Growing interest in visual languages has triggered new extended research into the specification and parsing of multi-dimensional structures. Though some approaches to general descriptive formalisms for visual structures have been made in the last years, there is still no visual specification formalism for visual languages. The paper discusses the need for a visual formalism and introduces such a technique by augmenting logic programming with picture terms. Picture terms can be considered as partially specified pictures. We define how to match picture terms and how to integrate matching with the execution of logic programs. Based upon this extension, picture clause grammars (PCGs) are introduced. Due to their declarative nature PCGs are formal visual specifications of visual languages and executable parsers at the same time. Like Definite Clause Grammars, PCGs can be used for parsing and syntax directed translation. The executability of PCGs is demonstrated by defining their translation to logic programs employing picture terms. From the different types of PCGs that are needed to describe a picture a Chomsky-like hierarchy of regular, context-free, and non-context-free pictures can be derived. We will give examples for picture languages and the grammars that are used for parsing and translating them.

1. Introduction

In recent years visual languages (VL) have become more and more important in several different application areas like database query languages and programming languages. Certainly, they will even gain in importance when paper-like interfaces [WRE89, Kim91] become everyday tools. Moreover, the development of our understanding of visual languages and their theory can be regarded as a precondition for the useful application of paper-like interfaces.

Most of the visual languages proposed or implemented have only been defined informally. Several textual specification formalisms have been introduced recently, and many of them have suggested important new aspects. A line of research oriented towards parsing has used extended textual grammars. One approach is to describe pictures by sequences of symbols containing special elements that describe relative positions of picture objects [Tor90]. This technique can take advantage of the fact that such structures can be parsed with adapted standard algorithms [CC90, CTC91] and has been used as the basis for the SIL visual language compiler [YC90].

Instead of sequential structures Relation Grammars [CGN+90] use sets of objects and spatial relations between them to describe a picture. A parser for Relation Grammars is described in [FPT+91]. A very expressive formalism that supports non tree-like parse structures are Picture Layout Grammars [GR89]. The interesting logic approach advocated in [HM86, HM90, HMO91] uses first order theories for the specification of pictures and constraint solvers for parsing. Also algebraic formalisms have been described for some classes of pictures [Jun89]. In [BL89] unmodified standard grammars that work on textual descriptions of pictures are used, i.e., the grammar formalism itself does not have any special properties. In contrast to all these approaches which use declarative picture descriptions Graphic Functional Grammars [KM91] parse sequences of graphic function calls that generate a picture. Other related areas of research go into specifications aimed at supporting syntax-directed editing of VLs [ElK91, Göt89] and into non-universal formalisms for individual languages [Mar90, MTW91].

But surprisingly there is no type of executable specification that is a visual language itself except for Lakin’s executable graphics [Lak86, Lak87], which has not been formalized. The approach presented here tries to address this issue. If the long-term goal for the implementation of VLs is to build visual compiler development environments for VLs — and we feel this should be an aim: to catch up with the state of the art in conventional compiler implementation environments one day — then the most important key requirements for a VL specification formalism are:

- The formalism should be visual itself, i.e., there should be a very tight, intuitively comprehensible correspondence between the object of description and the description itself.
- It should be flexible enough to support more than simple diagram languages\(^2\) [Shu88], e.g., advanced visual formalisms like higraphs or statecharts [Har88].
- Specifications should have a formally well-defined semantics.
- It should be possible to derive parsers from the specification automatically.
- As with conventional grammars, the specification should be declarative.

Actually, the most important feature of all might be the first one because it is not only the key to visual specifications of VLs but also the general key to VLs that talk about visual (or spatial) scenarios: If we want such VLs to be more than standard languages in disguise, we have to learn how to describe pictures by pictures. The potential for such languages that use schematic example sketches to talk about multi-dimensional arrangements (blackboard languages [Mey92], an extension of iconic visual information processing languages [Cha87, Cha89]) can be found in many areas, not only in visual grammars for VLs but also in visual query languages for spatial information systems and visual languages for pictorial databases.

The structure of the paper is as follows: Section 2 addresses the general problem of propositional picture representation. The next two sections introduce picture terms (partially specified pictures) and define how picture terms can be matched with pictures. Section 5 integrates the idea of picture terms with logic programming, which leads to a picture programming language. Picture clause grammars (PCGs) — an extension to picture programming — are explained in the sixth section, and a subclass of them, regular PCGs, is defined. The following section extends this notion to context-free PCGs. Both sections include examples of the grammar types explained. The last section concludes with an outlook on to the future development of the formalism.

---

\(^2\) By simple diagram languages we mean languages whose sentences are graphs consisting of icons (nodes) and labelled connections between them (edges), like electronic circuit diagrams, dataflow graphs, etc. [BGST90]
2. What is a Picture?

Before we are going to discuss the description of pictures by pictures we must make clear what the object of description is. Which kind of pictures do we want to handle? What do we think are the basic features of pictures? Having answered these questions, we can reformulate the problem of picture description as finding a mapping between a describing picture and the picture described by it. In this context we regard pictures as composed of basic visual or spatial objects and spatial relationships between these objects. The contrary position, which will not be considered, is to view a picture as an \( n \)-dimensional space with color- or grey-value attributes, i.e. raster pictures. Of course, this choice is not absolutely general and has been influenced by the kind of languages we want to describe. Some types of image processing languages can not be described sensibly in such a framework, but it is easy to see that iconic languages can be described in this way. This “objects embedded in space” approach is as well suitable for blackboard languages.

