On the use of Graph Grammars for defining the Syntax of Graphical Languages*

J. Rekers
Department of Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
email: rekers@wi.leidenuniv.nl

ABSTRACT

In order to implement graphical editors which allow both for structured and free editing, a parsing algorithm is needed which can analyze a diagram according to a graphical syntax, and derive the structure depicted.

We propose to split this analysis in two phases. The first phase reads the picture objects as they were drawn, determines the spatial relations between them, and stores the whole in a graph. The second phase of the analysis searches this graph for patterns which form constructs of the language under consideration, and generates an abstract syntax graph of the depicted structure.

We investigate whether graph grammars are a suitable formalism to define both graphs and to specify the translations between the two. The description of the translation would define the graphical syntax of L; graph rewriting according to this description would implement the corresponding graphical parser which performs the second phase of the analysis.

1 INTRODUCTION AND MOTIVATION

Workstations with high-resolution displays are more and more common. These displays provide superb capabilities for the display of graphics, and graphical languages which make use of these capabilities are emerging. Programming environments which offer support for graphical languages often use a syntax-directed graphical editor as input medium. The data structure as it lives internally in such an editor is specific for the graphical language L under consideration and is often is an abstract syntax graph. This graph is updated directly by the edit operations issued by the user. The architecture of such an editor can be depicted as in Fig. 1. It has as major advantage that no parsing is necessary and that syntactically incorrect graphs are impossible. Examples of graphical programming environments which work according to this model are [13] and [4].

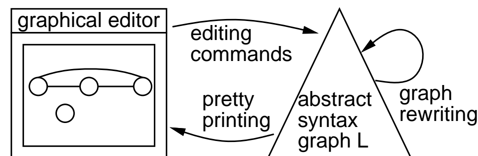


Figure 1: A syntax-directed editor for the graphical language L

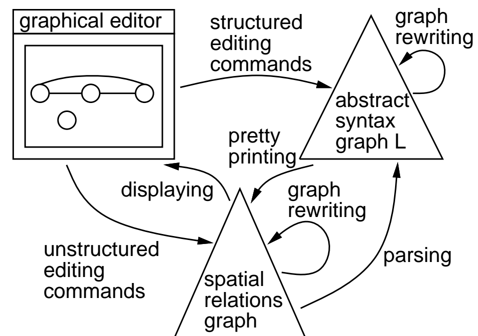


Figure 2: A hybrid editor for the graphical language L

However, just as pure syntax-directed editors have been a failure for textual languages, they will be for graphical languages. Users of an editor need absolute freedom in the order in which they develop their diagrams; structured support on demand is welcome, but shouldn't be a straight-jacket as it is in a pure syntax-directed editor. We strive for a hybrid editor which fills both the need for free editing as for structured help.

Our approach to realize such a hybrid editor is to introduce an intermediate level which is not specific for the graphical language L. Graphs are used again for the representation at this level, but the nodes are pictorial objects like *boxes*, *circles*, *arrows* and *strings*. The edges in this graph are the spatial relations between these objects, like *contains*, *connects*, *touches* and *labels*. The

*Presented at the *Colloquium on Graph Transformation and its application in Computer Science*, Palma de Mallorca, Spain, March 1994. Also available as technical report 94-11, Leiden University, ftp site ftp.wi.leidenuniv.nl, file pub/cs-techreports/tr94-11.ps.gz

architecture of the graphical editor becomes as depicted in Fig. 2. Here the user can use structured commands in terms of L, but can also perform unrestricted modifications to the picture with so-called unstructured editing commands. This means however that graphical syntax analysis is needed at a certain moment in order to discover the structure depicted.

We propose to split this analysis in two phases, corresponding to the representation levels of the hybrid graphical editor of Fig. 2. The unstructured edit commands create or modify graphical objects. The coordinates and shapes of these objects are in the first phase of the analysis used to determine the spatial relations between these objects and the other objects in the diagram. We call this process *graphical scanning* and it generates the spatial relations graph of Fig. 2. In the second phase of the analysis patterns in the spatial relations graph are used to derive constructs in terms of the graphical language L. We call this process *graphical parsing* and it generates the abstract syntax graph of Fig. 2. If the user applies structured editing commands to modify the abstract syntax graph directly, the spatial relations graph must be updated also. This means that some kind of pretty printing is necessary in order to perform the reverse mapping.

