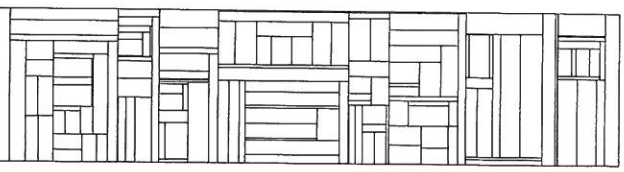
ISSN 0105-8517

Graphics and Their Grammars

L. Hess
B.H. Mayoh

DAIMI PB – 223
March 1987

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 83 55 *Telex: 64767 aausci dk*



Graphics and Their Grammars

L. Hess, Instituto Militar de Engenharia, Brasil
B.H. Mayoh, Aarhus University, Denmark

Abstract
Graphics are graphs with attributes at their vertices. Graphic grammars are natural extensions of graph and attribute grammars with rules that are attributed extensions of the "pushout" productions of graph grammars. The theory of graphic grammars is presented and various programming implementations will be discussed. Many motivating examples will be given, including

- the development of biological organisms
- the "semantic net" representation of expert system knowledge.

Introduction

Graphics have been defined as graphs whose vertices have attributed values. When the assignment of values to vertices is allowed, graph models become even more useful than they already are. In figure 1 we indicate some possible applications of graphics and their grammars.

Once one accepts that graphics are a useful generalisation of graphs, it is natural to look for a definition of a graphic grammar that generalises both graph and attribute grammars appropriately. The two definitions of graphic grammars in the literature have chosen to generalise a form of graph rewriting that may be useful in certain applications but seems arbitrary even so. In Section 1 we base our definition of graphic grammars on graphic morphisms and pushouts. This form of graph rewriting seems to be the natural generalisation of string rewriting in attribute grammars. A graphic is a new representational entity, because its attributes can be Σ-algebra terms; graphic grammars can benefit from the results of the algebraic approach to graph grammars and Σ-algebras. In Section 2 we look at the use of graphics for biological organisms and the "data base and expert system" problem of representing real world knowledge. In Section 3 we describe various ways of "implementing" graphic grammars on a computer.

Image analysis & Generation:	Aircraft identification (Fu).
	Stochastic terrain models (HM).
	Scene Analysis (Bu).
Diagram analysis & Generation:	Program Structure (G).
	Structured pictures (HM).
	Interactive picture editor (HM).
	Program Animation (HM).
	Circuit diagrams (G).
	Art (N).
	Window manager (HM).
Numerical analysis:	Generating grids for solving DDE's (Ka).
Biological Models:	Morphogenesis (Li).
Knowledge representation:	Conceptual schemes in databases.
	Semantic nets in AI (So).

Figure 1: Applications of graphics.

#1. Graphics and Their Grammars

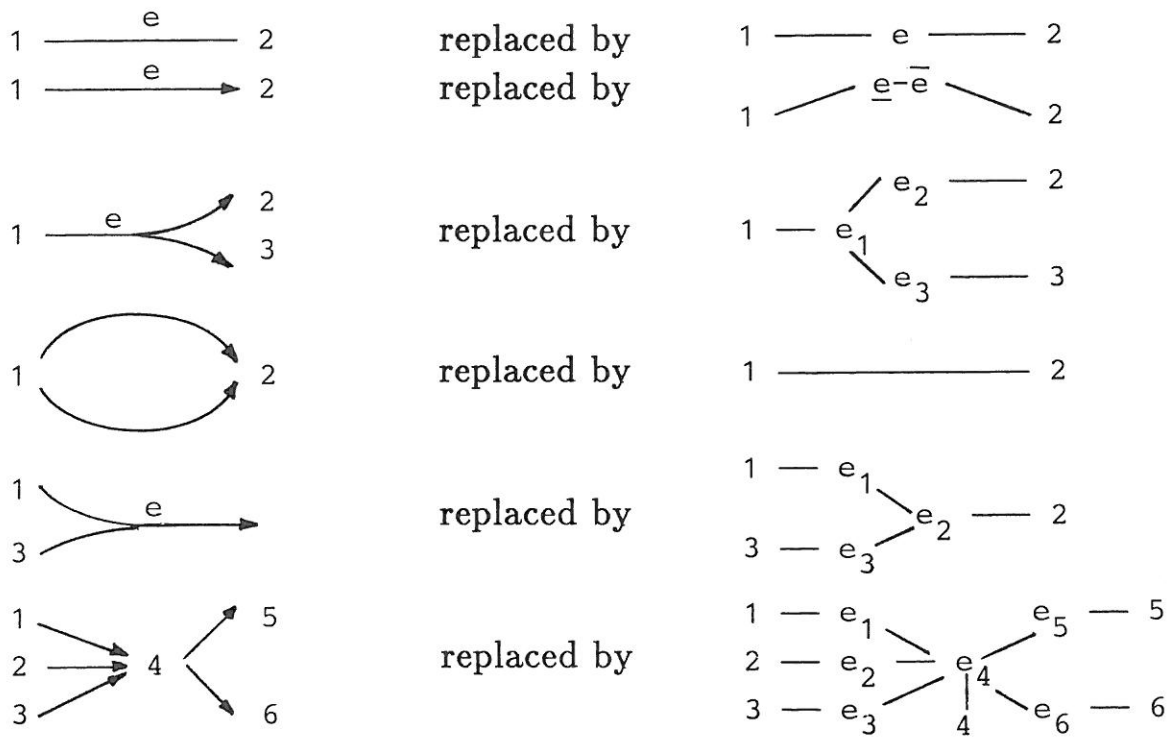
The natural generalisation of a graph is also the natural generalisation of the formal language concept of a word or string: a totally ordered set of vertices labelled by symbols from a finite alphabet. If we take a symmetric relation instead of a total ordering, we get:

Definition

A **graphic** G on an algebra A and label set V is a 4-tuple $G = (K, E, l, f)$ where

K is a finite non-empty set of elements called vertices,
E is a finite set of unordered pairs of vertices called edges,
l is a function from K to V
f is a function from K to A.

We seem to have lost generality by not allowing edge labels or attributes, directed edges, edges with more than two vertices, or multiple edges. This is not so because we can introduce edge nodes as new vertices whose labels and attributes can capture ordering information. As examples of new nodes capturing edge information,



consider the replacements. It is clear that edge replacing rules can be simulated by graphic productions and one can map a replacement graphic back to a graph with directed, labelled, attributed edges.

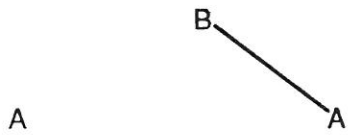
A **graphic morphism** $g : G \to G'$ consists of maps ver: $K \to K'$, edge: $E \to E'$, lab: $V \to V'$, att: $A \to A'$ such that

$$edge(v_1, v_2) = (ver(v_1), ver(v_2))$$
$$l'(ver(v)) = lab(l(v))$$
$$f'(ver(v)) = att(f(v)).$$

We say that G is a **subgraphic** of G' ($g : G \to G'$ is an *embedding*) when "ver" is an injection and "lab" is identity.

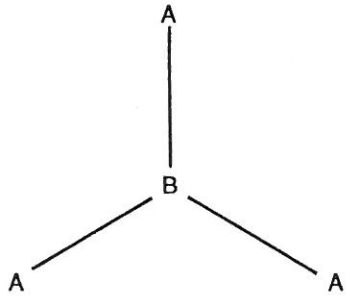
Comment:
With this definition of morphism, graphics form a category when the algebras form a category. We could specify the graphic category as a subcategory of STRUCT (EKMRW). Graphic morphisms are natural combinations of graph morphisms and morphisms of algebras. The definition of subgraphic is a natural generalisation of the definition of subgraph. We will be particularly interested in the case when the category of algebras is the much studied Σ-algebra for some signature Σ. In this case not only are graphics elements of the Cartesian products $(A \times V)^K$ but these Cartesian products are themselves Σ-algebras.

Example:
The graphic on the algebra $(R^2, +, \times, \sqrt{}, \sin, \cos, 0, \ldots)$



$$
\begin{array}{ll}
K = \{1,2,4\} & E = \{(2,4)\} \\
l(1) = A & f(1) =< x, y > \\
l(2) = A & f(2) =< x + a\cos\phi, y + a\sin\phi > \\
l(4) = B & f(4) =< x + a\cos(\phi + 30)/\sqrt{3}, \\
 & \quad y + a\sin(\phi + 30)/\sqrt{3} >
\end{array}
$$

is a subgraphic of the graphic



$$
\begin{array}{ll}
K' = \{1,2,3,4\} & E' = \{(1,4),(2,4),(3,4)\} \\
l'(1) = A & f'(1) = (0,0) \\
l'(2) = A & f'(2) = (a,0) \\
l'(3) = A & f'(3) = (a/2, a\sqrt{3}/2) \\
l'(4) = B & f'(4) = (a/2, a\sqrt{3}/6)
\end{array}
$$

on the same algebra because we have the graphic morphism

$$ver(v) = v \quad lab(l) = l \quad att(u,v) = (Su, Sv)$$

where S is given by subsituting 0 for x, y and ϕ. The trigonometric expressions in the subgraphic give the projections of an edge to the x- and y-axes.

In this section all graphics will have two spatial attributes which are given by the representation of the graphic as a diagram. Notice that all graphics have the empty graphic 0 as a subgraphic. The empty graphic is useful for removing vertices.

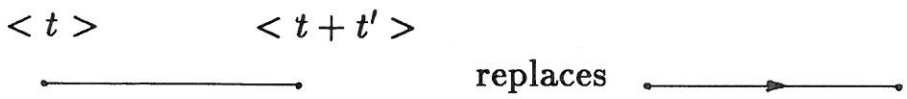
Definition

A graphic sequential rewriting system $\Gamma = (V, \Sigma, I, P)$ consists of

V, a finite non-empty set of symbols
Σ, a signature
I, the start graphic on the term algebra $T(\Sigma \cup V)$
P, a non-empty set of productions.

A **production** $p = B_1 \xleftarrow{B_1} K \xrightarrow{B_2} B_2$ is a pair of graphic morphisms connecting three graphics whose vertices have attribute values in $T(\Sigma \cup V)$ – terms formed from variables V and the operations in the signature Σ.

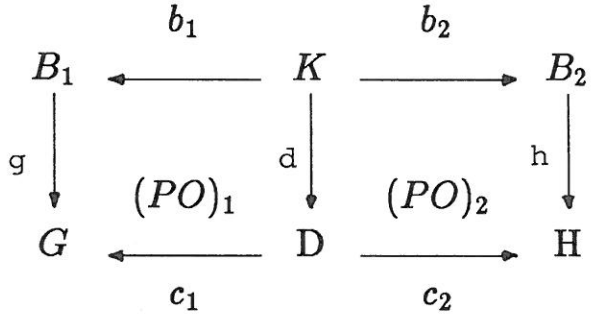
Comment:
One might well disagree with our requirement that the attribute values in B_1, K, B_2 are given by terms in $T(\Sigma \cup V)$. This requirement is connected with our choice of morphism rewriting; it ensures that graph pushouts coincide with graphic pushouts. The requirement also allows the attribute values to carry edge information; if a directed graph is partially ordered, the edge ordering can be captured by



Not only "later", "above" and "to the right of" are examples of this, but also many orderings in database and AI semantic nets.

Definition

A direct derivation $G \underset{p}{\Rightarrow} H$ is defined by giving a production, a context graphic D, a graphic morphism $K \rightarrow D$, and two pushouts $(PO)_1$ and $(PO)_2$.



We say that the graphic H is derived from G by the production p, based on the morphism $g : B_1 \rightarrow G$.

A graphic language is defined as the set of all graphics derived from the initial graphic by productions in P.

Comment:
As explained in [Eh, p. 11], the pushout of two graph morphisms $f : K \rightarrow B$ and $g : K \rightarrow D$ is given by "gluing together the items $f(k)$ in B and $g(k)$ in D for each item k in K". For labelled graphs we want

$$l_G([x]) = \textbf{if } x \in B \textbf{ then } l_B(x) \textbf{ else } l_D(x)$$

for each vertex $[x]$ in G, the pushout graph of f and g. This is a proper definition if $l_B(f(k)) = l_D(g(k))$ for all k in K (satisfied when f and g are graphic morphisms, and we try to define attributes for vertices in the pushout graphic G by

$$(+) \qquad f_G([x]) = \textbf{if } x \in B \textbf{ then } f_B(x) \textbf{ else } f_D(x)$$

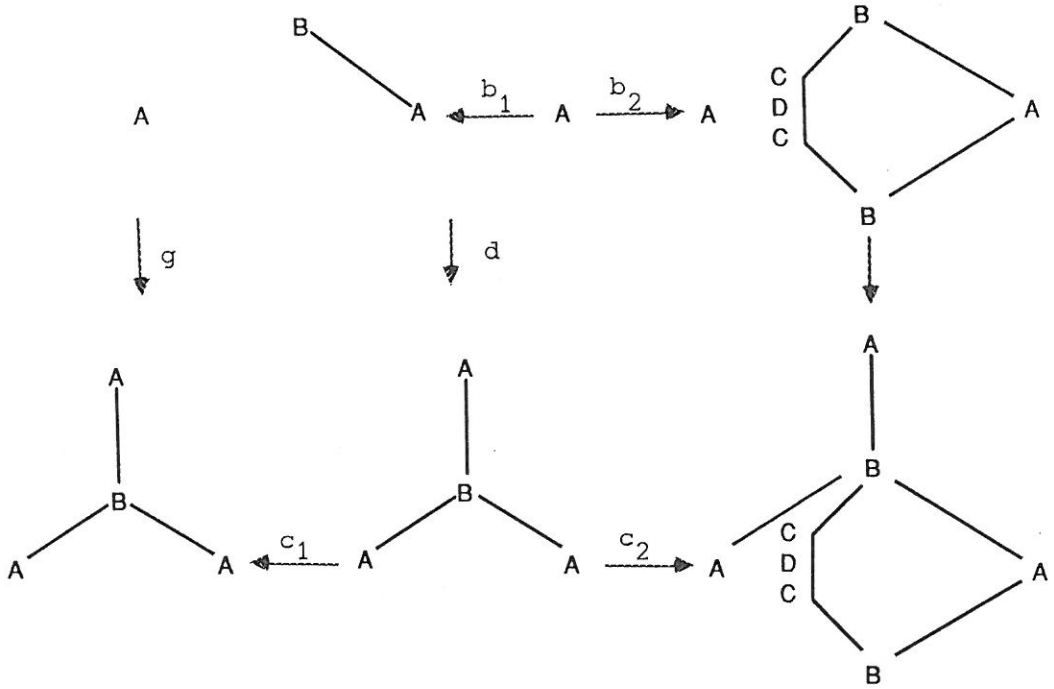
for each vertex $[x]$ in G. This works when $f_B(f(k)) = f_D(g(k))$ for all k in K (satisfied when f and g are attribute preserving). We would like a more general way of defining attribute values in the pushout graphic. If we require f and g to be either attribute preserving or "compatible" substitutions, then we can define $f_G([x])$ by $(+)$ when $[x]$ is disjoint from $f(K) \cup g(K)$

$$f_B(f(K) \cap [x]) \wedge f_D(f(K) \cap [x])$$

when $[x]$ meets $f(K) \cup g(K)$. Category theory tells us that graphics on Σ-algebras always have pushouts because "A has pushouts" implies "A^B has pushouts".

Comment:
The reader is invited to reformulate the above discussion of graphic pushouts in the terminology of the algebraic approach to general structures (EKMRW, Pa).

Example:
Perhaps we should give an example of a direct derivation.



All the morphisms in this diagram are embeddings (sources are subgraphics of sinks); the top two morphisms give a production and the morphism g was described in our previous example. The left square is a pushout when d assigns the coordinates $< 0,0 >$ to the lower right A in its target. The right square is also

a pushout with this assignment of values to attributes. One may well ask what has happened to the parameters a and ϕ in the previous example. The answer is that these parameters fix the morphisms, b_1 and b_2 – they parametrise a family of productions.