Describing a picture as a composition of objects and their relationships requires a process that splits a picture into its constituting objects. This can be done in two ways: a picture given as a raster image can be decomposed by a pattern recognition process or it can directly be entered as a set of objects, e.g., by using a graphical editor for a visual language, whose basic operations are to draw single picture objects\(^3\). We will not discuss this phase here and assume that the picture is given already decomposed by some preprocess. Such a preprocess can in analogy to conventional compilers be considered as the lexical analysis [CR91]. The picture in Figure 1 can, e.g., be described by the following set of tokens represented by facts:

\[
\begin{align*}
\text{object}(K_1, \text{circle}). \\
\text{object}(K_2, \text{rectangle}). \\
\text{object}(K_3, \text{line}). \\
\text{relation}(\text{touches}, K_3, K_1). \\
\text{relation}(\text{touches}, K_3, K_2).
\end{align*}
\]

![Figure 1](image)

where the \( K_n \) are object identifiers, the second argument of every object is its type, and the first argument of every relation is the type of spatial relation for the given objects. The propositional representation by objects and relations eliminates every constraint on the exact positions, distances, etc. Rotating the picture, e.g., would not change its meaning. This simple representation satisfies only in the most simple cases. Certainly, it is not reasonable to invent a new object type for every different flavor of an object. A filled rectangle, an empty rectangle, say, are both rectangles, as well as a dashed grey line and a solid green line are both lines. Another case is the representation of labels, which is very important for most types of diagrams. This can be modelled by allowing objects to have attributes. A label can then be regarded as a box with the text contained in it as one of its attributes. The modified picture in Figure 2 will now be represented as:

---

\(^3\) Resolving ambiguities is considered a task for the recognition process. Interactively entered pictures composed of separate objects step by step will normally not be ambiguous in this way.
Certainly, much of our interpretation of the picture is captured by the selection of the concepts that are used in the representation, i.e., of object and relationship types. Furthermore, it seems impossible to define a “complete” set of concepts that is capable of describing every picture. For the discussion of picture grammars, we will use the set of types given in Tables 1 and 2, which is obviously sufficient for most of the simple diagrammatic VLs and blackboard languages.

<table>
<thead>
<tr>
<th>point</th>
<th>line</th>
<th>curve</th>
<th>polygon</th>
<th>closed_curve</th>
<th>rectangle</th>
<th>circle</th>
<th>label</th>
</tr>
</thead>
</table>

Table 1

<table>
<thead>
<tr>
<th>any</th>
<th>$2d \times any$</th>
<th>$1d \times 1d$</th>
<th>$0d \times 1d$</th>
<th>label $\times any$</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists</td>
<td>contains</td>
<td>intersects</td>
<td>touches</td>
<td>attached</td>
</tr>
</tbody>
</table>

Table 2

While the existence of an object in a picture is represented simply by an object fact, the four binary relation types are represented by relation facts. Like the set of object types, the set of relationship types cannot be “complete”. For every fixed set of relations, there are concepts that cannot easily be expressed in terms of these relations, e.g., a concept like $on\_top\_of$ that uses gestalt principles could be introduced, which is not expressible by $contains$ or $touched$ or $intersects$.

The second row in Table 2 gives the signatures of the spatial relation types. In addition to the basic types in Table 1 we use the following union types in signatures:

$$
\begin{align*}
2d & := polygon \cup rectangle \cup closed\_curve \cup circle, \\
1d & := line \cup curve \cup 2d, \\
0d & := point \cup 1d, \\
any & := label \cup 0d.
\end{align*}
$$

While some relations are (conceptually) reflexive (e.g. $touched$), others may be irreflexive (e.g. $contains$; the outer object must be distinguished from the inner object). In the latter cases, the order of arguments in a relation is relevant. It has been mentioned that the exact spatial locations, $\ldots$ It is not even obvious what completeness should mean in this context.

$\ldots$ We do not consider them as real types in the sense of supertypes but as a means of abbreviation. Thus, e.g., $attached: label \times any$ is an abbreviation for an eight times overloaded predicate. The type names $nd$ have to be read as “n or more dimensional objects”.

4 It is not even obvious what completeness should mean in this context.

5 We do not consider them as real types in the sense of supertypes but as a means of abbreviation. Thus, e.g., $attached: label \times any$ is an abbreviation for an eight times overloaded predicate. The type names $nd$ have to be read as “n or more dimensional objects”.

4 It is not even obvious what completeness should mean in this context.

5 We do not consider them as real types in the sense of supertypes but as a means of abbreviation. Thus, e.g., $attached: label \times any$ is an abbreviation for an eight times overloaded predicate. The type names $nd$ have to be read as “n or more dimensional objects”.

4 It is not even obvious what completeness should mean in this context.

5 We do not consider them as real types in the sense of supertypes but as a means of abbreviation. Thus, e.g., $attached: label \times any$ is an abbreviation for an eight times overloaded predicate. The type names $nd$ have to be read as “n or more dimensional objects”.
sizes, etc., should not affect the representation of the picture for the goal is a schematic representation. Nevertheless, relative orientations can be of importance for the meaning of the picture: a flow chart, e.g., may have to be read starting at the top. Because it is not desirable to be forced to specify the orientations always we introduce meta-symbols (dashed arrows) to denote them (Table 3). If such an orientation marker is present, the relative orientation of the two objects connected by the arrow is taken into account, otherwise it is ignored. Meta-symbols are not really part of the picture like normal objects. They only control the interpretation and representation of the picture.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 3

One aspect of spatial relationships has not yet been considered: Often they do not only relate two objects to one another, but they also give an object/subobject relation. Two intersecting lines, e.g., split each other into two segments and all four segments share a common point (Figure 3). More precisely, a binary relationship between two objects, \( O_1 \) and \( O_2 \), can define three additional sets of objects: the set of subobjects belonging only to \( O_1 \) (\( \{s_1, s_2\} \)), the set of subobjects of \( O_2 \) (\( \{s_3, s_4\} \)), and the set of shared subobjects (\( \{p_0\} \)).

We are now prepared to define a picture formally. First we must define a language of object- and relation types:

**Def.:** A picture language is a triple \( PL = (O, R, \Sigma) \) where \( O \subseteq \mathcal{F}(V^*) \) is a finite set\(^6\) of object types over a fixed alphabet \( V \), \( R \subseteq \mathcal{F}(V^*) \) is a finite set of spatial relationship types, and \( \Sigma = \{ \Sigma_r \mid r \in R \} \) with \( \Sigma_r = r: o_1 \times o_2 \times \text{set}(o_3) \times \text{set}(o_4) \times \text{set}(o_5) \), for all \( r \in R \). Let \( S_r \) be the (possibly empty) list of its attribute values. A picture relation in \( PL \) is a six-tuple \( pr = (r, po_1, po_2, s_3, s_4, s_5) \) where \( r \in R \), \( po_1, po_2 \in ID \), and each \( s_k \) is a (possibly empty) set of object identifiers for subobjects. The sets of all admissible picture objects and picture relations in \( PL \) are denoted as \( PO_{PL} \) and \( PR_{PL} \), respectively.

