldots, \varphi_n = \psi_n)) = \{y_{k1}, y_{k2}, \ldots\}$.

(8) $FP(\text{atom } p) = FP(p)$.

3.1. Example. A program of computing the sum of elements of a list of integers as in 1.1 can be defined as follows:

```
$SUM(\xi,\zeta)$       where
($SUM(\xi,\zeta)$      =                        replace $sum(\xi,\zeta)$ by $sum'(\xi,0,\zeta)$;
                                                $SUM'(\xi,0,\zeta)$,
$SUM'(\xi,\eta,\zeta)$ =    $\Sigma X\ \Sigma Y$   if    $\xi$ represents $[X|Y]$
                                          then  $\Sigma\varrho$ (replace $sum'(\xi,\eta,\zeta)$
                                                by $plus(X,\eta,\varrho)$ and $sum'(Y,\varrho,\zeta)$;
                                                $(PLUS(X,\eta,\varrho) \parallel SUM'(Y,\varrho,\zeta))$)
                                          else  if $\xi$ represents $[]$
                                                then replace $sum'(\xi,\eta,\zeta)$ by $\zeta = 0$
                                                else $SUM'(\xi,\eta,\zeta)$),
$PLUS(\xi,\eta,\zeta)$ =    $\Sigma x\ \Sigma y$   if    $\xi$ represents $x$ and $\eta$ represents $y$
                                          then  replace $plus(\xi,\eta,\zeta)$ by $\zeta = x + y$
                                          else  $PLUS(\xi,\eta,\zeta)$
```

The instructions of replacement which occur in this program are parametrized productions as in 2.5. Formulas occurring in the program are abbreviations of formulas of the first order language Λ. For instance, the formula ξ *represents* $[X|Y]$ is an abbreviation of a formula which states the existence of an edge with the colour $|$, the source node ξ, and the target nodes X and Y.

4 Semantics

The way in which pairs consisting of graphs and valuations of variables in these graphs are transformed by executing programs can be described in the form of a labelled transition system which consists of a universe *conf* of configurations and a transition relation $\rightarrow$. When considered together with suitable fairness assumptions, such a system allows to define all practically possible program computations.

The universe *conf* consists of configurations of the form $c = (p, G, v)$, where p is a program expression, G is a graph, and v is a (possibly partial) valuation of variables in G such that the defined values of node-, edge-, and colour variables are respectively nodes, edges, and colours of G.

The transition relation $\rightarrow$ consists of transitions of the form $t = (c, \alpha, c')$, where c, c' are configurations and α is either an invisible action τ which does not change the configuration (that is such that $c = c'$) or an action of applying a production or executing an atomic program. Denoting by *actions* the set of possible actions we can define this relation as the smallest $\to \subseteq conf \times actions \times conf$ satisfying the following conditions:

(1) $(p, G, v) \xrightarrow{p} (\mathrm{nil}, H, w)$ for each parametrized production p and G, H, v, w such that there is a rewriting step $\sigma = (p, i)$ with the diagram i as in figure 2.5 and $g_s(x) = v(m_s(x))$, $h_s(y) = w(m_s(y)$ for each sort s and all x, y such that $m_s(x)$ and $n_s(y)$ are defined.

(2) $(\mathrm{atom}\ p, G, v) \xrightarrow{\mathrm{atom}\ p} (\mathrm{nil}, G', v')$ for each program p and G, G', v, v' such that there exists a finite sequence of transitions of the form

$$(p, G, v) \xrightarrow{a_1} \dots \xrightarrow{a_n} (\mathrm{nil}, G', v').$$

(3) If $(p, G, \gamma \circ v) \xrightarrow{a} (p', G', \gamma \circ v')$ then $(p\gamma, G, v) \xrightarrow{a\gamma} (p'\gamma, G', v')$.