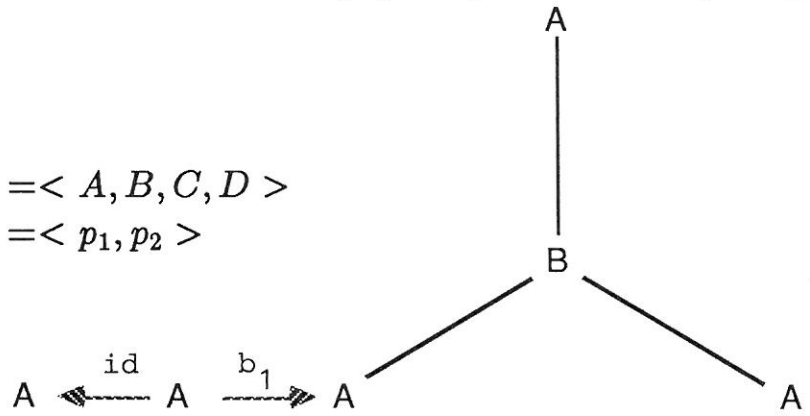
Example:

A graphic grammar to describe a "wall paper" pattern with planegroup p 31m [Ma]

$G =< V, I, P >$ $\quad V =< A, B, C, D >$

$I = A < 0, 0 >$ $\quad P =< p_1, p_2 >$

p1:



$K = \{1\}$ $\quad K = \{1, 2, 3, 4\}$

$E = \{\}$ $\quad E = \{< 1, 4 >, < 2, 4 >, < 3, 4 >\}$

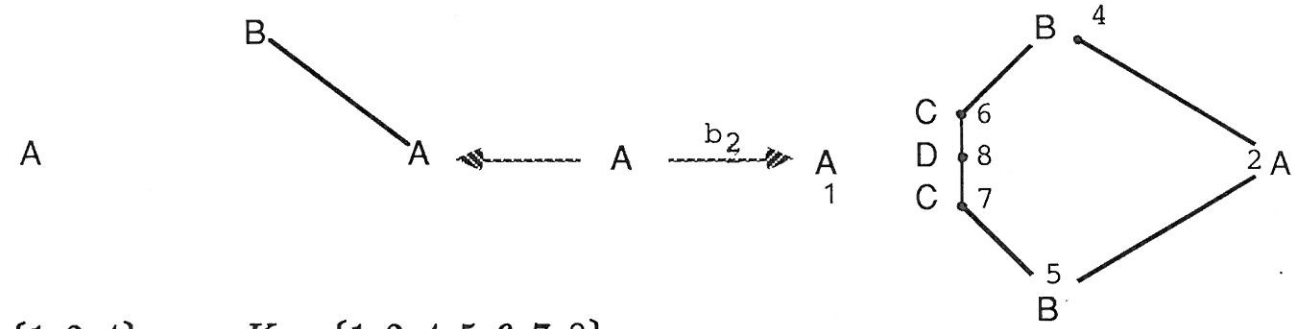
$L(1) = A$ $\quad L(1) = A \;\; L(2) = A \;\; L(3) = A \;\; L(4) = B$

$f(1) =< x, y >$ $\quad f(1) =< x, y > \;\; f(2) =< x + a, y >$

$f(3) =< x + a/2, y + \sqrt{3}a/2 >$

$f(4) =< x + a/2, y + \sqrt{3}a/6 >$

p2:



$K = \{1, 2, 4\}$ $\quad K = \{1, 2, 4, 5, 6, 7, 8\}$

$E = \{2, 4\}$ $\quad E = \{< 2, 4 >, < 4, 6 >, < 6, 8 >, < 8, 7 >, < 7, 3 >, < 5, 2 >\}$

$L(1) = A \;\; L(2) = A \;\; L(4) = B \;\; L(5) = B \;\; L(6) = C \;\; L(7) = C \;\; L(8) = D$

$f(1) = (x, y) \;\; f(2) =< x + a\cos\phi, y + a\sin\phi >$

$f(4) =< x + a\cos(\phi + 30)/\sqrt{3}, y + a\sin(\phi + 30)/\sqrt{3} >$

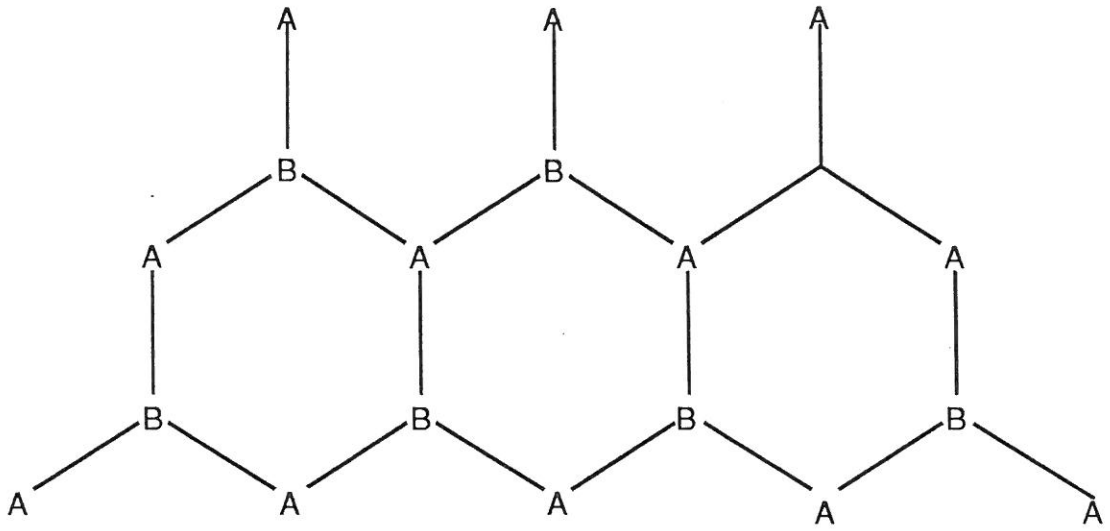
$f(5) =< x + a\cos(\phi - 30)/\sqrt{3}, y + a\sin(\phi - 30)/\sqrt{3} >$

$f(8) =< x + a\cos\phi/5, y + a\sin\phi/5 >$

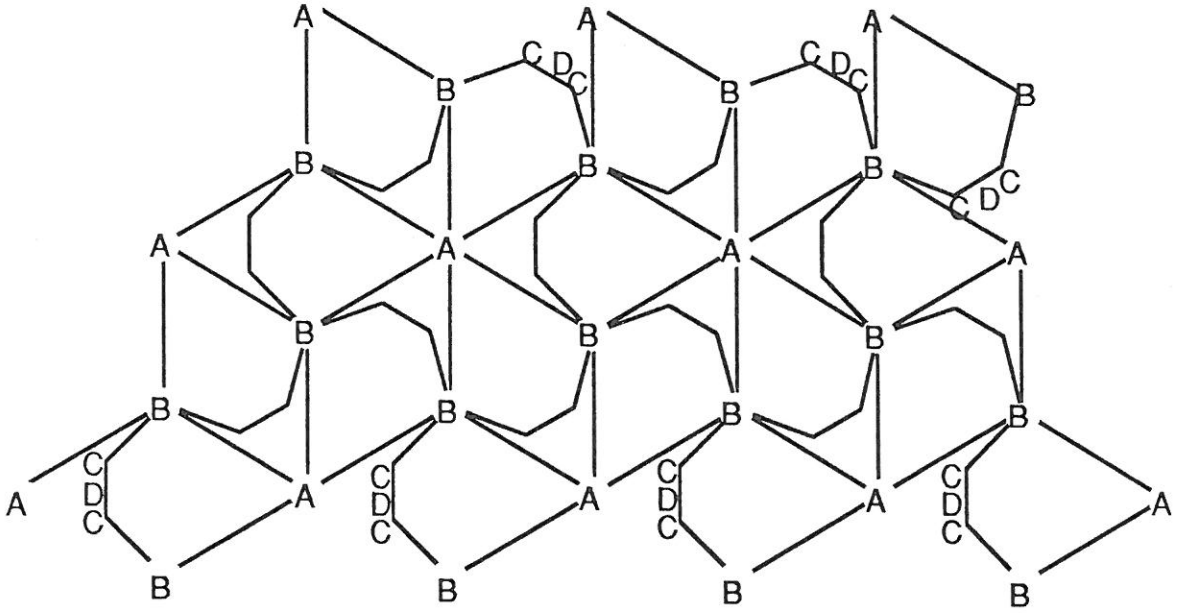
$f(6) =< x + a\cos(\phi)/5 + a\cos(\phi + 90)\sqrt{3}/24, y + a\sin\phi/5 + a\sin(\phi + 90)\sqrt{3}/24 >$

$f(7) =< x + a\cos(\phi)/5 + a\cos(\phi - 90)\sqrt{3}/24, y + a\sin(\phi - 90)\sqrt{3}/24 >$

Repeated use of the production p1 gives the pattern:



Repeated use of the production p2 now gives:



We have described the derivation of a graphic G in a graphic language as if the attributes were evaluated during the derivation. However, there is no objection to first deriving the underlying graph of G in the graph language, corresponding to L, then deriving the attribute values. This corresponds to the usual evaluation method in "string" attribute grammars: first derive the string in the underlying context-free grammar, then solve the attribute equations.

Final remark:

If one has an Σ-morphism μ to any Σ-algebra A from the Σ-algebra of the initial graphic of a grammar, then one gets a transformation of the graphic language by applying μ at every step of every derivation in the language. This captures geometric transformations of pictures and much else.

#2. Reality Models

Are graphics useful models of reality?

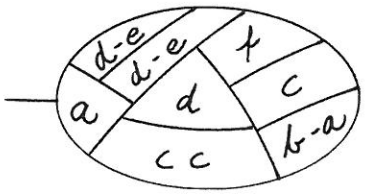
Graph models have become popular, and they become more accurate and convenient when vertices can have attribute values. However, the usefulness of graph grammars is strongly dependent on the kinds of rewriting rules that are allowed. "Push out" rewriting gives a flexibility that single morphism rewriting (identity morphisms from K to B_1, and D to G) and simple node and/or edge replacement rules do not have.

It is well established that the development of biological organisms (Li) can be modelled by graphs and their grammars and Example 1 shows that the extension to graphics gives more natural models. In the development of expert systems semantic nets are a popular way of representing "real world knowledge" (Sowa) and Example 2 shows that graphics are a convenient formalism for semantic nets.

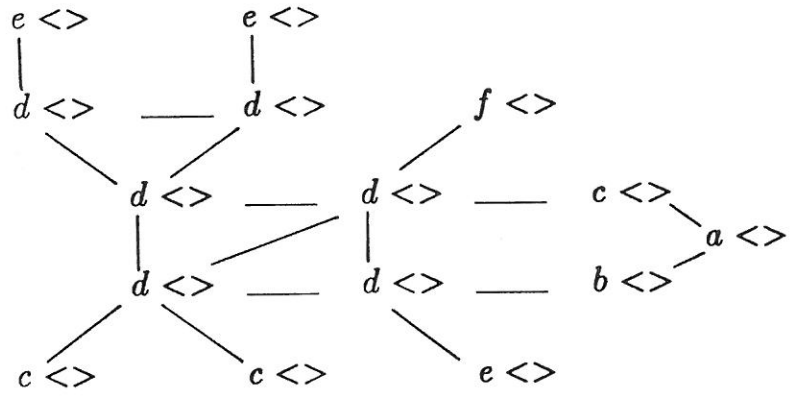
Example 1: "Phascum Cuspidatum"

The grammar for the development of the organism "Phascum Cuspidatum" and the graphic for a particular life stage is shown in

I: a



P: $a \Leftarrow a \Rightarrow b-a$
$b \Leftarrow a \Rightarrow c$
$c \Leftarrow c \Rightarrow d-e$
$d \Leftarrow d \Rightarrow d$
$e \Leftarrow e \Rightarrow f$
$f \Leftarrow f \Rightarrow cc$



As the attributes of the "Phascum Cuspidatum" cells are not used in this graphic grammar, it is no more than a traditional graph grammar. However, this grammar illustrates two points that are valuable in biological modelling.

- Grammars work best when there are only a finite number of cell labels, but the development of an organism is most naturally described in terms of

infinitely many states, and graphic attributes can capture this infinity.

- Grammars usually give an unambiguous result when a rule is applied, but the development of an organism is most naturally described by allowing very flexible applications of rules, and graphic attributes can capture this flexibility.

Example 2: "Semantic Nets"

A grammar for the representation of human relationships and the graphic for a particular family are shown in

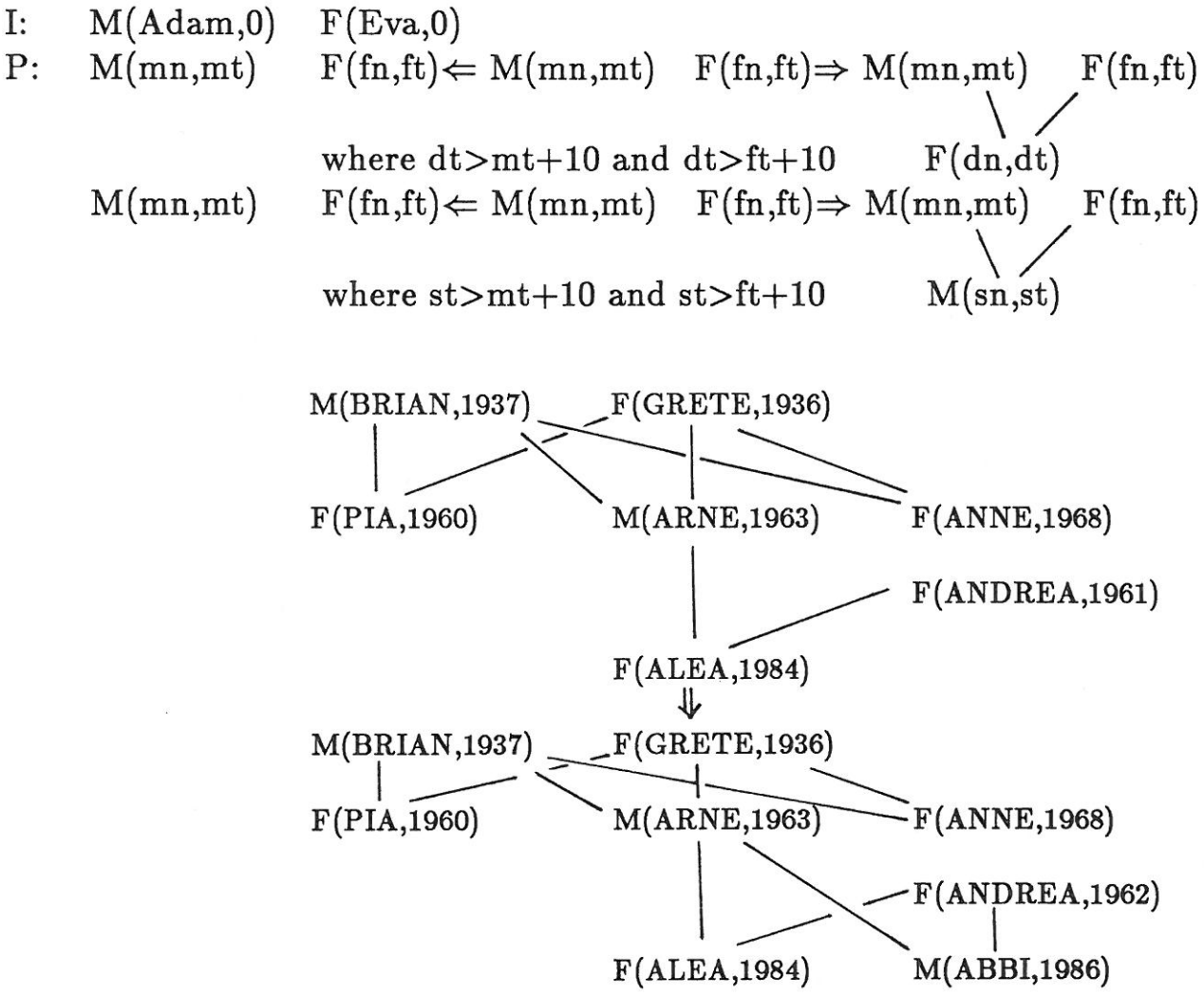


Figure 2.1: The Graphic Grammar Genesis.

The application of the second rule in this example is flexible because the fixing of the parameters – mn=Arne, mt=1963, fn=Andrea, ft=1962 – did not determine the attributes: sn=Abbi and st=1968. The example is instructive because it shows

how "time" attributes allow graphics to capture natural orderings which would be expressed by directed edges in graph models. Other examples would show how graphics can capture other kinds of edge information that expert system builders put into semantic nets. (Remember the discussion in Section 1 on edge information.)

Example 3: Erasing

In (So) Sowa introduced a form of semantic nets called conceptual graphs for the graphic grammar in Example 2, and he described the powerful operations for extracting information and manipulating conceptual graphs. All these operations can be extended to graphics. To convince doubters we show how edges and vertices can be removed ... consider the addition of rules

M(mn,mt)⇐0⇒		– vertex removal		
F(fn,ft)⇐0⇒0		– using the empty graphic 0		
M(mn,mt)-F(fn,ft)	⇐ M(mn,mt)	F(fn,ft)	⇒ M(mn,mt)	F(fn,ft)
M(mn,mt)-M(n,t)	⇐ M(mn,mt)	M(n,t)	⇒ M(mn,mt)	M(n,t)
F(fn,ft)-F(n,t)	⇐ F(fn,ft)	F(n,t)	⇒ F(fn,ft)	F(n,t)

The last three rules give edge removal.

#3. Implementation

Once one accepts that graphics and their grammars are useful models of reality, one wants to implement them on the computer; one wants an *analysis* program to recognise whether a given graphic can be generated by a given grammar, and one wants a *synthesis* program for generating graphics from grammars. The many ways of implementing graphics and their grammars can be grouped into – traditional, concurrent, logical and syntactical – so we shall look at each of these in turn.

