Abstract Visual Syntax

Martin Erwig

FernUniversität Hagen, Praktische Informatik IV
D-58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract. We propose a separation of visual syntax into concrete and abstract syntax, much like it is often done for textual languages. Here the focus is on visual programming languages; the impact on visual languages in general is not clear by now. We suggest to use unstructured labeled multi-graphs as abstract visual syntax and show how this facilitates semantics definitions and transformations of visual languages. In particular, disregarding structural constraints on the abstract syntax level makes such manipulations simple and powerful. Moreover, we demonstrate that – in contrast to the traditional monolithic graph definition – an inductive view of graphs provides a very convenient way to move in a structured (and declarative) way through graphs. Again this supports the simplicity of descriptions for transformations and semantics.

1. What is Abstract Visual Syntax?

To specify (the syntax of) a visual language one can choose among a large variety of formalisms, for a comparison, see [MM96]. All these approaches are concerned with the concrete syntax of visual languages, and there are only a few authors explicitly mentioning abstract visual syntax [AER96, RS95, RS96]. This is surprising since work on textual programming languages has demonstrated that the use of abstract syntax can be very helpful in many different areas, for example, specification of type systems, compiler construction, program transformations, and semantics definition. One reason for this situation might be that the right level of abstractness has not been identified yet: we believe that abstract visual syntax must be "more abstract" than in the textual case.

We explain this by a simple example. Consider the following (textual) grammar describing part of a concrete syntax for expressions.

expr ::= *n-expr* | *b-expr* | *if-expr*
n-expr ::= *term* | *n-expr* + *term*
term ::= *factor* | *term* * *factor*
factor ::= *id* | (*n-expr*)
b-expr ::= *id* | *b-expr* $\vee$ *b-expr* | …
if-expr ::= **if** *b-expr* **then** *expr* **else** *expr*

A corresponding abstract syntax would ignore many details, such as the choice of key

words, grammar rules for defining associativity of operators, or rules restricting the typing of operations (see also [Mos90]):

expr ::= *id* | *expr op expr* | **if** (*expr*, *expr*, *expr*)
op ::= + | * | ∨ | …

This grammar is much more concise. It does not introduce nonterminals for expressions of different types, and it also ignores associativity of operators. (Omitting the key words from the conditional does not make the grammar essentially simpler in this example.) Further operations on sentences of the language can rely on syntax being already checked by a parser and can thus work with the simpler abstract syntax.

In a similar way, the abstract syntax of visual languages need not be concerned with all the details that a concrete syntax specification has to care about. This means we can abstract from the choice of icons or symbols (comparable to the choice of key words in the textual case) and from geometric details such as size and position of objects (at least up to topological equivalence, that is, as long as relevant relationships between objects are not affected). We can also ignore associativities used to resolve ambiguous situations during parsing much like in the textual example. Moreover, typings of relationships that restrict relationships to specific subsets of symbols can be omitted. This corresponds to grouping operations, such as + or ∨, under one nonterminal.

But we can do even more – and this is the point where abstract visual syntax gets more abstract than in the textual case: the above abstract syntax for expressions is still given by a grammar and thus retains some structural information about the language. This is absolutely adequate since the description is very simple and can be easily used when defining, for example, an interpreter for expressions. However, to do so for a visual language requires, in most cases, some effort in the consideration of context information which unnecessarily complicates definitions of transformations. Therefore, we suggest to forget about this structural information, too, and to consider a picture just as a directed, labeled multi-graph where the nodes represent objects and the edges represent relationships between objects. A class of graphs is then just given by two types defining node and edge labels, that is, the types of objects and relationships in the abstractly represented visual language.

Definition 1. A *directed labeled multi-graph of type* (α, β) is a quintuple $G = (V, E, \iota, \nu, \varepsilon)$ consisting of a set of nodes V and a set of edges E where $\iota: E \to V \times V$ is a total mapping defining for each edge the nodes it connects. The (partial) mappings $\nu: V \to \alpha$ and $\varepsilon: E \to \beta$ define the node and edge labels.