Picture objects are identifiable typed entities with attributes. They can participate in spatial relations which can define object/subobject relationships:

**Def.:** A picture object in a picture language \( PL = (O, R, \Sigma) \) is a triple \( po = (id, o, l) \) where \( id \in ID \) is the object’s identity taken from a set of identifiers \( ID \), \( o \in O \) is its type, and \( l \) is the (possibly empty) list of its attribute values. A picture relation in \( PL \) is a six-tuple \( pr = (r, po_1, po_2, s_3, s_4, s_5) \) where \( r \in R \), \( po_1, po_2 \in ID \), and each \( s_k \) is a (possibly empty) set of object identifiers for subobjects. The sets of all admissible picture objects and picture relations in \( PL \) are denoted as \( PO_{PL} \) and \( PR_{PL} \), respectively.

Now, a picture is simply a set of attributed objects and their relations:

**Def.:** A picture in a picture language \( PL \) is a tuple \( p = (PO, PR) \) where \( PO \subseteq \mathcal{F}(PO_{PL}) \) and \( PR \subseteq \mathcal{F}(PR_{PL}) \). Let \( r: o_1 \times o_2 \times \text{set}(o_3) \times \text{set}(o_4) \times \text{set}(o_5) \) be a type in \( \Sigma \). Then for all \( (r, po_1, po_2, s_3, s_4, s_5) \in PR \) the following conditions must hold: (a) \((po_1, o_1, l_1) \in PO\), (b) \((po_2, o_2, l_2) \in PO\), and (c) \( p \in s_k : (p, o_k, l) \in PO\). The set of all possible pictures in \( PL \) is denoted as \( PP_{PL} \). Conditions (a) and (b) say that only objects of an

\(^6\) \( \mathcal{F}(M) \) denotes the set of all finite subsets of \( M \).
appropriate type may be used as the constituting objects of relationships, (c) states
the same for subobjects.

Pictures in the sense of the above definition can easily be visualized as a kind of directed graphs.
Two different types of nodes will be used: an object node for every \( po \in PO \) and a relation node
for every \( pr \in PR \). A special layout is used in which each relationship node is split once. Relation
nodes have two incoming edges and up to three outgoing edges. An edge is drawn from an object
node \( po = (id, o, l) \) to a relation node \( pr = (r, po_1, po_2, s_1, s_2, s_3) \) if \( id = po_1 \lor id = po_2 \). If \( id = po_1 \) the edge points to the left half of the relation node, if \( id = po_2 \) it points to the right half.

An edge is drawn from a relation node \( pr \) to an object node \( po \) if \( id \) is member of \( s_1 \) or \( s_2 \) or \( s_3 \).
Instead of labeling the edges to maintain the subobject relationship between \( s_1 \) and \( po_1 \) (\( s_2 \) and \( po_2 \), respectively) the convention is used that edges coming out of one half of a relationship node point
to the set of subobjects of that object that is attached
to the incoming edge of the same half. The edge
coming out of the splitting line points to the set of
shared subobjects. If a relationship does not define
any subobjects (like attached) no outgoing edges are
drawn. No edges exist between any two subobjects
derived from the same relationship node and no additional edges are placed between some object
and its own subobjects. Figure 4 gives an example: the graph representation of the picture\(^7\) in
Figure 3.

3. Picture Terms

So far we have means to describe pictures whose contents is entirely known. In order to be able to
describe the contents of a picture only partially, we introduce picture variables, which can be used
inside a picture like picture objects. A picture containing picture variables will be called a visual
picture term. Its foroal representation will simply be called a picture term and the corresponding
graph will be named a picture term graph\(^8\). There are four different types of picture variables:
object variables, group variables, backgrounds, and frames. The most simple type of them are
object variables. An object variable is given by its name and its type, which can be any of the
object types defined by the picture language. An object variable can be bound to a single picture
object. Inside of picture terms and their graphs the variable will be visualized like a normal object,
but instead of the identifier, its name will be used as a label\(^9\).

Group variables can not only be bound to single objects but to entire parts of pictures. The only
restriction on the pictures that a group variable can be bound to is that the corresponding picture
graph must be connected, i.e., there may not be two objects in such a picture that are neither
connected directly by a relation, nor indirectly through some objects and relations. A group
variable is given only by its name. It is not typed. Inside of picture terms group variables are
depicted as filled areas labelled with their name. Group variables are a language element that can
be used to express that the picture contains some (connected) substructure, which is connected to
other parts of the picture by a given relation and which may consist of more than a single object.
Consider the picture in Figure 5: It is an instance of the picture term in Figure 6 because both
pictures are identical if we let \( X = \bigcup b^c \) (c and b are connected by the contains relation).

\(^7\) Object identifiers (like O1), of course, do not appear in the picture normally.
\(^8\) All three forms will be abbreviated picture term if the intended meaning is apparent from the context.
\(^9\) Labels in picture terms must not be confused with labels in real pictures (e.g. “A Line” in Figure 2).
If even the number of groups contained in a picture is unknown a more powerful language element is necessary to build a picture term that can be matched with this picture. This element is called a background. A background can contain entire pictures that may consist of one or more groups. Only a single background may be present in a picture term. During the matching of a picture term with a picture, all objects in the picture that do not occur in the term and can neither be bound to some variable in the term will become part of the background variable’s binding. In a visual picture term a background is shown as a box framing the picture term and is labelled with its variable name.

The easiest way to imagine what can be done with picture variables is to regard a picture as a stack of slides. There is one slide for each object or object variable, one for each group variable and at most one for the background. Thus, an object uses a separate layer if it is given as a constant or bound to an object variable. A connected cluster of objects can be depicted on a slide for a group variable, and all the objects that are not consumed by one of these slides are pushed into the background. As long as the stack of slides remains untouched, the picture retains its entire structure. But what happens, if we remove some of the slides from the stack and put them back afterwards? There are several different possibilities to put a slide on the stack. Consider the bindings given in Figures 5 and 6. Let \( Y = X \). What is the picture belonging to the picture term in Figure 7? Is it the same as Figure 5, as one might expect, or is it Figure 8, or...?