#3.1 Traditional implementations

If one wants to implement graphics and their grammars in an imperative language like PASCAL or an object oriented language like SmallTalk, it is natural to represent *graphics as data objects.* It is also natural to represent a grammar rule by a pair of routines – a recognition "bottom-up" routine to substitute the right side

for the left side, and a generation "top-down" routine to substitute the left side for the right side.

```
type        Vertex
            access record; colour: V; x co, y co: Real
            end record;

type        Edge    is record source, sink: vertex
                    end record;


procedure Generate p2 ( in i1, i2, i4: Vertex; e: Edge; a, phi: Real
                        out i5, i6, i7, i8: Vertex; success: Boolean
                            e1, e2, e3, e4, e5: Edge);
          x, y, u, v: Real;


   begin x       := i1.x co; y := i1.y co;
         success := (i1.colour = A) and (e.source = i2) and (e.sink = i4)
                    and (i2.colour = A) and (i4.colour = B)
                    and (i2.x co = x + a × cos phi)
                    and (i2.y co = x + a × sin phi)
                    and (i4.x co = x + a × cos(phi + 30)/√3)
                    and (i4.y co = x + a × sin(phi + 30)/√3);


      if success then
        i5  := new Vertex(colour ⇒ B; x co ⇒ x + a cos(phi − 30)/√3);
               y co ⇒ x + a sin(phi − 30)/√3);
        u   := x + cos(phi)/5; v := y + a × sin(phi)/5;
        i8  := new Vertex(colour ⇒ D, x co ⇒ u, y co ⇒ v);
        i6  := new Vertex(colour ⇒ C,
                                        x co ⇒ u + a × cos(phi + 90) × sqrt(3)/24,
                                        y co ⇒ v + a × sin(phi + 90) × sqrt(3)/24);
        i7  := new Vertex(colour ⇒ C,
                                        x co ⇒ u + a × cos(phi − 90) × sqrt(3)/24,
                                        y co ⇒ v + a × sin(phi − 90) × sqrt(3)/24);
        e1  := (source ⇒ i2, sink ⇒ i5);
        e2  := (source ⇒ i4, sink ⇒ i6);
        e3  := (source ⇒ i6, sink ⇒ i8);
        e4  := (source ⇒ i7, sink ⇒ i8);
        e5  := (source ⇒ i5, sink ⇒ i7);
      end if;
end Generate p2;
```

Figure 3.1: Traditional Implementation of Genesis.

Using recursive descent and other familiar methods of compiler writers it is not difficult to write analysis and synthesis programs once one has routines for "choosing a production" and "finding an occurrence". The only difference between an analysis program and a traditional parsing program is that a test for "reaching the initial graphic I of a grammar" may be rather complicated; the usual distinction between terminal and non-terminal symbols in a formal language is irrelevant.

Comment:
The routine for finding an occurrence of a rule $B_1 \longleftarrow K \longrightarrow B_2$ corresponds to finding a morphism $K \longrightarrow D$; in the implementation it corresponds to setting some of the actual parameters for a call of the function for the rule (the other parameters may be set by the user). In later sections we will describe database implementations and expert system implementations but they could have been included in this section as "traditional" implementations of graphics and their grammars.

#3.2 Concurrent implementations

If one wants to implement graphics and their grammars in a concurrent language like OCCAM/CSP with their processes, CCS with its agents, or ADA with its tasks, it is natural to identify *vertices* in a graphic with processes/agents/tasks. The edges in a graphic can be given by links/channels; the attributes of a vertex can be given by local variables of a process and the label of a vertex corresponds to the type of its process. Since a concurrent language can capture the inherent non-determinacy of grammars, it is not difficult to write analysis and synthesis programs once one has recognition and generation routines for the grammar rules. As Figure 1 shows, one needs a rather flexible programming language in order to write recognition and generation routines.

operation Generate p22 $(i1, i2, i4, i5, i6, i7, i8, a, \phi)$
condition old Task$(i1, A, x, y)$;
old Task$(i2, A, x + a\cos\phi; y + a\sin\phi)$
old Task$(i4, B, x + a\cos(\phi + 30)/\sqrt{3}, y + a\sin(\phi + 30)\sqrt{3})$
old Link$(i2, i4)$
action new Task$(i5, B, x + a\cos(\phi - 30)/\sqrt{3}, y + a\sin(\phi - 30)/\sqrt{3})$
new Task$(i8, D, u := x + a\cos(\phi)/5, v := y + a\sin(\phi)/5)$
new Task$(i6, C, u + a\cos(\phi + 90)\sqrt{3}/24, v + a\sin(\phi + 90)\sqrt{3}/24)$
new Task$(i7, C, u + a\cos(\phi - 90)\sqrt{3}/24, v + a\sin(\phi - 90)\sqrt{3}/24)$
new Link$(i2, i5)$ new Link$(i4, i6)$ new Link$(i6, i8)$;
new Link$(i7, i8)$ new Link$(i5, i7)$
end operation

operation Recognise p2 $(i1, i2, i4, i5, i6, i7, i8, a, \phi)$

condition old Task($i1, A, x, y$);
old Task($i2, A, x + a\cos\phi,\ y + a\sin\phi$)
old Task($i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3}$)
old Task($i5, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)/\sqrt{3}$)
old Task($i8, D, u := x + a\cos\phi/5,\ v := y + a\sin\phi/5$)
old Task($i6, C, u + a\cos(\phi + 90)\sqrt{3}/24,\ v + \sin(\phi + 90)\sqrt{3}/24$)
old Task($i7, C, u + a\cos(\phi - 90)\sqrt{3}/24,\ v + \sin(\phi - 90)\sqrt{3}/24$)
old Link($i2, i4$) old Link($i2, i5$) old Link($i4, i6$)
old Link($i6, i8$) old Link($i7, i8$) old Link($i5, i7$)
action drop Link($i2, i5$) dropLink($i4, i6$)
drop Link($i6, i8$) drop Link($i7, i8$) drop Link($i5, i7$)
drop Task($i5$) drop Task($i6$) drop Task($i7$) drop Task($i8$)
end operation

Figure 3.2: Concurrent Implementation of Genesis.

#3.3 Logical implementations

If one wants to implement graphics and their grammars in a logic programming language like PROLOG, it is natural to represent graphics as sets of atomic formulas. One way of capturing edges in a graphic is to index the vertices and connect the indices by atomic formulas.

Since grammar rules produce new graphics from old, the formulas that were true before applying the rule may be false afterwards. This non-monotonicity can be captured in three ways:

(A) Introducing a new "stage" attribute in all formulas, and expressing the grammar rule by a logical implication.

(B) Using non-logical operators like *assert* and *retract* in PROLOG.

(C) Moving to the metalevel.

Because of the "frame" problem (A) seems unworkable, so let us illustrate (B) and (C) in our semantic net example:

(A) Current graphic $P(1, A, 0).\ P(2, A, 6, 0).$
$P(3, A, 3, 3\sqrt{3}).\ P(4, B, 3, \sqrt{3}).$
$E(1, 4).\ E(2, 4).\ E(3, 4).$

(B) Generate-p2 $(i1, i2, i4, a, \phi) :- P(i1, A, x, y)\ P(i2, A, x + a\cos\phi,\ y + a\sin\phi)$
$P(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})$

$E(i2, i4)$
assert $P(< i1, i2, i4, 5 >, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)/\sqrt{3})$
assert $P(< i1, i2, i4, 8 >, D, x + a\cos\phi/5,\ y + a\sin\phi/5)$
assert $P(< i1, i2, i4, 6 >, C, x + a\cos\phi/5 + a\cos(\phi + 90)\sqrt{3}/24,$
$y + a\sin\phi/5 + a\sin(\phi + 90)/\sqrt{3}/24)$
assert $P(< i1, i2, i4, 7 >, C, x + a\cos\phi/5 + a\cos(\phi - 90)/\sqrt{3}/24,$
$y + +a\sin\phi/5 + a\sin(\phi - 90)/\sqrt{3}/24)$
assert $E(< i1, i2, i4, 6 >, < i1, i2, i4, 8 >)$
assert $E(< i1, i2, i4, 7 >, < i1, i2, i4, 8 >)$
assert $E(< i1, i2, i4, 5 >, < i1, i2, i4, 7 >)$

Recognise-p2 $(i1, i2, i4, i5, i6, i7, i8, a, \phi) : - P(i1, A, x, y)$
$P(i2, A, x + a\cos\phi,\ y + a\sin\phi)$
$P(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})\ E(i2, i4)$
$P(i5, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)/\sqrt{3})\ E(i2, i5)$
u is $x + a\cos\phi/5$ v is $y + a\sin\phi/5$, $P(i8, D, u, v)$
$P(i6, C, u + a\cos(\phi + 90)\sqrt{3}/24,\ v + a\sin(\phi + 90)\sqrt{3}/24)$
$P(i7, C, u + a\cos(\phi - 90)\sqrt{3}/24,\ v + a\sin(\phi - 90)\sqrt{3}/24)$
$E(i6, i8)\ E(i8, i7)\ E(i5, i7)$
retract $P(i5, B, u + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})$
retract $P(i8, D, u, v)$ retract $E(i2, i5)$ retract $E(i5, i7)$
retract $P(i6, C, u + a\cos(\phi + 90)\sqrt{3},\ v + a\sin(\phi + 90)\sqrt{3}/24)$
retract $P(i7, C, u + a\cos(\phi - 90)\sqrt{3},\ v + a\sin(\phi - 90)\sqrt{3}/24)$
retract $E(i4, i6)$ retract $E(i6, i8)$ retract $E(i8, i7)$

(C) Gen-p2 (G,Union(G,Assertions)) : – In(G,Union($P(i1, A, x, y) \ldots E(i2, i4)$)).
Rec-p2 (Union(G,Retractions),G) : – In(G,Union($P(i1, A, x, y) \ldots E(i5, i7)$)).

Figure 3.3: Logical Implementation of Wallpaper Rule.

As numbering graphic vertices is inelegant, the reader is probably unimpressed by this example. Much more impressive examples of the elegance and power of logical programming languages for "declarative graphics" can be found in (HM). There the authors distinguish between specification and instances of "pictures"; a picture instance $P < G, R >$ is given by adding geometric G and restriction R transformations to a specification P of a picture (see final remark in Section 1). There are two ways of combining picture specifications

Composition $P \longleftarrow R_1, \ldots, R_m, P_1 \& \ldots \& P_n$
Collection $P \longleftarrow \{P \& \ldots \& P_n\}$.

Collection allows independent pictures to be grouped together, whereas composition combines pictures that are interrelated by the shared attributes and relations: $R_1 \ldots R_m$. The authors present algorithms for recognising pictures, generating pictures, and breaking pictures into their subpictures, so they can implement rules

and graphic grammars. However, there does seem to be a tension between our rewriting approach and their algebraic approach; we intend to reduce this tension by introducing "graphic operators" like & into our theory of graphics.

#3.4 Syntactic implementations

Sometimes the most natural way to implement a graphic grammar is to use a term rewriting system like REVE (L). In such a system a graphic is represented by a term with components for vertices and edges. A grammar rule is represented by a "conditional" rewrite rule; rewriting in one direction gives a generation routine; rewriting rules for rearranging terms are used to find occurrences of grammar productions.

- generation rule
  $P(i1, A, x, y);\ P(i3, A, x_a \cos\phi,\ y + a\sin\phi);$
  $P(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3});\ E(i2, i4)$
  $\Rightarrow\ P(i1, A, x, y) \ldots E(i1, i4)$
  $P(< i1, i2, i4, 5 >, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)) \ldots$
  $E(< i1, i2, i4, 5 >, < i1, i2, i4, 7 >)$

- rearrangement rules
  $E(i, j);\ E(k, l) \Rightarrow E(k, l);\ E(i, j) \qquad E(i, j) \Rightarrow E(j, i)$
  $E(i, j);\ P(k, l, x, y) \Rightarrow P(k, l, x, y);\ E(i, j)$
  $P(k, l, x, y);\ P(kk, ll, xx, yy) \Rightarrow P(kk, ll, xx, yy);\ P(k, l, x, y)$

Figure 3.4: Syntactic Implementation of Wallpaper Grammar.

We have chosen to present the rewriting rules in the synthesis direction. The last four rules do not change the underlying graphic, but they have to be there for the term rewriting system to simulate all derivations in the graphic grammar. If we reverse all the rewriting rules and stop when we derive the term I we can analyse any graphic in the language given by the grammar.

Can every graphic grammar be simulated so accurately by a term rewriting system? This would be true if we could show that graphic grammars are algebraic data type grammars (EHHB). To show this we have to:

- Find an abstract data type specification whose initial algebra is isomorphic to the set of graphics.
- Describe how graphic productions $B_1 \longleftarrow K \longrightarrow B_2$ can be converted into a pair $< t_1, t_2 >$ of Σ-terms such that:

$$\begin{array}{ccccc} B_1 & \longleftarrow & K & \longrightarrow & B_2 \\ \downarrow & & \downarrow & & \downarrow \\ G_1 & \longleftarrow & D & \longrightarrow & G_2 \end{array}$$

if and only if $G_1 = [D_1(t_1)]$ and $G_2 = [D(t_2)]$ for some Σ-term $D(x)$.

The version of graph grammars, shown to be algebraic data type grammars in (BC), was:

1. Edges are node strings with labels.
2. Nodes can have labesl and attributes in the form of unary edges.
3. All productions have a finite discrete graph $\bar{n}$ as K.

Graphics satify (1) (2), "V_1V_2 is an edge $\rightarrow$ is unlabelled & V_2V_1 is an edge" and "only unary and binary edges". Productions in graphic grammars are not required to satisfy (3), but the generative power of a graphic grammar is not affected if we enforce (3) by dropping all edges from K (in the application of the production these edges disappear from D but reappear in G_1 and G_2 because they are still present in B_1 and B_2).

Final remark

One can also implement graphics and their grammars when one has access to a powerful data base system or a "system for building expert systems" of the kind now flooding the market. In such systems graphics can be represented by knowledge bases, and grammar rules can be represented by transactions. In a data base system a transaction is a sequence of find, insert, delete and modify actions; in an expert system a transaction is a production rule of the form "if condition **then** action".

Acknowledgements

The first author would like to thank CNP and IBM-Brasil for supporting her travel to the workshop. Both authors would like to thank the editors for suggesting improvements to the presented paper.

References

(BC) M. Bauderon, B. Courcelle: "An algebraic formalism for graphs", in *CAAP '86, Springer LNCS* 214, 1986.

(Bu) H. Bunke: "Graph Grammars as a Generative Tool in Image Understanding", in *Springer LNCS* 153 (1983) 8-19.

(EH) H. Ehrig: *"Introduction to the algebraic theory of graph grammars (a survey)"* in Springer LNCS 73 (1978) 1-64.

(EHHB) H. Ehrig, A. Habel, U. Hummart, P. Boehm: "Towards algebraic datatype grammars: a junction between algebraic specifications and graph grammars", *Bull. EATCS* 29 (1986) 22-27.

(EKMRW) H. Ehrig, H.J. Kreowski, A. Maggiolo-Schettini, B.K. Rosen, J. Winkowski: "Transformations of structures: an algebraic approach", *Math. Sys. Th.* 14 (1981) 305-334.

(G) H. Gottler: "Attributed Graph Grammars for Graphics", in *Springer LNCS* 153 (1983) 130-142.

(GS) E.H. Lockwood, R.H. Macmillan: *Geometric Symmetry*, Cambridge Univ. Press, 1978.

(HM) R. Helm, K. Marriott: "Declarative Graphics", in *3rd Int. Conf. Logic Programming. Springer LNCS* 225 (1986).

(Ka) R. Kalaba, J.L. Casti (eds.): "Numerical grid Generation", *Applied Math. Computation* (1982) 1-895.

(L) P. Lescanne: "Computer experiments with the REVE Term Rewriting Systems Generator", *Proc. 10th Symp. Pr. Prog. Lang.* (1983) 99-108.

(Li) G. Rozenberg, A. Salomaa (eds.): *The Book of L*, North-Holland 1986.

(Ma) C.H. Macgillaray: *Fantasy & Symmetry. The Periodic Drawings of M.C. Escher*, Abrahs, N.Y. 1976.

(Pa) P. Padawitz: "Graph grammars and operational semantics", *Th. Comp. Sci.* 19 (1982) 117-141.

(So) J.F. Sowa: *Conceptual Structures*, Addison-Wesley 1984.

Graphics and Their Grammars

L. Hess
Instituto Militar de Engenharia
Rio de Janeiro, Brasil

B.H. Mayoh
Computer Science Department, Aarhus University
Aarhus, Denmark

Abstract. Graphics are graphs with attributes at their vertices. Graphic grammars are natural extensions of graph and attribute grammars with rules that are attributed extensions of the "pushout" productions of graph grammars. The notion of graphic grammars is presented and various programming implementations will be discussed. Many motivating examples will be given, including

the development of biological organisms
the "semantic net" representation of expert system knowledge.

Keywords: Graphs, attributes, grammars.

Introduction

Graphics have been defined as graphs whose vertices have attributed values. When the assignment of values to vertices is allowed, graph models become even more useful than they already are. In figure 1 we indicate some possible applications of graphics and their grammars.

Once one accepts that graphics are a useful generalisation of graphs, it is natural to look for a definition of a graphic grammar that generalises both graph and attribute grammars appropriately. The two definitions of graphic grammars in the literature have chosen to generalise a form of graph rewriting that may be useful in certain applications but seems arbitrary even so. In Section 1 we base our definition of graphic grammars on graphic morphisms and pushouts. This form of graph rewriting seems to be the natural generalisation of string rewriting in attribute grammars. A graphic is a new representational entity, because its attributes can be Σ-algebra terms; graphic grammars can benefit from the results of the algebraic approach to graph grammars and Σ-algebras. In Section 2 we look at the use of graphics for biological organisms and the "data base and expert system" problem of representing real world knowledge. In Section 3 we describe various ways of "implementing" graphic grammars on a computer.

