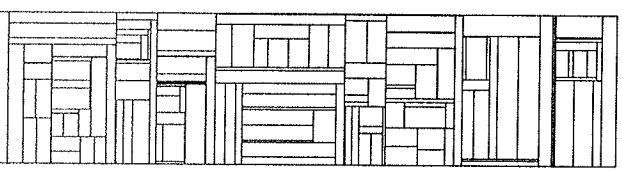
ISSN 0105-8517

Graph Grammars for Knowledge Representation

Lilia Hess
Brian H. Mayoh

DAIMI PB – 304
February 1990

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark



Graph Grammars for Knowledge Representation

Abstract

Two papers to be presented at the March 1990 GRAGRA meeting in Bremen: the more general

"Representation of knowledge using graph grammars"

which argues for graphs as the universal KR formalism. The more specific

"The four musicians: analogies and expert systems - a graphic approach"

which demonstrates the use of graphics for type inheritance and analogical reasoning.

Representation of knowledge using graph grammars

Many ingenious ways of representing knowledge have been devised and incorporated in "knowledge based" programs-for surveys see (Enc, Th ch.3). However only some of these KR techniques have been formalised. Usually logic is used for these formalisations, and the author has suggested "institutions" as a universal formalism (Ma), but graphs seem to be an attractive alternative. One of the attractions is that rules in graph grammars can be more expressive than logical proof rules so that nonmonoticity and other logical troubles are less apparent. The first two sections of this paper demonstrate this expressiveness by surveying some of the ways graphs can represent knowledge. Section 1 starts with a survey of various kinds of graphs, and section 2 starts with a discussion of 'dynamic' graphs where the vertices or edges represent actions, processes, procedures or productions. Section 3 gives a solution to the tricky problem of "when can a graph rewriting rule be applied to a graph?". Section 4 compares the graph and logic approachs to knowledge representation. Section 5 is devoted to the graph approach to uncertainty. Throughout the paper we use MapSee as the running example because it was the running example in a recent paper(RM) that argued for logic as a universal formalism.

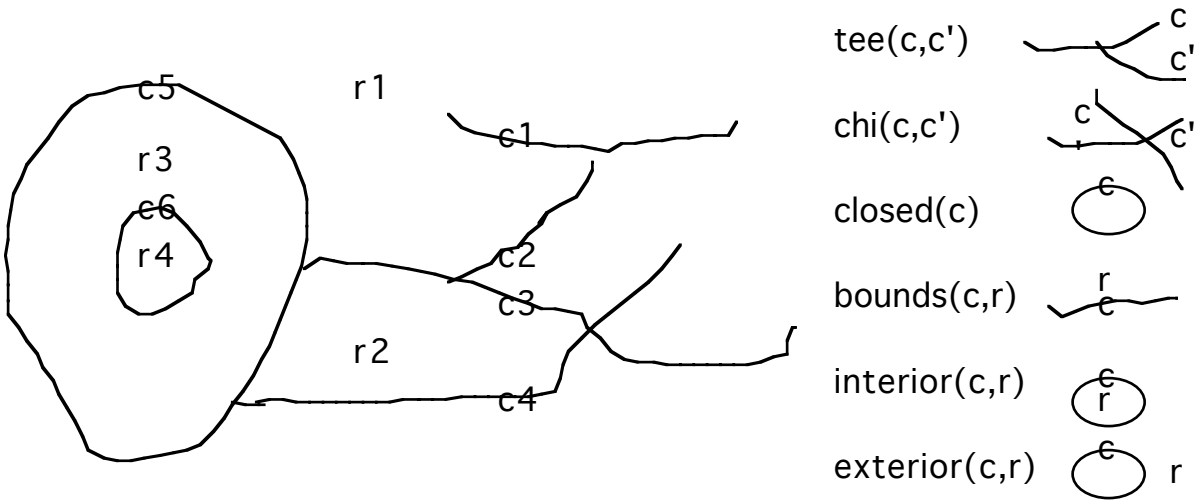


fig.1 Sketch map

MapSee is used to illustrate the representation of knowledge in data-bases(#1.1), semantic nets(#1.2,#2.2), type inheritance(#1.3), conceptual structures(#1.4), logical programs(#1.5), planning(#2.1), hypermedia(#2.3), simulation(#2.4), and linguistic attribute grammars(#2.5).

#1 Representation of static knowledge

Let us start by trying to bring some order in the wide variety of mathematical objects that have been called labelled graphs.Let us agree on the name index for a pair of label sets <VL,EL>. EL and VL may be ordered sets and even have operations. In our theory label morphisms from (VL,EL) to (VL',EL') will play an important role, particularly the morphisms given by type assignments. A label morphism is a pair of functions,

la = <vla:VL->VL', ela:EL->EL'>

that satisfy various requirements. These requirements depend on what one means by a graph G over index <VL,EL>. For any kind of labelled graph G one has two functions

lab: Vertex(G) -> VL edge: Edge(G) -> EL

where Vertex(G) is a set of vertices and Edge(G) is a set of edges. But what is an edge? Possible answers are shown in figure 2.

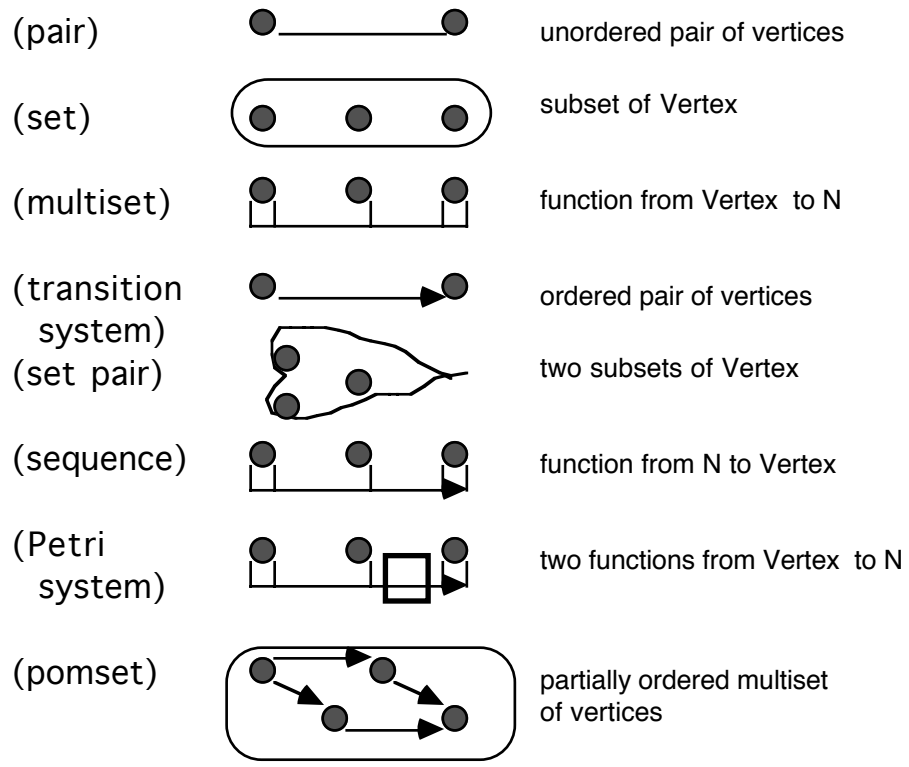


fig2 various kinds of grap h edges

fig 2 Various kinds of graph edges

The most general notion of edge is given by (pomset) but the other notions are much more convenient in practice. Each notion of edge gives a natural notion of morphism from one unlabelled graph G to another G' :

functions ve:Vertex(G) -> Vertex(G'),ed:Edge(G) -> Edge(G')

such that for each edge e in G we have

ed(e) is identical to the edge ve(e)

where ve(e) is the edge e after ve has renamed its vertices. Note that we allow different edges in a graph to be identical as such edges may be given different edge labels.

For each notion of edge we have a natural category $\mathbb{C}$ of unlabelled graphs that is well-behaved - $\mathbb{C}$ has all sums and pushouts. For each notion of edge we also have a tempting notion of morphism from a labelled graph G over (VL,EL) to a graph G' over (VL',EL'):

label morphism (vla,ela) and graph morphism (ve,ed)

such that

lab;vla = ve;lab' and edge;ela = ed;edge'

This definition works well for the examples in this section and the next, but it gives a category which behaves so badly that there are grave implementation problems. In section 3 the definition is modified so that the resulting category is well behaved because it is the flattening of an indexed category. In sections 1.4 and1.5 we will meet several examples of reflexive graph grammars - static graph grammars where vertices and/or edges can themselves be static graphs. More study should be devoted to this special case of static graph grammars.

#1.1 Relational databases

In the conceptual design of a relational data base one devises a set of relation names and one assigns attributes to each relation name. The conceptual design is given by a signature $\sum$; for each subset M of attributes we have a set $\sum(M)$ of relation names. This conceptual design can be represented by a labelled hypergraph with a vertex for each attribute and a hyperedge for each relation name. The label of a hyperedge is the corresponding relation name; the label of a vertex is the corresponding attribute.

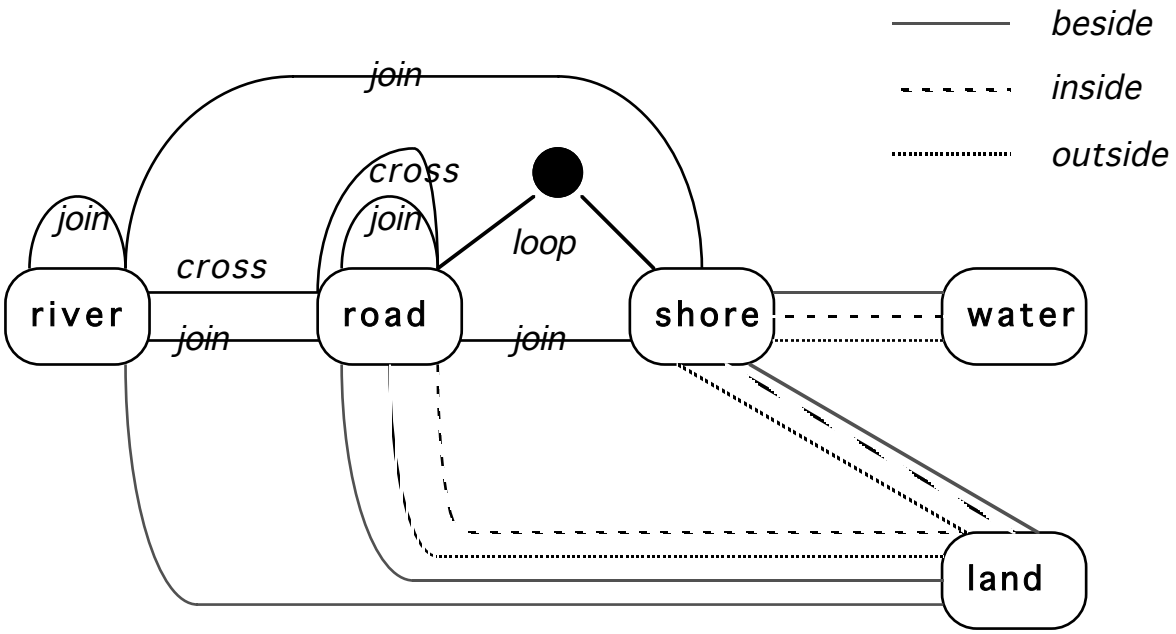


fig 3 Scene design graph SDG

At any time during the life of design ∑ the actual data base is a collection of 'tuples',instances of relations. The actual database can also be represented as a labelled hypergraph with an edge for each tuple and a vertex for each attribute that occurs in a tuple.

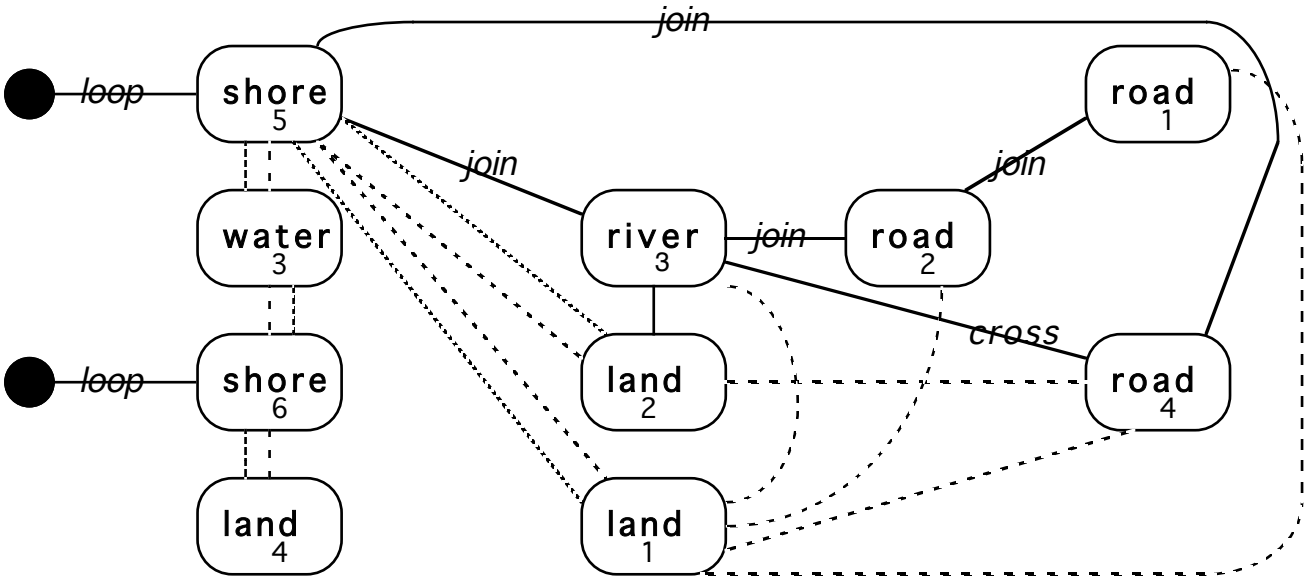


fig 4 Scene instance graph SIG

There is a graph morphism from the instance graph SIG to the design graph SDG, and there is a label morphism from our design ∑ to an alternative design ∑'