For a graph G, $V(G)$ and $E(G)$ denote the set of nodes and edges of G. Whenever G is fixed, we also might simply use V and E. For brevity, we sometimes denote a node or edge x together with its label l simply by $x{:}l$.

The *source* of an edge $e \in E$ with $\iota(e) = (v, w)$ is defined as $\sigma(e) = v$, and the *target* of e is defined as $\tau(e) = w$. A *path* in a graph G is an alternating sequence of nodes and edges $p = [v_1, e_1, v_2, e_2, \ldots, v_n, e_n, v_{n+1}]$ (with $e_i \in E(G)$, $1 \le i \le n$) such that for all $1 \le i \le n$: $\sigma(e_i) = v_i$ and $\tau(e_i) = v_{i+1}$. In particular, an empty path is a sequence $[v_1]$ containing just a single node. When no ambiguities can arise, that is, if each e_i is uniquely determined by v_i and v_{i+1}, we also denote a path more concisely by $[v_1, v_2, \ldots, v_{n+1}]$.

We extend the definition of σ and τ to paths by: $\sigma(p) = v_1$ and $\tau(p) = v_{n+1}$. $P(G)$ denotes the set of all paths in G. Note that $P(G)$ does not just include simple paths, but also cycles and paths in which edges might occur multiple times.

The label types α and β might just be sets of symbols, or they can be complex structures to enable the labeling with terms or semantic values, say. The set of all graphs of type (α, β) is denoted by $\Gamma(\alpha, \beta)$.

Definition 2. A *visual language of type* (α, β) is a set of graphs $VL \subseteq \Gamma(\alpha, \beta)$.

In the sequel we will look at visual languages on this very abstract level, that is, the abstract syntax of a visual language is specified as a set of graphs of a specific type.

Of course, structural information has to be explored in some way when defining transformations or semantics, but with the presented approach this happens through a specific kind of pattern matching so that context is explored only as far as really needed for a particular task. This pattern matching provides an inductive view on graphs which facilitates the analysis and transformation of graphs in many situations.

The abstractness of representations is affected by yet another factor: the choice of relationships. A representation based on elementary relationships, such as *touches* or *inside*, gets much larger than a graph whose edges denote relationships like "are connected by an arrow labeled *x*". We will illustrate this point further in Section 2. Before that we compare our approach with other work in Section 2. Graph pattern matching and the inductive graph view is described in Section 4, and applications of the abstract syntax representation are shown in Section 5. Conclusions follow in Section 6.

2. Related Work

Using graphs to describe pictures is a common and wide-spread approach. However, general models that apply to a broad range of visual languages are few. Examples are Harel's higraphs [Har88] and the theory of graph grammars [Cou90].

Higraphs are a combination of hierarchical graphs and Euler/Venn diagrams and provide a perfect representation for those visual languages that exactly fit that model. However, since higraphs have a fixed structure, their applicability is restricted, and only a certain class of visual languages can be expressed in terms of them. Moreover, higraphs do not offer an inductive view of graphs which makes some specification or transformation tasks more difficult, if not impossible.

Graph grammars, on the other hand, provide a fairly general model of visual languages. Graph grammars are very powerful, and they have been extensively used to describe graph transformations. Graph grammars enjoy a large body of theoretical results, and they also provide, in a certain sense, an inductive view of graphs. So why should we need yet another graph model? A major difficulty with graph grammars is that they consider the graphs they operate on as global variables that can be updated destructively. This means that changes performed by grammar rules are implicitly propagated, and thus a declarative treatment of graphs is prohibited. Things are complicated by the fact that the semantics of graph grammars themselves is rather complex due to advanced embedding rules and nondeterminism. In contrast, the inductive graph view to be presented in Section 4 is quite simple, and it treats graphs as explicit param-

eters of transformations.