Image analysis & Generation:	Aircraft identification (Fu).
	Stochastic terrain models (HM).
	Scene Analysis (Bu).
Diagram analysis & Generation:	Program Structure (G).
	Structured pictures (HM).
	Interactive picture editor (HM).
	Program Animation (HM).
	Circuit diagrams (G).
	Art (N).
	Window manager (HM).
Numerical analysis:	Generating grids for solving DDE's (Ka).
Biological Models:	Morphogenesis (Li).
Knowledge representation:	Conceptual schemes in databases.
	Semantic nets in AI (So).

Figure 1: Applications of graphics.

This paper is not intended to be a presentation of the theory of graphics and their grammars — just suggestions for definitions, examples and how they can be implemented.

#1. Graphics and Their Grammars

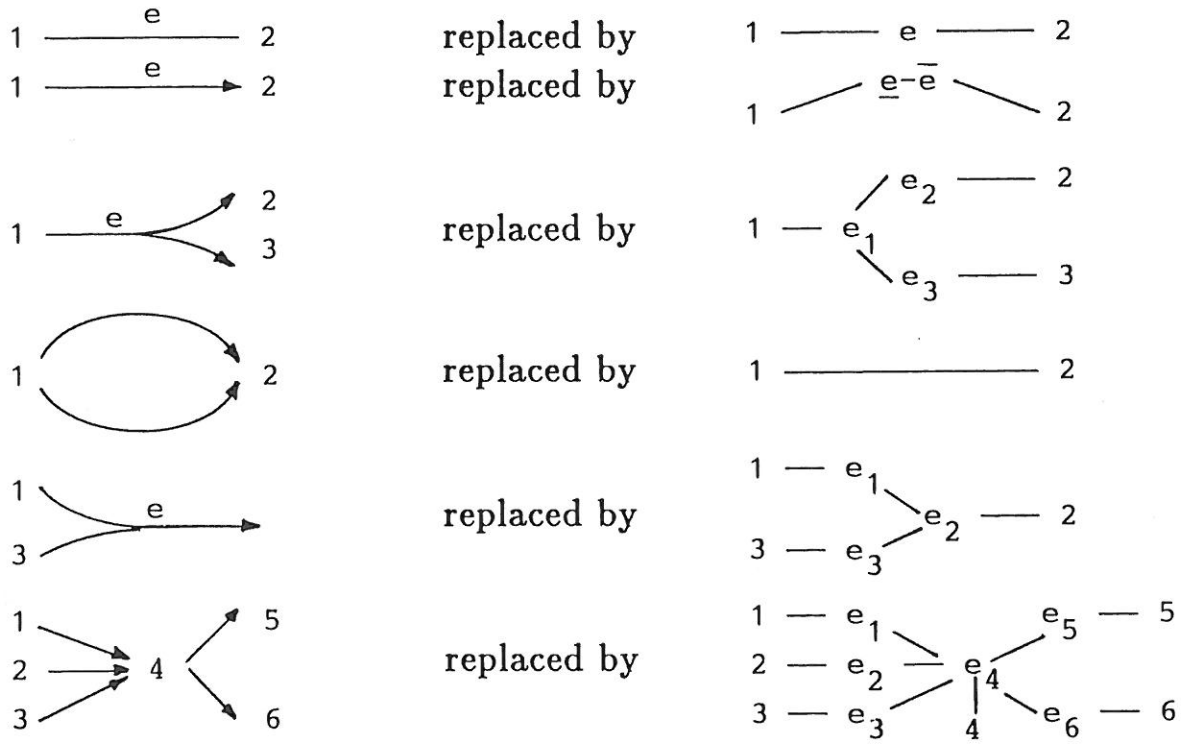
The natural generalisation of a graph is also the natural generalisation of the formal language concept of a word or string: a totally ordered set of vertices labelled by symbols from a finite alphabet. If we take a symmetric relation instead of a total ordering, we get:

Definition

A **graphic** G on an algebra A and label set W is a 4-tuple $G = (K, E, l, f)$ where

K is a finite non-empty set of elements called vertices,
E is a finite set of unordered pairs of vertices called edges,
l is a function from K to W,
f is a function from K to A.

We seem to have lost generality by not allowing edge labels or attributes, directed edges, edges with more than two vertices, or multiple edges. This is not so because



we can introduce edge nodes as new vertices whose labels and attributes can capture ordering information. As examples of new nodes capturing edge information, consider the replacements. It is clear that edge replacing rules can be simulated by graphic productions and one can map a replacement graphic back to a graph with directed, labelled, attributed edges.

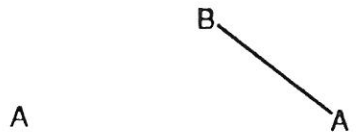
A **graphic morphism** $g : G \to G'$ consists of maps ver: $K \to K'$, edge: $E \to E'$, att: $A \to A'$ such that att is a homomorphism.

$$edge(v_1, v_2) = (ver(v_1), ver(v_2))$$
$$l'(ver(v)) = l(v)$$
$$f'(ver(v)) = att(f(v)).$$

We say that G is a **subgraphic** of G' ($g : G \to G'$ is an *embedding*) when "ver" is an injection.

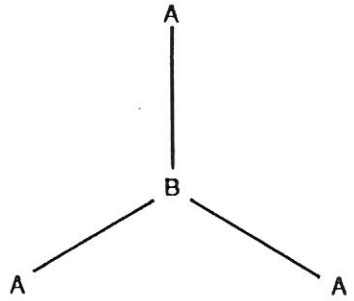
Comment:
With this definition of morphism, graphics form a category when the algebras form a category. We could specify the graphic category as a subcategory of STRUCT (EKMRW). Graphic morphisms are natural combinations of graph morphisms and morphisms of algebras. The definition of subgraphic is a natural generalisation of the definition of subgraph. We will be particularly interested in the case when the category of algebras is the much studied Σ-algebra for some signature Σ.

Example:
The graphic on the term algebra $T(\Sigma \cup V)$ – where $\Sigma = (R^2, +, \times, \sqrt{}, \sin, \cos, 0, \ldots)$ and $x, y, a, \phi \in V$ –



$$K = \{1,2,4\} \qquad E = \{(2,4)\}$$
$$l(1) = A \qquad f(1) = < x, y >$$
$$l(2) = A \qquad f(2) = < x + a\cos\phi, y + a\sin\phi >$$
$$l(4) = B \qquad f(4) = < x + a\cos(\phi + 30)/\sqrt{3},$$
$$y + a\sin(\phi + 30)/\sqrt{3} >$$

is a subgraphic of the graphic (on the algebra $T(\Sigma \cup V)/\approx$ where $\approx$ is given by the equations for evaluating $+, \times, \sin, \cos \ldots$)



$$K' = \{1,2,3,4\} \qquad E' = \{(1,4),(2,4),(3,4)\}$$
$$l'(1) = A \qquad f'(1) = (0,0)$$
$$l'(2) = A \qquad f'(2) = (a,0)$$
$$l'(3) = A \qquad f'(3) = (a/2, a\sqrt{3}/2)$$
$$l'(4) = B \qquad f'(4) = (a/2, a\sqrt{3}/6)$$

because we have the graphic morphism

$$ver(v) = v \quad att(u,v) = (Su, Sv)$$

where S is given by substituting 0 for x, y and ϕ. The trigonometric expressions in the subgraphic give the projections of an edge to the x- and y-axes.

In this section all graphics will have two spatial attributes which are given by the representation of the graphic as a diagram. Notice that all graphics have the empty graphic 0 as a subgraphic. The empty graphic is useful for removing vertices.

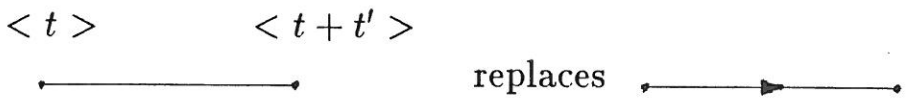
Definition

A graphic sequential rewriting system $\Gamma = (V, \Sigma, I, P)$ consists of

V, a finite non-empty set of symbols
Σ, a signature
I, the start graphic on some Σ-algebra
P, a non-empty set of productions.

A **production** $p = B_1 \xleftarrow{B_1} K \xrightarrow{B_2} B_2$ is a pair of attribute preserving graphic morphisms connecting three graphics whose vertices have attribute values in $T(\Sigma \cup V)$ – terms formed from variables V and the operations in the signature Σ.

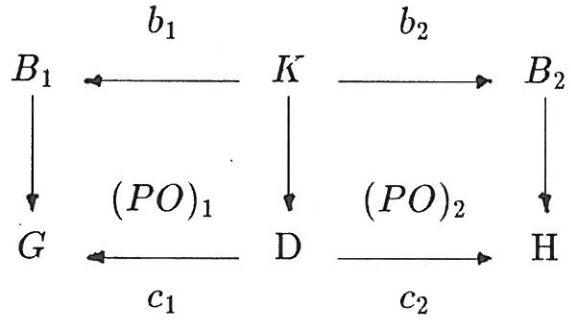
Comment:
One might well disagree with our requirement that the attribute values in B_1, K, B_2 are given by terms in $T(\Sigma \cup V)$. This requirement is connected with our choice of morphism rewriting; it ensures that graph pushouts coincide with graphic pushouts. The requirement also allows the attribute values to carry edge information; if a directed graph is partially ordered, the edge ordering can be captured by



Not only "later", "above" and "to the right of" are examples of this, but also many orderings in database and AI semantic nets.

Definition

A direct derivation $G \underset{p}{\Rightarrow} H$ is defined by giving a production, a context graphic D, a graphic morphism $K \rightarrow D$, and two pushouts $(PO)_1$ and $(PO)_2$ in the category of graphics and graphic morphisms defined above.



We say that the graphic H is derived from G by the production p, based on the morphism $g : B_1 \rightarrow G$.

A graphic language is defined as the set of all graphics derived from the initial graphic by productions in P.

Comment:
The attribute components in the vertical morphisms g, d, h need not be identities, they can be the same term evaluation from $T(\Sigma \cup V)$ to a Σ-algebra A. As explained in [Eh, p. 11], the pushout of two graph morphisms $b : K \rightarrow B$ and $d : K \rightarrow D$ is given by "gluing together the items $b(k)$ in B and $d(k)$ in D for each item k in K". For labelled graphs we want

$$l_G([x]) = \textbf{if } x \in B \textbf{ then } l_B(x) \textbf{ else } l_D(x)$$

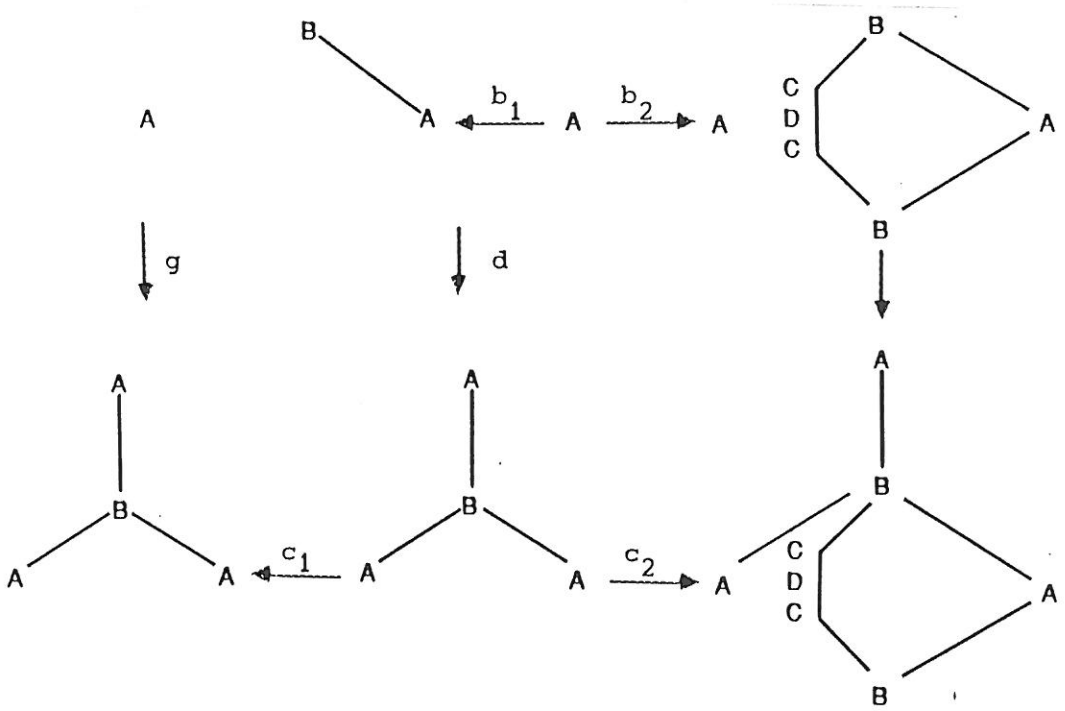
for each vertex $[x]$ in G, the pushout graph of b and d. This is a proper definition if $l_B(b(k)) = l_D(d(k))$ for all k in K (satisfied when b and d are graphic morphisms), and we try to define attributes for vertices in the pushout graphic G by

$$(+) \qquad f_G([x]) = \textbf{if } x \in B \textbf{ then } f_B(x) \textbf{ else } f_D(x)$$

for each vertex $[x]$ in G. This works when $f_B(b(k)) = f_D(d(k))$ for all k in K (satisfied when b and d are attribute preserving). We would like a more general way of defining attribute values in the pushout graphic. In a later paper we will study the case of "productions" whose graphic morphisms are not attribute preserving. The reader is invited to reformulate the above discussion of graphic pushouts in the terminology of the algebraic approach to general structures (EKMRW, Pa).

Notice that our definition of a direct derivation $G \underset{p}{\Rightarrow} H$ does *not* depend our our discussion of when pushouts exist – if the required pushouts do not exist, then we do not have $G \underset{p}{\Rightarrow} H$.

Example:
Perhaps we should give an example of a direct derivation.



All the morphisms in this diagram are embeddings (sources are subgraphics of sinks); the top two morphisms give a production and the morphism g was described in our previous example. The left square is a pushout when d assigns the coordinates $< 0, 0 >$ to the lower right A in its target. The right square is also a pushout with this assignment of values to attributes. One may well ask what has happened to the parameters a and ϕ in the previous example. The answer is that these parameters fix the morphisms, b_1 and b_2 – they parametrise a family of productions.

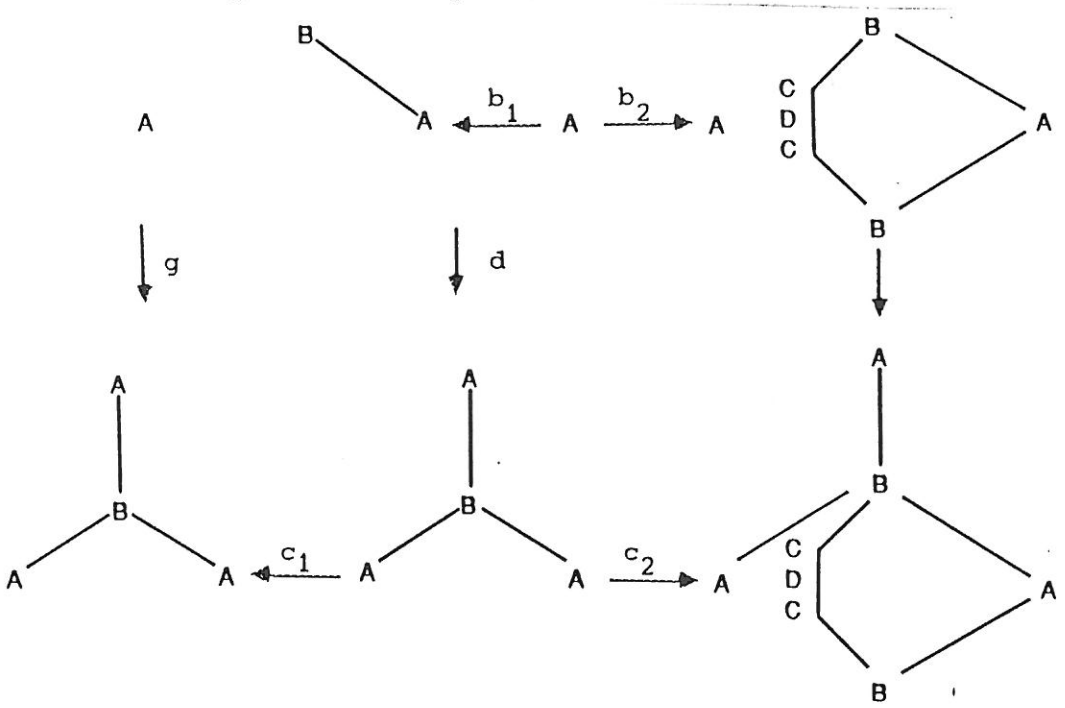
for each vertex $[x]$ in G, the pushout graph of b and d. This is a proper definition if $l_B(b(k)) = l_D(d(k))$ for all k in K (satisfied when b and d are graphic morphisms), and we try to define attributes for vertices in the pushout graphic G by

$$(+) \qquad f_G([x]) = \textbf{if } x \in B \textbf{ then } f_B(x) \textbf{ else } f_D(x)$$

for each vertex $[x]$ in G. This works when $f_B(b(k)) = f_D(d(k))$ for all k in K (satisfied when b and d are attribute preserving). We would like a more general way of defining attribute values in the pushout graphic. In a later paper we will study the case of "productions" whose graphic morphisms are not attribute preserving. The reader is invited to reformulate the above discussion of graphic pushouts in the terminology of the algebraic approach to general structures (EKMRW, Pa).