The problem is that we do not have any information on the exact positions of the slides. In other words, the original picture contains spatial relationships between objects, say \( a \) and \( b \), that are assigned to two different slides, \( s_1 \) and \( s_2 \). Consequently, the relation between \( a \) and \( b \) can only be given as a relation between \( s_1 \) and \( s_2 \), i.e., as the relative position and orientation of these slides because there is no layer that contains both objects. Now, if \( s_1 \) and \( s_2 \) are separated from each other, this information is no longer retained and there is no way to restore the original picture. More formally, since a picture variable may only contain relations on objects that it contains itself, and since there is no variable that contains both objects \( a \) and \( b \) no variable may contain any relation between \( a \) and \( b \).

This is the point where Frames come in. Frames can never be used as pictures themselves, they can only be used in combination with a picture term and contain all the information on spatial relationships that cannot be assigned to any of the variables, i.e., that cannot be depicted on a single slide. In picture terms, frames are visualized as labelled dashed boxes around the picture term. They can be imagined as slides that contain only markers for the position in which the other

---

10 Otherwise the corresponding picture graph would contain a relation node with edges “pointing to nowhere”.
slides have to be fixed. Only a single frame may be present in a picture term. Technically, a frame contains only relation nodes and the subobjects defined by them, i.e., it cannot be depicted as a graph. Given the bindings from above for \(X\) and \(Y\) and a frame that contains the relations between \(a\) and the objects in \(X\), the equation in Figure 9 holds\(^{11}\).

![Figure 9](image)

We have already defined how to formalize pictures built from picture objects. Now a formalization of picture terms will be given. First, variables must be defined:

**Def.:** A **picture variable** is a triple \(pv = (n, o, B)\), where \(n \in V^*\) is the name of the variable and \(o \in O \cup \{\varepsilon\}\) is its type in the case of object variables and \(\varepsilon\) otherwise. \(B\) is the binding for \(pv\). For unbound variables \(B = \varepsilon\). For bound **object variables** \(B \in POPL\). For bound **backgrounds** \(B \in PPL\). For bound **group variables** \(B \in PPL\) with the restriction that only completely connected pictures may be used for \(B\). **Frames** are an exception: they contain only relations and subobjects. The bindings admitted for Frames are \(B \in \mathbf{F}(POPPL) \times \mathbf{F}(PRPL)\). The sets of all admissible variables are called \(V_{obj}, V_{group}, V_{back},\) and \(V_{frame}\) for object variables, group variables, backgrounds, and frames, respectively.

Now we redefine picture relations so that variables can be used instead of objects. The type of picture relations defined previously will be called **ground relations**:

**Def.:** A **picture relation** in \(PL\) is a six-tuple \(pr = (r, po_1, po_2, s_1, s_2, s_3)\) where \(r \in R\) is the relation type, \(po_1, po_2 \in ID \cup V^*\) is either an object identifier or a variable name, and each \(s_j\) is a (possibly empty) set of object identifiers and variable names \(p \in ID \cup V^*\). The sets of all possible picture relations in \(PL\) (including ground relations) is denoted as \(RPL\).

A picture term consists of a set of objects, a set of relations, and a set of variable names that are used in the set of object variables, in the set of group variables, and for the background and the frame. Background and frame may be omitted:

**Def.:** A **picture term** in a picture language \(PL\) is a seven-tuple \((PO, N, PR, V_o, V_g, B, F)\), where \(PO \in \mathbf{F}(POPPL)\), \(N \in \mathbf{F}(V^*)\), \(PR \in \mathbf{F}(PRPL)\), \(V_o \in \mathbf{F}(V_{obj})\), \(V_g \in \mathbf{F}(V_{group})\), \(B \in V_{back} \cup \{\varepsilon\}\), and \(F \in V_{frame} \cup \{\varepsilon\}\). Again, objects and variables in relations must have the appropriate types. The set of all possible picture terms in \(PL\) is denoted as \(PTPL\).

### 4. How to Match Pictures

Earlier we argued that the problem of picture description can be reduced to the problem of matching a picture against an example picture; picture terms are (possibly partially specified) example pictures. To use them for descriptions it must be defined how to match a term with a picture, i.e., how to compute bindings for the variables contained in the term such that the picture term exactly describes a given picture if such bindings can be found. Only matching a picture

\(^{11}\) Equality means that both sides of the equation describe exactly the same picture (see Section 4).
term with a given picture will be considered for this can be done by term matching. Finding variable substitutions that make two different picture terms identical is a problem of unification [Kni89] and not yet solved in this context. The restrictions that result from being constrained to matching will be explained in the next section.

Intuitively, the picture described by a picture term in which all variables are bound is obvious: It must contain all the picture objects given by $PO, V_o, F, V_g$, and the object parts of $B$ and $V_g$, and it must satisfy all the spatial relations given by $PR, F$, and the relation parts of $B$ and $V_g$. Only those elements of $F$ are used that define a relation for objects contained in $PO, V_o, V_g$, or $B$. Of course, picture terms can be constructed that can hardly be visualized because of denoting spatial relations which would be contradictory in the real world. But during matching against visualizable pictures such terms will never arise, because no spatial relations exceeding those really depicted in the original picture will be added.

Let $pt=(PO, N, PR, V_o, V_g, B, F)$ be a picture term containing only bound variables. Then $eval(pt) = p = (O, R) \in PP_L$ is the picture belonging to $pt$.

$O' = PO \cup obj_o(V_o) \cup obj_g(V_g \cup \{B\})$

is a preliminary set of all objects given by the bindings from $pt$ in which only the subobjects given by $F$ are missing. $obj_o$ and $obj_g$ are functions that extract the objects from a given set of object- or background variables, respectively.