Apart from graph representations, the specification of visual language syntax has been addressed in algebraic frameworks [WL93] and by several logical formalisms, for an overview, see [MMW96]. In the formalism of [WL93] a picture is represented by the set of terms generated from a sub-signature of the signature defining the visual language. Actually, this representation is not very different from the logical approach in which a picture is represented by set of facts. It is interesting to note that these formalisms always deal with concrete syntax, although it would be possible, in principal, to derive more abstract representations by appropriate predicate definitions. In any case, it is possible to transform a logical description into a graph and vice versa. So why prefer graphs over logical representations? To some degree, it might be a matter of taste, but the applications of abstract visual syntax often require the decomposition of graphs performed in specific order, and for that graphs seem to be more convenient.

Finally, the notion of abstract visual syntax is explicitly mentioned in [AER96, RS95, RS96]. Although the authors recognize the need to separate concrete and abstract syntax of visual languages, they do not really achieve this separation since for their applications a one-to-one correspondence between both levels is required.

3. Examples: State Diagrams and VEX

Consider a visual language for state diagrams. Our goal is to define a semantics for this visual language by specifying which set of strings is accepted by the corresponding finite (nondeterministic) automaton.

In a state diagram, circles represent states and arrows represent state transitions. One arrow is labeled with a special symbol "*start*" and is only connected with its head to a circle. This circle represents the initial state of the automaton. All other arrows are connected with their head and tail to circles and are labeled by a set of symbols drawn from some alphabet A or by the symbol $\Diamond$ denoting the empty word. Final states are represented by double circles.

The picture in Figure 1 denotes an automaton accepting strings of a's and b's containing two consecutive a's or b's:

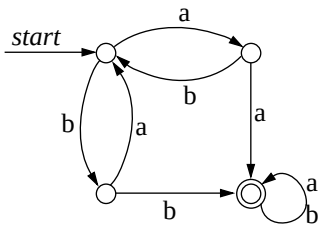

Figure 1. Finite Automaton for the Language (a∪b)*(aa∪bb)(a∪b)*

What is the general structure of state diagrams? A state diagram consists of the following objects: Circles, double circles, symbols (from an alphabet A), arrows, and the special symbol *start*. Thus the type of node labels is $\{\bigcirc, \circledcirc, \rightarrow, \mathit{start}\} \cup A$. Relevant relationships are: Connection of arrows to (double) circles and attachment of symbols

to arrows. We therefore have the type of edge labels {src, tgt, lab} where “src” (“tgt”) labels edges that represent the connection of an arrow to its source (target) and “lab” labels edges that represent the attachment of labels to arrows. The abstract syntax graph for the above automaton is shown in Figure 2.

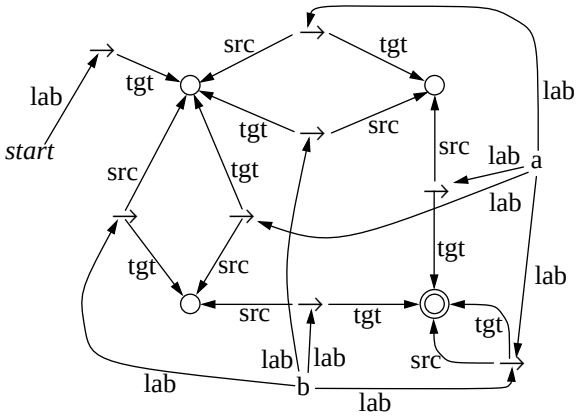

Figure 2. Abstract Syntax Graph for the Automaton

Since we ignore structural constraints, it is clear that the visual language defined by graphs of type ($\{\bigcirc, \circledcirc, \rightarrow, \mathit{start}, \Diamond\} \cup A$, {src, tgt, lab}) is actually a proper superset of correctly formed state diagrams.