Notice that our definition of a direct derivation $G \underset{p}{\Rightarrow} H$ does *not* depend on our discussion of when pushouts exist – if the required pushouts do not exist, then we do not have $G \underset{p}{\Rightarrow} H$.

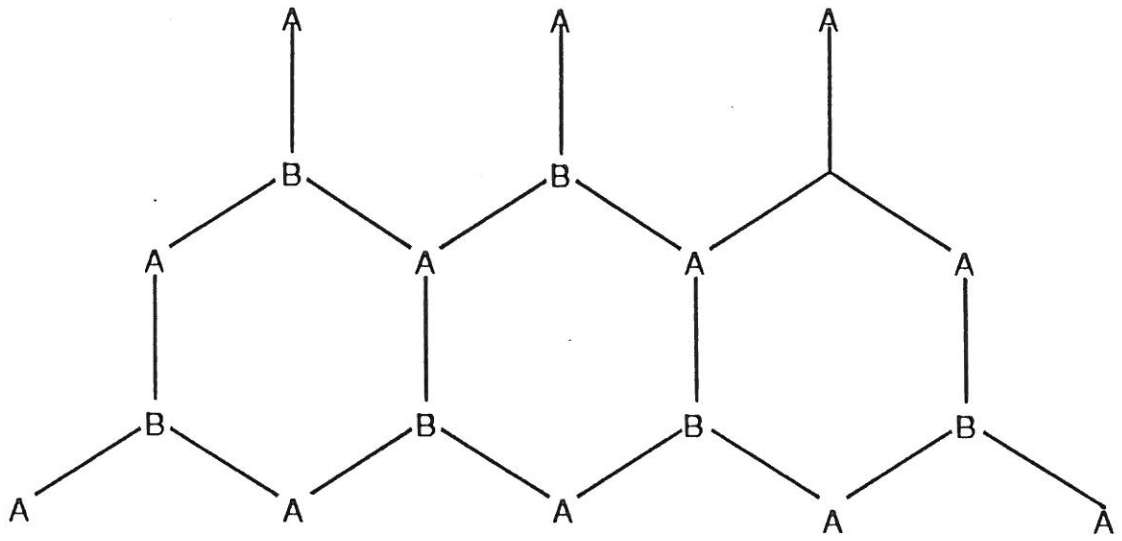
Example:
Perhaps we should give an example of a direct derivation.



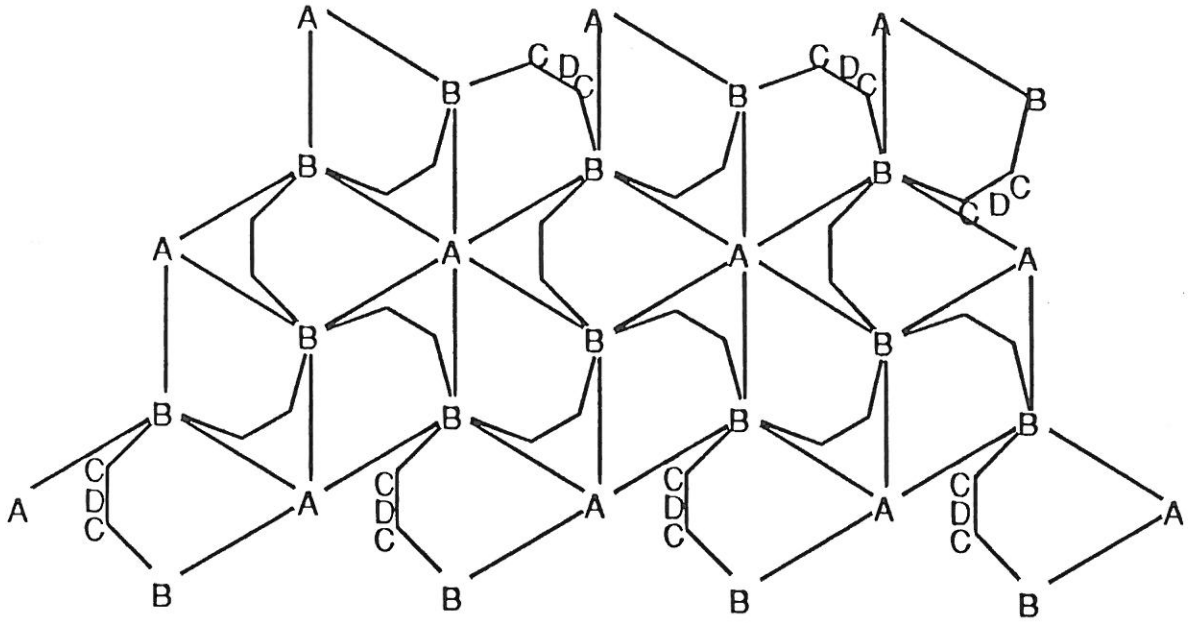
All the morphisms in this diagram are embeddings (sources are subgraphics of sinks); the top two morphisms give a production and the morphism g was described in our previous example. The left square is a pushout when d assigns the coordinates $< 0, 0 >$ to the lower right A in its target. The right square is also a pushout with this assignment of values to attributes. One may well ask what has happened to the parameters a and ϕ in the previous example. The answer is that these parameters fix the morphisms, b_1 and b_2 – they parametrise a family of productions.

Please insert p. 7 from manuscript already with you.

Repeated use of the production p1 gives the pattern:



Repeated use of the production p2 now gives:



We have described the derivation of a graphic G in a graphic language as if the attributes were evaluated during the derivation. However, there is no objection to first deriving the underlying graph of G in the graph language, corresponding to L, then deriving the attribute values. This corresponds to the usual evaluation method in "string" attribute grammars: first derive the string in the underlying context-free grammar, then solve the attribute equations.

Final remark:
If one has an Σ-morphism μ to any Σ-algebra A from the Σ-algebra of the initial graphic of a grammar, then one gets a transformation of the graphic language by applying μ at every step of every derivation in the language. This captures geometric transformations of pictures.

#2. Reality Models

Are graphics useful models of reality?

Graph models have become popular, and they become more accurate and convenient when vertices can have attribute values. However, the usefulness of graph grammars is strongly dependent on the kinds of rewriting rules that are allowed. "Push out" rewriting gives a flexibility that single morphism rewriting (identity morphisms from K to B_1, and D to G) and simple node and/or edge replacement rules do not have.

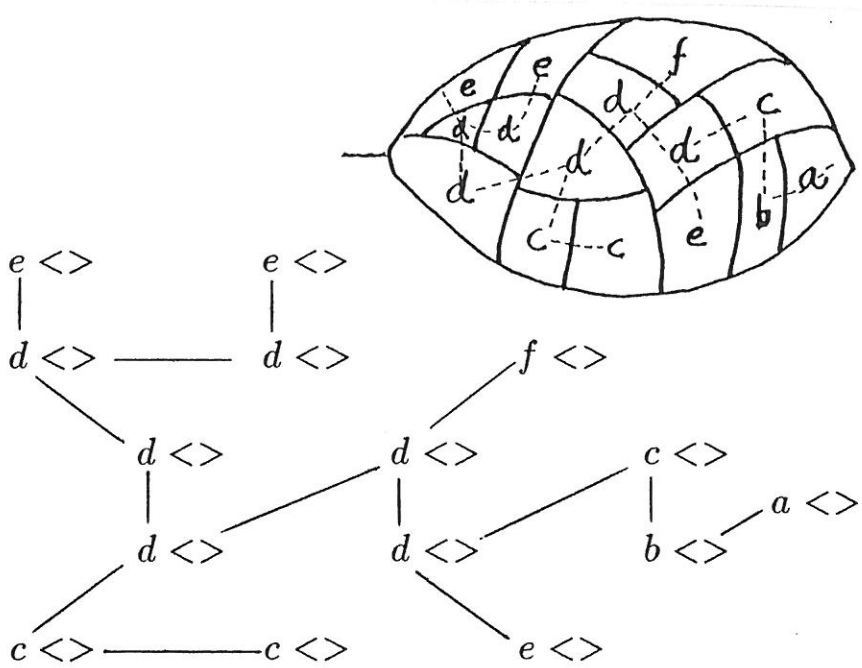
It is well established that the development of biological organisms (Li) can be modelled by graphs and their grammars and Example 1 shows that the extension to graphics gives more natural models. In the development of expert systems semantic nets are a popular way of representing "real world knowledge" (Sowa) and Example 2 shows that graphics are a convenient formalism for semantic nets.

Example 1: "Phascum Cuspidatum"

The grammar for the development of the organism "Phascum Cuspidatum" and the graphic for a particular life stage is shown in

I: a

P: $a \Leftarrow a \Rightarrow b - a$
$b \Leftarrow b \Rightarrow c$
$c \Leftarrow c \Rightarrow d - e$
$d \Leftarrow d \Rightarrow d$
$e \Leftarrow e \Rightarrow f$
$f \Leftarrow f \Rightarrow c - c$



The edges in the graph reappear as dotted lines in the picture. As the attributes of the "Phascum Cuspidatum" cells are not used in this graphic grammar (except for the convention that the only vertex in the left graphic of a production is rewritten to the leftmost vertex of its right graphic), it is no more than a traditional graph grammar. However, this grammar illustrates two points that are valuable in biological modelling.

- Grammars work best when there are only a finite number of cell labels, but the development of an organism is most naturally described in terms of infinitely many states, and graphic attributes can capture this infinity.
- Grammars usually give an unambiguous result when a rule is applied, but the development of an organism is most naturally described by allowing very flexible applications of rules, and graphic attributes can capture this flexibility.

Example 2: "Semantic Nets"

A grammar for the representation of human relationships and the graphic for a particular family are shown in

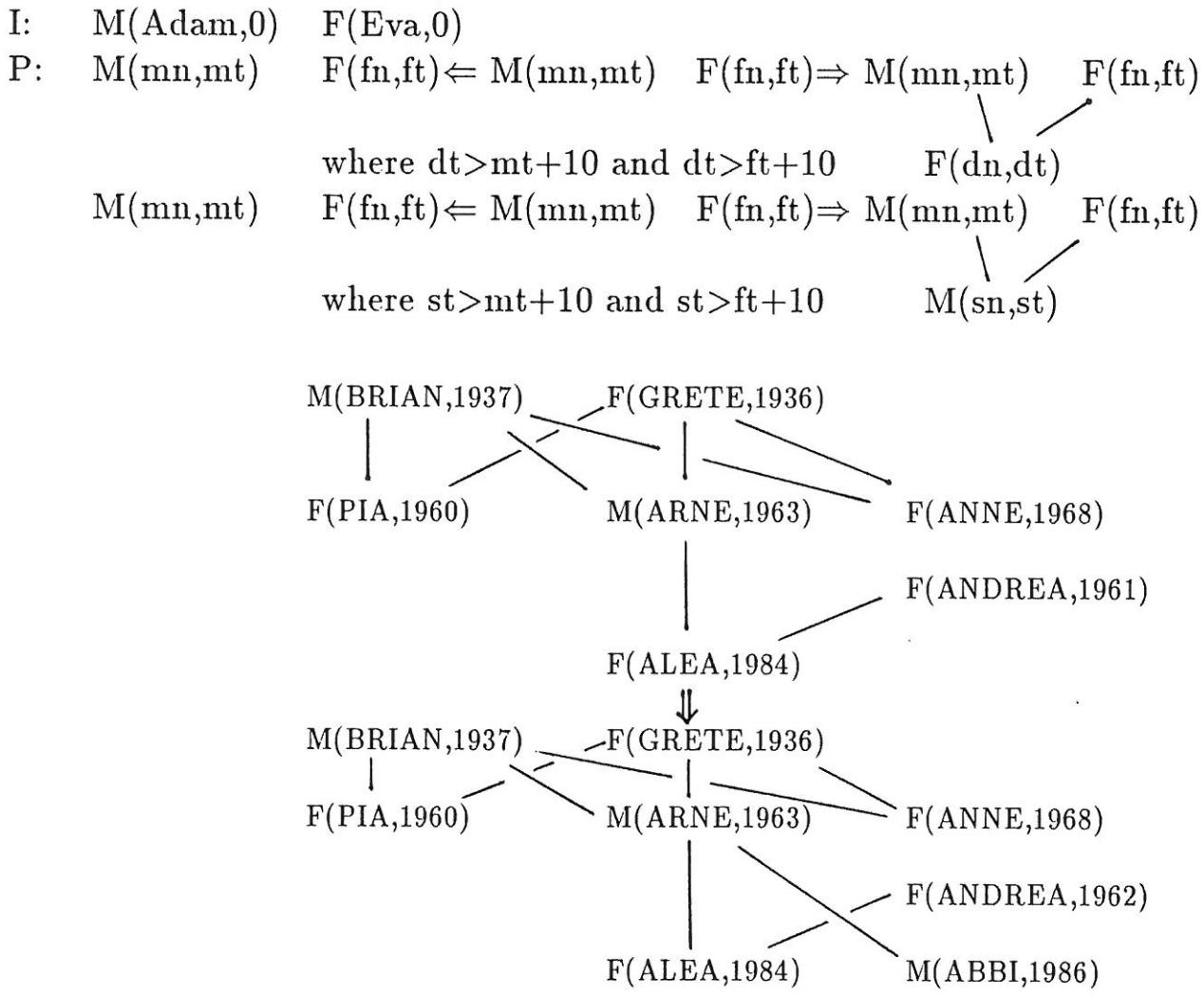


Figure 2.1: The Graphic Grammar Genesis.

The application of the second rule in this example is flexible because the fixing of the parameters – mn=Arne, mt=1963, fn=Andrea, ft=1962 – did not determine the attributes: sn=Abbi and st=1968. The example is instructive because it shows

how "time" attributes allow graphics to capture natural orderings which would be expressed by directed edges in graph models. Other examples would show how graphics can capture other kinds of edge information that expert system builders put into semantic nets. (Remember the discussion in Section 1 on edge information.)

Example 3: Erasing

In (So) Sowa introduced a form of semantic nets called conceptual graphs for the graphic grammar in Example 2, and he described the powerful operations for extracting information and manipulating conceptual graphs. All these operations can be extended to graphics. To convince doubters we show how edges and vertices can be removed ... consider the addition of rules

M(mn,mt)	$\Leftarrow 0 \Rightarrow 0$		– vertex removal	
F(fn,ft)	$\Leftarrow 0 \Rightarrow 0$		– using the empty graphic 0	
M(mn,mt)-F(fn,ft)	$\Leftarrow$ M(mn,mt)	F(fn,ft)	$\Rightarrow$ M(mn,mt)	F(fn,ft)
M(mn,mt)-M(n,t)	$\Leftarrow$ M(mn,mt)	M(n,t)	$\Rightarrow$ M(mn,mt)	M(n,t)
F(fn,ft)-F(n,t)	$\Leftarrow$ F(fn,ft)	F(n,t)	$\Rightarrow$ F(fn,ft)	F(n,t)

The last three rules give edge removal.

#3. Implementation

Once one accepts that graphics and their grammars are useful models of reality, one wants to implement them on the computer; one wants an *analysis* program to recognise whether a given graphic can be generated by a given grammar, and one wants a *synthesis* program for generating graphics from grammars. The many ways of implementing graphics and their grammars can be grouped into – traditional, concurrent, logical and syntactical – so we shall look at each of these in turn.

#3.1 Traditional implementations

If one wants to implement graphics and their grammars in an imperative language like PASCAL or an object oriented language like SmallTalk, it is natural to represent *graphics as data objects.* It is also natural to represent a grammar rule by a pair of routines – a recognition "bottom-up" routine to substitute the right side for the left side, and a generation "top-down" routine to substitute the left side for the right side.

```
type        Vertex
            access record; colour: V; x co, y co: Real
            end record;

type        Edge    is record source, sink: vertex
                    end record;

procedure Generate p2 ( in i1, i2, i4: Vertex; e: Edge; a, phi: Real
                        out i5, i6, i7, i8: Vertex; success: Boolean
                            e1, e2, e3, e4, e5: Edge);
          x, y, u, v: Real;


    begin x       := i1.x co; y := i1.y co;
          success := (i1.colour = A) and (e.source = i2) and (e.sink = i4)
                     and (i2.colour = A) and (i4.colour = B)
                     and (i2.x co = x + a × cos phi)
                     and (i2.y co = x + a × sin phi)
                     and (i4.x co = x + a × cos(phi + 30)/√3)
                     and (i4.y co = x + a × sin(phi + 30)/√3);


        if success then
          i5  := new Vertex(colour ⇒ B; x co ⇒ x + a cos(phi − 30)/√3);
                 y co ⇒ x + a sin(phi − 30)/√3);
          u   := x + cos(phi)/5; v := y + a × sin(phi)/5;
          i8  := new Vertex(colour ⇒ D, x co ⇒ u, y co ⇒ v);
          i6  := new Vertex(colour ⇒ C,
                                        x co ⇒ u + a × cos(phi + 90) × sqrt(3)/24,
                                        y co ⇒ v + a × sin(phi + 90) × sqrt(3)/24);
          i7  := new Vertex(colour ⇒ C,
                                        x co ⇒ u + a × cos(phi − 90) × sqrt(3)/24,
                                        y co ⇒ v + a × sin(phi − 90) × sqrt(3)/24);
          e1  := (source ⇒ i2, sink ⇒ i5);
          e2  := (source ⇒ i4, sink ⇒ i6);
          e3  := (source ⇒ i6, sink ⇒ i8);
          e4  := (source ⇒ i7, sink ⇒ i8);
          e5  := (source ⇒ i5, sink ⇒ i7);
        end if;
end Generate p2;
```

Figure 3.1: Traditional Implementation of Genesis.