$\begin{align*}
obj_o(M) &= \{ \text{po} \mid (n, o, po) \in M \} \\
obj_g(M) &= \bigcup PO \text{ forall } (n, \epsilon, (PO, PR)) \in M.
\end{align*}$

The computation of the set of relations $R$ in $p$ is more complicated. $R$ consists of four parts: (1) the relations given by $PR$ in which all variable names occuring in some relation are substituted by the objects they are bound to, (2) the relations given by the group variables, (3) the relations contained in the background, and (4) those relations in the frame whose constituting objects are contained in the picture term. Relations from the frame always take precedence over conflicting relations derived from (1 — 3), i.e., the conflicting relations must be removed. Thus,

$R = \text{remove}(R_1, \text{rel}([F])) \cup \text{restrict}(\text{rel}([F]), O')$, where

$R_1 = \text{subst}(PR, V_o, V_g) \cup \text{rel}(V_g \cup \{B\})$.

is the set of relations. $\text{remove}$ eliminates conflicting relations from $R_1$:

$\begin{align*}
\text{remove}(R_1, R_f) &= \{ \text{pr}= (r, po_1, po_2, s_1, s_2, s_3) \mid \text{pr} \in R_1 \land (r, po_1, po_2, s'_1, s'_2, s'_3) \in R_f \\
&\quad \land (r, po_2, po_1, s'_1, s'_2, s'_3) \not\in R_f \}. \\
\text{restrict} \text{ eliminates those relations from the frame whose constituting objects are not in } O':
\end{align*}$

$\begin{align*}
\text{restrict}(R_f, O') &= \{ (r, po_1, po_2, s_1, s_2, s_3) \in R_f \mid (po_1, o_1, l_1) \in O' \land (po_2, o_2, l_2) \in O' \}.
\end{align*}$

$\text{rel}$ selects the relation part of group-, background-, and frame variables:

$\text{rel}(M) = \bigcup PR \text{ forall } (n, \epsilon, (PO, PR)) \in M.$

$\text{subst}(PR, V_o, V_g)$ replaces every variable name $n$ occuring in some relation of $PR$ by the identifier to which the variable is bound, i.e., with an identifier $i$ for which the following condition holds:

$\begin{align*}
(n, o, (i, o, l)) \in V_o \lor (n, \epsilon, (PO, PR)) \in V_g \land (i, o, l) \in PO .
\end{align*}$

Finally, the set of all objects $O$ is the union of $O'$ and the set of relevant subobjects in $F$:

$O = O' \cup \text{used}(F, R)$

where used selects those subobjects from $F$ that are used by some relation in $R$:  

9
\[
\text{used}(F, R) = \{ (i, o, l) \in \text{obj}_g(\{F\}) \mid \exists (r, po_1, po_2, s_1, s_2, s_3) \in R \land i \in s_1 \cup s_2 \cup s_3 \} .
\]

Given \(\text{eval}(pt)\) we finally are prepared to define matching. Now, matching a non-ground picture term \(pt=(PO, N, PR, \varepsilon, \varepsilon, \varepsilon, \varepsilon)\) with a picture \(p\) simply means to find admissible bindings \(V_o, V_g, B\) and \(F\) of the variables \(N\) in \(pt\) to objects and relations from \(p\) such that \(p = \text{eval}((PO, N, PR, V_o, V_g, B, F))\). No object or relation may be contained in more than a single variable. Matching is not deterministic because, e.g., some objects contained in a group variable can sometimes as well be pushed into the background. We will reconsider this problem in a later section.

Note that matching is defined purely declarative. We omit the rather lengthy procedural definition of matching here. All this looks a little complicated on the formal level, but defines a very intuitive semantics, if we look at visual picture terms.

5. Picture Programming

Having defined what it means to match picture terms, we can use them as an extension to a logic programming language (we will use Prolog [CM87] here). But we must ensure that only matching is needed, i.e., that no attempt of unification is made during the evaluation of a logic program that uses picture variables. Of course, this requirement forces us to take the procedural aspects of the language into account. The consequences are quite similar to those implied by using numerical evaluations or other built-in evaluable predicates in Prolog programs. We have to distinguish between input and output arguments of a predicate, and we must ensure that every input argument is completely bound in a call to this predicate. This is quite a usual technique in logic programming, several implementations of Prolog have incorporated such mode declarations [MW88]. The rules for admissible usage of picture terms are: (1) every picture variable occurring in a picture term that is used in some input position of a rule body must occur either in an output term to the left of this position in the rule body or in an input term in the rule head, and (2) every picture variable occurring in an output term in the rule head must either be used in an output term in the rule body or in an input term in the rule head. If we obey these conditions, the rules can be evaluated by Prolog’s standard SLD-resolution [Llo87]. Of course, “normal” terms and picture terms may both be combined in the same rule. Due to its declarative semantics, such definite clause picture programs can not only be used for automated reasoning about pictures, but as a specification method for visual languages, as well.

For the sake of convenience, we define some syntactical simplifications: a picture term \(pt\) consisting only of a single variable \(X\) (\(pt=(\emptyset, \{X\}, \ldots )\) ) may simply be denoted by this variable instead of using a picture, e.g., \(X\) instead of \(\square X\). Anonymous variables may be used without giving them a name. We assume that matching two terms is done by calling a non-deterministic (i.e. backtrackable) built-in predicate \(\text{match}(+P, ?Pt)\)\(^{12}\), whose arguments are an entirely bound picture term in the first position and another picture term in the second position. \(\text{match}(P, Pt)\) evaluates the picture term \(P\) and computes bindings \(B\) for the variables in \(Pt\) such that \(\text{eval}(P)=\text{eval}(Pt \circ B)\), where \(Pt \circ B\) is \(Pt\) augmented by the set of bindings \(B\). The usage of \(\text{match}/2\) may be abbreviated: a variable used as an input term in the rule head whose only occurrence in the body is in a positive call to \(\text{match}/2\) as the first argument may be omitted from the rule. Instead the picture term used as the second argument of this call to \(\text{match}\) is directly used in the head: Thus,

\(^{12}\) This is the usual notation of a mode declaration: +P is an input argument, i.e., it must be bound, and ?Pt is an output argument, which may be bound, partially bound, or unbound.
\[ p(P) \quad : \quad \ldots, \quad q(...), \quad \text{match}(P, \quad \begin{array}{c} a \\ X \end{array} \quad ), \quad r(...), \quad \ldots \]

is the same as

\[ p(\begin{array}{c} a \\ X \end{array} \quad ) \quad : \quad \ldots, \quad q(...), \quad r(...), \quad \ldots \]