We attempt in this paper to apply graph grammars to define the graphs at both levels and to specify the translation between the two levels. The description of the translation would define the graphical syntax of L; graph rewriting according to this description would implement the corresponding graphical parser.

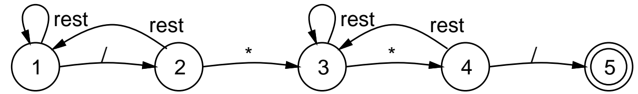
We limit our work to diagram-like pictures, the kind which can be drawn with graphical editors like `idraw`, `xfig` or `MacDraw`; we have no intention of processing bitmap-based pictures. Neither do we, at this moment, consider *incremental* parsing of pictures.

1.1 Organization of the paper

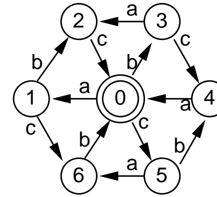
Section 2 explains the different representations of a diagram in more detail and explains which kind of translations must be performed between the levels. Next, section 3 applies the graph grammar formalism PROGRES to define the translation between two of the levels. Section 4 provides some entry points to other proposals for defining graphical syntax. Finally, section 5 comments on the specification achieved and discusses the application of graph grammars in general to the specification of graphical languages.

2 PICTURE REPRESENTATIONS

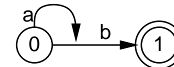
The running example of this paper will be the definition of the syntax of the language of Deterministic Finite Automata (DFA's). For example,



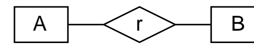
is a well-formed sentence in the language of DFA's, and



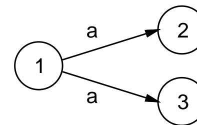
is syntactically correct, but



is not, and neither is



A sentence like



is syntactically correct, but it is incorrect according to the static semantics of DFA's (it is non-deterministic).

As mentioned in the introduction, we split the analysis of a picture according to a graphical syntax in two phases. As a result, there will be three representation levels which are increasingly abstract.

Level 1: Picture Objects

The first level describes the picture in a general format. The objects at this level are, for example, *Text*, *Ellipse*, *B-Spline* and *Line*, and these objects have attributes describing their physical appearance in detail. At this moment, it is not fully determined how to represent this level, but we imagine something like the following data structures:

Text
- Font-Family
- Font-Size
- Foreground-Color
- StartPos
- String

Circle
- Pen-Kind
- Foreground-Color
- Background-Color
- Center-Position
- Radius

B-spline
- Pen-Kind
- Foreground-Color
- Background-Color
- Position*

Box
- Pen-Kind
- Foreground-Color
- Background-Color
- LowerLeftPosition
- UpperRightPosition

Line
- ...

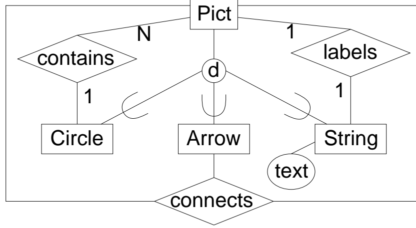


Figure 3: The (level 2) representation of pictorial objects and their spatial relations

Everything needed to (re)draw the picture is available at this level, we only abstract from the specific graphical editor used.

Level 2: Spatial Relations

At the second level we abstract from attributes which are not of interest for the graphical language L we want to recognize. These are for DFA's the *color* and *font* information.

More importantly, we use the positional information of the picture objects to derive higher level *spatial relations* between the objects. Such relations could, for example, be *contains*, *touches*, *connects* or *labels*. Once these relations are established, we can also abstract from the specific locations and sizes of the objects.

The data structures at the second level can be expressed as in Fig. 3 (we use an Entity-Relationship model in the notation of [2]). We model the spatial relations graph for DFA's with three kinds of objects, Circle, Arrow and String, which all are specializations of the more general object Pict. A circle can contain other picture objects, an arrow can connect two picture objects, and a string can label other picture objects. The string object has the string value itself as attribute. For other graphical languages, the spatial relations graph will differ of course.