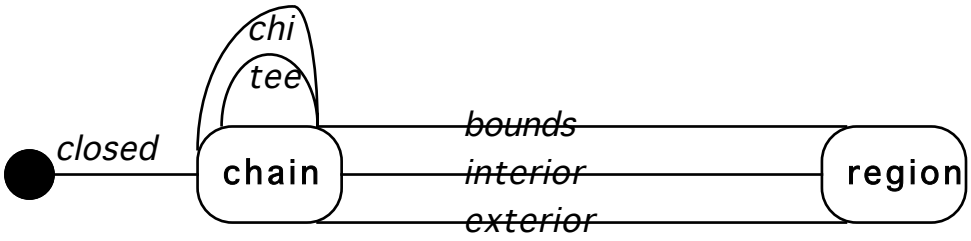


fig 5 Image design graph IDG

The instance graph SIG is inferred from the "observed" instance graph IIG

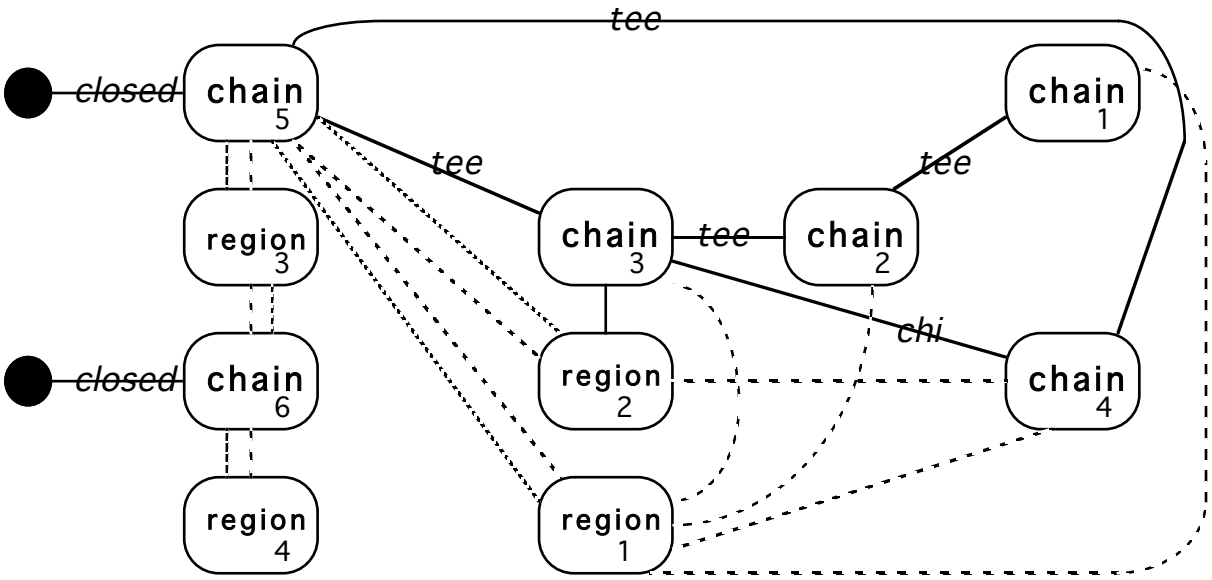


fig 6 Image instance graph IIG

using graph productions like

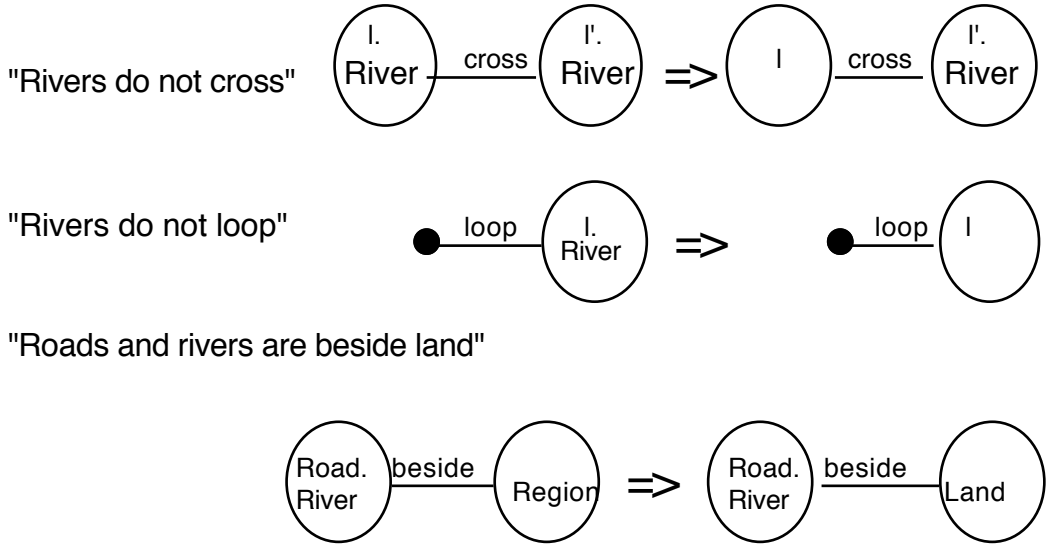


fig 7 Scene-Image graph productions

In (RM) these graph productions appear as logical formulas, but the graph formalism is simpler because it incorporates contextual, situational and semantic constraints.

Our example of a label morphism from Scene to Image is unusual. It is usual in databases to assign a type ty(a) to each attribute "a" so each relation R:a1,a2,... is assigned a product type

ty(a1)*ty(a2)*....

An actual database gives a map Val:Vertex -> Values such that

Val(v) is a value of the type ty(lab(v)).

The lowest morphism in figure 8 gives an example.

	river	road	shore	land	water	join	cross	loop	beside	inside	outside
la1	chain	chain	chain	region	region	tee	chi	closed	bounds	interior	exterior
la2	I*I*N	I*N	I*N	I*N	I*N	N*N	N*N	N	N*N	N*N	N*N
la3	N	N	N	N	N	N*N	N*N	N	N*N	N*N	N*N

fig 8 Three label morphisms from Scene

Many database designers follow the entity-relationship approach in which one considers only unary and binary relations so our hypergraphs become graphs.In relational data bases one usually insists on "no repetitions and flat values" - Val is an injection and ty(a) is a basic type like Integers, Characters, or Strings. There is no theoretical reason for these restrictions, and one gets semantic nets if one relaxes them.

#1.2 Static semantic nets

Frames,schemes and many other popular AI methods of representing knowledge are examples of semantic nets - objects connected together by links. If there are no "actions or methods" associated with the objects, then a net can be converted to a graph by

- vertex for each object
- vertices labelled by sets of attribute value pairs
- edge for each link labelled by 'link' labels.

Example ctd: The semantic net in figure 9 becomes the scene design graph SDG in figure 3.

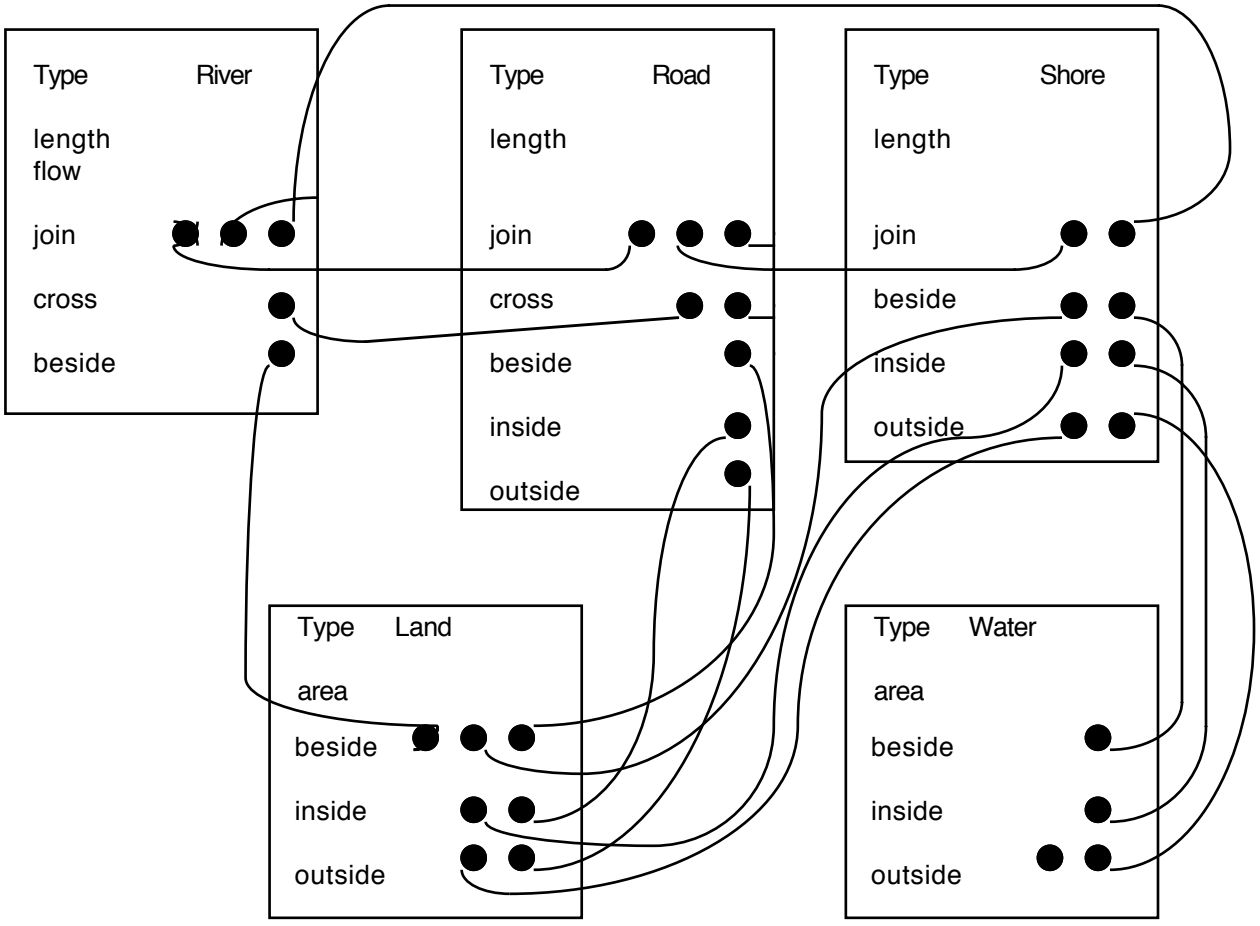


fig 9 Semantic net version of graph SDG

Several authors have formalised semantic nets by reducing them to 'unnormalised' relational databases. The idea is that objects are instances of classes and classes are just relations. Links are special kinds of attributes whose values are instances of relations. Databases with such attributes are unnormalised.

Semantic nets differ from relational databases in being object-oriented, but capture databases by taking tuples as vertices not edges.The type

assignment ty:VL->Type can take vertex labels into product types; la2 and la3 in figure 8 are examples. For the theory to go through we must also assign types to edges.So far we have only used product types, but figure 9a shows some of the other possibilities (the upper row shows the constructors studied by type theorists,the lower row is borrowed from (GHS)).

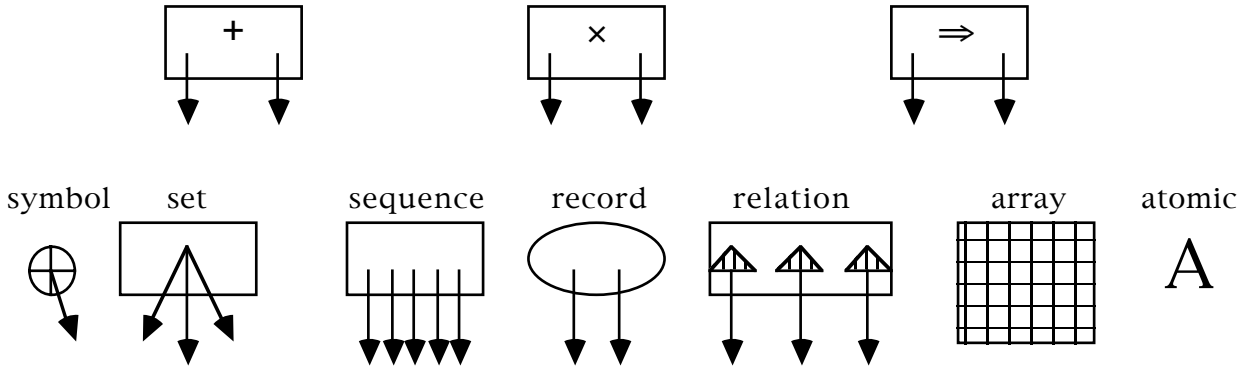


fig 9a Type constructors for structuring labels

In section 4 we show how Type can be a family of sets of formulas, so the graph approach can be reduced to the logic approach.

#1.3 Type inheritance

Many ingenious graph representations have been devised by those interested in 'multiple inheritance' problems,and it is surprising that noone seems to have used graph productions to capture 'Type inheritance' reasoning. As we do this in the companion paper(HM), we will only give an example here.The essence of 'Type inheritance' is that there is an order on types(=concepts = vertexlabels). This order may be part of the conceptual design or it may be derived from a label morphism la = <vla:VL->VL', ela:EL->EL'>. This morphism orders VL"=VL+VL' and EL"=EL+EL' by

vl ≤ vl' iff vla(vl)=vl' el ≤ el' iff ela(el)=el'

In our example the order on VL" and EL" is

river,road,shore < chain land,water < region
join=tee,cross=chi,loop=closed, bounds=beside
inside=interior,outside=exterior

and type inheritance can be given by graph productions like

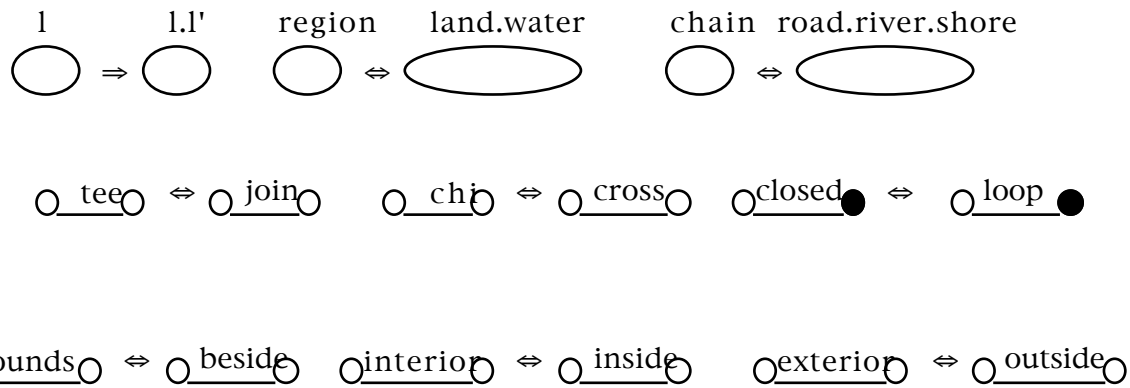


fig 10 Graph productions for type inheritance

Notice that all 4 graphs in section 1.1 are now over the same index (VL",EL"). What happens to our assignment ty:VL->Type?All goes well if Type has an order and vl< vl' implies ty(vl)< ty(vl'). In section 3 we will see the advantage of making orders complete by introducing "top and bottom" labels. Complete orders give meet or join as a 'gluing' operation and all goes well with type assignment if Type has a 'gluing' operation "." and

vl" = vl.vl' implies ty(vl") =ty(vl).ty(vl').

Thus the typing assignment should be an order morphism or an algebraic homomorphism.

#1.4 Conceptual structures

An extremely popular way of representing 'linguistic' knowledge is to use the conceptual graphs invented by Sowa(So). In designing a family of conceptual graphs for a 'knowledge domain' or even a 'language', one devises not only a set of relation domains and an ordered set of attributes and types, but also allows 'nesting'. Usually nesting in conceptual graphs is illustrated by linguistic modalities as in " John believes that Mary knows...", but in our example we will use nesting to capture: interior,exterior,closed, bounds,inside,outside,beside and loop.

Example ctd: Consider the special role played by 'region' in the image instance graph IIG in figure 6. This suggests the nested graph in figure 11 where we have a tree of regions and each region has a 'chain graph'. The scene instance graph SIG in figure 4 also suggests the nested graph in the figure 11 where the labels have been changed appropriately (water or land label for each region in the tree); it suggests also that region2 should be eliminated from our graphs.

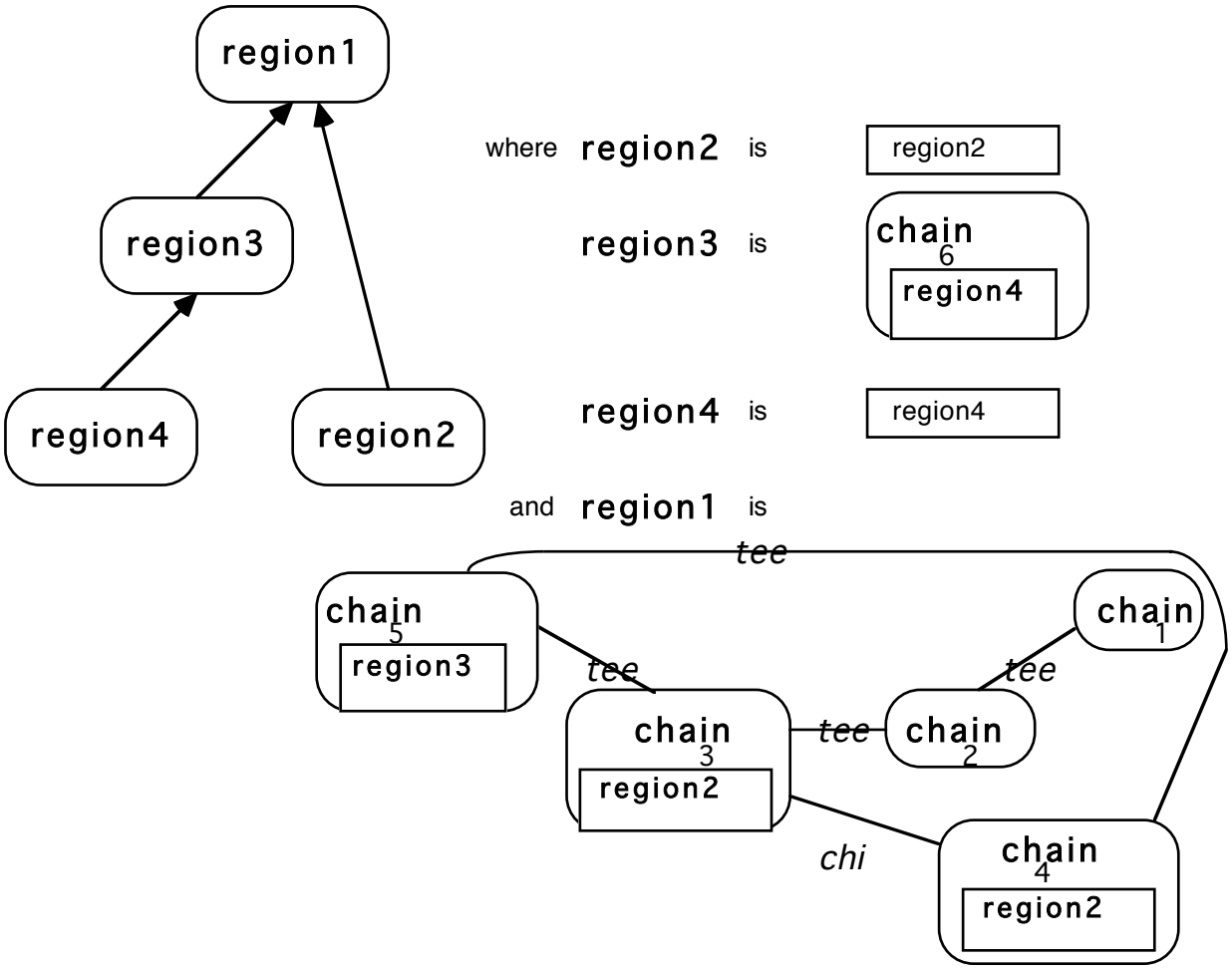


fig 11 A nested graph

Now that nesting has eliminated so many labels, we can give a concise graph grammar for the historical developement of scenes.