We will now give a tiny example program, which exploits these features. The program is used to compute reachable nodes in a graph. The graph is given as a picture in which nodes are visualized by labelled circles and edges by lines. This picture is passed as the first argument. The second argument is the label of the node that is to be used as the start node. A list of all labels on paths starting at this node is returned as the third argument.

\[ \text{path}(\begin{array}{c} L1 \\ X \end{array} \quad , \quad \begin{array}{c} S \\ [N \mid \text{Ns}] \end{array} \quad ) \quad : \quad \text{attr}(L2, \quad \begin{array}{c} S \mid \_ \end{array} \quad ), \quad \text{attr}(L1, \quad \begin{array}{c} N \mid \_ \end{array} \quad ), \quad \text{path}\quad \begin{array}{c} L1 \\ N \end{array} \quad , \quad \text{Ns} \quad ) \quad . \]

\[ \text{path}(P, \quad \begin{array}{c} S \quad [\_] \end{array} \quad ) \quad : \quad \text{not} ( \quad \text{match}(P, \quad \begin{array}{c} L1 \quad \_ \end{array} \quad ), \quad \text{attr}(L1, \quad \begin{array}{c} S \mid \_ \end{array} \quad ) \quad ) \quad . \]

The interpretation of the first clause is: If the picture contains two nodes connected by a straight line, and the label of one of the nodes is that given for the start point, then the path consists of the label of the second point concatenated with the path starting at the second point. The path is empty, if the starting point cannot be found in the picture. The definition of the predicate uses the connection predicate \text{attr}/2 to access the values of attributes of picture objects. We assume that the string of a label is the first attribute in its attribute list. Note that, due to the removal of the start node from the picture, cycles in the graph will not lead to infinite recursion. As an example, a call to

\[ p(\begin{array}{c} a \\ b \end{array} \quad , \quad a, \quad \text{Ls} \quad ) \quad 

will return the list \text{Ls}=[b, c] at first and \text{Ls}=[b, d] upon backtracking. A trace of an execution will be shown in the next section.

Given the execution model of a picture logic program, it is easy to imagine how the relations \textit{left_of}, \textit{above}, etc., are managed. Since the admitted usage of picture terms is restrictive enough these relations can simply be implemented as evaluable predicates, which access the attribute list of an object by connection predicates. The main goal of the program must always be called with a completely specified picture. Imagine this picture as being embedded in a two-dimensional cartesian space. The positions and extensions of the objects can then be derived from the picture, and they can be stored in the objects’ attribute lists. A call to one of the predicates \textit{left_of}, \textit{above}, etc., simply computes the relative spatial positions of the objects given as arguments by comparing their spatial attributes.

6. Picture Clause Grammars
Based upon picture logic programming we can define picture clause grammars (PCGs). These are related to picture programs in the same way as Definite Clause Grammars (DCGs [CM87]) are related to normal Prolog programs. PCGs are more restrictive than arbitrary picture programs, i.e., they can only recognize a subclass of the pictures that can be recognized by picture programs. We will give a preliminary definition of PCGs first which recognizes a subclass of pictures that we will term regular pictures. It is of particular interest for it can — in contrast to complex picture programs — efficiently be executed, because the reduction of pictures by regular productions requires only the selection and removal of a single object from a picture. This class of picture languages is termed regular because the formal structure of picture grammars resembles that of grammars for regular languages. Unfortunately, many types of diagrams are not captured by regular PCGs. We will therefore extend the notion of PCGs to context-free picture grammars without sacrificing the advantage of efficient executability. Context-free pictures are not only theoretically interesting, but also practically interesting because they comprise a great many of important diagram types. The structure of PCGs is simple but very powerful because PCGs may arbitrarily be attributed.

Let us quickly review regular grammars. Every regular language can be defined by a grammar consisting only of productions

\[(i) \quad P \rightarrow a \, Q \quad (ii) \quad P \rightarrow a \quad (iii) \quad P \rightarrow \varepsilon\]

where \(P\) and \(Q\) are non-terminal symbols, \(a\) is a terminal symbol and \(\varepsilon\) is the empty word [HU79]. From the viewpoint of parsing, every production of a regular grammar specifies two things: which terminal symbol \(a\) (or none) is to be removed from the beginning of the input and whether parsing continues with some production \(Q\) or stops (types \((ii)\) and \((iii)\)). The DCG-counterparts of these production types and their clausal translations are:

\[(i) \quad p \rightarrow [a], \, q. \quad \quad p([a]|X], \, Y) :- q(X,Y).\]

\[(ii) \quad p \rightarrow [a]. \quad \quad p([a]|X], \, X).\]

\[(iii) \quad p \rightarrow \[]. \quad \quad p(X, \, X).\]

The procedural meaning of the clauses corresponding to regular DCG productions is quite simple. They remove the given element from the head of the tokenlist if it matches (types \((i)\) and \((ii)\)) and fail otherwise. Type \((i)\) continues parsing by calling the next subgoal for the remaining input list, types \((ii)\) and \((iii)\) stop the derivation.

We can use the same simple schema to parse pictures: in every step a single picture object is removed from the input picture and parsing is continued with the remaining picture. But because there is no particular order in which the objects in a picture have to be used, unlike textual languages that are inherently sequential, every production must define where the intended object can be found. This is done by defining its spatial relation to the rest of the picture, in particular to a second object of the picture. A picture production consists of four parts: (1) the name of the non-terminal symbol it defines, (2) an example picture specifying two objects and their relation, (3) the specification which of the two objects is to be removed, and (4) the name of the non-terminal which is used to continue (or none). The production below\(^{14}\), e.g., defines a non-terminal

---

\(^{13}\) Every production of type \((ii)\) can be replaced by a production of type \((i)\) and one of type \((iii)\). In fact, type \((ii)\) is unnecessary. It is used here only for reasons of clarity.