The spatial relations at level 2 are specific for L, but the derivation of them from level 1 objects is independent of the context in which they appear. This is equivalent to the process of tokenization in the textual world, which is also specific for the language tokenized but independent of the context in which recognized string appears. The objects and relationships at level 2 can somehow be seen as the graphical counterparts of lexical tokens.

It is necessary to specify which objects, attributes and spatial relations are relevant for L, and how these relations can be derived from the positional information of the picture objects of level 1. The former is specific for L, but the latter might be shared by several graphical syntaxes.

The computations involved to discover these spatial relations will be of the kind: "is this a point on that line?", "do these two shapes overlap?", or "are these two objects near to each other?". The definition of the spatial relations is thus expressed in terms of coordinates and shapes and will need some geometry-like formalism. We are currently investigating the needs in this area in more detail and will not go deeper into them in this paper.

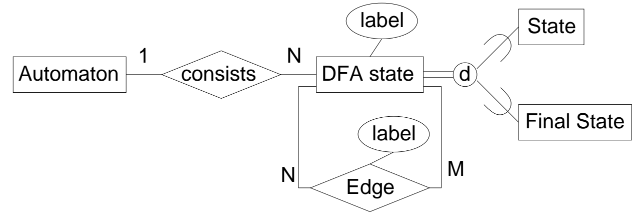


Figure 4: The (level 3) data structure of DFA's

Level 3: The language L

At the third level the abstract syntax of the graphical language L itself appears. For the abstract syntax of DFA's one could imagine a data structure as shown in Fig. 4. Entities at this level are automaton states and the labeled edges between DFA states are relationships. As a single diagram can contain more than one automaton, we introduce a an Automaton entity which takes all states contained in it together. Note that there does not exist any relation yet between the graphs at level 2 and the graphs at level 3. It is exactly this relation which we seek to define in the graphical syntax of the language.

The process of deriving a level 3 graph from level 2 pictorial objects and spatial relations is roughly equivalent to the parsing process of the textual world, where combinations of tokens and their relative positions are used to derive the abstract syntax tree.

There is a need for a context-free kind of grammar to translate the objects and relations in the picture world of level 2 to objects and relations in the L world of level 3. As both representation are graphs, it seems most appropriate to express the translation with graph rewrite rules and to apply a graph replacement system here. However, these systems describe a graph transformation instead of a translation from one kind of graphs to another. This means that, in order to apply a graph replacement system, we have to unify the representations at level 2 and 3 to a single representation. The distinction between the two levels will in that case become less clear. This approach will be pursued in the next section.

Back-links

Instances at all levels should keep so-called back-links to the objects and/or relationships at lower levels from which they were derived. This is necessary as information is lost in the abstraction, while this information is vital to obtain a graphical representation of an instance of L again. This also means that, if L instances are modified or created directly, that some layout algorithm must create structures at lower levels to represent them.

3 PARSING WITH A GRAPH GRAMMAR

There are a number of possibilities to express the transformation between the graphical level 2 and the DFA constructs of level 3. In this paper we will use graph grammars and will use the graph grammar formalism PROGRES [3, 15] to express this translation. A PROGRES specification has two parts, a static part in which the types of the nodes and the allowed edges are defined, and a dynamic part in which the allowed graph transformations are specified. We start with the data definition part.

3.1 Data structure definition

We simulate the two distinct graphs by defining two top classes, `Pict` and `DFAobject`, for the different levels. The abstract syntax of the spatial relations level (Fig. 3) can be expressed in PROGRES as follows:

```
section NODE_CLASSES;
  node class Pict end;
end;

section NODE_TYPES;
  node type Circle : Pict end;
  node type Arrow : Pict end;
  node type Text : Pict
    external text: STRING
  end;
end;

section EDGE_TYPES;
  edge type contains: Circle -> Pict;
  edge type labels:   Text   -> Pict;
  edge type startsin: Arrow  -> Pict;
  edge type endsin:   Arrow  -> Pict;
end;
```