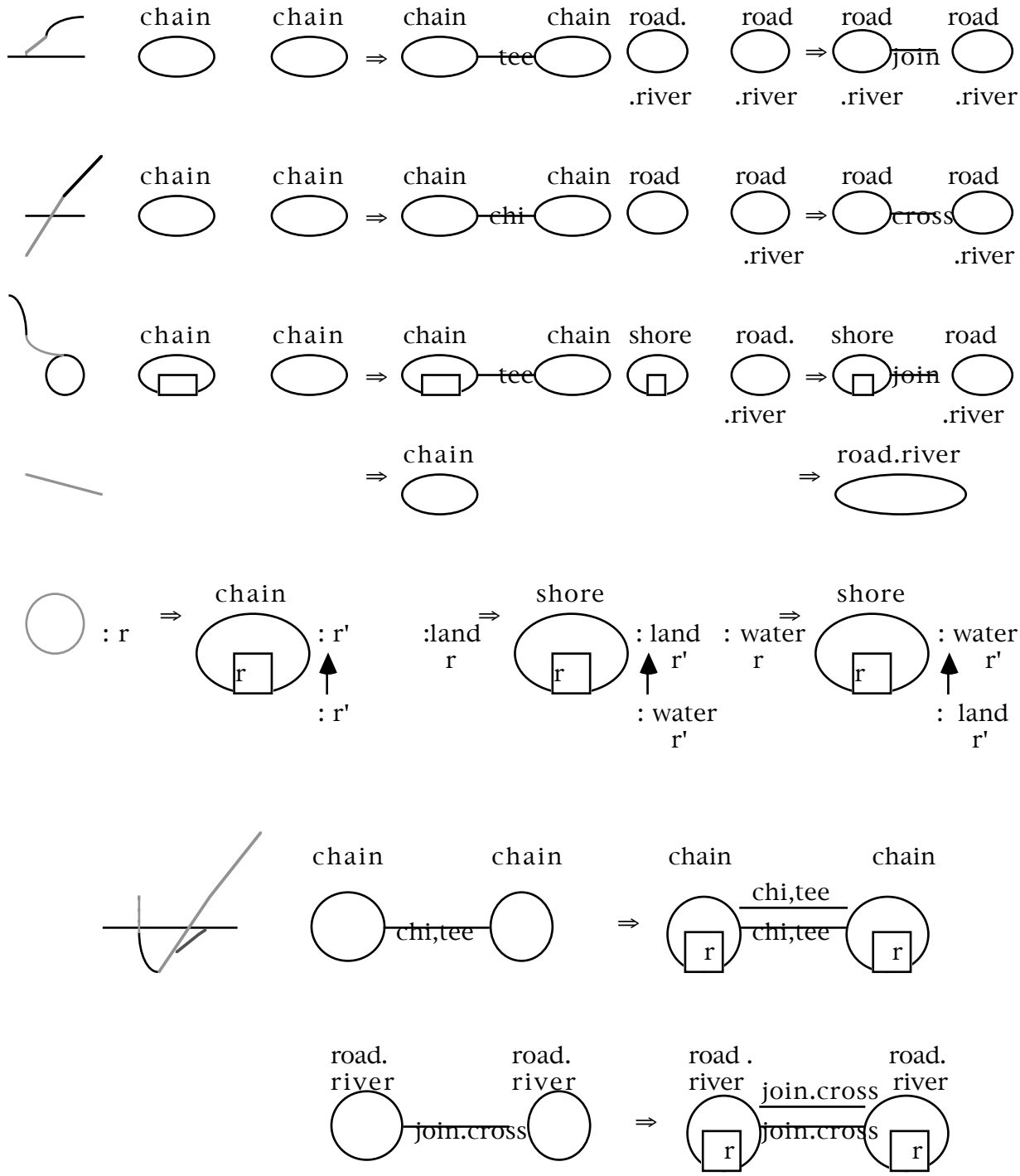


fig 12 Historical graph productions

Note that we have captured the essential content of "shores always loop" and "shores separate land and water".

#1.5 Logical programming

Logical programming languages can express most of the information in graph representations of databases,semantic nets,type inheritance and conceptual graphs. It is not difficult to express most of the graphs and graph productions in this paper in a language such as Prolog. If a logical language is sufficiently modular, it can capure nesting in conceptual graphs by using 'worlds' or viewpoints. Some logical languages, like Omega (ACDS), have an implicit metalevel and they can capture most of the information in reflexive graphs, in particular they can express combinations of viewpoints. In conceptual graphs the nested graphs are *rigid* viewpoints, they cannot be restructured by graph morphisms. Let us meet the Omega challenge by giving an example of *fluid* viewpoints.
Example ctd: In the last section we gave the "history" productions for the construction of scene graphs.

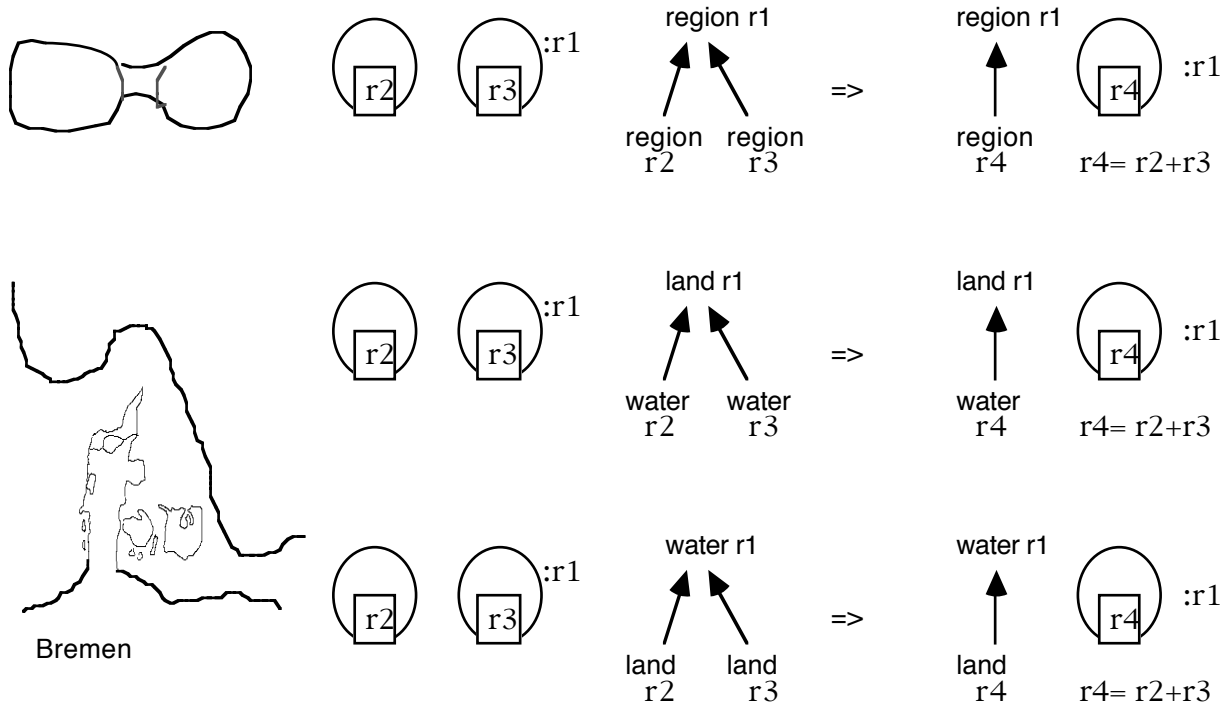


fig 13 Fluid historic graph productions

#2 Representation of dynamic knowledge

It is often natural to represent knowledge by dynamic graphs where the vertices or edges represent actions,processes, procedures or productions.Much of the computer science literature on concurrency uses the theory of transition systems and/or Petri nets. Transition systems are just dynamic graphs whose edges are ordered pairs of vertices; Petri nets are just

dynamic graphs whose edges are ordered pairs of multisets of vertices. Both kinds of dynamic graphs can be converted to graph grammars by:

> for each edge e we have the graph production L=>R
> where L is the "input" component
> and R is the "output" component of e .

Applying one of these productions to a dynamic graph G corresponds to a "joint action by the processors carrying G".

There is no objection to concurrent/parallel applications of productions. In section 3 we will show that any two graphs have sums, so one can consider the concurrent/parallel application of productions, L=>R and L'=>R', as the application of the sum production L+L' => R+R'. Naturally one can have "conflict"- productions, L=>R and L'=>R', can both be applied to a graph G but the sum production cannot be applied to G.

In dynamic graphs it is natural to assign States to vertices and State functions or relations to edges. The category minded might prefer to assign objects in a category C to vertices and morphisms to edges. We will see that it is sometimes convenient to assign static graphs to vertices and applications of graph productions to edges. This idea of 'graph productions as actions' can be lifted to the label level to give metagraph grammars. In sections 2.4 and 2.5 we will meet several examples of metagraph grammars - dynamic graph grammars with static graphs as vertex labels and applications of static graph productions as edge labels. More study should be devoted to this special case of dynamic graph grammars.One approach is to treat a metagraph grammar as a 2-category with static graphs as objects and static graph productions as morphisms. The close connection between "rewriting" and 2-categories is well-known.

#2.1 Planning

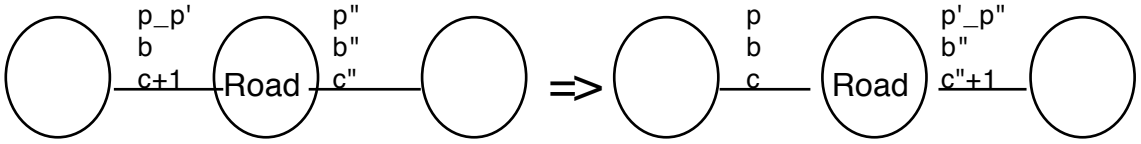
Plans are combinations of primitive actions. One has a repertoire of action types and the actions in a plan are occurrences of these types.It is natural to think of actions as edges and action types as edge labels, but what are the vertices. Usually one assigns pre- and post-conditions to action types,and one can take states or situations as vertices. A more sophisticated view (SR,Ba) is that action types also have prevail- and keep-conditions that constrain the 'joint actions' that can occur in a plan. In this view one should take local(partial) states as vertices and give dynamic productions for permitted joint actions.

Example ctd. We could consider the historical productions in figures 12 and 13 as action types(appropriate edge labels are given at the extreme left of the figures), and a joint action could be the developement of an island at the same time as the building of a road. Instead we give a more dynamic

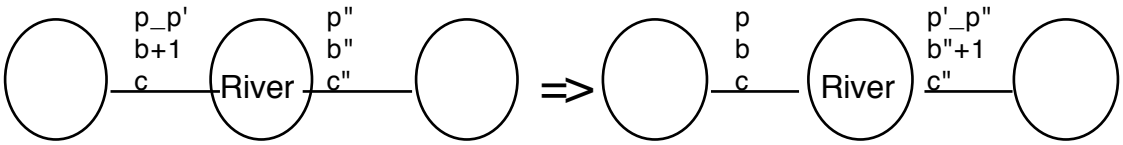
example.Vertices are scene graphs with enlarged edge labels for shores and places where roads and rivers meet. Edge labels - loop,join,cross- are extended by triples < p,b,c> where

p is a set of person names
b is the number of available boats
c is the number of available cars.

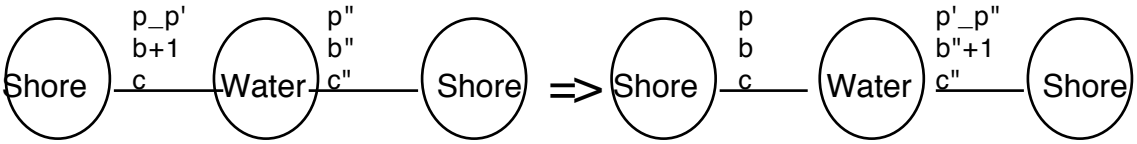
Figure 14 shows the three action types and a plan for a person to go from 'shore6' to 'road1'



DRIVE: if road between X and Y & car at X then one can drive from X to Y & car at Y



ROW: if river between X and Y & boat at X then one can row from X to Y & boat at Y



SAIL: if X and Y are shores & boat at X & X beside W & Y beside W then one can sail from X to Y & boat at Y

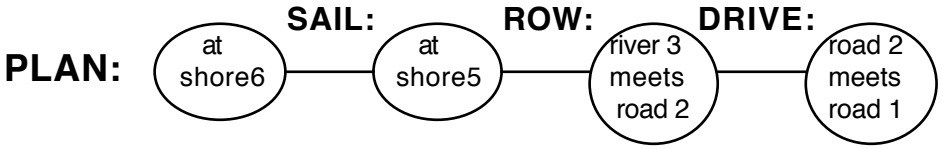


fig 14 Three action types and a plan

Note that this simple plan will be frustrated if there is no car available when the boat trip is over, so one might prefer joint actions and the more elaborate plan:

"telephone to a friend on road1, so he drives to meet me"

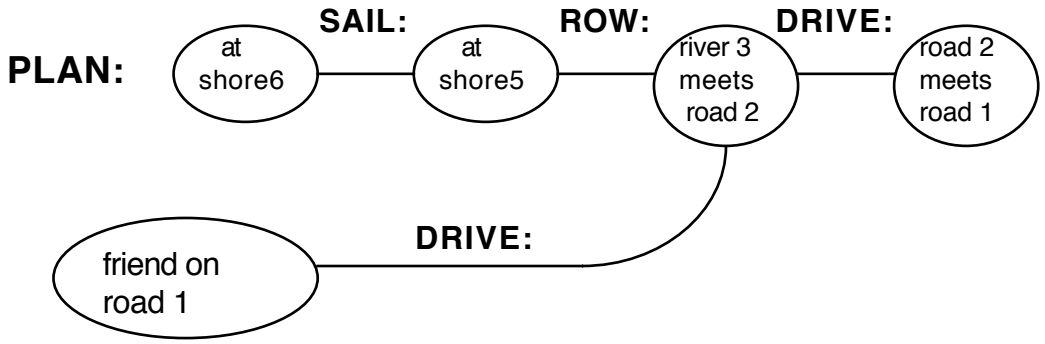


fig 15 Another plan

#2.2 Dynamic Semantic nets

Usually actions are represented in semantic nets as "methods" attached to objects, but some more refined net representations (e.g.LINCKS(Pa)) allow actions to be objects.

Example ctd: The upper row in figure 16 shows the kind of objects which capture the edge information in the MapSee net.The lower row in figure 16 shows the kind of objects which capture the action information in the MapSee net. Using both kinds of objects one can build a net, that contains so much knowledge about scenes that it supports a realistic multimedia simulation.