Using recursive descent and other familiar methods of compiler writers it is not difficult to write analysis and synthesis programs once one has routines for "choosing a production" and "finding an occurrence". The only difference between an analysis program and a traditional parsing program is that a test for "reaching the initial graphic I of a grammar" may be rather complicated; the usual distinction between terminal and non-terminal symbols in a formal language is irrelevant.

Comment:
The routine for finding an occurrence of a rule $B_1 \longleftarrow K \longrightarrow B_2$ corresponds to finding a morphism $B_1 \longrightarrow G$; in the implementation it corresponds to setting some of the actual parameters for a call of the function for the rule (the other parameters may be set by the user). In later sections we will describe database implementations and expert system implementations but they could have been included in this section as "traditional" implementations of graphics and their grammars.

#3.2 Concurrent implementations

If one wants to implement graphics and their grammars in a concurrent language like OCCAM/CSP with their processes, CCS with its agents, or ADA with its tasks, it is natural to identify *vertices* in a graphic with processes/agents/tasks. The edges in a graphic can be given by links/channels; the attributes of a vertex can be given by local variables of a process and the label of a vertex corresponds to the type of its process. Since a concurrent language can capture the inherent non-determinacy of grammars, it is not difficult to write analysis and synthesis programs once one has recognition and generation routines for the grammar rules. As Figure 1 shows, one needs a rather flexible programming language in order to write recognition and generation routines.

operation Generate p22 $(i1, i2, i4, i5, i6, i7, i8, a, \phi)$
condition old Task$(i1, A, x, y)$;
old Task$(i2, A, x + a\cos\phi; y + a\sin\phi)$
old Task$(i4, B, x + a\cos(\phi + 30)/\sqrt{3}, y + a\sin(\phi + 30)\sqrt{3})$
old Link$(i2, i4)$
action new Task$(i5, B, x + a\cos(\phi - 30)/\sqrt{3}, y + a\sin(\phi - 30)/\sqrt{3})$
new Task$(i8, D, u := x + a\cos(\phi)/5, v := y + a\sin(\phi)/5)$
new Task$(i6, C, u + a\cos(\phi + 90)\sqrt{3}/24, v + a\sin(\phi + 90)\sqrt{3}/24)$
new Task$(i7, C, u + a\cos(\phi - 90)\sqrt{3}/24, v + a\sin(\phi - 90)\sqrt{3}/24)$
new Link$(i2, i5)$ new Link$(i4, i6)$ new Link$(i6, i8)$;
new Link$(i7, i8)$ new Link$(i5, i7)$
end operation

operation Recognise p2 $(i1, i2, i4, i5, i6, i7, i8, a, \phi)$
condition old Task$(i1, A, x, y)$;

Please insert pages [redacted] 14+15+16+17
from manuscript already
with you.

References

(BC) M. Bauderon, B. Courcelle: "An algebraic formalism for graphs", in *CAAP '86, Springer LNCS* 214, 1986.

(Bu) H. Bunke: "Graph Grammars as a Generative Tool in Image Understanding", in *Springer LNCS* 153 (1983) 8-19.

(EH) H. Ehrig: "Introduction to the algebraic theory of graph grammars (a survey)" in *Springer LNCS 73 (1978) 1-64.*

(EHHB) H. Ehrig, A. Habel, U. Hummert, P. Boehm: "Towards algebraic datatype grammars: a junction between algebraic specifications and graph grammars", *Bull. EATCS* 29 (1986) 22-27.

(EKMRW) H. Ehrig, H.J. Kreowski, A. Maggiolo-Schettini, B.K. Rosen, J. Winkowski: "Transformations of structures: an algebraic approach", *Math. Sys. Th.* 14 (1981) 305-334.

(G) H. Gottler: "Attributed Graph Grammars for Graphics", in *Springer LNCS* 153 (1983) 130-142.

(GS) E.H. Lockwood, R.H. Macmillan: *Geometric Symmetry*, Cambridge Univ. Press, 1978.

(HM) R. Helm, K. Marriott: "Declarative Graphics", in *3rd Int. Conf. Logic Programming. Springer LNCS* 225 (1986).

(Ka) R. Kalaba, J.L. Casti (eds.): "Numerical grid Generation", *Applied Math. Computation* (1982) 1-895.

(L) P. Lescanne: "Computer experiments with the REVE Term Rewriting Systems Generator", *Proc. 10th Symp. Pr. Prog. Lang.* (1983) 99-108.

(Li) G. Rozenberg, A. Salomaa (eds.): *The Book of L*, North-Holland 1986.

(Ma) C.H. Macgillaray: *Fantasy & Symmetry. The Periodic Drawings of M.C. Escher*, Abrahs, N.Y. 1976.

(N) M. Nagl: *Graph-Grammatiken*, Braunschweig, Vieweg, 1979.

(Pa) P. Padawitz: "Graph grammars and operational semantics", *Th. Comp. Sci.* 19 (1982) 117-141.

(So) J.F. Sowa: *Conceptual Structures*, Addison-Wesley 1984.

Graphics and Their Grammars

L. Hess
Instituto Militar de Engenharia
Rio de Janeiro, Brasil

B.H. Mayoh
Computer Science Department, Aarhus University
Aarhus, Denmark

Abstract. Graphics are graphs with attributes at their vertices. Graphic grammars are natural extensions of graph and attribute grammars with rules that are attributed extensions of the "pushout" productions of graph grammars. The theory of graphic grammars is presented and various programming implementations will be discussed. Many motivating examples will be given, including

- the development of biological organisms
- the "semantic net" representation of expert system knowledge.

Keywords: Graphs, attributes, grammars.

Introduction

Graphics have been defined as graphs whose vertices have attributed values. When the assignment of values to vertices is allowed, graph models become even more useful than they already are. In figure 1 we indicate some possible applications of graphics and their grammars.

Once one accepts that graphics are a useful generalisation of graphs, it is natural to look for a definition of a graphic grammar that generalises both graph and attribute grammars appropriately. The two definitions of graphic grammars in the literature have chosen to generalise a form of graph rewriting that may be useful in certain applications but seems arbitrary even so. In Section 1 we base our definition of graphic grammars on graphic morphisms and pushouts. This form of graph rewriting seems to be the natural generalisation of string rewriting in attribute grammars. A graphic is a new representational entity, because its attributes can be Σ-algebra terms; graphic grammars can benefit from the results of the algebraic approach to graph grammars and Σ-algebras. In Section 2 we look at the use of graphics for biological organisms and the "data base and expert system" problem of representing real world knowledge. In Section 3 we describe various ways of "implementing" graphic grammars on a computer.

Image analysis & Generation:	Aircraft identification (Fu).
	Stochastic terrain models (HM).
	Scene Analysis (Bu).
Diagram analysis & Generation:	Program Structure (G).
	Structured pictures (HM).
	Interactive picture editor (HM).
	Program Animation (HM).
	Circuit diagrams (G).
	Art (N).
	Window manager (HM).
Numerical analysis:	Generating grids for solving DDE's (Ka).
Biological Models:	Morphogenesis (Li).
Knowledge representation:	Conceptual schemes in databases.
	Semantic nets in AI (So).

Figure 1: Applications of graphics.

This paper is not intended to be a presentation of the theory of graphics an their grammars — just suggestions for definitions, examples and how they can b implemented.

#1. Graphics and Their Grammars

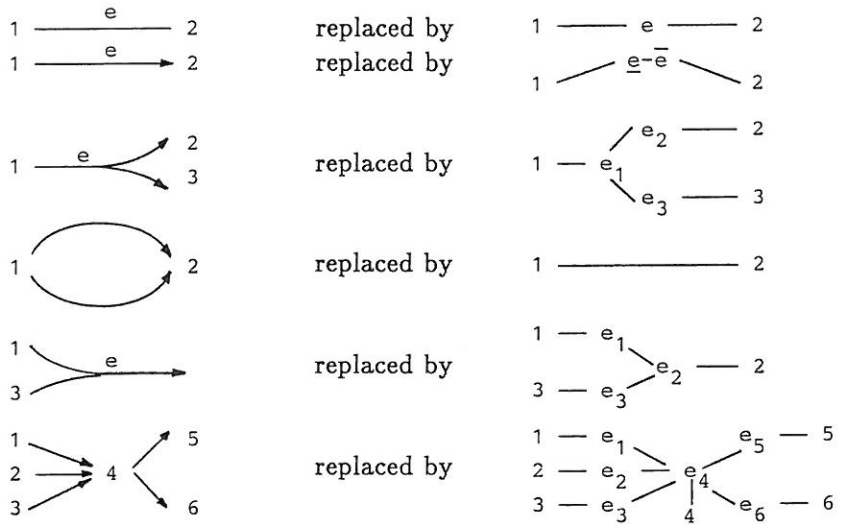
The natural generalisation of a graph is also the natural generalisation of th formal language concept of a word or string: a totally ordered set of vertice labelled by symbols from a finite alphabet. If we take a symmetric relation instea of a total ordering, we get:

Definition

A **graphic** G on an algebra A and label set W is a 4-tuple $G = (K, E, l, f)$ wher

K is a finite non-empty set of elements called vertices,
E is a finite set of unordered pairs of vertices called edges,
l is a function from K to W,
f is a function from K to A.

We seem to have lost generality by not allowing edge labels or attributes, directe edges, edges with more than two vertices, or multiple edges. This is not so becaus



we can introduce edge nodes as new vertices whose labels and attributes can capture ordering information. As examples of new nodes capturing edge information, consider the replacements. It is clear that edge replacing rules can be simulated by graphic productions and one can map a replacement graphic back to a graph with directed, labelled, attributed edges.

A **graphic morphism** $g : G \to G'$ consists of maps ver: $K \to K'$, edge: $E \to E'$, ~~lab: $V \to W_\ell$~~, att: $A \to A'$ such that att is a homomorphism.

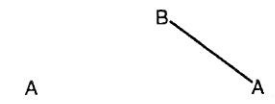
$$edge(v_1, v_2) = (ver(v_1), ver(v_2))$$
$$l'(ver(v)) = ~~lab~~(l(v))$$
$$f'(ver(v)) = att(f(v)).$$

We say that G is a **subgraphic** of G' ($g : G \to G'$ is an *embedding*) when "ver" is an injection and ~~"lab" is identity~~.

Comment:
With this definition of morphism, graphics form a category when the algebras form a category. We could specify the graphic category as a subcategory of STRUCT (EKMRW). Graphic morphisms are natural combinations of graph morphisms and morphisms of algebras. The definition of subgraphic is a natural generalisation of the definition of subgraph. We will be particularly interested in the case when the category of algebras is the much studied Σ-algebra for some signature Σ.

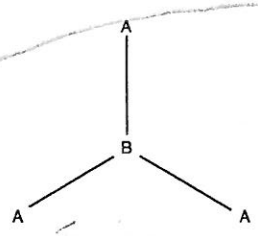
(on the algebra $T(\Sigma \cup V)/\approx$ where $\approx$ is given by the equations for evaluating

Example:
The graphic on the term algebra $T(\Sigma \cup V)$ – where $\Sigma = (R^2, +, \times, \sqrt{}, \sin, \cos, 0, \ldots)$ and $x, y, a, \phi \in V$ –



$K = \{1,2,4\}$ $E = \{(2,4)\}$
$l(1) = A$ $f(1) =< x, y >$
$l(2) = A$ $f(2) =< x + a\cos\phi, y + a\sin\phi >$
$l(4) = B$ $f(4) =< x + a\cos(\phi + 30)/\sqrt{3},$
$y + a\sin(\phi + 30)/\sqrt{3} >$

is a subgraphic of the graphic



$K' = \{1,2,3,4\}$ $E' = \{(1,4),(2,4),(3,4)\}$
$l'(1) = A$ $f'(1) = (0,0)$
$l'(2) = A$ $f'(2) = (a,0)$
$l'(3) = A$ $f'(3) = (a/2, a\sqrt{3}/2)$
$l'(4) = B$ $f'(4) = (a/2, a\sqrt{3}/6)$

~~on the same algebra~~ because we have the graphic morphism

$$ver(v) = v \quad ~~lab(l) = l~~ \quad att(u,v) = (Su, Sv)$$

where S is given by subsituting 0 for x, y and ϕ. The trigonometric expressions in the subgraphic give the projections of an edge to the x- and y-axes.

In this section all graphics will have two spatial attributes which are given by the representation of the graphic as a diagram. Notice that all graphics have the empty graphic 0 as a subgraphic. The empty graphic is useful for removing vertices.

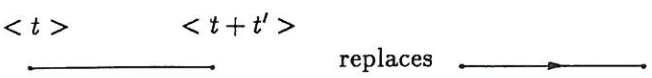
Definition

A graphic sequential rewriting system $\Gamma = (V, \Sigma, I, P)$ consists of

V, a finite non-empty set of symbols
Σ, a signature
I, the start graphic on some Σ-algebra
P, a non-empty set of productions.

A **production** $p = B_1 \xleftarrow{B_1} K \xrightarrow{B_2} B_2$ is a pair of attribute preserving graphic morphisms connecting three graphics whose vertices have attribute values in $T(\Sigma \cup V)$ – terms formed from variables V and the operations in the signature Σ.

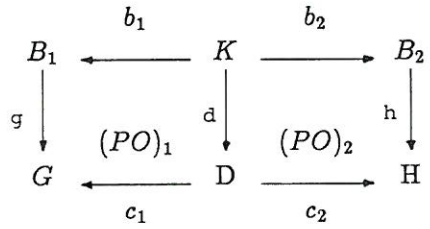
Comment:
One might well disagree with our requirement that the attribute values in B_1, K, B_2 are given by terms in $T(\Sigma \cup V)$. This requirement is connected with our choice of morphism rewriting; it ensures that graph pushouts coincide with graphic pushouts. The requirement also allows the attribute values to carry edge information; if a directed graph is partially ordered, the edge ordering can be captured by



Not only "later", "above" and "to the right of" are examples of this, but also many orderings in database and AI semantic nets.

Definition

A direct derivation $G \underset{p}{\Rightarrow} H$ is defined by giving a production, a context graphic D, a graphic morphism $K \to D$, and two pushouts $(PO)_1$ and $(PO)_2$ in the category of graphics and graphic morphisms defined above.



We say that the graphic H is derived from G by the production p, based on the morphism $g : B_1 \to G$.

A graphic language is defined as the set of all graphics derived from the initial graphic by productions in P.

Comment:
The attribute components in the vertical morphisms g, d, h need not be identities, they can be the same term evaluation from $T(\Sigma \cup V)$ to a Σ-algebra A. As explained in [Eh, p. 11], the pushout of two graph morphisms f: $K \to B$ and $g : K \to D$ is given by "gluing together the items $f(k)$ in B and $g(k)$ in D for each item k in K". For labelled graphs we want

$$l_G([x]) = \textbf{if } x \in B \textbf{ then } l_B(x) \textbf{ else } l_D(x)$$

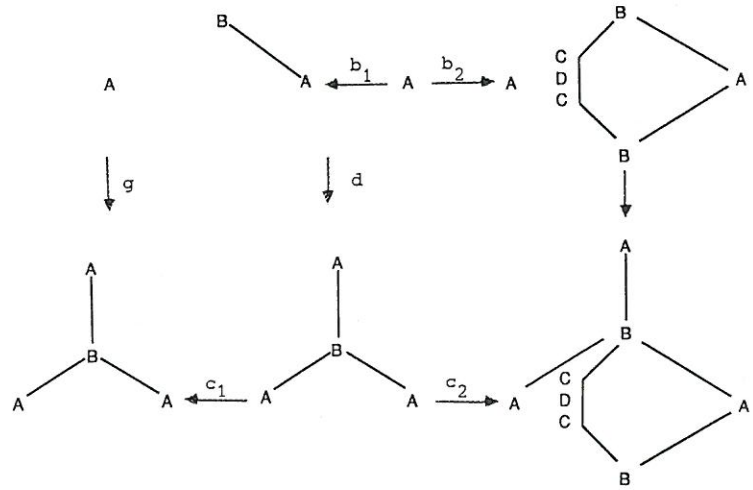
for each vertex $[x]$ in G, the pushout graph of f and g. This is a proper definition if $l_B(f(k)) = l_D(g(k))$ for all k in K (satisfied when f and g are graphic morphisms), and we try to define attributes for vertices in the pushout graphic G by

$$(+) \qquad f_G([x]) = \textbf{if } x \in B \textbf{ then } f_B(x) \textbf{ else } f_D(x)$$

for each vertex $[x]$ in G. This works when $f_B(f(k)) = f_D(g(k))$ for all k in K (satisfied when f and g are attribute preserving). We would like a more general way of defining attribute values in the pushout graphic. In a later paper we will study the case of "productions" whose graphic morphisms are not attribute preserving.

Comment:
The reader is invited to reformulate the above discussion of graphic pushouts in the terminology of the algebraic approach to general structures (EKMRW, Pa).

Example:
Perhaps we should give an example of a direct derivation.