It is interesting that – at least for graphs representing syntactically correct state diagrams – some objects and relationships always occur in fixed configurations. For example, all $\rightarrow$-nodes (except one) have two outgoing edges connected to circles, one labeled “src” and the other labeled “tgt”, and one incoming edge labeled “lab” which comes from a symbol-node, that is, from a node with a label $\in A \cup \{\Diamond\}$. Now we can safely replace (that is, without loosing essential picture information) such a subgraph representing a fixed configuration of objects and relationships by a more abstract relationship, that is, by just one edge going from the target of the “src”-edge to the target of the “tgt”-edge labeled like the source of the “lab”-edge.We have to decide what to do with arrows that are labeled by more than one symbol. For convenience we introduce a different edge for each label. After this translation we can drop all symbol-nodes since they are isolated and do not take part in any relationship. Finally, we can get rid of the path [u:*start*, v:$\rightarrow$, w:$\bigcirc$] by marking the last node as a start node. Since this can, in general, also be a final node, we have three kinds of node labels: Just a start node (*S*), just a final node (*F*), or a start and final node (*SF*). All other nodes can be left unlabeled. Now we have a much more succinct abstract representation of state diagrams given by ($\{S, F, SF\}, A \cup \{\Diamond\}$)-graphs, see Figure 3.

The similarity of this representation to the original visual language is purely accidental. In general, higher abstraction levels cause the representations to differ significantly from the visual original.

To give an example, consider the language VEX, which offers a visual notation for the lambda calculus [CHZ95]: empty circles represent identifiers, non-empty circles

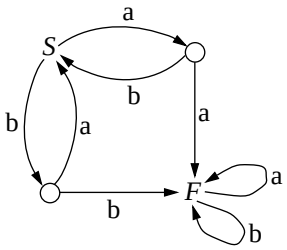

Figure 3. More Abstract Syntax Graph for the Automaton

denote abstractions, and an application is depicted by two externally tangent circles with an arrow at the tangent point leading from the function to the argument. Each circle representing an identifier is connected by a straight line to a so-called root node, which is either internally tangent to an abstraction circle and represents then the parameter of the abstraction or which lies outside any other circle and then represents a free variable. Figure 4 shows the VEX expression for the lambda term $\lambda y.((\lambda x.yx)z)$.

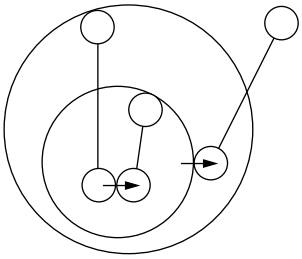

Figure 4. A simple VEX picture

We can, again, give abstract syntax on different levels of abstractness. First, we can stay in a representation rather close to the original by simply replacing lines and arrows by *def*- and *apply*-labeled edges and replacing the *inside*- and *internally-touches*-relationships by *body*- and *par*-edges, respectively. We do not need node labels at all since abstraction nodes can be distinguished from variable nodes by looking at the incident edges. Thus the abstract syntax for VEX can be given by graphs of type ($\emptyset$, {*def*, *apply*, *par*, *body*}). The abstract syntax for the VEX example is shown in the left part of Figure 5.

On the other hand, we can represent VEX pictures by DAGs to facilitate concise semantics definitions, see [Erw97b] for details. Such a representation consists of application-, abstraction- and variable-nodes with corresponding node labels: @, λ, $\bigcirc$ (unlabeled). An @-node has an outgoing *fun*-edge and an outgoing *arg*-edge that lead to the function to be applied and the argument, respectively. A λ-node is connected by an outgoing *par*-edge to its parameter and by an outgoing *body*-edge to the node representing its body. Hence, this abstract syntax for VEX uses graphs of type ({@, λ}, {*fun*, *arg*, *par*, *body*}). This abstract syntax version is shown in the right part of Figure 5.