(4) If $(p, G, v) \xrightarrow{a} (p', G', v')$ and the formula f is satisfied for the valuation v then $(\mathrm{if}\ f\ \mathrm{then}\ p\ \mathrm{else}\ q, G, v) \xrightarrow{a} (p', G', v')$.

(5) If $(q, G, v) \xrightarrow{a} (q', G', v')$ and the formula f is not satisfied for the valuation v then $(\mathrm{if}\ f\ \mathrm{then}\ p\ \mathrm{else}\ q, G, v) \xrightarrow{a} (q', G', v')$.

(6) If $(p, G, v) \xrightarrow{a} (\mathrm{nil}, G', v')$ then $(p; q, G, v) \xrightarrow{a} (q, G', v')$.

(7) If $(p, G, v) \xrightarrow{a} (p', G', v')$ then $(p; q, G, v) \xrightarrow{a} (p'; q, G', v')$.

(8) If $(p, G, v) \xrightarrow{a} (p', G', v')$ then $(p \parallel q, G, v) \xrightarrow{a} (p' \parallel q, G', v')$.

(9) If $(q, G, v) \xrightarrow{a} (q', G', v')$ then $(p \parallel q, G, v) \xrightarrow{a} (p \parallel q', G', v')$.

(10) If $(p, G, v) \xrightarrow{a} (p', G', v')$ then $(p + q, G, v) \xrightarrow{a} (p', G', v')$.

(11) If $(q, G, v) \xrightarrow{a} (q', G', v')$ then $(p + q, G, v) \xrightarrow{a} (q', G', v')$

(12) $(\Sigma x\ p, G, v) \xrightarrow{a} (\mathrm{nil}, G, v)$ for each program expression p and all G, v such that $v(x)$ is not defined.

(13) If $(p, G, v) \xrightarrow{a} (p', G', v')$ for some v such that $v(x)$ is defined then $(\Sigma x\ p, G, w) \xrightarrow{a} (p', G', v')$ for $w = v - \{(x, v(x))\}$.

(14)

$$(\varphi_k(y_{k1}, y_{k2}, \dots)\ \mathrm{where}\ (\varphi_1 = \psi_1, \dots, \varphi_n = \psi_n), G, v) \xrightarrow{\tau}$$

$$(\varphi_j(y_{j1}, y_{j2}, \dots)\ \mathrm{where}\ (\varphi_1 = \psi_1, \dots, \varphi_n = \psi_n), G, v)$$

whenever $\varphi_j(y_{j1}, y_{j2}, \dots)$ is the first program expression in ψ'_k which is obtained by substituting in ψ_k the program expression

$$\varphi_i(y_{i1}, y_{i2}, \dots)\ \mathrm{where}\ (\varphi_1 = \psi_1, \dots, \varphi_n = \psi_n)$$

for each occurrence of $\varphi_i(y_{i1}, y_{i2}, \dots)$.

(15) If $(\psi'_k, G, v) \xrightarrow{a} (p', G', v')$ for ψ'_k obtained by substituting in ψ_k the program expression

$$\varphi_i(y_{i1}, y_{i2}, \ldots) \text{ where } (\varphi_1 = \psi_1, \ldots, \varphi_n = \psi_n)$$

for each occurrence of $\varphi_i(y_{i1}, y_{i2}, \ldots)$, then

$$(\varphi_k(y_{k1}, y_{k2}, \ldots) \text{ where } (\varphi_1 = \psi_1, \ldots, \varphi_n = \psi_n), G, v) \xrightarrow{a} (p', G', v').$$

The transition relation allows to define computations of programs and the respective relations between data and results.

Formally, a *computation* of a program p is defined as a sequence of transitions of the form

$$u = ((p, G, v) \xrightarrow{a_1} (p_1, G_1, v_1) \xrightarrow{a_2} \ldots),$$

where u is either countably infinite or it has a terminal configuration of the form (nil, G', v'). The pairs (G, v) and (G', v') (if u terminates) are called respectively the *data* and the *result* of u.