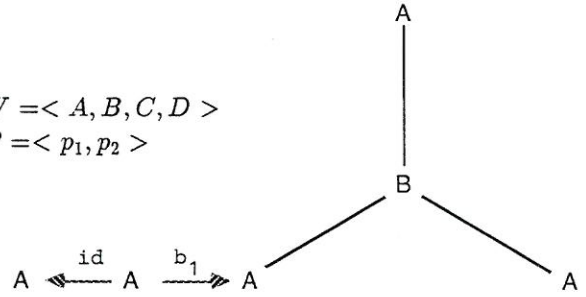


All the morphisms in this diagram are embeddings (sources are subgraphics of sinks); the top two morphisms give a production and the morphism g was described in our previous example. The left square is a pushout when d assigns the coordinates $< 0, 0 >$ to the lower right A in its target. The right square is also a pushout with this assignment of values to attributes. One may well ask what has happened to the parameters a and ϕ in the previous example. The answer is that these parameters fix the morphisms, b_1 and b_2 – they parametrise a family of productions.

Example:
A graphic grammar to describe a "wall paper" pattern with planegroup p 31m [Ma]

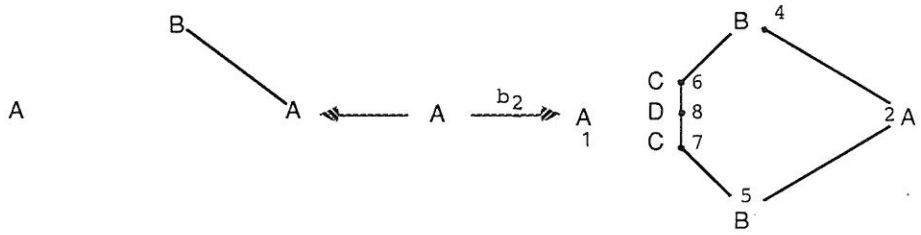
$G =< V, I, P >$ $W =< A, B, C, D >$
$I = A < 0, 0 >$ $P =< p_1, p_2 >$

p1:



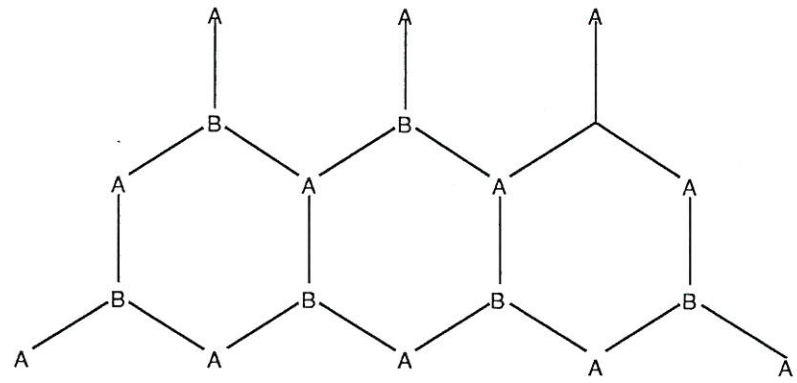
$K = \{1\}$ $K = \{1, 2, 3, 4\}$
$E = \{\}$ $E = \{< 1, 4 >, < 2, 4 >, < 3, 4 >\}$
$L(1) = A$ $L(1) = A \;\; L(2) = A \;\; L(3) = A \;\; L(4) = B$
$f(1) =< x, y >$ $f(1) =< x, y > \;\; f(2) =< x + a, y >$
$f(3) =< x + a/2, y + \sqrt{3}a/2 >$
$f(4) =< x + a/2, y + \sqrt{3}a/6 >$

p2:

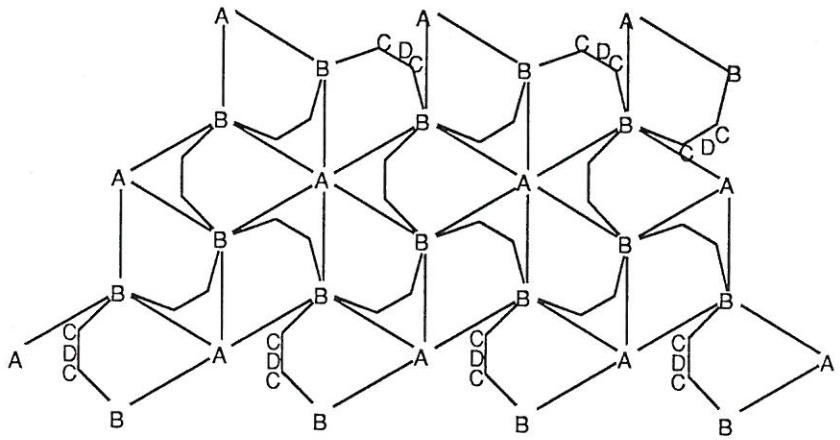


$K = \{1, 2, 4\}$ $K = \{1, 2, 4, 5, 6, 7, 8\}$
$E = \{2, 4\}$ $E = \{< 2, 4 >, < 4, 6 >, < 6, 8 >, < 8, 7 >, < 7, 3 >, < 5, 2 >\}$
$L(1) = A \;\; L(2) = A \;\; L(4) = B \;\; L(5) = B \;\; L(6) = C \;\; L(7) = C \;\; L(8) = D$
$f(1) =< x, y > \;\; f(2) =< x + a\cos\phi, y + a\sin\phi >$
$f(4) =< x + a\cos(\phi + 30)/\sqrt{3}, y + a\sin(\phi + 30)/\sqrt{3} >$
$f(5) =< x + a\cos(\phi - 30)/\sqrt{3}, y + a\sin(\phi - 30)/\sqrt{3} >$
$f(8) =< x + a\cos\phi/5, y + a\sin\phi/5 >$
$f(6) =< x + a\cos(\phi)/5 + a\cos(\phi + 90)\sqrt{3}/24, y + a\sin\phi/5 + a\sin(\phi + 90)\sqrt{3}/24 >$
$f(7) =< x + a\cos(\phi)/5 + a\cos(\phi - 90)\sqrt{3}/24, y + a\sin(\phi - 90)\sqrt{3}/24 >$

Repeated use of the production p1 gives the pattern:



Repeated use of the production p2 now gives:



We have described the derivation of a graphic G in a graphic language as if the attributes were evaluated during the derivation. However, there is no objection to first deriving the underlying graph of G in the graph language, corresponding to L, then deriving the attribute values. This corresponds to the usual evaluation method in "string" attribute grammars: first derive the string in the underlying context-free grammar, then solve the attribute equations.

Final remark:
If one has an Σ-morphism μ to any Σ-algebra A from the Σ-algebra of the initial graphic of a grammar, then one gets a transformation of the graphic language by applying μ at every step of every derivation in the language. This captures geometric transformations of pictures and much else.

#2. Reality Models

Are graphics useful models of reality?

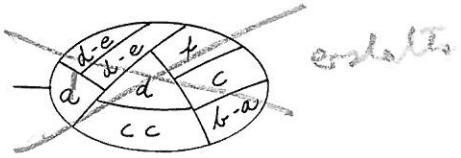
Graph models have become popular, and they become more accurate and convenient when vertices can have attribute values. However, the usefulness of graph grammars is strongly dependent on the kinds of rewriting rules that are allowed. "Push out" rewriting gives a flexibility that single morphism rewriting (identity morphisms from K to B_1, and D to G) and simple node and/or edge replacement rules do not have.

It is well established that the development of biological organisms (Li) can be modelled by graphs and their grammars and Example 1 shows that the extension to graphics gives more natural models. In the development of expert systems semantic nets are a popular way of representing "real world knowledge" (Sowa) and Example 2 shows that graphics are a convenient formalism for semantic nets.

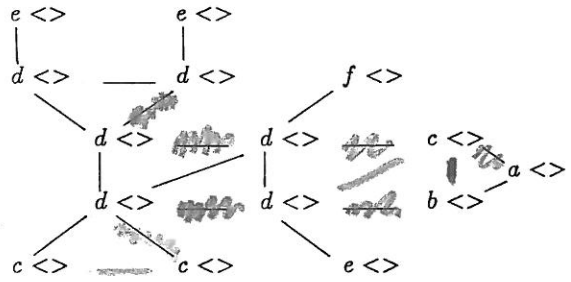
Example 1: "Phascum Cuspidatum"

The grammar for the development of the organism "Phascum Cuspidatum" and the graphic for a particular life stage is shown in

I: a



P: $a \Leftarrow a \Rightarrow b-a$
$b \Leftarrow b \Rightarrow c$
$c \Leftarrow c \Rightarrow d-e$
$d \Leftarrow d \Rightarrow d$
$e \Leftarrow e \Rightarrow f$
$f \Leftarrow f \Rightarrow e$

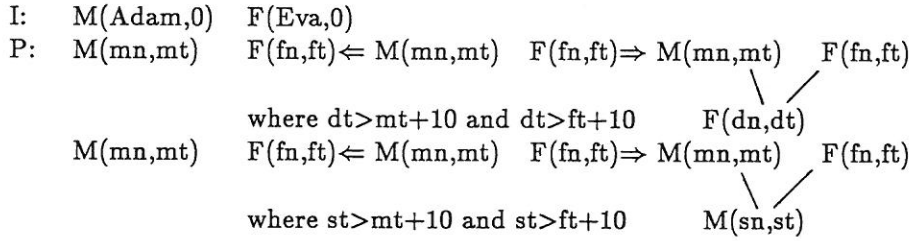


As the attributes of the "Phascum Cuspidatum" cells are not used in this graphic grammar, it is no more than a traditional graph grammar. However, this grammar illustrates two points that are valuable in biological modelling.

- Grammars work best when there are only a finite number of cell labels, but the development of an organism is most naturally described in terms of infinitely many states, and graphic attributes can capture this infinity.
- Grammars usually give an unambiguous result when a rule is applied, but the development of an organism is most naturally described by allowing very flexible applications of rules, and graphic attributes can capture this flexibility.

Example 2: "Semantic Nets"

A grammar for the representation of human relationships and the graphic for a particular family are shown in



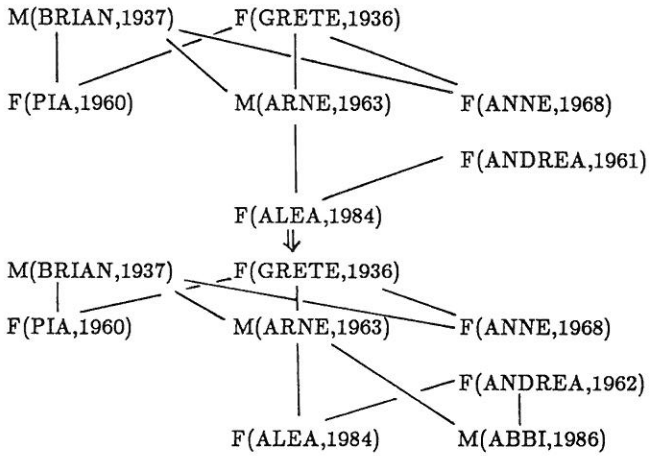


Figure 2.1: The Graphic Grammar Genesis.

The application of the second rule in this example is flexible because the fixing of the parameters – mn=Arne, mt=1963, fn=Andrea, ft=1962 – did not determine the attributes: sn=Abbi and st=1968. The example is instructive because it shows

how "time" attributes allow graphics to capture natural orderings which would be expressed by directed edges in graph models. Other examples would show how graphics can capture other kinds of edge information that expert system builders put into semantic nets. (Remember the discussion in Section 1 on edge information.)

Example 3: Erasing

In (So) Sowa introduced a form of semantic nets called conceptual graphs for the graphic grammar in Example 2, and he described the powerful operations for extracting information and manipulating conceptual graphs. All these operations can be extended to graphics. To convince doubters we show how edges and vertices can be removed ... consider the addition of rules

M(mn,mt)⇐0⇒		– vertex removal		
F(fn,ft)⇐0⇒0		– using the empty graphic 0		
M(mn,mt)-F(fn,ft)	⇐ M(mn,mt)	F(fn,ft)	⇒ M(mn,mt)	F(fn,ft)
M(mn,mt)-M(n,t)	⇐ M(mn,mt)	M(n,t)	⇒ M(mn,mt)	M(n,t)
F(fn,ft)-F(n,t)	⇐ F(fn,ft)	F(n,t)	⇒ F(fn,ft)	F(n,t)

The last three rules give edge removal.

#3. Implementation

Once one accepts that graphics and their grammars are useful models of reality, one wants to implement them on the computer; one wants an *analysis* program to recognise whether a given graphic can be generated by a given grammar, and one wants a *synthesis* program for generating graphics from grammars. The many ways of implementing graphics and their grammars can be grouped into – traditional, concurrent, logical and syntactical – so we shall look at each of these in turn.

#3.1 Traditional implementations

If one wants to implement graphics and their grammars in an imperative language like PASCAL or an object oriented language like SmallTalk, it is natural to represent *graphics as data objects*. It is also natural to represent a grammar rule by a pair of routines – a recognition "bottom-up" routine to substitute the right side

for the left side, and a generation "top-down" routine to substitute the left side for the right side.

```
type      Vertex
          access record; colour: V; x co, y co: Real
          end record;

type      Edge   is record source, sink: vertex
                 end record;

procedure Generate p2 ( in i1, i2, i4: Vertex; e: Edge; a, phi: Real
                        out i5, i6, i7, i8: Vertex; success: Boolean
                            e1, e2, e3, e4, e5: Edge);
          x, y, u, v: Real;


  begin x       := i1.x co; y := i1.y co;
        success := (i1.colour = A) and (e.source = i2) and (e.sink = i4)
                   and (i2.colour = A) and (i4.colour = B)
                   and (i2.x co = x + a × cos phi)
                   and (i2.y co = x + a × sin phi)
                   and (i4.x co = x + a × cos(phi + 30)/√3)
                   and (i4.y co = x + a × sin(phi + 30)/√3);


    if success then
      i5  := new Vertex(colour ⇒ B; x co ⇒ x + a cos(phi − 30)/√3);
             y co ⇒ x + a sin(phi − 30)/√3);
      u   := x + cos(phi)/5; v := y + a × sin(phi)/5;
      i8  := new Vertex(colour ⇒ D, x co ⇒ u, y co ⇒ v);
      i6  := new Vertex(colour ⇒ C,
                                  x co ⇒ u + a × cos(phi + 90) × sqrt(3)/24,
                                  y co ⇒ v + a × sin(phi + 90) × sqrt(3)/24);
      i7  := new Vertex(colour ⇒ C,
                                  x co ⇒ u + a × cos(phi − 90) × sqrt(3)/24,
                                  y co ⇒ v + a × sin(phi − 90) × sqrt(3)/24);
      e1  := (source ⇒ i2, sink ⇒ i5);
      e2  := (source ⇒ i4, sink ⇒ i6);
      e3  := (source ⇒ i6, sink ⇒ i8);
      e4  := (source ⇒ i7, sink ⇒ i8);
      e5  := (source ⇒ i5, sink ⇒ i7);
    end if;
end Generate p2;
```

Figure 3.1: Traditional Implementation of Genesis.

Using recursive descent and other familiar methods of compiler writers it is not difficult to write analysis and synthesis programs once one has routines for "choosing a production" and "finding an occurrence". The only difference between an analysis program and a traditional parsing program is that a test for "reaching the initial graphic I of a grammar" may be rather complicated; the usual distinction between terminal and non-terminal symbols in a formal language is irrelevant.

Comment:
The routine for finding an occurrence of a rule $B_1 \longleftarrow K \longrightarrow B_2$ corresponds to finding a morphism $K \longrightarrow D$; in the implementation it corresponds to setting some of the actual parameters for a call of the function for the rule (the other parameters may be set by the user). In later sections we will describe database implementations and expert system implementations but they could have been included in this section as "traditional" implementations of graphics and their grammars.

#3.2 Concurrent implementations

If one wants to implement graphics and their grammars in a concurrent language like OCCAM/CSP with their processes, CCS with its agents, or ADA with its tasks, it is natural to identify *vertices* in a graphic with processes/agents/tasks. The edges in a graphic can be given by links/channels; the attributes of a vertex can be given by local variables of a process and the label of a vertex corresponds to the type of its process. Since a concurrent language can capture the inherent non-determinacy of grammars, it is not difficult to write analysis and synthesis programs once one has recognition and generation routines for the grammar rules. As Figure 1 shows, one needs a rather flexible programming language in order to write recognition and generation routines.

operation Generate p22 $(i1, i2, i4, i5, i6, i7, i8, a, \phi)$
condition old Task$(i1, A, x, y)$;
old Task$(i2, A, x + a\cos\phi; y + a\sin\phi)$
old Task$(i4, B, x + a\cos(\phi + 30)/\sqrt{3}, y + a\sin(\phi + 30)\sqrt{3})$
old Link$(i2, i4)$
action new Task$(i5, B, x + a\cos(\phi - 30)/\sqrt{3}, y + a\sin(\phi - 30)/\sqrt{3})$
new Task$(i8, D, u := x + a\cos(\phi)/5, v := y + a\sin(\phi)/5)$
new Task$(i6, C, u + a\cos(\phi + 90)\sqrt{3}/24, v + a\sin(\phi + 90)\sqrt{3}/24)$
new Task$(i7, C, u + a\cos(\phi - 90)\sqrt{3}/24, v + a\sin(\phi - 90)\sqrt{3}/24)$
new Link$(i2, i5)$ new Link$(i4, i6)$ new Link$(i6, i8)$;
new Link$(i7, i8)$ new Link$(i5, i7)$
end operation

operation Recognise p2 $(i1, i2, i4, i5, i6, i7, i8, a, \phi)$

condition old Task$(i1, A, x, y)$;
old Task$(i2, A, x + a\cos\phi,\ y + a\sin\phi)$
old Task$(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})$
old Task$(i5, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)/\sqrt{3})$
old Task$(i8, D, u := x + a\cos\phi/5,\ v := y + a\sin\phi/5)$
old Task$(i6, C, u + a\cos(\phi + 90)\sqrt{3}/24,\ v + \sin(\phi + 90)\sqrt{3}/24)$
old Task$(i7, C, u + a\cos(\phi - 90)\sqrt{3}/24,\ v + \sin(\phi - 90)\sqrt{3}/24)$
old Link$(i2, i4)$ old Link$(i2, i5)$ old Link$(i4, i6)$
old Link$(i6, i8)$ old Link$(i7, i8)$ old Link$(i5, i7)$
action drop Link$(i2, i5)$ dropLink$(i4, i6)$
drop Link$(i6, i8)$ drop Link$(i7, i8)$ drop Link$(i5, i7)$
drop Task$(i5)$ drop Task$(i6)$ drop Task$(i7)$ drop Task$(i8)$
end operation

Figure 3.2: Concurrent Implementation of Genesis.