\(^{14}\) Note that the box framing the two example objects in the production is not a background. It is a visual clue only.
$p$ which removes a single node $N$ (circle) that is connected to some edge (line) from the picture and continues parsing with the non-terminal $q$:

$$p \rightarrow \begin{array}{c}
\text{\textbullet} \\
N : q \end{array}.$$  

Due to our definition of backgrounds and frames it is very easy to derive picture clauses from picture productions that exactly do what is required. For every production a single clause is generated which defines a binary predicate with the non-terminal’s name as its functor. Like in DCGs, the first argument of the predicate is the input picture and the second argument is the picture that remains after successful reduction by this production. The two-object example picture given by the production must be given a background and must be encapsulated in a frame. The picture generated in this way is matched against the first argument of the predicate. The same picture less the element to be removed is passed to the next non-terminal as the first argument. The second argument of this non-terminal is the same fresh variable as the second argument of the head, i.e., whatever the called production returns, is returned by the calling production. The above example results in the rule:

$$p (N, X) :\Rightarrow q (\text{\textbullet} ; F, X).$$

Of course, anonymous object variables (the edge) must now be named. When the example picture is matched against the calling picture, the two explicitly depicted object variables will be bound to the corresponding objects, all other objects and their relations are pushed into the background, and the frame contains the relations between objects in the background and the two explicitly depicted objects. Thus, the background plus one of the explicit objects plus the frame is exactly the original picture with the requested object removed. Nothing else is changed. A bit more formally: Every picture production

$$p \rightarrow \begin{array}{c}
X \\
N : q \end{array}.$$  

where $p$ and $q$ are non-terminal symbols and $N$ is a variable name used inside of the example picture $X$, will be translated into a clause

$$p (\text{\textbullet} ; F, R) :\Rightarrow q (\text{\textbullet} ; F, R).$$

where $Y$ is the picture derived from $X$ by removing $N$. $\varepsilon$-Productions are translated to

$$p (X, X).$$

Why did we use a second argument for returning the remainder of the picture? If we want to check if a given picture which we pass as the first argument belongs to the language defined by the PCG we always have to test, whether an empty picture is returned. It would have been simpler to use single argument predicates. But using a return argument provides a neat interface to general picture programming. Having translated the PCG to a picture logic program, the translated productions can be used like normal goals inside of a general picture logic program. Every production tries to isolate a phrase of the category defined by it from the picture and if it succeeds it returns that part of the picture it did not consume. The rest of the picture can then be processed by other parts of the program.

Furthermore, this scheme can be utilized to extend PCGs to capture what we will term context-free pictures. In a context-free textual grammar the above mentioned types of productions (i - iii) occur plus a new type.
\[(iv) \quad P \rightarrow Q \ R\]
in which \(P, Q,\) and \(R\) are all non-terminal symbols. A DCG production of type \(iv\) and its translation is:

\[(iv) \quad p \rightarrow q, r. \quad p(X, Z) \ni q(X, Y), r(Y, Z).\]

Note that the translated clause uses the return attribute \(Y\) (the remainder of the input) for chaining the subgoals. This type of production can be introduced into PCGs, as well. It specifies the names of two picture types (non-terminals) that must be contained in the input picture and a way in which these pictures are connected. This type has the form

\[p \rightarrow \begin{array}{c} X \end{array} : q \ & r. \quad \text{or} \quad p \rightarrow q \ & r.\]

and is translated to a picture clause

\[p(\begin{array}{c} X \end{array}, Z) \ni q(\begin{array}{c} X \end{array}, Y), r(Y, Z).\]

or, in the second case, simply

\[p(X, Z) \ni q(X, Y), r(Y, Z).\]

Like a regular production, a context-sensitive production may remove an object from the picture:

\[p \rightarrow \begin{array}{c} X \end{array} \backslash N : q \ & r.\]

The procedural interpretation is: The production is applicable if the input picture contains a subpicture matching \(X\) if this is given. It is applied by parsing the input picture with production \(q\) (which removes a substructure of type \(q\) from the input) and using whatever picture is returned by \(q\) as the input to production \(r\), possibly after having removed \(N\). This is roughly the same that is done in context-free DCGs.

As a further extension to the formalism, PCG productions may be attributed and augmented by arbitrary subgoals, including the usage of \texttt{match/2}, \texttt{attr/2}, and cuts. Of course, when these features are used the programmer is again saddled with the responsibility for really executable parsers.

In the previous section a picture logic program was shown, which computed the paths between nodes in a graph given by a picture. This program can readily be reformulated as a regular PCG. To keep the example small it is assumed this time that the start node is given by a picture object variable instead of its label. The path is still returned as a list of labels. The start symbol of the PCG is \textit{path}. All productions are augmented by normal attributes and the definition of \textit{nodelabel} makes use of \texttt{attr/2} by calling it as a standard subgoal. The curly brackets are escape symbols that enclose calls to arbitrary subgoals.
path( P, Ns ) → P \ E | P : edge( E, Ns ) .

path( P, [] ) → ε .

draw( E, Ns ) → E \ nodelabel( P, Ns ) .

nodelabel( P, [N | Ns] ) → P Label \ L : path( P, Ns ) ,
{ attr( L, [ N | _ ] ) } .

Let us trace the execution of the same example that was used in the previous section. The entire picture denoting the four-node graph from the example is given by the following picture term graph (circles are marked by c1, ..., c4, lines by l1, ..., l3, labels by lb1, ..., lb4, and points by p1, ..., p6. The spatial relationships are abbreviated: t=touches, a=attached):

Figure 10

Every application of a production removes some part from the graph. In Figure 10 these parts are marked and labelled by bold-face numbers which give the order of their removal. The dual parts marked 4a - 5a - 6a and 4b - 5b - 6b, respectively, mark different paths of the non-deterministic derivation that are followed upon backtracking. Obviously, a PCG can also be viewed as a graph grammar, which describes how to transform a picture term graph. Graph transformations described by PCGs are quite simple, because removal of objects is the only operation applied to the graph. But picture logic programs can in general be viewed as picture term graph transformations, which discloses the way to arbitrary complex manipulation of these structures.

Many picture programs and PCGs use non-deterministic matching and non-deterministic productions in a “don’t-care” style, i.e., it is sufficient if there is some match and it is irrelevant which of the valid matches will be selected as the first one. If, e.g., the above PCG example which computes paths between nodes is used to test whether a given path is contained in the picture, i.e., if path is called with all arguments bound, then we need not care about the non-deterministic nature of matching. A cut (!) can be used in the first production without changing the meaning of the program. Execution of cutted productions can be far more efficient, because no choice point
for this production must be pushed on to the stack used for backtracking. Nevertheless, cuts must be used very carefully because they can change the recognized/generated language of the PCG.