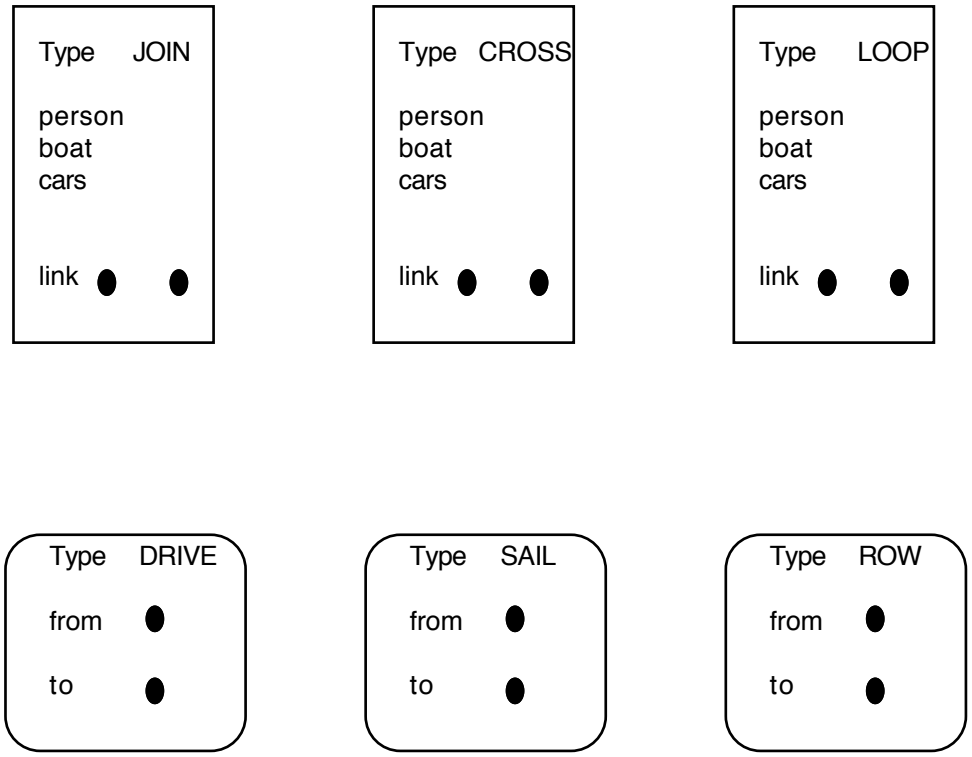


fig 16 Nodes for a dynamic semantic net

Objects of action type , corresponding to hypermedia 'buttons', are attached to JOIN/CROSS/LOOP nodes. The code for the buttons might have the specification:(DRIVE) transfer persons and car following links that go via roads or shores (ROW) transfer persons and boat following links that go via rivers (SAIL) transfer persons and boat following links that go via water. Running these codes may cause appropriate animation to be displayed on a screen.

#2.3 Hypermedia

Graphs seem to be the preferred formalisation of hypermedia systems (SF,To). Conversely any dynamic graph can be implemented as a hypermedia system. Each vertex corresponds to a "window",which may or may not be displayed on the screen (Mac's Hypercard only displays one window at a time, other systems allow more). Each edge in a dynamic graph corresponds to "pressing a button". Each edge label in a dynamic graph corresponds to a "button" (static graph production).

Example ctd:The dynamic graph described in the last section is suitable for a hypermedia presentation because it has a node for every 'join','cross',and 'loop' edge in SIG,the scene instance graph in figure 4. . One can have a window for each such edge.If the person of interest me is at the edge, then

the window is displayed - appropriate scenery appears on the screen, sounds of boats and cars with an intensity proportional to their number... For each of the DRIVE, ROW or SAIL actions that are currently possible, there can be a button on the screen.When a button is chosen, then a car or boat appears on the scene, people climb aboard, scenery changes, people dismount, and the car or boat disappears.

It is an interesting exercise to make the slight changes in our graph representation, so that our hypermedia representation is more natural - one should not insist that everybody gets out of the car at every road junction!

#2.4 Simulation

Once we have described the possible actions by dynamic graph productions we can build a Petri net simulation. In our MapSee example scene edges labelled by 'join', 'cross' or 'loop' correspond to place-triples in a Petri net. Place triples are connected by transitions for possible 'DRIVE','ROW' or 'SAIL' actions.

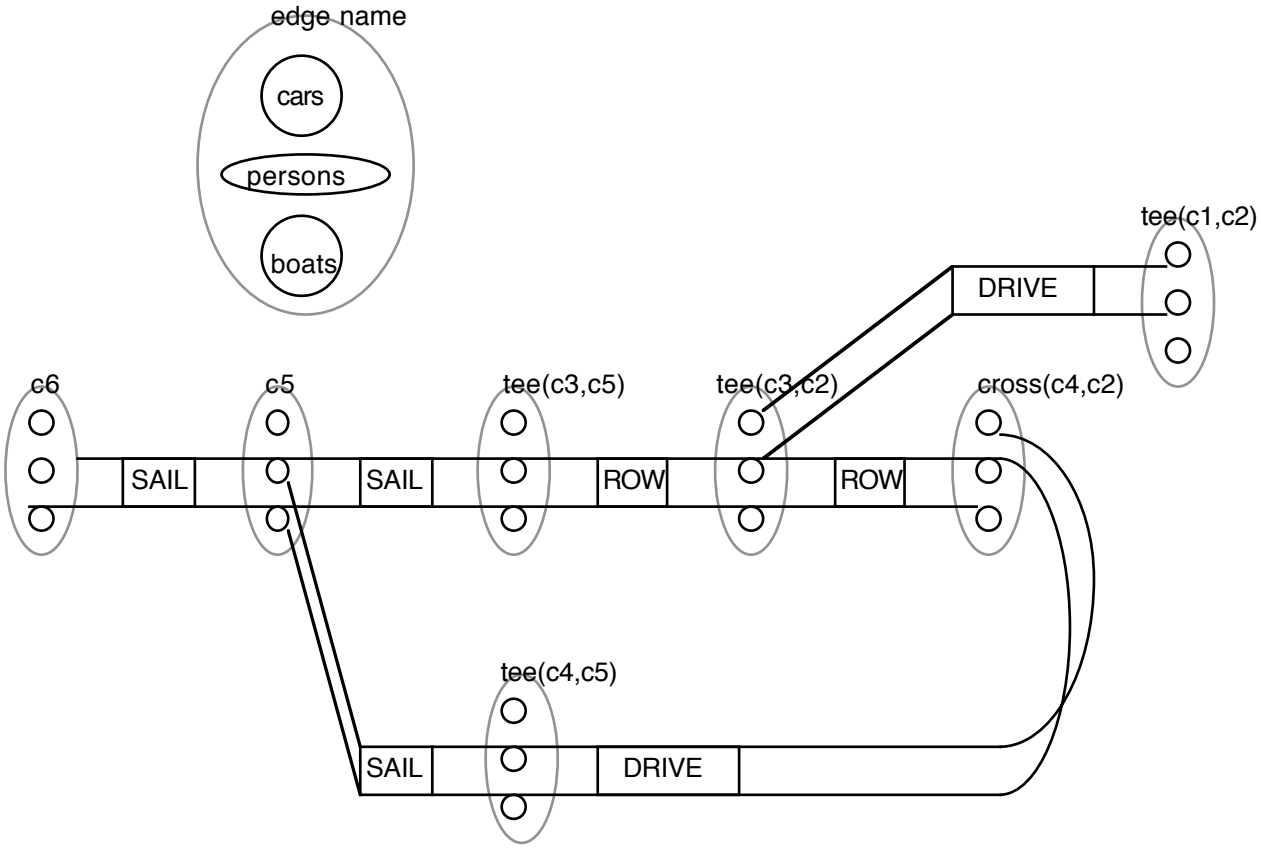


fig 17 Petri Net Simulation

Dynamic graphs given by the rules 'DRIVE', 'ROW' or 'SAIL' correspond to occurrence nets - possible runs of the simulation Petri net. For a more realistic simulation example of dynamic graph grammars one can look at the generation of Forrester diagrams in (DT).

#2.5 Linguistic attribute grammars

There is an interesting developement in linguistics (Jo), in which knowledge is represented by both static and dynamic graphs. A normal syntactic grammar for parsing sentences gives not only a 'derivation tree' but also a dynamic graph where the edges are applications of syntax rules. The vertices of this graph are not just sets of attribute-value pairs; their labels are static graphs.

Example ctd: Let us ignore linguistic reality and agree on the syntax:

< conditional > S ::= if P then P;< conjunction > P ::= S and S

The derivation tree for "if road between X and Y & car at X, then one can drive from X to Y & car at Y" gives the dynamic tree at the bottom of figure 18, and the static graphs for the tree nodes can be seen in the rest of the figure. These static graphs are the solution of the linguistic attribute equations.

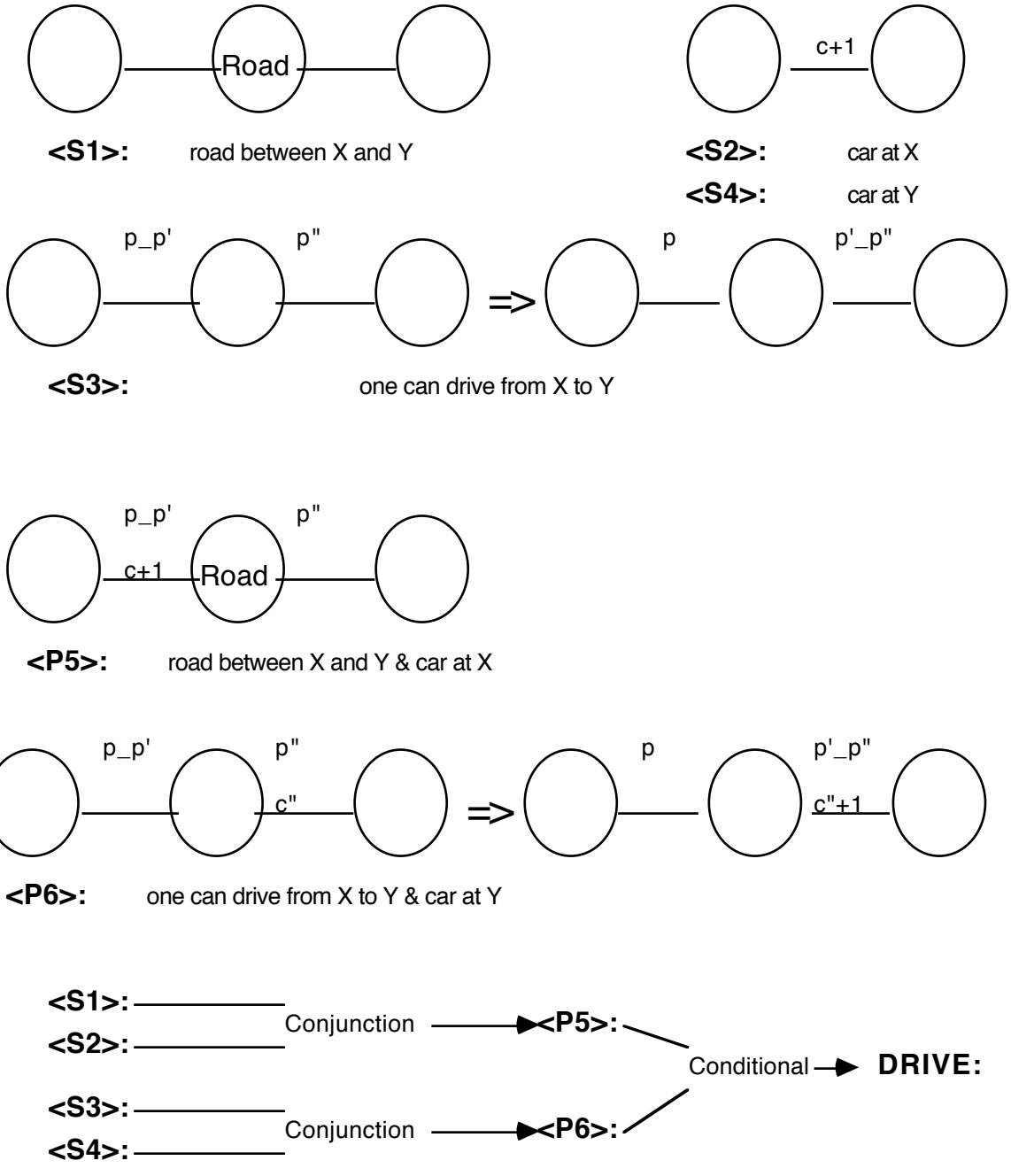


fig 18 Linguistic synthesis of an action

Clearly any attribute grammar can be represented as a dynamic graph - the syntax rules give dynamic edges and the corresponding semantic attribute functions are static graph morphisms.

#3 Morphisms and the applicability of productions

When can one apply a production L=>R to a graph G? The intuitive answer is - when there is an occurrence of the graph L in the graph G. This gives morphism requirements

(1) the definition of graph morphisms should cover all "ocurrences of one graph in another"
(2) if there is a graph morphism from L to G, then the result of applying any production L=>R to G should be well-defined.

Intuitively the result of applying L=>R to a graph is given by replacing L by R in G. This suggests the morphism requirements:

(3) if H is the result of applying L=>R to G, then there is a morphism from R to H
(4) if the production L=>R is a morphism, then the result of applying L=>R to G is the pushout of L=>R and the occurrence of L in G.

Do we want the definition of graph morphism to cover all possible graph productions? Intuitively one thinks of the application of L=>R to a graph is given by removing L, then inserting R. This suggests the flexible definition of a production as a pair of morphisms, < l: K->L, r: K->R >, and the requirements:

(5) the production < l: K->L, r: K->R > can be applied to a graph G if there is an occurrence of K in a graph D such that G is the pushout of l: K->L and this occurrence.
(6) the result of applying the production < l: K->L, r: K->R > to G is the pushout of r: K->R and the occurrence of K in D.

There are two attitudes to the morphism requirements (1-6) and their demands for the existence of pushouts. The neat attitude is to require

(7) the category of graphs has all pushouts;

the scruffy attitude is to restrict the graph morphisms allowed in productions and occurrences, or even to force an implementation to check if a pushout exists. Any readers happy with the scruffy attitude can skip the rest of this section.

Are there any categories of graphs that satisfy requirements (1-7)? Yes, if we are prepared to modify "well-defined" in (2) to "well-defined for each pushout complement". In section 1 we saw many kinds of graphs could be associated with label sets (VL,EL).

If VL and EL are sets with a 'gluing' operation "."and we impose the continuity requirements:

vla (V li) = V vla(li) ela (V ei) = V ela(ei)

on the functions in label morphisms, then all label pushouts exist. If we have a graph morphism from a graph G over (VL,EL) to an unlabelled graph G', then we have a fiber morphism:

lab" (v") = V (lab(v) ! ve(v) = v")
edge"(e") = V (edge(e) ! ed(e) = e")

which takes G into a labelled graph G" over (VL,EL). Now we can give the correct definition of the morphism between labelled graphs.

Definition1 A gluing morphism from a labelled graph G over (VL,EL) to a graph G' over (VL',EL') consists of :

continuous label morphism (vla,ela) and fiber morphism (ve,ed)
such that lab' = lab";vla and ed' = ed";ela.

Theorem The category of labelled graphs, given by gluing morphisms, has all sums and pushouts.

Proof
This is given by a general theorem on indexed categories (TBG) but the argument for our particular case is instructive.

If one 'places' a labelled graph G over (VL,EL) 'beside' a graph G' over (VL',EL') , one gets a graph over (VL+VL',EL+EL') that is the sum G+G'. Now for the construction of the pushout of the graphic morphisms r: K => R and k: K => D. We have just constructed the sum graphic R+D and we have to glue some of its vertices and edges together coherently. The graph pushout tells which vertices and edges must be glued together, and the label pushout tells what the labels of the glued vertices and edges must be. The continuity requirements on label and fiber morphisms ensure that this construction does give the pushout of r and k.

Where do the 'gluing' operations "." come from? The simplest case is when VL and EL are power sets and "." is set intersection or union. Another possibility is that VL and EL are families of closed sets and "." is given by

l.l' = closure of the union of l and l'.

If G" is a graph over (VL",EL") and there is no obvious gluing operation on VL" and EL", then one can identify G" with a graph
over (VL,EL) :

lab(v) = singleton (lab" (v))
edge(e) = singleton (edge"(e))

where VL is the power set of VL" and EL is the power set of EL". If there are natural preorders on VL" and EL" , then one can identify G" with a graph over (VL,EL) :

lab(v) = predecessor (lab" (v))
edge(e) = predecessor (edge"(e))

where VL is the ideal family of VL" and EL is the ideal family of EL". In both cases we get wellbehaved morphisms by attaching an index to a graph G, that is wider than its apparent index (the vertex and edge labels that appear in G).

#4 Logic, graphs and institutions

One can reduce any kind of labelled graph G to sets of logical formulas. Whatever the kind of edge,one can define a dart as the occurrence of a vertex in an edge. If one has constant symbols for each dart in G and unary predicates for each vertex and edge label, then G can be described completely by {lab(v), edge(e) ! <v,e> is a dart in G} , a set of atomic formulas. For most kinds of graphs we also have the structure and frame reductions.The label sets EL and VL give a signature ∑ with a predicate symbol for every edge or vertex label. Any graph G over EL and VL becomes a ∑ -algebra when the graph vertices are collected into the carrier domain. Thus labelled graphs can be reduced to structures in the institution of first order logic. If we extend the signature ∑ by adding constant symbols for the vertices and G, then all information about G can be expressed as formulas(frames) in the first order logic for the extended signature. Thus labelled graphs can be reduced to theories in the institution of first order logic.