#3.3 Logical implementations

If one wants to implement graphics and their grammars in a logic programming language like PROLOG, it is natural to represent graphics as sets of atomic formulas. One way of capturing edges in a graphic is to index the vertices and connect the indices by atomic formulas.

Since grammar rules produce new graphics from old, the formulas that were true before applying the rule may be false afterwards. This non-monotonicity can be captured in three ways:

(A) Introducing a new "stage" attribute in all formulas, and expressing the grammar rule by a logical implication.

(B) Using non-logical operators like *assert* and *retract* in PROLOG.

(C) Moving to the metalevel.

Because of the "frame" problem (A) seems unworkable, so let us illustrate (B) and (C) in our semantic net example:

(A) Current graphic $P(1, A, 0).\ P(2, A, 6, 0).$
$P(3, A, 3, 3\sqrt{3}).\ P(4, B, 3, \sqrt{3}).$
$E(1, 4).\ E(2, 4).\ E(3, 4).$

(B) Generate-p2 $(i1, i2, i4, a, \phi) :- P(i1, A, x, y)\ P(i2, A, x + a\cos\phi,\ y + a\sin\phi)$
$P(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})$

$E(i2, i4)$
assert $P(< i1, i2, i4, 5 >, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)/\sqrt{3})$
assert $P(< i1, i2, i4, 8 >, D, x + a\cos\phi/5,\ y + a\sin\phi/5)$
assert $P(< i1, i2, i4, 6 >, C, x + a\cos\phi/5 + a\cos(\phi + 90)\sqrt{3}/24,$
$y + a\sin\phi/5 + a\sin(\phi + 90)/\sqrt{3}/24)$
assert $P(< i1, i2, i4, 7 >, C, x + a\cos\phi/5 + a\cos(\phi - 90)/\sqrt{3}/24,$
$y + +a\sin\phi/5 + a\sin(\phi - 90)/\sqrt{3}/24)$
assert $E(< i1, i2, i4, 6 >, < i1, i2, i4, 8 >)$
assert $E(< i1, i2, i4, 7 >, < i1, i2, i4, 8 >)$
assert $E(< i1, i2, i4, 5 >, < i1, i2, i4, 7 >)$

Recognise-p2 $(i1, i2, i4, i5, i6, i7, i8, a, \phi) :- P(i1, A, x, y)$
$P(i2, A, x + a\cos\phi,\ y + a\sin\phi)$
$P(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})\ E(i2, i4)$
$P(i5, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)/\sqrt{3})\ E(i2, i5)$
u is $x + a\cos\phi/5$ v is $y + a\sin\phi/5$, $P(i8, D, u, v)$
$P(i6, C, u + a\cos(\phi + 90)\sqrt{3}/24,\ v + a\sin(\phi + 90)\sqrt{3}/24)$
$P(i7, C, u + a\cos(\phi - 90)\sqrt{3}/24,\ v + a\sin(\phi - 90)\sqrt{3}/24)$
$E(i6, i8)\ E(i8, i7)\ E(i5, i7)$
retract $P(i5, B, u + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3})$
retract $P(i8, D, u, v)$ retract $E(i2, i5)$ retract $E(i5, i7)$
retract $P(i6, C, u + a\cos(\phi + 90)\sqrt{3},\ v + a\sin(\phi + 90)\sqrt{3}/24)$
retract $P(i7, C, u + a\cos(\phi - 90)\sqrt{3},\ v + a\sin(\phi - 90)\sqrt{3}/24)$
retract $E(i4, i6)$ retract $E(i6, i8)$ retract $E(i8, i7)$

(C) Gen-p2 (G,Union(G,Assertions)) :- In(G,Union($P(i1, A, x, y) \ldots E(i2, i4)$)).
Rec-p2 (Union(G,Retractions),G) :- In(G,Union($P(i1, A, x, y) \ldots E(i5, i7)$)).

Figure 3.3: Logical Implementation of Wallpaper Rule.

As numbering graphic vertices is inelegant, the reader is probably unimpressed by this example. Much more impressive examples of the elegance and power of logical programming languages for "declarative graphics" can be found in (HM). There the authors distinguish between specification and instances of "pictures"; a picture instance $P < G, R >$ is given by adding geometric G and restriction R transformations to a specification P of a picture (see final remark in Section 1). There are two ways of combining picture specifications

Composition $P \longleftarrow R_1, \ldots, R_m, P_1 \& \ldots \& P_n$
Collection $P \longleftarrow \{P \& \ldots \& P_n\}$.

Collection allows independent pictures to be grouped together, whereas composition combines pictures that are interrelated by the shared attributes and relations: $R_1 \ldots R_m$. The authors present algorithms for recognising pictures, generating pictures, and breaking pictures into their subpictures, so they can implement rules

and graphic grammars. However, there does seem to be a tension between our rewriting approach and their algebraic approach; we intend to reduce this tension by introducing "graphic operators" like & into our theory of graphics.

#3.4 Syntactic implementations

Sometimes the most natural way to implement a graphic grammar is to use a term rewriting system like REVE (L). In such a system a graphic is represented by a term with components for vertices and edges. A grammar rule is represented by a "conditional" rewrite rule; rewriting in one direction gives a generation routine; rewriting rules for rearranging terms are used to find occurrences of grammar productions.

- generation rule
$P(i1, A, x, y);\ P(i3, A, x_a\cos\phi,\ y + a\sin\phi);$
$P(i4, B, x + a\cos(\phi + 30)/\sqrt{3},\ y + a\sin(\phi + 30)/\sqrt{3});\ E(i2, i4)$
$\Rightarrow P(i1, A, x, y) \ldots E(i1, i4)$
$P(< i1, i2, i4, 5 >, B, x + a\cos(\phi - 30)/\sqrt{3},\ y + a\sin(\phi - 30)) \ldots$
$E(< i1, i2, i4, 5 >, < i1, i2, i4, 7 >)$

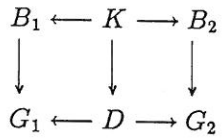
- rearrangement rules
$E(i, j);\ E(k, l) \Rightarrow E(k, l);\ E(i, j)$ $\qquad E(i, j) \Rightarrow E(j, i)$
$E(i, j);\ P(k, l, x, y) \Rightarrow P(k, l, x, y);\ E(i, j)$
$P(k, l, x, y);\ P(kk, ll, xx, yy) \Rightarrow P(kk, ll, xx, yy);\ P(k, l, x, y)$

Figure 3.4: Syntactic Implementation of Wallpaper Grammar.

We have chosen to present the rewriting rules in the synthesis direction. The last four rules do not change the underlying graphic, but they have to be there for the term rewriting system to simulate all derivations in the graphic grammar. If we reverse all the rewriting rules and stop when we derive the term I we can analyse any graphic in the language given by the grammar.

Can every graphic grammar be simulated so accurately by a term rewriting system? This would be true if we could show that graphic grammars are algebraic data type grammars (EHHB). To show this we have to:

- Find an abstract data type specification whose initial algebra is isomorphic to the set of graphics.
- Describe how graphic productions $B_1 \longleftarrow K \longrightarrow B_2$ can be converted into a pair $< t_1, t_2 >$ of Σ-terms such that:



if and only if $G_1 = [D_1(t_1)]$ and $G_2 = [D(t_2)]$ for some Σ-term $D(x)$.

The version of graph grammars, shown to be algebraic data type grammars in (BC), was:

1. Edges are node strings with labels.
2. Nodes can have labesl and attributes in the form of unary edges.
3. All productions have a finite discrete graph $\bar{n}$ as K.

Graphics satify (1) (2), "V_1V_2 is an edge $\rightarrow$ is unlabelled & V_2V_1 is an edge" and "only unary and binary edges". Productions in graphic grammars are not required to satisfy (3), but the generative power of a graphic grammar is not affected if we enforce (3) by dropping all edges from K (in the application of the production these edges disappear from D but reappear in G_1 and G_2 because they are still present in B_1 and B_2).

Final remark

One can also implement graphics and their grammars when one has access to a powerful data base system or a "system for building expert systems" of the kind now flooding the market. In such systems graphics can be represented by knowledge bases, and grammar rules can be represented by transactions. In a data base system a transaction is a sequence of find, insert, delete and modify actions; in an expert system a transaction is a production rule of the form "if condition then action".

Acknowledgements

The first author would like to thank CNP and IBM-Brasil for supporting her travel to the workshop. Both authors would like to thank the editors for suggesting improvements to the presented paper.

References

(BC) M. Bauderon, B. Courcelle: "An algebraic formalism for graphs", in *CAAP '86, Springer LNCS* 214, 1986.

(Bu) H. Bunke: "Graph Grammars as a Generative Tool in Image Understanding", in *Springer LNCS* 153 (1983) 8-19.

(EH) H. Ehrig: *"Introduction to the algebraic theory of graph grammars (a survey)"* in Springer LNCS 73 (1978) 1-64.

(EHHB) H. Ehrig, A. Habel, U. Hummert, P. Boehm: "Towards algebraic datatype grammars: a junction between algebraic specifications and graph grammars", *Bull. EATCS* 29 (1986) 22-27.

(EKMRW) H. Ehrig, H.J. Kreowski, A. Maggiolo-Schettini, B.K. Rosen, J. Winkowski: "Transformations of structures: an algebraic approach", *Math. Sys. Th.* 14 (1981) 305-334.

(G) H. Gottler: "Attributed Graph Grammars for Graphics", in *Springer LNCS* 153 (1983) 130-142.

(GS) E.H. Lockwood, R.H. Macmillan: *Geometric Symmetry*, Cambridge Univ. Press, 1978.

(HM) R. Helm, K. Marriott: "Declarative Graphics", in *3rd Int. Conf. Logic Programming. Springer LNCS* 225 (1986).

(Ka) R. Kalaba, J.L. Casti (eds.): "Numerical grid Generation", *Applied Math. Computation* (1982) 1-895.

(L) P. Lescanne: "Computer experiments with the REVE Term Rewriting Systems Generator", *Proc. 10th Symp. Pr. Prog. Lang.* (1983) 99-108.

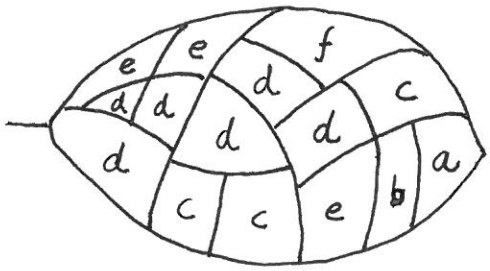
(Li) G. Rozenberg, A. Salomaa (eds.): *The Book of L*, North-Holland 1986.

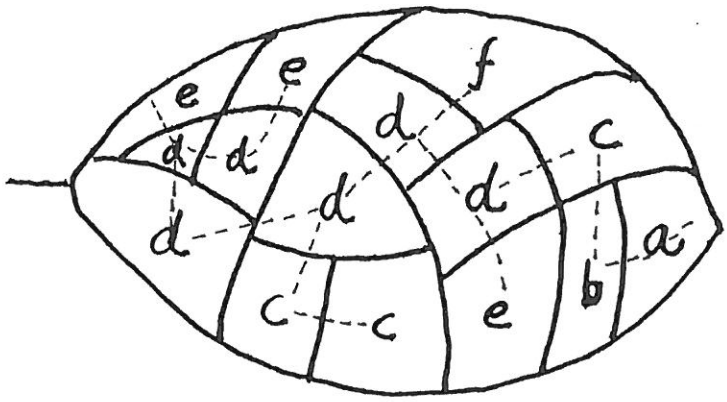
(Ma) C.H. Macgillaray: *Fantasy & Symmetry. The Periodic Drawings of M.C. Escher*, Abrahs, N.Y. 1976.

(Pa) P. Padawitz: "Graph grammars and operational semantics", *Th. Comp. Sci.* 19 (1982) 117-141.

(So) J.F. Sowa: *Conceptual Structures*, Addison-Wesley 1984.

(N) M Nagl Graph-Grammatiken Braunschweig, Vieweg, 1979


e
e
f
d
d
c
d
d
d
d
d
c
c
e
b
a


e
e
f
d
d
c
d
d
d
d
d
c
c
e
b
a

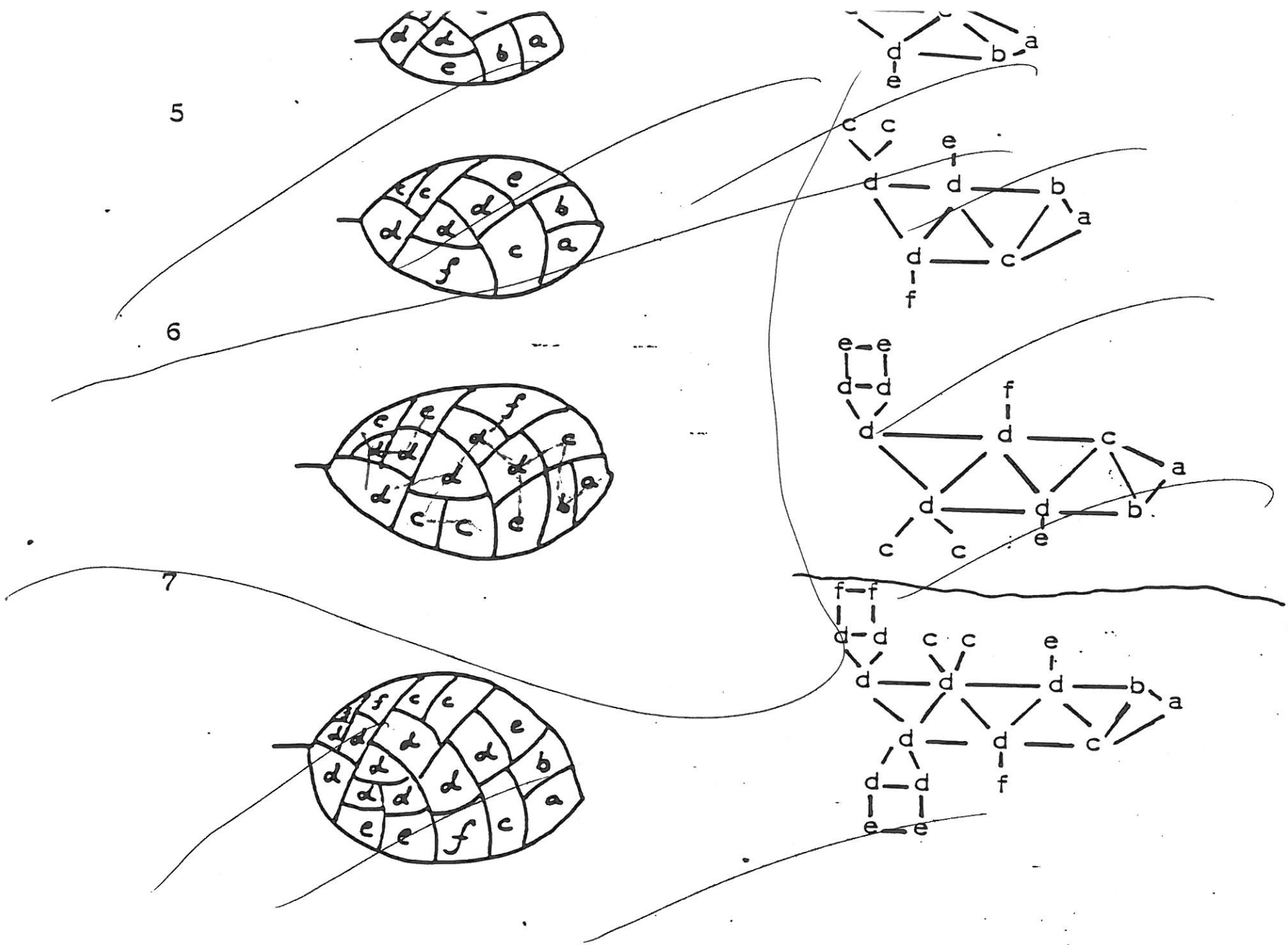


Figure 4. The web model for Figure 3