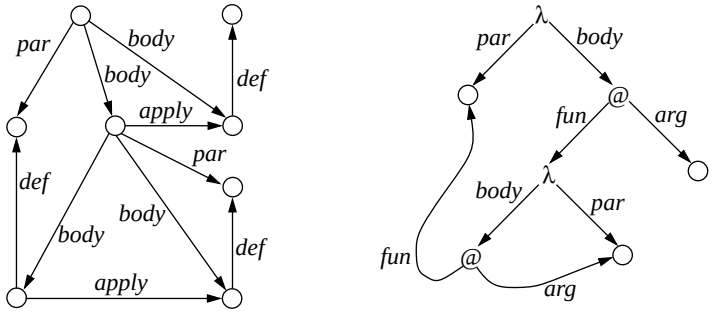

Figure 5. Abstract VEX syntax

4. Inductive Graph View

We can view a graph in the style of algebraic data types found in functional programming languages: a graph is either empty, or it is constructed by a graph *g* and a new node *v* together with edges from *v* to its successors in *g* and edges from its predecessors in *g* leading to *v*. This way we can construct graphs with a constant constructor *Empty* and a constructor *N* taking as arguments a triple (*pred-spec*, *node-spec*, *succ-spec*), called *node context*, and the graph *g* to be extended. Here, *node-spec* is a node identifier not already contained in *g* possibly followed by a label (for example, *v*:*S* or *y*:*F*), and *pred-spec* (*succ-spec*) denotes a list of predecessor (successor) nodes possibly extended by labels for the edges that come from (lead to) the nodes. For instance, [*x*›a, *w*›b] denotes a list of two predecessor nodes *x* and *w* where the edges coming from *x* and *w* have the respective labels "a" and "b". Similarly, [a›*y*, b›*y*] denotes a single successor *y* that is reached via two differently labeled edges. The graph from Figure 3, for example, is given by the following expression:

(*N* ([*x*›a, *w*›b], *v*:*S*, [a›*w*, b›*x*])
(*N* ([], *w*, [a›*y*]) (*N* ([], *x*, [b›*y*]) (*N* ([], *y*:*F*, [a›*y*, b›*y*]) *Empty*)))

Here *v*, *w*, *x*, and *y* are arbitrary node identifiers that are pairwise distinct. In the sequel we make use of two abbreviations: (1) empty sequences can be omitted, and (2) a cascade of *N*-constructors is replaced by a single *N**-constructor. So the above term can be simplified to:

*N** ([*x*›a, *w*›b], *v*:*S*, [a›*w*, b›*x*]) (*w*, [a›*y*]) (*x*, [b›*y*]) (*y*:*F*, [a›*y*, b›*y*]) *Empty*

Graph expressions are by no means unique, in particular, the order in which nodes appear in a constructing term can be arbitrarily changed. (Of course, the predecessor and successor specifications have to be adjusted accordingly.) It should be clear that any labeled multi-graph can be represented by a graph expression. A proof for this can be found in [Erw97a]. There we also give a formal semantics for graph types and graph constructors.

The main use of graph constructors in the context of this work is not to build new graphs but to take part in pattern matching on graphs. Especially useful for graphs is

the concept of *active patterns* [Erw96]: as known from pattern matching on data types, matching a pattern like *N* (*p*, *v*:*l*, *s*) *g* to a graph expression binds the last inserted node context to *p*, *v*, *l*, and *s* and the remaining graph to *g*. However, in order to move in a controlled way through the graph, it is necessary to match the context of a specific node. This is possible if *v* is already bound to the node to be matched. Then the context of *v* is bound to the remaining variables. For example, matching the pattern *N* (*p*, *y*:*l*, *s*) *g* against the above graph expression results in the following bindings:

$p \rightarrow$ [*w*›a, *x*›b], $l \rightarrow F$, $s \rightarrow$ [a›*y*, b›*y*], $g \rightarrow$ "*rest-graph*"

where *rest-graph* is an arbitrary representation of the matched graph without node *y* and its incident edges, for example,