An idempotent gluing operation "." can be captured by the equivalence

vl1(x) or vl2(x) iff vl3(x)

whenever vl1.vl2=vl3. For reflexive graphs it is natural to have an extra viewpoint parameter in each atomic formula. Once one has viewpoints, one can capture graph productions L=>R by using 'left' viewpoints for L and 'right' viewpoints for R. Usually a graph production L=>R can also be captured by structure morphism from the structure for L to the structure for R. However structure morphisms correspond to equivalence classes on sums, and our viewpoint construction is more general.

Example ctd: The structure representation of the scene instance graph SIG has the carrier domain {c1,c2,c3,c4,c5,c6,r1,r2,r3,r4} and predicates:

Shore{c5,c6}, River{c3}, Road{c1,c2,c4}, Land{r1,r2,r4}, Water{r4},
Join{c1c2,c2c1,c2c3,c3c2,c3c5,c5c3}, Cross{c3c4,c4,c3},
Loop{c5,c6}, Inside{c5r3,c6r4}, Outside{c5r1,c5r2,c6r3},
Beside{c1r1,c2r1,c3r1,c3r2,c4r1,c4r2,c5r2,c5r3,c6r3,c6r4}.

The frame representation is given by introducing constant symbols {c1,c2,c3,c4,c5,c6,r1,r2,r3,r4} and converting the structure representation into atomic formulas. The structure representation of the graph morphism from SIG to the image instance graph IIG is given by adding predicates:

Chain{c1,c2,c3,c4}, Region{r1,r2,r3,r4},
Tee{c1c2,c2c1,c2c3,c3c2,c3c5,c5c3}, Chi{c3c4,c4,c3},
Closed{c5,c6}, Interior{c5r3,c6r4}, Exterior{c5r1,c5r2,c6r3},
Bounds{c1r1,c2r1,c3r1,c3r2,c4r1,c4r2,c5r2,c5r3,c6r3,c6r4}.

In this example SIG and IIG share no predicate, so viewpoints are not needed.

The above reduction is suitable for static graphs, but for dynamic graphs one may prefer a signature of function symbols. We will only describe the reduction when edges with the same label have the same number of input and output vertices. Then the signature $\sum$ can have a set of function symbols for each edge label, one for each output. Any graph G over EL and VL becomes a $\sum$-algebra when the graph vertices are collected into the carrier domain, which also has an 'undefined' vertex "?". Thus labelled graphs can be reduced to structures in the institution of equational logic. If we extend the signature $\sum$ by adding constant symbols for the vertices of G, then all information about G can be expressed as equations in the first order logic for the extended signature. Thus labelled graphs can be reduced to theories in the institution of equational logic.

Example ctd: The plan in figure 14 can be described by the equations

SAIL(v1) = v2 ROW(v2) = v3 DRIVE(v3) =v4

and the theories for the viewpoints: v1= "at shore6", v2= "at shore5", v3= "where river3 meets road2", v4= "where road2 meets road1". These theories can be combined if edge labels have viewpoint parameters.

Conversely institutions can be converted to graphs, if we have a way of

(1) converting structures to theories(diagramming models in logic)
(2) representing theories as graphs

In most institutions (1) is not a problem as one can extend the signature by adding symbols as we did above. If the theories given by (1) can be generated from finitely many "atomic formulas" and we have a natural way of decomposing a signature into edge and vertex components, then we can achieve (2). In a discussion of the reduction of logic to graphs, we should mention the reduction of rule-based systems to Petri nets in such papers as (MZ,PM,Zi).

Where do graph morphisms and productions come from? In an institution we have structure morphisms and signature morphisms. The signature morphisms give theory morphisms and (2) converts these into graph morphisms. Usually (1) converts structure morphisms into theory

morphisms and (2) converts these into graph morphisms. Usually we have a domain theory which specifies which structures are possible models of the domain.

Example ctd: Most of the domain constraints in [MR] are captured by the 'type inheritance' graph productions in figure 9. The six remaining constraints are:

(1) Rivers do not cross
(2) Shores form closed loops
(3) Rivers do not loop
(4) Shores separate land from water
(5) Roads and rivers are beside land
(6) Rivers flow into other rivers or into shores.

Constraints (1), (3) and (5) are captured both by the scene design graph in figure 3 and by the productions in figure 7. Constraints (2),(3) and (4) are captured by the historical productions in figure 12. A slight modification of these historical productions also captures constraint (6).

Many of the domain axioms can be converted to proof rules and thence to theory and graph morphisms, but what of those axioms that can not? Our attitude is that they are constraints that control the uncertainty of the knowledge represented in a structure, and they too can and should be captured in graph productions.

#5 Uncertainty in graph representations

So far we have not exploited the fact that orderings on an index <VL,EL> give a natural order on graphs over the index. One can define "G' is an extension of G" as the "weak fiber morphism"

lab' (v') ≥ V (lab(v) ! ve(v) = v')
edge'(e') ≥ V (edge(e) ! ed(e) = e')

Any graph G can have many extensions G' and we are uncertain about which extension is "the actual state of affairs". Intuitively a knowledge representation graph G is "the known/believed state of affairs", and we should reject the notorious "closed world assumption" (that we tell "all the truth,and nothing but the truth" - sworn by every witness to british jury trials).

Example ctd: All MapSee images can be interpreted as scenes by "all regions are land and all chains are roads", but interpretations with water regions and chains, that are shores or rivers, are much to be preferred. Heuristic graph productions must be used to get the scene instance graph SIG from the

graph IIG in figure 6.With the productions in figures 7 and 19 one can only get the extension of SIG,in which c1,c2 and c4 are still labelled as chains. This extension is also an extension of the scene graph SRG, in which c1 and c2 are rivers, but there is no morphism between SRG and the extension.

There is an enormous literature on uncertainty .... and most of the ideas can be translated into graph ideas. One can have probabilities and uncertainty factors on graph productions, vertex labels, and edge labels. This corresponds to applying productions L=>R ,not to a graph G to get a new graph H, but to probability distributions over graphs to get new probability distributions. In calculating the new probability distribution one should pay due attention to the 'pushout complement' phenomenon that for there may be 0,1,or many choices for K-> D in the application of a production. This phenomenon can give many chaos and fractal effects, particularly if the our 'probability' distributions are really Schaeffer-Dempster or fuzzy distributions.

In the last decade there has been a trend away from statistics towards circumscription, default logics, and truth maintenance. In (Sh), Shoham shows that most of these methods of handling uncertainty are captured by preference relations on structures. One can maintain that representing knowledge about a domain can also include the representation of preferences - and preferences can also be represented by graph productions.

Example ctd: Heuristic variants of two domain constraints

(2') Closed loops are usually shores
(6') Chains joined to shores or rivers are usually rivers

can be given by graph productions (notice that the graph SRG, introduced above, should be preferred to SIG)

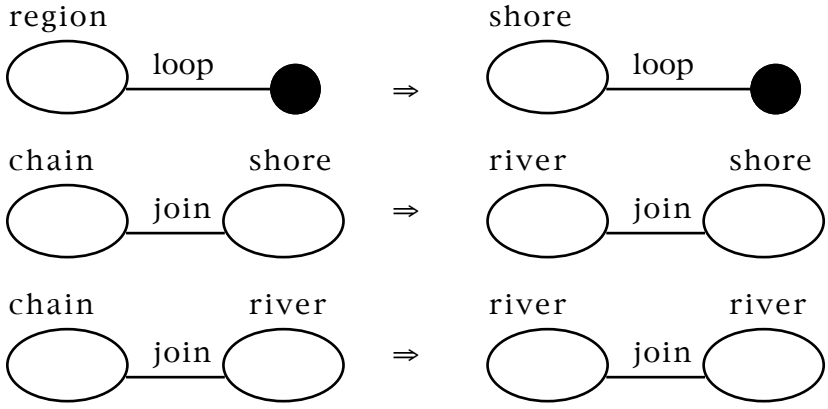


fig 19 Heuristic graph productions

What about the natural inclusion order on structures ,mentioned at the beginning of this section? It corresponds to Occam's Razor, and we may or may not want to make these preferences for simplicity explicit as graph productions.

In a series of papers (Do) one of the truth maintenance pioneers has been arguing for "rationality" rather than "logic".

Several AI authors have distinguished two kinds of beliefs

- manifest = explicit=assertions=axioms=base beliefs
- constructive=implicit=theorems=derived=inheritable=inferable.

In the logical approach the constructive beliefs of an agent are a subset of the deductive consequences of its manifest beliefs. In the rational approach the manifest beliefs of an agent are specifications of its constructive beliefs and it can choose rationally between various ways of interpreting(=construing - hence constructive) its specifications. Different rational choices give different sets of constructive beliefs. One can think of the use of graph grammars to represent knowledge as an example of the rational approach to uncertainty. The manifest beliefs of an agent can be captured by a graph G and constructive beliefs are given by all possible applications of graph productions in the grammar to G.

Example ctd. One can consider the image instance graph,IIG, as the manifest beliefs of an agent, and the scene instance graph,SIG, as one of several possible constructive beliefs.

Conclusion

In (Do) we find: "To paraphrase Hamming , the purpose or aim of thinking is to increase insight or understanding, to improve one's view,so that, for instance,answering the question of interest is easy, not difficult. This conception of reasoning is very different from incremental deduction of implications. Instead of seeking more conclusions, rationally guided reasoning seeks better ways of thinking, deciding, and acting. Rational reasoning does not preserve truth, but instead destroys and abandons old ways of thought to make possible invention and adoption of more productive ways of thought. Correspondingly, the purpose of representation is to offer the best conclusions to draw rather than all the logically possible conclusions, to guide the reasoner towards the the useful conclusions, whether sound or unsound, and away from the useless ones, whether true or false."

This can be taken as an intuitive argument that formalising steps of rational reasoning as graph productions is sometimes better than formalising steps of logical reasoning as logical rules or implications. One has also the pragmatic argument: graph productions can take account of contextual and default information.

References

(ACDS) G. Attardi, A. Corradini, S. Diomedi, M. Simi "Taxonomic reasoning", in 'Advances in artificial intelligence 2' 1987 N. Holland.

(Ba) C. Backstrom "Arepresentation of coordinated actions" Proc. Scand. Art. Int. (1988) 193-207 Tromsø.

(Do) J. Doyle "Constructive belief and rational representation" Comput. intell 5 (1989)1-11

(DT) J.J. Dolado,F.J. Torrealdea "Formal manipulation of Forrester diagrams by graph grammars", IEEE Trans. Sys. Man. Cyb. 18 (1988) 981-996.

(Enc) ed.St.C. Shapiro "Encyclopedia of artificial intelligence" Wiley 1987, ISBN 0-471-80748-6

(GHS) A.M. Goodman,R.M. Haralick,L.G. Shapiro "Knowledge-based Computer vision" IEEE Computer ??dec (1989) 43-52.

(HM) L. Hess,B.H. Mayoh "The four musicians:analogies and expert systems - a graphic approach", this volume.

(Jo) M. Johnson "Attribute-value logic and the theory of grammar" CSLI lecture notes16, ISBN 0-937073-36-9

(Ma) B.H. Mayoh "Unified theory of knowledge representation" in ed. W. Bibel,B. Petkoff 'Artificial Intelligence methodology, systems, applications', N. Holland 1985, ISBN0-4444-87743-6.

(MZ) T. Murata, D. Zhang "A predicate-transition net model for parallel interpretation of logic programs",IEEE Trans. SE 14 (1988)4 81-497.

(PM) G. Peterka, T. Murata "Proof procedure and answer extraction in Petri net model of logic programs" IEEE Trans.SE15 (1989)....

(PR) L. Padgham, R. Ronnquist "LINCKS:an imperative object oriented system" Proc. Hawaii Int. Conf. Sys. Sci.1987.

(RM) R. Reiter, A.K. Mackworth "A logical framework for depiction and image interpretation", Art. Int. 41 (1989/90) 125-155.

(SR) E. Sandewall, R. Ronnquist "A representation of action structures" Proc. AAAI-86, Philadelphia.

(SF) P. David Stotts, R. Futura "Petri-net-based hypertext: Document structure with browsing semantics", ACM Trans. Inf. Sys. 7 (1989) 3-29

(Sh) Y. Shoham "Reasoning about change". MIT press1988, ISBN 0-262-19269-1

(So) J. Sowa "Conceptual structures" Addison-Wesley 1983, ISBN 0-201-14472-7

(TBG) A. Tarlecki, R. Burstall, J. Goguen "Indexed categories" LFCS report 88-60, Edinburgh Univ.

(Th) ed.A. Thayse "From standard logic to logic programming" ch.3, Wiley 1988, ISBN 0-471-91838-5

(To) F.W. Tompa "A data model for flexible hypertext database systems", ACM Trans. Inf. Sys. 7 (1989) 85-100.

(Zi) M.D. Zisman "Use of production systems for modelling asyn-chronous concurrent processes" in 'Pattern directed inference

systems" ed. D.A. Watterman, F. Hayes-Roth, Academic Press 1978.

The four musicians:

analogies and expert systems - a graphic approach

In their paper "Graph rewriting with unification and composition" in the last GraGra conference (PEM) Parisi-Presicce, Ehrig and Montanari suggested that graph grammars might be useful in rule based expert systems. The idea is that graphs capture the relationships between facts, while graph productions capture rules for deriving new facts. In this paper we develop this idea using "graphics" (HM) instead of the usual arc and node labelled graphs. Graphics have the advantage of incorporating variables directly (pointed out to one of the authors by Ehrig) but they seem to have the apparent disadvantage that arcs are neither directed nor labelled.

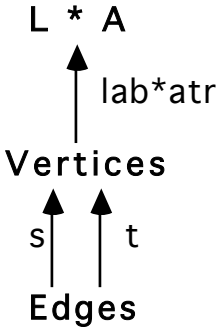
Section1 describes how graphics can capture the information in the labels on directed arcs, so familiar from the semantic nets, conceptual schemes and other knowledge representations in data bases and expert systems. Section 2 describes how graphic productions can capture "rule" information: in traditional IF-THEN rules, in Prolog rules with *assert* and *retract*, and in type reasoning in inheritance hierarchies. Section 3 shows how graphic productions can also capture reasoning by *analogy*, not just logical reasoning. The final section gives various technical results about substitutions, ∑-algebra changes, and the pushout problems that plague both (PEM) and (HM). It also shows that graphic grammars are yet another example of the general theory of institutions and galleries.

#1 Graphics & directed arc labels

A graphic is a graph where all vertices get elements of a ∑-algebra as extra labels. More precisely:

Definition 1 A graphic G over a ∑-algebra A and set L consists of 4 functions

s,t: Edges => Vertices
lab: Vertices => L
atr: Vertices => A



As an example of a graphic consider

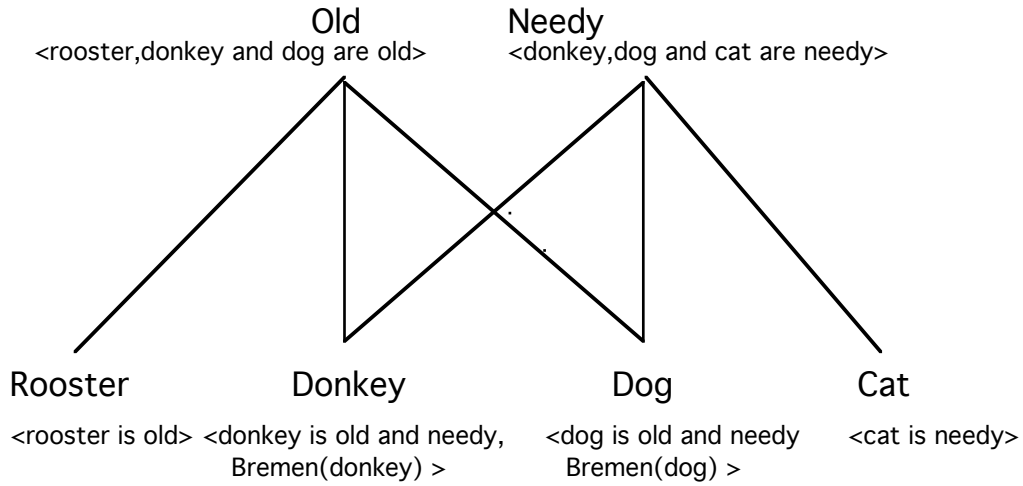


fig 1 Graphic D1 over SIGMA

This represents a small database with a binary relation "is" and a unary relation "Bremen". In the style of (PEM) this database would be represented by the graph

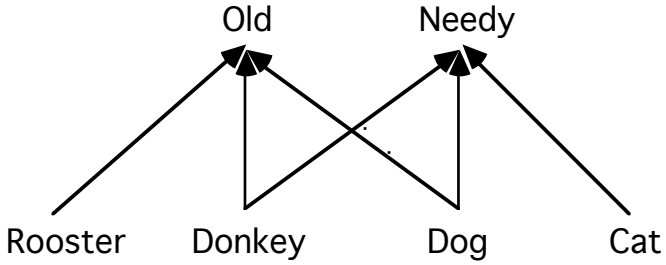


fig 2 SC-graph for D1

where the arcs should be labelled "is" and there should be two loops labelled "Bremen".