The allowed objects in the input graph are circles, arrows and strings. These are all specializations of Pict. Circles can contain other objects, strings can label other objects and arrows can connect objects. PROGRES does not support three-way relations and we replaced the *Connect* relationship of our EER definition of Fig. 3 by two edges, *startsin* and *endsin*. In order to provide

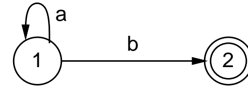


Figure 5: A simple DFA

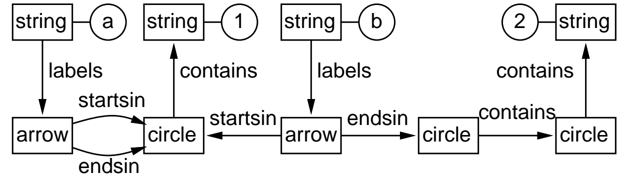


Figure 6: The spatial relations graph of Fig .5

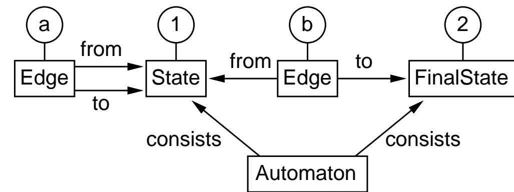


Figure 7: The internal representation of Fig .5

an example of an instance of this data structure, the spatial relations graph for the DFA of Fig. 5 is depicted in Fig. 6.

The abstract syntax of DFA's is expressed as follows in PROGRES:

```
section NODE_CLASSES;
  node class DFAobject end;
  node class DFAstate is a DFAobject
    external label: STRING;
  end;
end;

section NODE_TYPES;
  node type Automaton: DFAobject end;
  node type State: DFAstate end;
  node type Finalstate: DFAstate end;
  node type Edge: DFAobject
    external label: STRING;
  end;
end;

section EDGE_TYPES;
  edge type consists: Automaton -> DFAstate;
  edge type from:     Edge      -> DFAstate;
  edge type to:       Edge      -> DFAstate;
end;
```

The output of the parser will be a graph of DFA states and edges with their connections. We also introduce an *Automaton* object which has a *consists* relation with

every state belonging to the automaton. This automaton object and the the relations it must obtain with all states mainly serves to make the definition of the graphical syntax harder; we introduced it as a challenge to the syntax specification formalism. We represent the *DFA edge* relationship of Fig. 4 by a PROGRES type. According to this data structure, the DFA of Fig. 5 would be represented by the graph of Fig. 7.

Finally, we need to define the back-links between the objects in the DFA world and the graphical objects from which they were derived. These back-links will also be used in the translation from the level 2 objects to level 3 objects.

```
section EDGE_TYPES;
  edge type drawn_by: DFAobject -> Pict;
end;
```

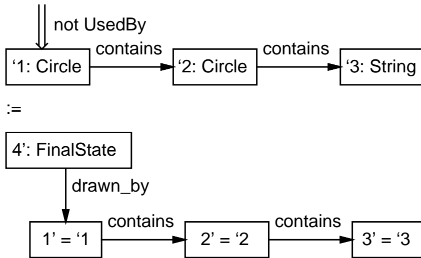
3.2 Translation rules

In the graph transformation part we are finally able to express what DFA states and DFA edges look like and how they are to be composed out of pictorial objects and their spatial relations. That would define the correspondence between the graph of Fig. 6 and the one of Fig. 5.

3.2.1 Recognizing DFA states

The first rule recognizes a double circle which contains a string (eg. ③) as being a final DFA state:

```
production ParseDFA =
```



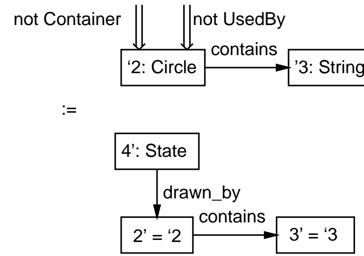

```
  transfer 4'.label := '3.text;
end;

restriction UsedBy: DFAobject =
  with <-drawn_by--
end;
```

The left-hand side of a PROGRES rule denotes the pattern to be recognized in the existing graph. For each instance in the graph that matches this pattern, the right-hand side states what modification must be performed on it. The transformation specified above introduces a node for a DFAstate and relates it to the outer circle recognized in the left-hand side. By using the construct `1' = '1` we state that node `'1` and all its connections with the rest of the graph are to be left intact. This means that we only add to the existing graph, no information is deleted. The line "**`transfer label of 4' := '3.text`**" provides the DFA state recognized with its label, which is the text of the string contained in the double circle.