*N** ([*x*›a, *w*›b], *v*:*S*, [a›*w*, b›*x*]) (*w*) (*x*) *Empty*

We can restrict patterns further by adding labels that must be present or by replacing list variables like *s* or *p* by more concrete lists patterns. For example, *x*::*l* matches any nonempty list and binds *x* to its head and *l* to its tail. It is also possible to match lists of a specific length, and we can also ignore bindings altogether by simply omitting the corresponding parts of the pattern. For instance, we can match node *v* binding the "a"- and "b"-successor nodes to variables *i* and *j*, respectively, by using the pattern: *N* (*v*, [a›*i*, b›*j*]) *g*. Then, *i* and *j* will be bound to the nodes *w* and *x*, respectively. Since we did not specify anything for the predecessor list, no binding will be produced. If we wanted to ensure that the matched node has no predecessors, we would have used the pattern *N* ([], *v*, [a›*i*, b›*j*]) *g* instead. This, however, fails to match our example graph. In cases like this, the next pattern (in a function definition) is tried.

Matching a cascading pattern *N** $c_1\ c_2\ \ldots\ c_n\ g$ against a graph *g*' works as follows: let $g_1, \ldots, g_n$ be auxiliary variables to be bound to intermediate decomposed graphs. Now first, *N* $c_1\ g_1$ is matched against *g*', and the bindings produced by this match, especially the node bindings in c_1 and the rest graph g_1, are then used to match *N** $c_2\ \ldots\ c_n\ g$ against g_1, that is, *N* $c_2\ g_2$ is matched against g_1, *N* $c_3\ g_3$ is matched against g_2, and so on, until *N* $c_n\ g_n$ is matched against $g_{n\text{-}1}$. Then *g* is bound to g_n. In this way, *N** patterns can actually be used to conveniently find paths (of fixed length) in the graph.

We also shall use an additional edge constructor *E* (*v*›*l*›*w*) *g* which simply inserts an edge with label *l* between the two nodes *v* and *w* (*v* and *w* must be already present in *g*). Strictly, the *E* constructor is not needed, it can be expressed in terms of *N* by, for example, matching *v* and re-inserting *v* with *l*›*w* as an additional successor:

E (*v*›*l*›*w*) (*N* (*p*, *v*:*m*, *s*) *g*) := *N* (*p*, *v*:*m*, (*l*›*w*)::*s*) *g*

In some cases *E* is much more convenient to use than nested applications of *N*.

5. Abstract Visual Syntax in Action

In this section we first demonstrate how to define semantics for state diagrams using the second given abstract visual syntax. Then we show how the inductive graph view

can be used to map between different abstract syntax levels. The use of inductive graphs in semantics definitions is illustrated in further detail in [Erw97b].

5.1 Path Semantics for State Diagrams

The semantics definition maps paths of the abstract syntax graph to words over *A*. Actually, this happens in two steps: First, we define the set of paths of the state diagram that correspond to trails in the automaton taken when a word is accepted. These are all finite (simple and nonsimple) paths for which the source node is labeled *S* (or *SF*) and the target node is labeled *F* (or *SF*). Then the set of accepted words is obtained by concatenating the labels along each such path. This concatenation is defined by an aggregation function over paths:

$$agg\ (f, u)\ [v] := u$$
$$agg\ (f, u)\ [v_1, e_1, v_2, \ldots, v_{n+1}] := f\ (\varepsilon(e_1), agg\ (f, u)\ [v_2, \ldots, v_{n+1}])$$

This means that an empty path is mapped to *u*, and a nonempty path is aggregated by combining the label of the first edge with the aggregation of the rest path by *f*. (*agg* is actually the *fold*/*reduce* operator found in functional languages.) Now the word represented by a path in an automaton can be obtained by aggregating the path with *u* being the empty word and *f* defining the concatenation of a character *a* in front of a word *s*. This is usually denoted by $a{\cdot}s$ where $\Diamond{\cdot}s = s{\cdot}\Diamond = s$.