This example illustrates our general method of converting label information on directed arcs to atomic formulas in attribute values:

- each arc label becomes a predicate symbol
- each node label becomes a constant symbol
- each arc becomes an atomic fact
- each atomic fact is attached to the nodes at the end of the corresponding arc as part of their attribute value.

For the sake of readability we use infix notation for binary predicates and obvious linguistic conventions, so

<is(rooster,old),is(donkey,old),is(dog,old)>

becomes < rooster,donkey and dog are old>.

Remark

As there is no reason why arc labels should be binary predicate symbols,our conversion method also works for directed hypergraphs. Some data base systems (Ge,Ul) use such hypergraphs for "representing conceptual knowledge".

We must show that our attribute values,sets of atomic formulas, are elements of a ∑-algebra. Let ∑' be the signature of predicate and constant symbols given by the arc and node labels of a graph. Define ∑ as the extension of ∑' given by adding a sort atom , and operations

, : individual x individual => individual
• : atom x atom => atom .

Attributes take values in the term algebra T(∑,V) where V is a set of variables.

In our examples the signature ∑ will be :

rooster,donkey,dog,cat,crow,hoof,teeth,claws,		
judge,monster,assassin,witch,judgement,club,knife,nails,		
old,needy,worried,creature,domestic,cottage,	:	individual
, : individual x individual	=>	individual
≈: individual => individual,Bremen : individual	=>	atom
is,uses,Musician,*attacks_with* : individual x individual	=>	atom
: atom x atom	=>	atom
•		

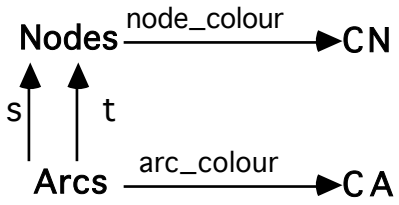
We will use three ∑-algebras:

SIGMA	the term algebra T(∑)	with no variables
SIGVAR	the term algebra T(∑,{x,y})	with two individual variables
ROBBER	a term algebra we introduce in section 3.	

When we write that a graphic is over a ∑-algebra A, we also specify that its label set is A's carrier domain for "individual". We will often write "formula set" for the combination of terms using ".".

The kinds of graphs used in (PEM) for representing relationships between facts are the SC-graphs where one has preordered sets, CA and CN, and functions

s,t: Arcs => Nodes
arc_colour : Arcs => CA
node_colour: Nodes => CN



Any SC-graph can be converted into a graphic. One can take CN as L and define ∑ as: an individual constant for each graph node, a predicate Ca: Node*Node -> atom for each arc colour in CA, and a conjunction operator ".". As the ∑ -algebra A one can take the term algebra T(∑) .For each node n in the SC-graph, the attribute value is the formula set:

Ca(m,n)	for each arc from m to n with colour Ca
Ca'(n,m)	for each arc from n to m with colour Ca'

and its label is its nodecolour. In the same way that we added "." earlier, we can convert the preorders in CA and CN into equations

Cn = Cn,Cn'	for Cn' ≤ Cn
Ca(m,n) =Ca(m,n).Ca'(m,n)	for Ca ≤ Ca'.

Remark

We use "≤" for both preorders (reflexive and transitive relations) and x~y for the "interchangeability" relation: x≤y and y≤x. When the preorders are trivial (x≤y iff x=y), we get the usual labelled graphs. If CN has only one element and CA has the trivial preorder, then we have the much studied *labelled transition systems.* When the preorders are flat (x≤y iff x=y or y=top) we get the partially coloured graphs with top as a new colour for "unknown","absent" or "transparent". The sets CA and CN can be lattices,unified algebras(Mo), or Boolean algebras.

In her work on type inheritance hierarchies (Pa) Lin Padgham has introduced an interesting kind of graph in which node labels have a lattice structure. These graphs, which she uses to clarify the traditional Clyde, Tweety and Nixon examples, can also be converted to graphics. Her graphs capture the distinctions made in earlier type inheritance hierarchies; in particular they capture Etherington's distinction in (Et) between "strict is a", "default is a", "strict is not a", "default is not a", and "exception" arcs. In (Pa) nodes are labelled by "sets of characteristics" and there are several kinds of nodes. There are two kinds of arcs: directed arcs for ≤ in the label lattice (usually inclusion), and undirected arcs for "inconsistent labels". As an example consider

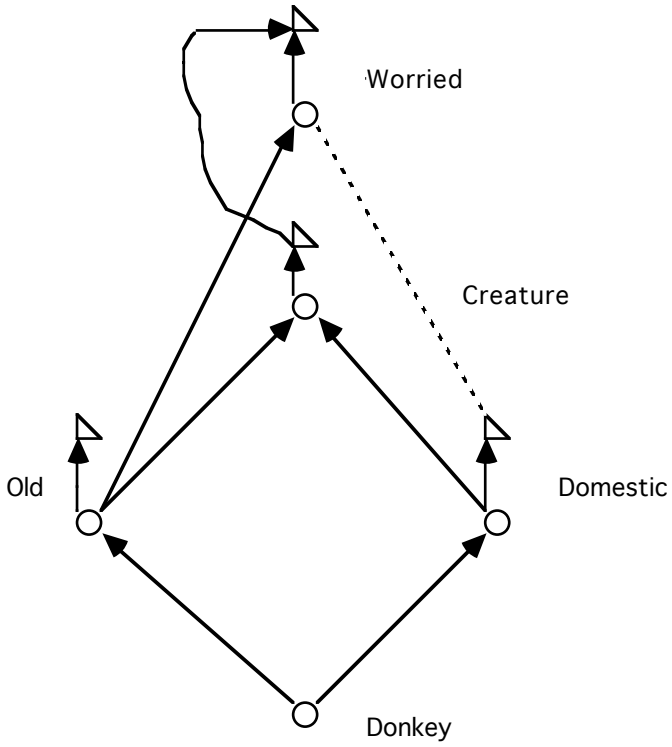


fig 3 Type inheritance graph

where circles represent "core nodes" and triangles represent "default nodes". Here the undirected arc represents

'Typical domestic creatures never worry'

and the directed arcs represent such beliefs as

'Old creatures always worry'
'Creatures often worry'

There are two ways of converting type inheritance hierarchies into graphics. One can introduce new predicate symbols in ∑ so our example becomes the graphic

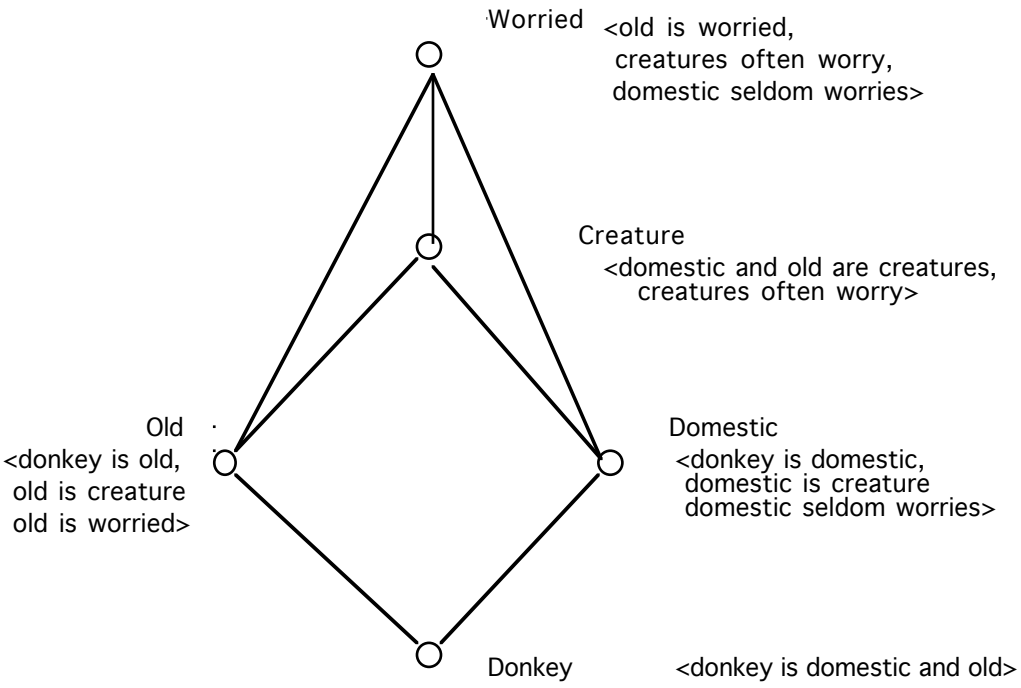


fig 4 Graphic D2 over an extension of SIGMA

Alternatively one can introduce a unary function symbol '≈' so our example becomes the graphic

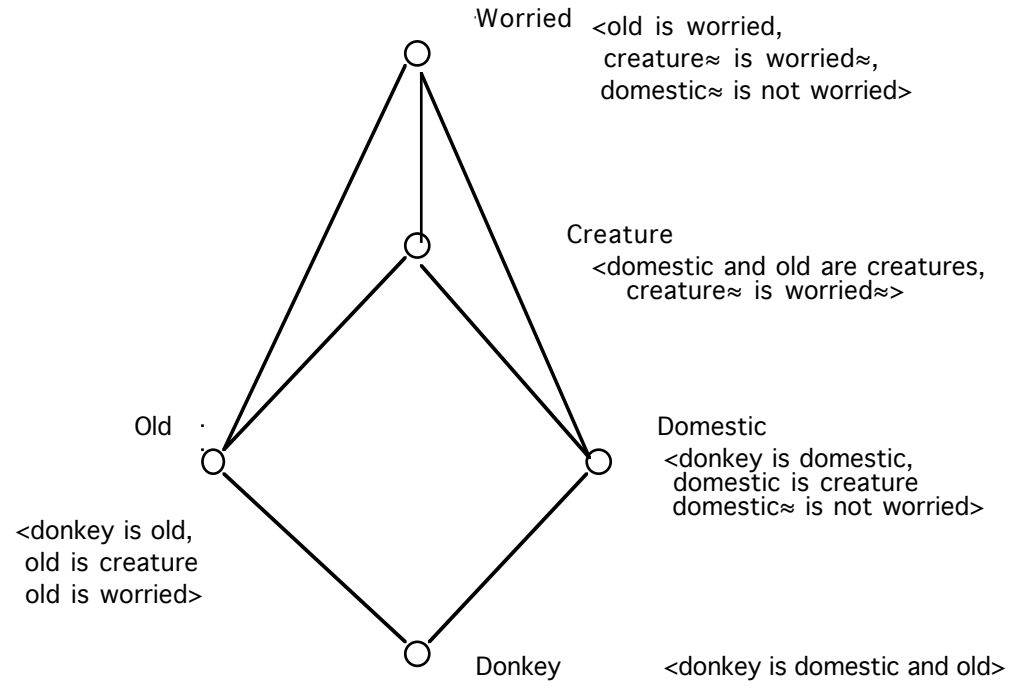


fig 5 Graphic D3 over SIGMA

In the next section we show how graphic productions can capture type inheritance reasoning.

From the logical programming viewpoint what we have done in this section is very natural. A graphic can be built on any collection C of facts in a logical programming language such as PROLOG. For each constant or variable cv, there is a vertex in G with label cv. The attribute value of the vertex cv is the set of facts in C that mention cv. There is an edge between cv1 and cv2, whenever there is some fact in C that mentions both cv1 and cv2. For an example the reader can take any graphic in this section as G, and the union of the attribute values at its vertices as C. Note also that one can have vertices in a graphic for Prolog predicate symbols. The attribute value for such a vertex is the atomic formulas using the corresponding predicate.We shall often use this option,our decision that Bremen' should be a predicate and 'old' an individual was arbitrary.

The approach in the last paragraph can be used for any institution(GB) or gallery(Ma). Any finitely presented theory in any institution or gallery can be represented as a graphic. However the approach may give pretty weird graphics, if one does not keep our separation between atomic formulas and the rules to drive more complicated formulas. Such rules should be represented as graphic productions.

#2 Productions and rules

In this section we describe how graphic productions can capture "rule" information: in traditional IF-THEN rules, in Prolog rules with *assert* and *retract*, and in type reasoning in inheritance hierarchies. Consider a rule like "if two old and needy creatures meet in Bremen, then they can form a musical group". In (PEM) this rule would be represented as a graph production

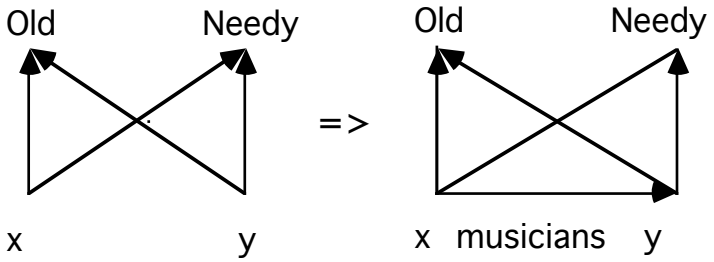
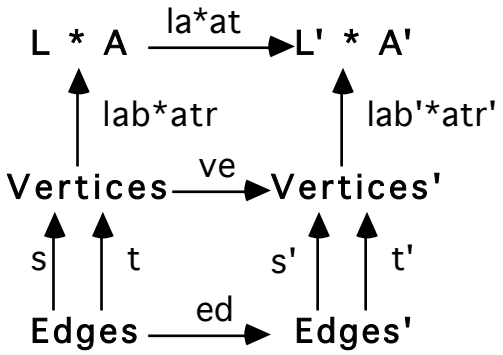


fig 6 SC-graph production

and the partial ordering x ≤ Bremen, y ≤ Bremen. We will give a graphic production for this rule,after we have defined what strict graphic morphisms are.

Definition 2 A *strict morphism* from a graphic (s,t,lab,atr,L,A) to a graphic (s',t',lab',atr',L',A') consists of a graph morphism (ve,ed) and

la: L=> L'
at: A=>A'
such that
lab';ve = la;lab
atr';ve = at;atr



where at is a ∑-algebra homomorphism. Strict graphic morphisms suffice for all our examples, but technical problems force us to define a more general notion of graphic morphism in the last section. Let us show that there is a strict morphism from the graphic L1

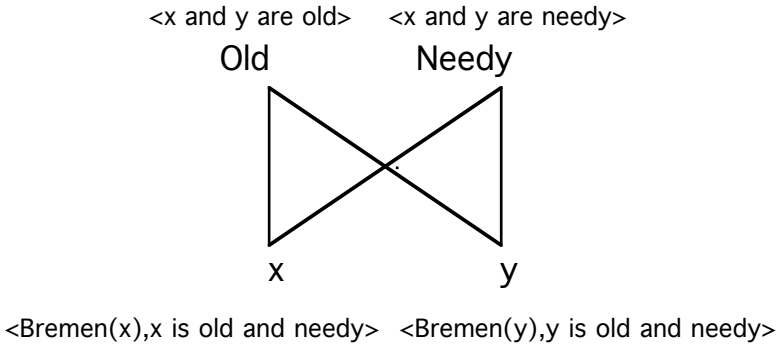


fig 7 Graphic L1 over SIGVAR

to the graphic R1

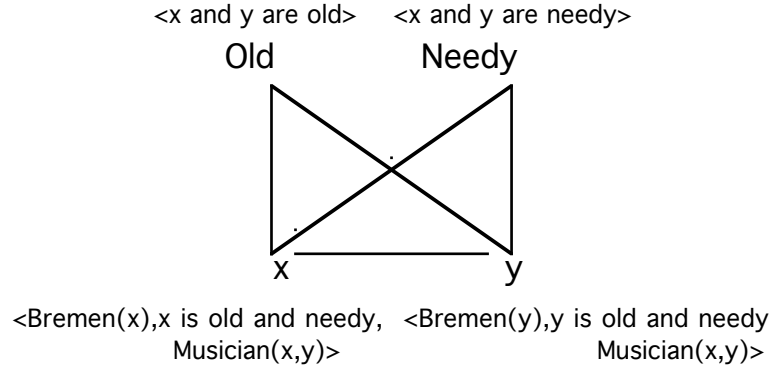


fig 8 Graphic R1 over SIGVAR

The obvious embedding as labelled graphs and the definition:

at(S) = if Bremen(x) or Bremen(y) in S then S U{Musician(x,y)} else S

give the required graphic morphism r1 from L1 to R1.Graphic morphisms can do many things: (1) add or remove vertices, (2) add or remove edges, (3) change vertex labels, (4) change attribute values.Our morphism r1: L1 => R1 illustrates only (2) and (4) and it is monotonic in that: it is an embedding of labelled graphs and at(S) always contains S.