In reality it is usually ensured that only such computations are possible which enjoy a fairness property. Consequently, in the sequel by a computation of a program p we shall mean only such a computation u of p which is *fair* in the sense that there is no transition of p which is permanently possible starting from a configuration c of u and does not occur among the transitions of u which follow c.

The *resulting relation* of a program p, $res(p)$, is defined as the relation which holds between the data and the results of finite computations of p: $(G, v)\ res(p)\ (G', v')$ iff there exists a computation of p of the form: $(p, G, v) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (\mathrm{nil}, G', v')$.

5 Resulting relations of programs

The possibility of using computations of programs to define the respective resulting relations suggests that an input-output semantics could be defined directly in terms of resulting relations. Unfortunately, such a definition is impossible because of the lack of compositionality of the correspondence between programs and their resulting relations. There are two sources of this situation. One of them is the parallel composition, where the resulting relation of a program obtained by composing given programs depends not only on the resulting relations of these programs, but also on the resulting relations of smaller program components. For example, for a program $p \parallel q$, where $p = p_1; p_2$, $q = q_1; q_2$, and p_1, p_2, q_1, q_2 are parametrized productions the resulting relation is

$$res(p \parallel q) = res(p_1; p_2; q_1; q_2) \cup res(p_1; q_1; p_2; q_2) \cup res(p_1; q_1; q_2; p_2)$$

$$\cup res(q_1; p_1; p_2; q_2) \cup res(q_1; p_1; q_2; p_2) \cup res(q_1; q_2; p_1; p_2)$$

and it cannot be expressed in terms of $res(p) = res(p_1; p_2)$ and $res(q) = res(q_1; q_2)$. Another source of the mentioned situation is recursion, where the resulting relation of a program defined by a mutual recursion need not be definable by the corresponding system of equations in the algebra of relations. For example, for the program $p = X$ where $(X =$

nil;X), each finite graph G, the resulting relation is *identity* and it is different from the least relation R satisfying $R = identity \circ R$ since such the latter is the empty relation $\emptyset$.

The lack of compositionality w. r. to the parallel composition and recursion shows that there is no chance for a direct input-output semantics of the considered programs. In particular, an operational semantics as presented cannot be avoided even if we are interested only in the resulting relations of programs. On the contrary, in the situation in which such relations cannot be derived from programs directly, a semantics of this kind becomes an important tool of defining the resulting relations.

The reasonning about resulting relations of programs can be supported by a number of properties of such relations.

5.1. Proposition. The following properties hold true for the resulting relations of programs:

(1) $res(\text{skip}) = identity$.

(2) If p is a parametrized production then $(G,v)res(p)(H,w)$ iff there is a rewriting step $\sigma = (p,i)$ with the diagram i as in figure 2.5 and $g_s(x) = v(m_s(x))$, $h_s(y) = w(m_s(y)$ for each sort s and all x,y such that $m_s(x)$ and $n_s(y)$ are defined.

(3) $(G,v)res(p\gamma)(G',v')$ iff $(G,\gamma \circ v)res(p)(G',\gamma \circ v')$.

(4) $(G,v)res(\text{if } f \text{ then } p \text{ else } q)(G',v')$ iff either f is satisfied for G and v and $(G,v)res(p)(G',v')$ or f is not satisfied and $(G,v)res(q)(G',v')$.

(5) $res(p;q) = res(p) \circ res(q)$.

(6) $res(p+q) = res(p) \cup res(q)$.

(7) $res(\Sigma x\ p) = \bigcup(res(p) : i \in I)$, where I is the set of possible values of valuations for x. $\Box$

A proof is of this proposition is straightforward.

The class of resulting relations of programs is rich enough to represent the usual derivability relation.

Taking into account the definition of the resulting relation of a program and the definition of the semantics of programs, we obtain the following realization of the relation of derivability.