7. Example: Parsing and Translating ER Diagrams

As a concluding, more complex example we present a PCG that parses a simplified version of entity relationship diagrams and translates them to an abstract textual description, thereby defining their semantics. To get an impression of the textual representation compare the example diagram in Figure 11 to its representation.

```
diag([ works_at(person, office), is_a([man, woman], person)]).
```

![Diagram](image)

The simplified ER version uses circles for entities, diamonds for relationships and triangles for generalizations. Syntactically, a diagram consists of a relation or a generalization inside of a diagram:

```
diag([R|D])    →    rel(R) & diag(D), !.
diag([R|D])    →    gen(R) & diag(D), !.
diag([])       →    ε .
```

The cut is used to avoid the application of the ε-production until there are no more generalizations or relations to be parsed. A relation consists of a diamond connected to two objects by lines:
rel(R) \rightarrow \begin{array}{c}
\text{Label} \\
L : \text{rel}_\text{obj}(D, \text{Left}) \& \text{rel}_\text{obj}(D, \text{Right}), \\
\{ \text{rotated}(D, 90), \text{attr}(L, [\text{Rel} | _]), \text{R} = .. [\text{Rel, Left, Right}] \}.
\end{array}

\text{rel}_\text{obj}(D, O) \rightarrow \begin{array}{c}
\text{Label} \\
E : \text{obj}(C, O).
\end{array}

\text{obj}(C, O) \rightarrow \begin{array}{c}
\text{Label} \\
\{ \text{attr}(L, [O | _]) \}.
\end{array}

The production \text{rel}_\text{obj} uses a generalized syntax: the matching picture contains not only two but three object variables. This extension is straightforward and can be converted to the normal form of productions described in the previous section by repetitively removing a terminal (object variable) from the picture and adding a new non-terminal instead. A generalization can be defined in the same way as the relation but it can be connected to several base objects. We assume that a triangle is defined as a basic picture object type. Otherwise productions must be defined that synthesize triangles from lines.

gen(G) \rightarrow \begin{array}{c}
\text{Label} \\
L : \text{derived}(T, \text{Derived}) \& \text{base}_\text{set}(T, \text{Base}), \\
\{ \text{attr}(L, [\text{Gen} | _]), \text{G} = .. [\text{Gen, Derived, Base}] \}.
\end{array}

derived(T, O) \rightarrow \begin{array}{c}
\text{Label} \\
E : \text{obj}(C, O).
\end{array}

\text{base}_\text{set}(T, [O|Os]) \rightarrow \begin{array}{c}
\text{Label} \\
E : \text{obj}(C, O), !, \& \text{base}_\text{set}(T, Os).
\end{array}

\text{base}_\text{set}(T, []) \rightarrow \varepsilon.

Being describable by a context-free PCG, this type of diagram could be called context-free.

8. Conclusions and Visions

PCGs are a method to specify pictures visually, but they break down the picture into several tiny subunits. A conceptual more adequate way is to use complex picture terms. Unfortunately, picture terms do not always lead to efficiently executable programs. Using them means to trade efficiency for comprehensibility. Consider, e.g., the problem of loop detection in flow charts (Figure 12) which can be solved easily by matching with complex picture terms (Figure 13). It would cost much more effort to write an attributed PCG for this problem. However, the loss of efficiency is irrelevant if the formalism is used for specification purposes.
An example of a class of pictures that can not at all be described by context-free PCGs (without tricky usage of attributes) but by picture clause programs are flow charts that contain equal numbers of nodes on parallel paths.

We have presented a formalism for the specification of visual languages that is visual itself and leads to clear and easily comprehensible specifications of visual languages. The level of description covers the entire range between highly visual but only inefficiently executable picture clauses and less expressive but efficiently executable grammars. The grammars derived from this formalism are not only capable of describing simple diagrammatic languages, like, ER diagrams, flow-charts, or graphs. Moreover, they are as well suitable for the description of non- iconic visual free-form languages and can be adopted to new languages by extending the basic object and relationship types the formalism uses. Thus, the formalism is one step towards natural human computer interaction by graphics. Of course, it does not solve the semantical problems of graphical languages, but it offers a formal basis for the investigation of these languages and their problems.

Though in the first place, the formalism was developed to describe and specify visual languages formally, it turned out that it can likewise be used as a suitable basis for the automatic generation of parsers for visual languages. Being logic programs, such parsers can not only be used to recognize picture languages, but also to generate example pictures from their specifications. Naturally, there are still some problems to be solved before we can expect such parsers to be useful in the realm of real applications. The main problem is the inherent non-determinism of picture matching (and the inefficiency caused by it). If a deterministic redefinition can be found, or if at least an order in which the different possible bindings are established can be defined for it, it would be much easier to improve the efficiency of the deduction by controlled pruning of the derivation tree, i.e., by using cuts.

Another important improvement to the formalism is to use unification of picture terms instead of matching. If picture terms were unifiable, picture variables could be equipped with the full power of logic variables. Consequently, picture terms could then be used in logic programs without putting any constraint on the order of evaluation or the positions they may be used in. Furthermore, unification would provide an interesting and elegant way to talk about impossible pictures: infinitely recursive pictures like $X$ in Figure 14, etc.

A second direction for further research is to investigate parsing of interactive visual languages [WW90] with logic grammars, i.e., to develop incremental deductive parsing techniques that employ some kind of “intelligent” backtracking algorithm so that not the entire input picture must be reinterpreted when changes are made to it, but only relevant portions of the picture need to be reconsidered.
Of course, the last topics are only visions for a long-term future. Though the formalism presented here is just a first step on this way, we are confident that the combination of logic grammars and picture variables might open the door to a general framework for specification and implementation of visual languages.

Acknowledgements

Thanks to Ralf Hartmut Güting for encouragement, to Gerd D. Westerman and to Martin Erwig for detailed and helpful comments on a draft version of this paper, and to Jörg Poswig for interesting discussions.

References