There is an occurrence of L1 in our earlier graphic D1; if one substitutes 'donkey' for x and 'dog' for y, one gets a graphic morphism d: L1 => D1. The label part of the morphism is given by :

la(cv) = if cv is x then donkey else if cv is y then dog else cv

and this substitution gives the attribute part :

at(S) = the result of applying substitution la to S.

The pushout of this graphic morphism d with the 'rule' morphism r1: L1 => R1 gives the graphic H1:

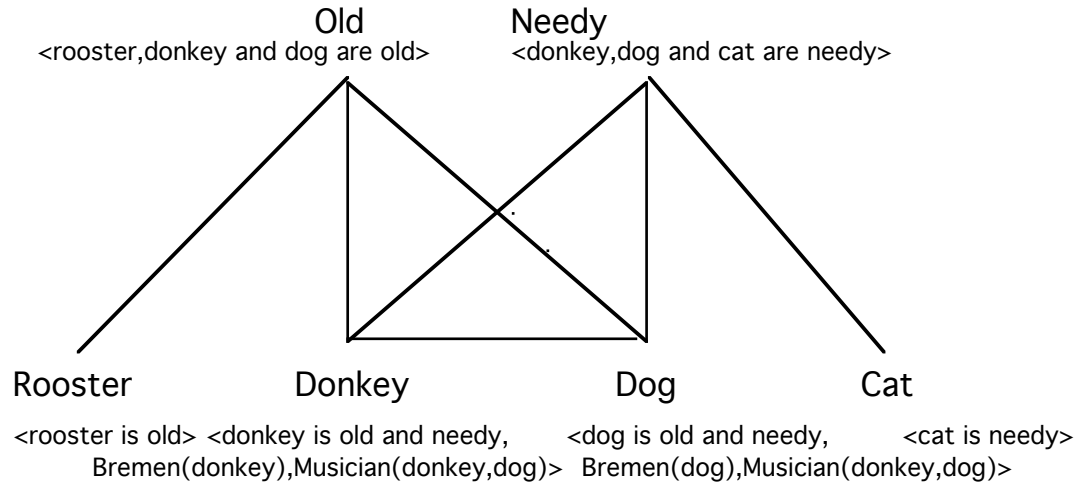


fig 9 Graphic H1 over SIGMA

Later we will show that the pushouts of graphic morphisms always exist, but now we define graphic productions.

Definition 3 A graphic production is an ordered pair of graphic morphisms l: K => L and r: K => R . It is simple if l is an identity.
It is logical if K,L,R are graphics over the same ∑-algebra.

The graphic production (l,r) can be applied when one has a graphic morphism d: K => D and the pushout diagram

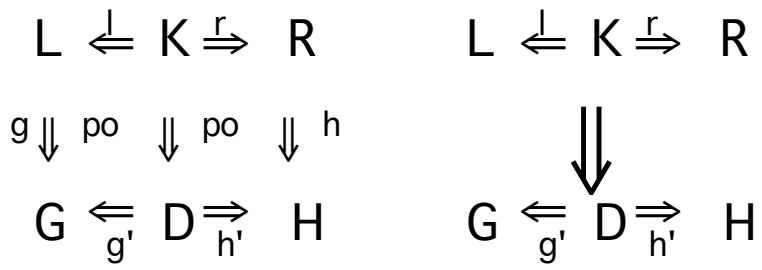


fig 10 (normal) Application of a (logical) production

shows how the production transforms G to the graphic H. Sometimes the application of graphic productions can give *surprises* (more on this in the last section), and it might be wise for an expert system to keep to normal applications.

Definition 4 The application of a logical graphic production (l,r) is normal if the label and attribute parts of the pushout morphisms g: L=>G, h: R=>H are the same as those in d: K=>D.

Our example showed a normal application of the simple graphic production (id,r1:L1=>R1), but more general graphic productions are also useful in expert systems; to quote (PEM):

> "This allows for a uniform treatment of 'forward chaining' systems, where changes in the working memory data produce a match for the left hand side of a rule which can then be applied, and 'backwards chaining' systems where rules are examined in search for a match with a fixed goal".

We should note the 'applicability' problem with graphic productions: to determine if a production (l:K=>L,r:K=>R) can be used to transform a graphic G, one needs the morphism d:K=>D - it is not enough to find a morphism g:L=>G. There may be zero,one or many morphisms d, whose pushout with l is g. Nevertheless for all simple productions and most productions, that arise in practice, there is a unique d for any g.

Our example of a graphic production is close both to the corresponding Prolog rule:

```
Musicians(x,y) :- Bremen(x),Old(x),Needy(x),Bremen(y),Old(y),Needy(y).
```

and to the running example of a rule in (PEM):

> "if two women have the same mother, then they are sisters".

From the discussion at the end of the last section, it is clear that any 'pure' Prolog rule gives a simple graphic production. If the rule also uses *assert*, the corresponding graphic production is still simple because *assert* can only add new edges and extend attribute values. Only when a rule uses *retract* do we have to go beyond simple productions and let the graphic morphism l:K=>L remove edges and reduce attribute values. Note however that *assert* and *retract* have no influence on the unifier when a Prolog rule is applied. This corresponds to the fact that the substitution map is determined by the occurrence map d:K=>D when a graphic production is applied.

For another example of (PEM)'s "rules containing reasoning knowledge ... represented by graph productions" let us look at type inheritance reasoning. Previous techniques for type reasoning(Et,Sa,To) are all captured in (Pa) where Padgham describes her version of type reasoning in a way that we can translate into graphic productions. One of these productions is

(INH) x _____ y => x _____ y
< x is y> <x is y,P(y)> <x is y,P(x)> <x is y,P(y)>

Using this repeatedly on the graphic D3 gives

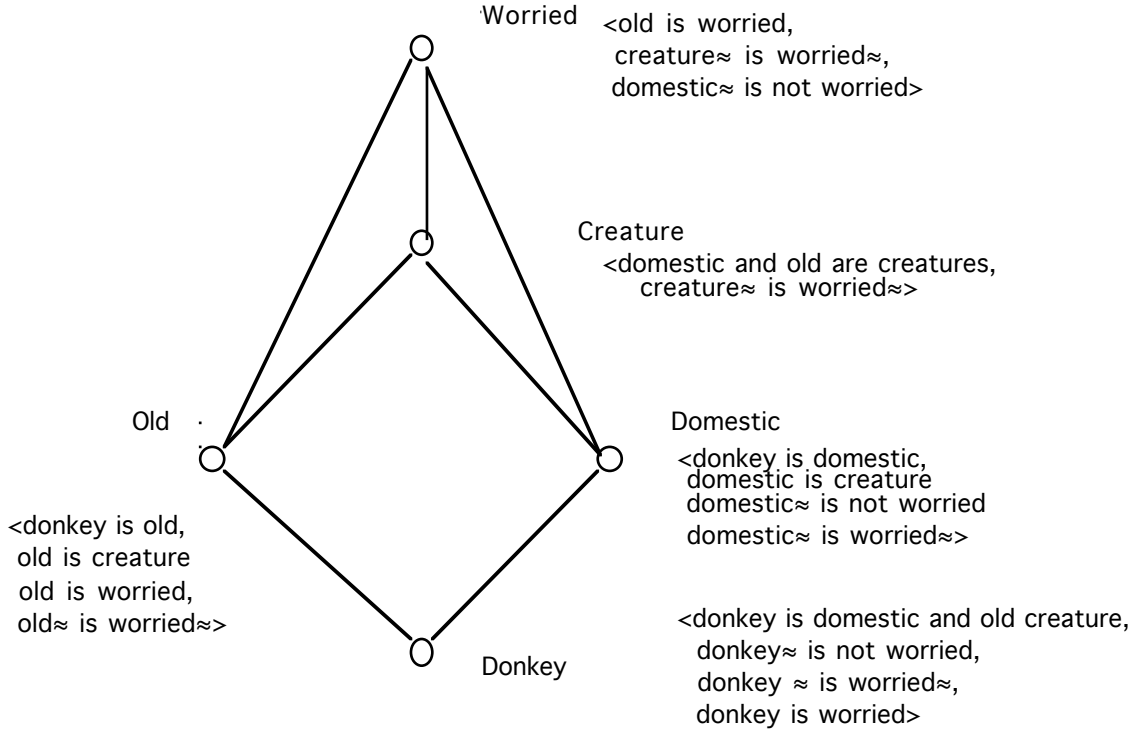


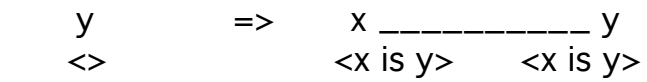
fig 11 Graphic H2 over SIGMA

Clearly we need productions like

(DEL) x_____y => x _____ y
<x is y,x≈ is not y> <x.is y> <x is y> <x is y>

to remove 'default' information when it conflicts with 'core' information. Using this production we can delete the unwanted
'donkey≈ is not worried' from the attribute of 'Donkey' in the graphic H2.

So far we have not seen graphic productions that create new vertices, so we give



Using this production one can add old roosters,cats and dogs to the graphic H2. For any symbol in the signature ∑ we can introduce a production for introducing facts using the symbol into graphics.

#3 Analogies

There is a large literature on reasoning by analogy (Pr), and some of it is concerned with whether an analogy is a map of a situation G into a situation H or whether G and H must have a common structure or pattern D. This dispute is related to whether analogies should be modelled by simple graphic productions or whether we need productions that are not simple. Although we do not take sides in this dispute, we simplify this section by only using simple graphic productions.

Frequently situations can be described by graphics and a map from situation G into situation H can be described by a graphic morphism. We maintain that the interesting 'analogy' graphic morphisms are pushouts of simple graphic productions.

Definition 5 A graphic production (l:K=>L, r:K=>R) is analogical if L and R are over different ∑-algebras.

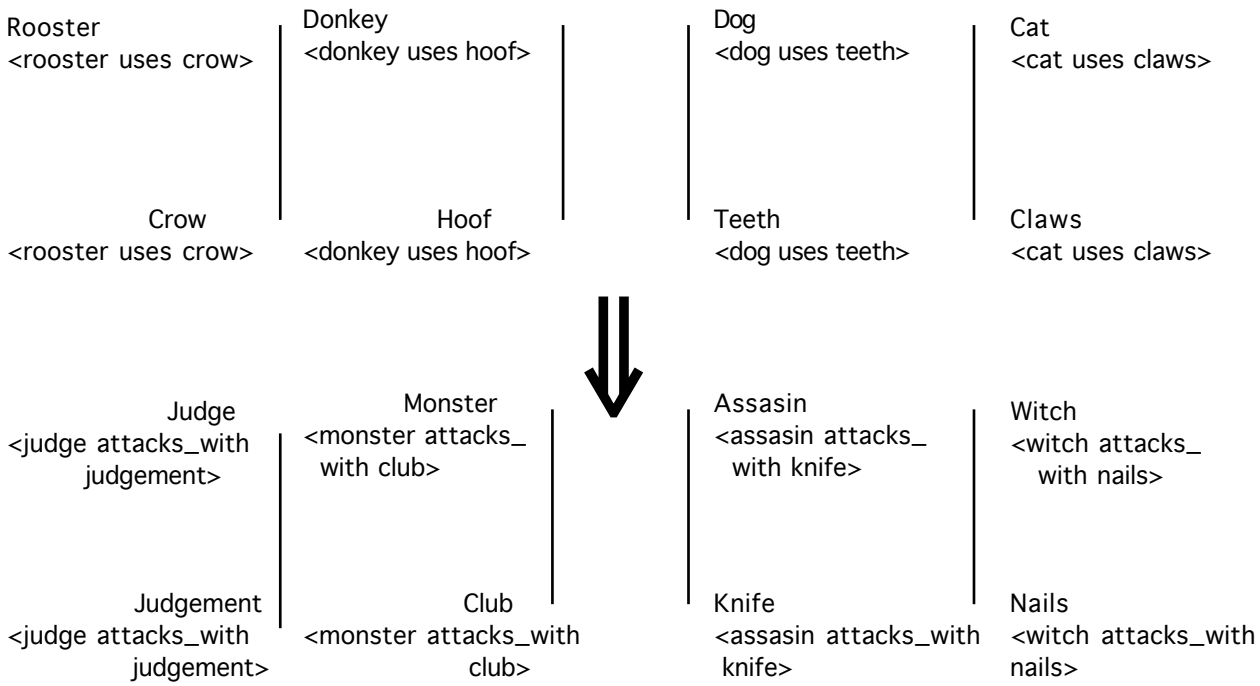


fig 12 Analogical graphic production from L4 to R4

In this example the signature ∑ is that given in section1, the graph part of the morphism is trivial and the other parts are given by the ∑-algebra homomorphism at: SIGMA => ROBBER

rooster	->	judge crow		->	judgement	
donkey	->	monster	hoof	->	club	
dog	->	assassin	teeth	->	knife	
cat	->	witch nails		->	claws	
uses	->	attacks_with.				

The domain of the morphism is the ∑-term algebra SIGMA, the codomain ROBBER is also a 'linguistic' ∑-algebra with 'words' (formula sets) as elements of the carrier for 'individual' (atom) The homomorphism at is a "quotient" and ROBBER is the result of dropping all terms with "rooster, donkey, dog, cat, uses, crow, hoof, teeth, nails" from SIGMA. Applying this production to the graphic

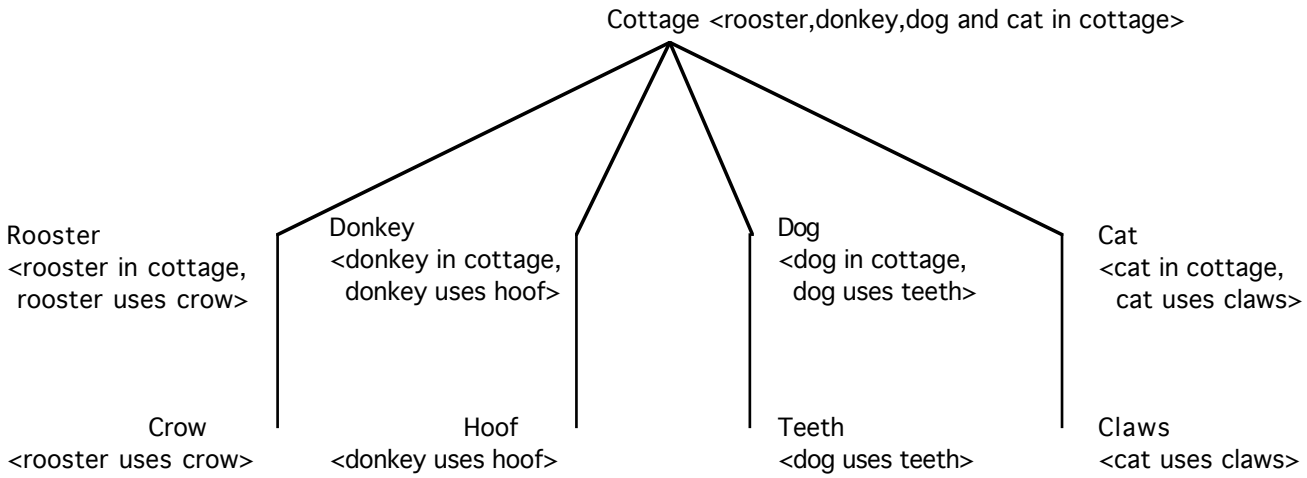


fig 13 Graphic D4 over ROBBER

gives the graphic

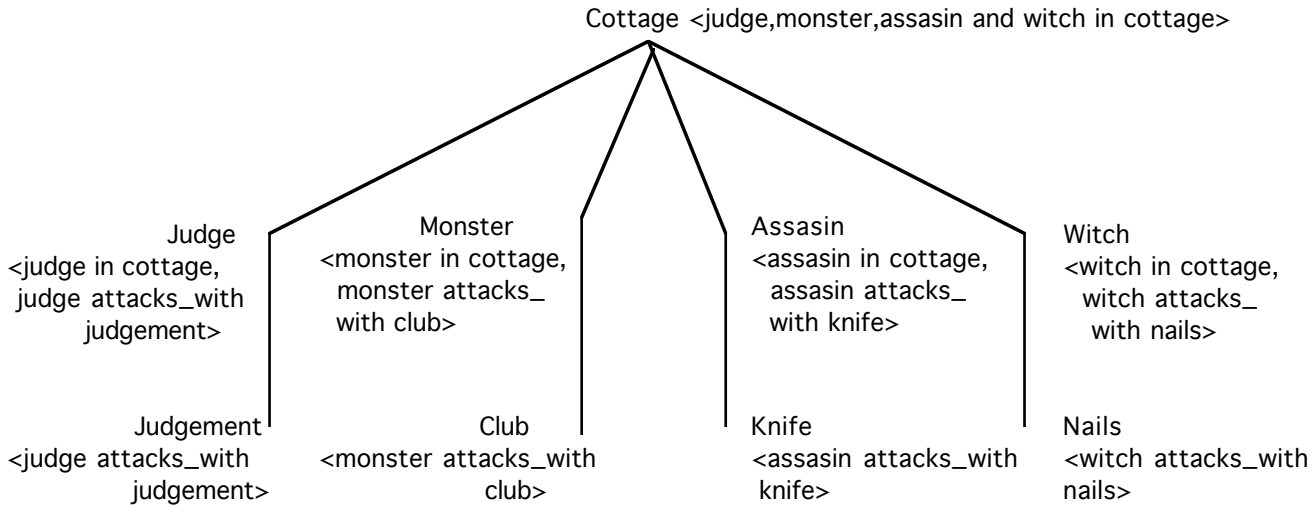


fig 14 Graphic H4 over ROBBER

This example shows no *surprises* because it illustrates a normal application of an analogical graphic production.