In order to prevent the above rule to match infinitely often we add the limitation that only pictorial objects that haven't been used yet can be used as left-hand side. This is expressed by the restriction **`UsedBy`** which returns the set of DFAobjects which have a **`drawn_by`** relation with the outer circle recognized. The above production may be fired only if this set is empty. Fig. 8 shows the graph after having recognized the final state in the graph of Fig. 6.

The rule to recognize ordinary DFA states (eg. ⑤) can be formulated as follows:

```
production ParseDFA =
```




```
  transfer 4'.label := '3.text;
end;

restriction Container: Pict =
  with <-contains--
end;
```

This rule uses two restrictions, one to disallow multiple matches of the same pictorial information and one to prevent that the inner circle of a double circle is recognized as being an ordinary state. Fig. 9 shows the graph after having recognized the ordinary state.

3.2.2 Recognizing DFA edges

Next, we can formulate the rule to recognize DFA edges (eg. $\xrightarrow{a}$) in the picture. This rule uses the **`drawn_by`** relation to learn which DFA states are connected by the arrow.

```
production ParseDFA =
```

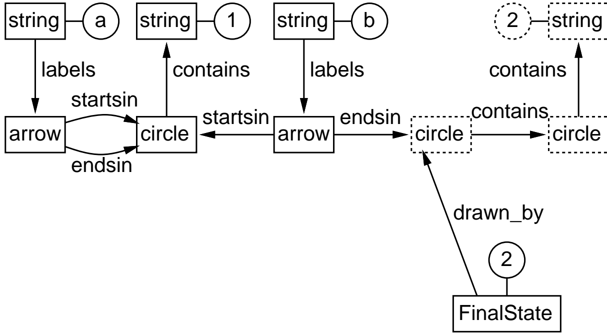


Figure 8: Final state 2 recognized

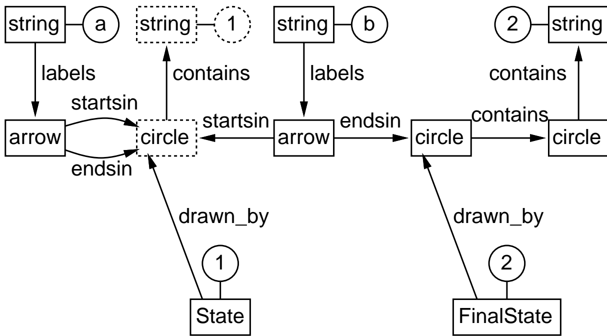


Figure 9: Ordinary state 1 recognized

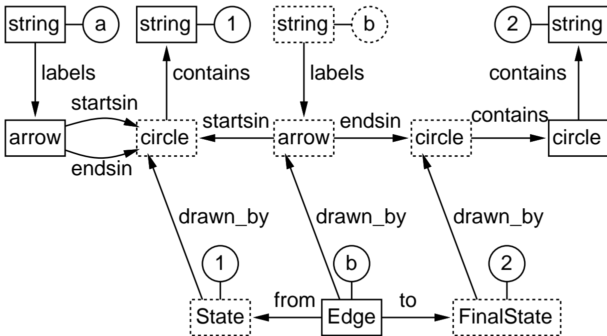


Figure 10: Edge *b* recognized

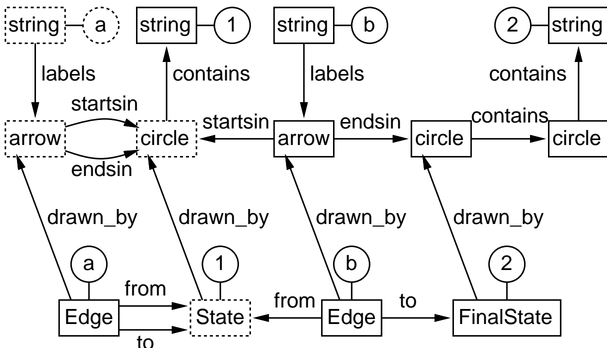


Figure 11: Edge *a* recognized

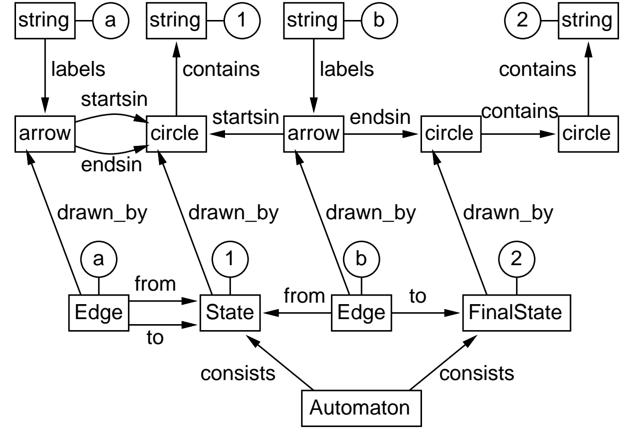
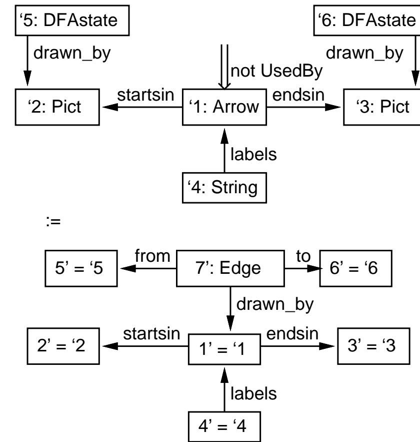


Figure 12: The automaton object recognized




```
  transfer 7'.label := ‘4.text;
end;
```

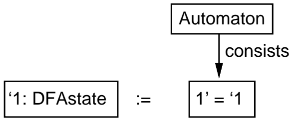