5.2. Proposition. Let $\Pi = \{p_1, ..., p_m\}$ be a finite set of parametrized productions. There is a program p such that H is derivable from G via productions from Π iff $(G,v)res(p)(H,v')$ for some v and v'. In particular, we may define such a program as

$$p = X \text{ where } (X = q; X + \text{nil})$$

where

$$q = (\Sigma x_1)...(\Sigma x_m)(p_1 + ... + p_m)$$

and $\{x_1, ..., x_m\}$ is the set of parameters of productions $p_1, ..., p_m$. $\Box$

6 Recapitulation

We have presented conceptual means for programming concurrent processes of rewriting graphs by applying productions. These means are flexible enough to cover the usual rewriting. However, their possibilities go much beyond such particular cases due to the powerful mechanisms of parameters, recursion, concurrency, and operating on colours.

The presented formalism is brought to the form of (a kernel of) a programming language with a precise syntax and semantics.

The semantics defines computations of each program p. These computations represent the possible ways of transforming a given graph G and a given valuation of variables in this graph, and thus they determine a resulting relation $res(p)$ of p. The possibility of determining such a relation is important since there is no simpler way of computing it directly from the program.

The semantics presented in the paper represents concurrency as an arbitrary interleaving of actions. Nevertheless, it contains implicitly all the information about the existing concurrency. Moreover, there seems to be a chance of refining it to a form in which concurrency would be reflected in a more explicit way as in [MR 92]. How to do it is however a problem which we leave open in the present paper.

Acknowledgements. The authors are grateful to the two anonymous referees of an early version of this paper for their comments and suggestions.

References

[CER 79] Claus, V., Ehrig, H., Rozenberg, G., (Eds.) *Proceedings of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, Springer LNCS 73, 1979.

[CMREL 91] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Löwe, M., *Logic Programming and Graph Grammars*, in [EKR 91], 221-237.

[CRP 91] Corradini, A., Rossi, F., Parisi-Presicce, F., *Logic Programming as Hypergraph Rewriting*, in the Proceedings of CAAP'91, Springer LNCS 493, 1991. 275-295.

[EKMRW 82] Ehrig, H., Kreowski, H.-J., Maggiolo-Schettini, A., Rosen, B.K., Winkowski, J., *Transformations of Structures: An Algebraic Approach*, Math. Systems Theory 14 (1981) 305-334.

[EKR 91] Ehrig, H., Kreowski, H.-J., Rozenberg, G., (Eds.) *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer LNCS 532, 1991.

[ENRR 87] Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A., (Eds.) *Proceedings of the 3rd Workshop on Graph-Grammars and Their Application to Computer Science*, Springer LNCS 291, 1987.

[JaRo 90] Jansssens, D., Rozenberg, G., *Structured Transformations and Computation Graphs for Actor Grammars*, in [EKR 91], 446-460.

[MR 92] Montanari, U., Rossi, F., *Graph Grammars as Context-Dependent Rewriting Systems: A Partial Ordering Semantics*, in Springer LNCS 581, 1992, 232-247.

[MW 83] Maggiolo-Schettini, A., Winkowski, J., *Towards a Programming Language for Manipulating Relational Data Bases*, in: Bjorner, D., (Ed.), Formal Description of Programming Concepts II, North-Holland, 1983, 265-278.

[MW 91] Maggiolo-Schettini, A., Winkowski, J., *Programmed Derivations of Relational Structures*, in [EKR 91], 582-598.

[MW 92] Maggiolo-Schettini, A., Winkowski, J., *A Programming Language for Deriving Hypergraphs*, in Springer LNCS 581, 1992, 221-231.

[Plo 81] Plotkin, G., *A Structural Approach to Operational Semantics*, Technical Report, Computer Sc. Dept., Aarhus Univ., Denmark, DAIMI-FN-19, 1981.

[P 91] Plump, D., *Graph-Reducible Term Rewriting Systems*, in [EKR 91], 622-636.

[Sh 89] Shapiro, E., *The Family of Concurrent Logic Programming Languages*, ACM Computing Surveys, 21, 1989, 413-510.