Definition 6 The application of an analogical graphic production (l,r) is normal if the label and attribute parts of the pushout morphisms

g': D=>G, h': D=>H

are the same as those in l: K=>L, r: K=>R respectively.

Our approach to analogies assumes that 'situations' can be described by graphics. Whenever situations can be described by terms in a ∑-algebra, and a map from situation G into situation H can be described by a ∑-algebra homomorphism, this assumption is reasonable. The discussion at the end of

section 1 about graphics for collections of logical facts in Prolog - indeed any logical programming language, institution or gallery - shows that the assumption is highly reasonable.

#4 Technicalities

In this section we define the precise notion of graphic morphism in such a way that pushouts of graphics always exist. The underlying idea is that graphic morphisms must be continuous in operations for 'gluing' labels and attributes.

Presentations of a graphic only reveal part of the label set L and the underlying ∑-algebra A ; the range of 'lab' is only a subset of L, the range of 'atr' is only part of A and some of the ∑-symbols may not be mentioned. Presentations of graphics must be supplemented with a specification of L with its gluing operation "," and A with its gluing operations ".". Presentations of a graphic morphism only reveal part of the underlying labelling function 'la' and ∑-algebra homomorphism 'at'.

We will assume that each label set L has a binary operation "," and each ∑-algebra A has a binary operation ".". We assume these binary operations are associative and commutative. We will write

, S for l1.l2.l3	when S = (l1,l2,l3,..)	is a subset of L
. S for a1,a2,a3	when S = (a1,a2,a3...)	is a subset of A
la: L => L' for la: L => L'	such that la(l1,l2)	= la(l1),la(l2)
at: A => A' for at:A => A'	such that at(a1.a2)	= at(a1),.at(a2).

There is no loss of generality with our assumptions, because one can always replace L and each carrier domain of A by their power sets, so union is available for "." and ",".

Definition 7 A *morphism* from a graphic (s,t,lab,atr,L,A) to a graphic (s',t',lab',atr',L',A') consists of a graph morphism (ve,ed) and

la: L => L' such that lab';ve = la;lab
at:A => A' such that atr';ve = at;atr

where at is a ∑-algebra homomorphism,

lab(v) is ,{lab(v)! ver(v') = ver(v)} and atr(v) is .{atr(v')! ver(v') = ver(v)}.

Our earlier definition of strict graphic morphisms corresponds to the case when lab = lab and atr = atr. All of our examples have also had 'la' and 'at' generated from a total map on 'singletons', but partial maps also generate strict graphic morphisms.

Example

Let A and A' be SIGVAR,the term algebra when variables are allowed. Let L and L' be the carrier domain for *individual* in SIGVAR. Any substitution sub: Var -> L gives :

la: L => L' where at(l) is result of applying sub to l
at:A => A' where at(a) is result of applying sub to a.

Any graph morphism (ve,ed) gives a graphic morphism (ve,ed,la,at). Such graphic morphisms are called 'graphic substitutions'.

Any graphic morphism can be split into a graph morphism, a label function la and a ∑-algebra homomorphism 'at'. Each graphic morphism r = <rve,red,rla,rat> from K to R is the composition of two graphic morphisms:

<rve,red,id,id>, <id,id,rla,rat>

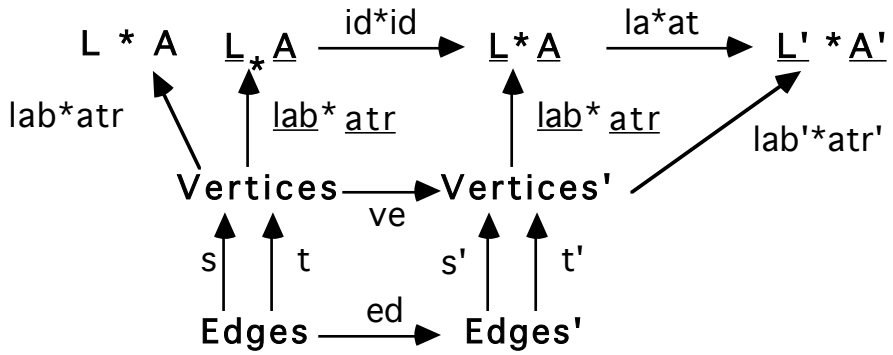


fig 15 Decomposition of a graphic morphism

in either order. In (PEM, lemma 3.7) there is a similar splitting of 'SC-graph morphisms' into first 'g-substitutions' then 'colour preserving graph morphisms',but this order matters because 'g-substitutions' may not be continuous.

Comment

Now we can be more precise about what can be done by a graphic morphism r: K => R

(1') r can add vertices - ve need not be a surjection
(1'') r can glue vertices- ve need not be an injection
(2') r can add edges - ed need not be a surjection
(2'') r can glue edges - ed need not be an injection
(3) r can change labels - even if la is identity,
the label of glued vertices may change
(4) r can change attributes - even if at is identity,
the attribute of glued vertices may change.

When a graphic production (l: K => L, r: K => R) is applied to a graphic G, it may remove vertices and edges because the morphism l can add vertices and edges.

Suppose we have another graphic morphism k = <kve,ked,kla,kat> from K to D. As graphs, sets and ∑-algebras have pushouts, one can form the pushout of our graphic morphisms. One might expect trouble with the vertices in R+D that must be glued together.However the morphism requirements:

Rlab;rve =	rla;Klab	Ratr;rve =	rat;Katr
Dlab;kve =	kla;Klab	Ratr;kve =	kat;Katr

show that glued together vertices get the label la''(Klab(v)) and attribute at''(Katr(v)), where la'' is the pushout of rla and kla, and at'' is the pushout of rat and kat. Thus pushouts of graphic morphisms always exist and the problems of (PEM) were caused by the fact that their 'g-substitutions' do not always have pushouts. Note also that their 'g-substitutions' are somewhat more general than our graphic substitutions, because they do not insist on our 'vertex-independent functions', la and at.

Definition 8 A ℂ-surprise is a pair (k,r) of graphic morphisms in the class C whose pushout is not in C.

A surprise is a pair (k,r) of graphic morphisms whose label and attribute parts satisfy neither k= r;f nor r= k;f for any morphism f.

Usually C-surprises are also surprises, because k = r;f gives the pushout k;id = r;f, r=k;f gives the pushout r;id = k;f, and C is a full subcategory of graphics.

For C we can take the class of graphic substitutions. Since substitutions do not usually commute, the pushout of two substitutions is rarely a substitution. As the pushout of two substitutions, s1 and s2, is s3(x) = (s1(x).s2(x), s2(x).s1(x)), we get a graphic substitution surprise, when d and r are graphic substitutions that do not commute. We are surprised to find that vertex labels and attributes are complex terms with gluing operations. In logical graphic productions for "logical reasoning in expert systems" one usually has identity substitutions in the algebraic part and there will be no overlap with any substitution in K=>D. Pushouts will be substitutions,and there will be no *surprises*.

Let us continue our search for natural classes of graphic morphisms that rarely give surprises.Remember our 2-way splitting of graphic morphisms. It gives a 2-way splitting of pushouts

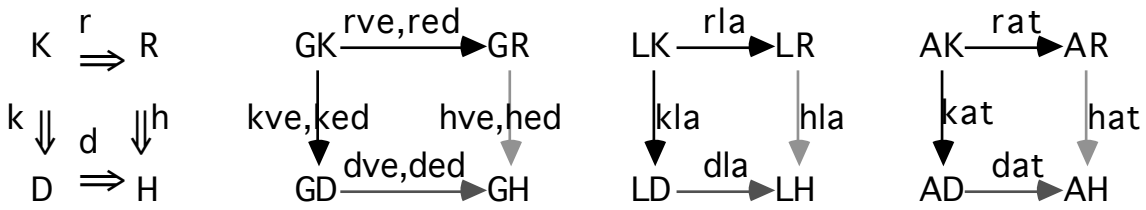


fig 16 Decomposition of pushouts

The graph pushout just tells about "gluing", it never gives surprises. The label pushout does not give surprises if we have either LK = LR LD = LH kla =hla or LK = LD LR = LH rla = dla. The attribute pushout does not give surprises, if we have either AK = AR AD = AH kat =hat or AK = AD AR = AH rat = dat. This analysis shows why normal applications of both logical and analogical productions do not give *surprises*.

Graphics and their morphisms form a category **Graphic** which has several interesting subcategories. Many of the graphics and graphic morphisms in this paper have four common properties; they are algebraic, proper, powered and linguistic.

Definition 9 A graphic over A is algebraic if its label set L is a subset of A. A graphic morphism r: K =>R is algebraic if rla is the restriction of rat to the label set of K. **Agraphic** is the category of algebraic graphics and morphisms.

This category is attractive because one does not have to treat labels and attributes separately. All the graphics in this paper are algebraic; we have followed the convention: the label set L for A is its carrier domain for 'individual'.

Definition 10 A graphic over A is powered if its attribute values are sets. A graphic morphism r: K =>R is powered if rat is a monotonic function on sets. **2graphic** is the category of powered graphics and morphisms.
This category is attractive because unions and intersections of sets are natural interpretations of our gluing operations. All the graphics in this paper are isomorphic to powered graphics; one can replace terms like d by the singleton {d} and terms like d1.d2.d3 by the set {d1,d2,d3}. In all our figures we have made this replacement.

Definition 11 A graphic over A is linguistic if A is given by a signature morphism from ∑ . A graphic morphism r: K =>R is linguistic if la and at are given by a signature morphism. **Lgraphic** is the category of linguistic graphics and morphisms.

A signature morphism from ∑ to ∑' is a function from the symbols and variables of ∑ to the symbols and variables of ∑'. Every signature

∑'corresponds to a context-free grammar by : constants map into terminals, sorts map into non-terminals, and for each operation op:s1.s2...-> s0 one has both a terminal 'op' and a grammar production

s0 ::= 'op(' s1 ',' s2 .... ')'

The language generated by this grammar is exactly the term algebra T(∑'), and it is a ∑ -algebra when one has a signature morphism from ∑ to ∑'.When ∑ is the signature in section 1, the identity signature morphism on ∑ gives SIGMA, the embedding of ∑ in ∑ U {x,y} gives SIGVAR, and the signature morphism in section 3 gives ROBBER. Each of these signature morphisms gives a linguistic morphism.

Definition 12 A graphic over A is reachable if its labelling and attribute functions can be factored through the term algebra. A graphic morphism r: K =>R is reachable if it can be factored through the term algebra. **Rgraphic** is the category of reachable graphics and morphisms.

This category is attractive because all vertices of a reachable graphic have "names" and two vertices with the same name have the same label and attribute values. Almost all of the graphics in this paper are reachable because their vertices have different "∑-individuals" as labels. For more on categories of reachable objects and the connection to the theory of institutions and algebraic specifications, one can consult (AT).

It is instructive to make the category **Rgraphic** into a gallery **RG.** The signatures of **RG** are the usual signatures of first order logic. The structures of **RG** are the usual ∑ - algebras, supplemented by a labelset.The frames of **RG** are the graphics over the term ∑ - algebras. The valuation function of **RG** is given by

val(A,L,e) = the graphic with the same graph as e but
atr is the A-interpretation of the e-attribute
lab is the L-interpretation of the e-labels

One gets interesting subgalleries of **RG**, if one places restrictions on the structures. One can restrict to the algebraic structures (A,L) where L is a designated subset of A and structure morphisms (at,la) have la as the restriction of at to L. One can restrict to the powered structures (A,L), where all terms interpreted as sets. One can restrict to the linguistic structures (A,L) , where all terms are interpreted as sentences in a grammar.

Let us close this paper by describing "the passage to the metalevel". Once we have **RG** or any of its subgalleries we can apply the construction at the end of section1 to get "metagraphics". Thus we can define C as the collection of graphics in this paper, we can introduce "metavertices" for each vertex label, "metaedges" between metavertices whose attributes

overlap - i.e. metavertices that occur together in some graphic in this paper-and then build a large metagraphic:

metavertexlabel	*metavertexattribute*
Rooster	H1,D1,D4,L4
Dog	H1,D1,D4,L4
Cat	H1,D1,D4,L4
Donkey	H1,H2,D1,D2,D3,D4,L4
Creature	H2,D2,D3
Worried	H2,D2,D3
Domestic	H2,D2,D3
Old	H2,D1,D2,D3,L1,R1,H1
Needy	L1,R1,H1,D1
x	L1,R1
y	L1,R1
Crow	L4,D4
Hoof	L4,D4
Claws	L4,D4
Teeth	L4,D4
Cottage	L4,D4,H4
Judge	R4,D4,H4
Monster	R4,D4,H4
Assassin	R4,D4,H4
Witch	R4,D4,H4
Judgement	R4,D4,H4
Club	R4,D4,H4
Knife	R4,D4,H4
Nails	R4,D4,H4

Our way of constructing metagraphics seems to mirror quite precisely the widespread use of metafacts and metarules in practical expert systems.

References

(BP) W. Bibel, B. Petkoff "Artificial Intelligence methodology, systems, applications",N.Holland 1985,ISBN 0-4444-87743-6

(ENRL) H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld "Graph grammars and their application to computer science" Springer LNCS 291.

(Et) D.W. Etherington "Formalizing nonmonotonic reasoning systems" Art.Int 31 (1987) 41-85 .

(GB) J.A. Goguen, R.M. Burstall "Introducing institutions" Springer LNCS 164 pp 221-256.

(Ge) I. Georgescu "The hypernets method for representing knowledge" pp 47-58 in (BP).

(HM) L. Hess, B.H. Mayoh "Graphics and their grammars" pp 232-249 in (ENRL).

(Ma) B.H. Mayoh "Unified theory of knowledge representation" pp 35-46 in (BP).

(Mo) P.D. Mosses "Unified algebras and institutions" LICS89, Fourth Ann. Symp. Logic in Comp. Sci.

(Pa) L. Padgham "A model and representation for type information and its use in reasoning with defaults" Proc. AAAI88 (1988) 409-414.

(PEM) F. Parisi-Presicce, H. Ehrig, U. Montanari "Graph Rewriting with unification and composition" pp 496-514 in (ENRL).

(Pr) A. Prieditis(ed) "Analogica" ISBN0-273-8780-0 Pitman1983.

(Sa) E. Sandewall "Non-monotonic inference rules for multiple inheritance with exceptions" Proc. IEEE 74(1986)1345-1353.

(ST) D. Sannella, A. Tarlecki "On observational equivalence and algebraic specification" J. Comp. Sys. Sci 34 (1987) 150-178.

(To) D.S. Touretzky "The mathematics of inheritance systems" M. Kaufmann Pub. 1986.

(Ul) J.D. Ullman "Principles of database systems" ISBN0-7167-8069-0 Comp. Sci Pr. 1982.