This rule states "if a labeled arrow connects two pictorial objects which have been recognized as being DFA states, then the arrow represents a DFA edge between those states". Note the use of the *drawn_by* edge in this transformation rule. The arrow has a *startsin* and an *endsin* relation with two objects of kind *Pict*. By stating that these two must have a *drawn_by* relation with two states again, we locate the two states which must be connected by the edge recognized in the arrow. We use the superclass *Pict* instead of *Circle* as object kind here, in order not to mention once more that states are depicted by a circle. Figures 10 and 11 show the successive graphs after the recognition of the two edges.

3.2.3 Taking the automaton together

Finally, we need to take all connected DFA states together under a single *Automaton* object. We specify this by first stating that every DFA state belongs to some automaton, and next by stating that DFA states which are connected by a DFA edge belong to the same

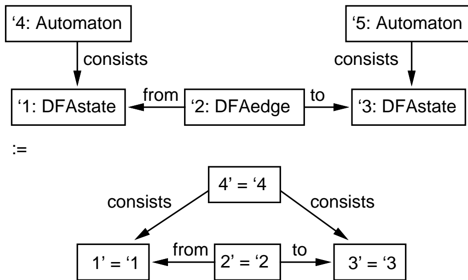
automaton. This leaves us with as many automata as there are distinct subgraphs in the picture.

```
production ParseDFA =
```




```
end;
```

```
production ParseDFA =
```




```
  condition not ( '4 = '5 );
  embedding redirect --consists-> from '5 to '4;
end;
```

Fig. 12 shows the final graph, after having applied all translation rules. This leaves us indeed with the desired graph of Fig. 7, annotated with back-links towards the objects in the spatial relations graph.

3.3 Discussion

PROGRES is well fit to describe the parsing process of DFA's as a transformation of graphs. The specification of this derivation only took a few rules and we feel that formulating graph transformation rules in a graphical manner improves the comprehensibility greatly.

Still, due to the fact that a graph grammar is a general tool and not specified towards the definition of graphical syntax, the rules need to express more than strictly necessary, as convenient defaults specialized towards the definition of graphical syntax are not available.

Evaluating the graph-grammar approach we consider the following points *positive* in it:

- The definition of graphical objects and their relations in a data definition formalism which allows for sub-classing and attributes leads to short and expressive definitions.
- The definition of the structure of the input graph and that of the output graph in a single formalism minimizes the number of formalisms which need to be understood.
- The fact that the grammar rules themselves can be defined in a graphical manner provides good insight in the sometimes complicated rules.
- The transformation rules are very powerful in terms of the structures they can match and the allowed graph transformations they can prescribe.

We consider the following points *negative* in our PROGRES definition of the graphical syntax of DFA's:

- There is no distinction between the universe in which the input graph exists and the one in which the output graph is generated. The only distinction between the two are the names used by the writer of a grammar.
- The creation of the *drawn_by* relation and the checking for it is cumbersome. This should be handled automatically by the underlying implementation.
- The spatial relations only exist between graphical objects; This forces the transformation rules to follow the *drawn_by* relations.
- The graphical part of the transformation rules lacks expressiveness, and there often is a need for additional constraints or actions which need to be defined in a textual manner.

4 RELATED WORK

The approach of Helm and Marriott [10, 11] towards graphical parsing is to define the graphical syntax entirely in a logic formalism. They do not split the translation in separate phases. Their parsers are implemented by a constraint solver such as PROLOG.

Wittenburg et al. [16, 17, 18] propose relational unification grammars to specify the parsing of pictorial objects and their spatial relations. They implement the parsers defined by these grammars with a chart-based relational parsing algorithm which has been developed by them also.

Golin et al. [5, 6, 7, 8] present a compiler generator for visual languages which is based on the notion of *picture layout grammars*. In such a grammar a syntax rule is limited to a single spatial relation which connects two symbols. This generally leads to a large number of non-terminals and syntax rules. Symbols may be defined as being "remote"; these result in additional connections in the derivation tree, effectively turning it into a graph. Spatial relations are based on the bounding box of graphical objects, non-terminals obtain the enclosing bounding box of their constituents. Picture layout grammars have been used successfully to define the syntax of several graphical languages.

Our approach to graphical parsing is to specify the syntax in a graph grammar. For a general introduction to this field the proceedings of the various workshops on graph grammars can best be scanned; [1] is a good entry point. A well known implementation of graph grammars is the PROGRES system [3, 15].

Göttler [9] also describes the application of graph grammars to the processing of graphical languages. He describes how diagrams can be represented by graphs and how graph grammar productions implement the edit operations to the diagrams represented. This is close to our spatial relations graph and the processing of the "unstructured edit commands" of Fig. 2 in it. Göttler stops here and does not address the next step in the analysis, the step which performs graphical parsing and derives the structure depicted by the diagram.

Kaul [12] describes a class of context-free precedence graph grammars for which he provides a linear time parsing algorithm. This parser produces a derivation tree according to the production rules in the graph grammar, and the technique could be very useful for the implementation of our graphical parsers. However, the fact that the graphs must reduce to a single non-terminal makes it harder to formulate the syntax of a graphical language. Furthermore, the spatial relations, on which parsing is based, may in our setting be ambiguous. Only a parser that can backtrack or that can process alternative derivations in parallel, is able to find correct derivations for these graphs (a future publication of ours discusses the problems related to parser implementation in more detail [14]). Therefore, it seems that the techniques proposed by Kaul are not directly applicable to our problem setting.

5 CONCLUSIONS AND FUTURE WORK

We want to develop a graphical parsing algorithm which is able to analyze a diagram according to a graphical syntax and which derives the structure depicted by that diagram. Such parsing algorithm would enable graphical editors which allow both structured and unstructured editing.

We have split the definition of the syntax of graphical languages in two parts, one part to define the relevant pictorial objects and their spatial relations, and a second part which defines how constructs in the language to be recognized are composed of these pictorial objects and relations. We consider this distinction to be very useful, as the two need entirely different definition and implementation strategies.

We have applied graph grammars to define both data structures and the transformation between the two, thus defining a graphical parser. We have carried out an experiment with a very simple graphical language and consider the results positive. However, it is unclear whether more complicated graphical grammars can be expressed with the same ease. This is subject to further research.

It could be profitable to use graph grammars to *define* the graphical grammars only, and not to use graph rewriting to implement the parser. Due to the general nature of graph rewriting, we expect this to be less efficient than a more specific graphical parsing algorithm. We can imagine that a translation of our graph grammar transformation rules to the relational grammars of Wittenburg [17] could lead to more efficient parsing.

ACKNOWLEDGEMENT

I would like to thank Gregor Engels for his numerous stimulating remarks and ideas.

REFERENCES

[1] Ehrig, Kreowski, and Rozenberg, editors. *4th international workshop on Graph Grammars and their application to Computer Science, LNCS 532*, Bremen, Germany, 1990. Springer Verlag.

[2] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1989.

[3] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments – part 1: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[4] G. Engels and P. Löhr-Richter. A highly integrated environment to support conceptual database design. In *Proceedings of the 5th international workshop on Computer-Aided Software Engineering – CASE'92*, pages 19–22, Montréal, Québec, Canada, 1992.

[5] E. J. Golin. *A method for the specification and parsing of visual languages.* PhD thesis, Brown University, May 1991.

[6] E. J. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, December 1991.

[7] E.J. Golin and T. Magliery. A compiler generator for visual languages. In *Proceedings 1993 IEEE Symposium Visual Languages*, pages 314–321, Bergen, Norway, August 1993.

[8] E.J. Golin and S.P. Reiss. The specification of visual language syntax. *Journal of Visual Languages and Computing*, 1(2):141–156, June 1990.

[9] H. Göttler. Graph grammars, a new paradigm for implementing visual languages. In C. Ghezzi and

J. A. McDermid, editors, *the 2nd European Software Engineering Conference (ESEC'89), LNCS 387*, pages 336–350. Springer–Verlag, 1989. Also in: Proceedings Eurographics 1989, pages 505–516.

[10] R. Helm and K. Marriott. Declarative specification of visual languages. In *Proceedings 1990 IEEE Workshop Visual Languages*, pages 98–103, Skokie, Illinois, October 1990. Extended version in technical report RC 15813 of the IBM T.J. Watson Research Center.

[11] R. Helm and K. Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2(4):311–332, December 1991.

[12] M. Kaul. Parsing of graphs in linear time. In *2nd international workshop on Graph Grammars and their application to Computer Science, LNCS 153*, pages 206–218, 1982.

[13] M.F. Kleyn and J.C. Brown. A high level language for specifying graph based languages and their programming environments. In *Proceedings 15th international conference on software engineering*, pages 324–335, Baltimore, Maryland, 1993.

[14] J. Rekers. Graphical definition of graphical syntax. Submitted for presentation at the *IEEE/CS Symposium on Visual Languages (VL'94)*, St. Louis, Missouri, October 1994.

[15] A. Schürr. PROGRESS: A VHL–language based on graph grammars. In Ehrig, Kreowski, and Rozenberg, editors, *4th international workshop on Graph Grammars and their application to Computer Science, LNCS 532*, pages 641–659. Springer Verlag, 1990.

[16] K. Wittenburg. Parsing with relational unification grammars. In *Proceedings of the second International Workshop on Parsing Technologies – IWPT'91*, pages 225–234, Cancun, Mexico, 1991. Association for Computational Linguistics.

[17] K. Wittenburg and L. Weitzman. Visual grammars and incremental parsing for interface languages. In *Proceedings 1990 IEEE Workshop Visual Languages*, pages 111–118, Skokie, Illinois, October 1990.

[18] K. Wittenburg, L. Weitzman, and J. Talley. Unification–based grammars and tabular parsing for graphical languages. *Journal of Visual Languages and Computing*, 2(4):347–370, December 1991.