Hence the semantics of an automaton represented by an abstract syntax graph *g* can be simply defined as:

$$S[g] := \{agg\ (\cdot, \Diamond)\ p \mid p \in P(g) \wedge \nu(\sigma(p)) \in \{S, SF\} \wedge \nu(\tau(p)) \in \{F, SF\}\}$$

5.2 Syntax Transformations

We have seen that the abstract visual syntax can be defined on quite different levels of abstraction. On the one hand, the syntactic description can be very fine-grained reflecting more or less the exact spatial syntax. This representation level is suitable, for instance, for a parser. On the other hand, the syntax might abstract from many geometric details using advanced relationships between objects that are represented on a lower level by several elementary relationships. This representation is often much better suited for semantics definitions and for compiling purposes. Now having formally defined mappings between these different levels we can on the one hand tie the semantics of the abstract level to the concrete visual appearance of the language, on the other hand we can join a parsing front end with a compiling back end providing a complete compiler/interpreter for a visual language. This supports the modular design of visual language implementations.

As an example we show how to transform the first automaton syntax into the second. The main work is to replace →-nodes with incident edges by new edges. This is done by the function *repl* which consists of three cases.

The first line applies when a node *u* with label → exists that has at least one predecessor (*x*) and two successors (*v* and *w*) connected to *u* by a “src”-, respectively, “tgt”-edge. In that case an edge from *v* to *w* with label *l* is inserted into the graph. This is

done with the edge constructor *E*. Here, *l* is obtained by matching *x* after *u* in the cascade pattern. Note that *E* (*v*›*l*›*w*) is not just applied to *g*, the decomposed graph without *u* and *x*. Instead, first *repl* is applied recursively to *g* into which *x* and *u* have been re-inserted. The re-insertion of *x* is necessary since it might be used to label other edges, and the re-insertion of *u* with all remaining predecessors *p* is needed to enable the edge insertion for possibly other labels (given by *p*). The second case simply deletes →-nodes which have no labels anymore, and the third case is for termination.

repl (*N** (*x*::*p*, *u*:→, [src›*v*, tgt›*w*]) (*x*:*l*, *s*) *g*) :=
    E (*v*›*l*›*w*) (*repl* (*N** (*p*, *u*:→, [src›*v*, tgt›*w*]) (*x*:*l*, *s*) *g*))
repl (*N* ([], *u*:→) *g*) := *repl g*
repl g := *g*

Deleting isolated label nodes is done by the function *del*:

del (*N* ([], *v*, []) *g*) := *del g*
del g := *g*

Finally, the replacement of the *start*-node and the relabeling of final states is accomplished by the function *relab*. The first pattern determines the path from the *start*-node via the →-node *u* to the actual start node *w* of the automaton.

relab (*N** (*u*:*start*, [*v*]) (*v*, [*w*]) (*w*:*l*, *s*) *g*) := *N* (*u*:**if** *l*=◎ **then** *SF* **else** *S*, *s*) (*relab g*)
relab (*N* (*p*, *v*:◎, *s*) *g*) := *N* (*p*, *v*:*F*, *s*) (*relab g*)
relab g := *g*

Finally, the complete transformation is simply given by:

transform g := *relab* (*del* (*repl g*))

6. Conclusions

We have shown how graphs can serve as an abstract representation of visual languages. Taking graphs *without* structural constraints allows to easily switch between different representations for the same language and to choose the abstraction level suited best for a particular task. Moreover, adopting an inductive graph view facilitates mapping between different representations of the same visual language which might be helpful in combining different phases of language elaboration.

References

[AER96] Andries, M., Engels, G. & Rekers, J.: How to Represent a Visual Program?, *Workshop on Theory of Visual Languages*, 1996.

[CHZ95] Citrin, W., Hall, R. & Zorn, B.: Programming with Visual Expressions, *IEEE Symp. on Visual Languages*, pp. 294-301, 1995.

[Cou90] Courcelle, B.: Graph Rewriting: An Algebraic and Logic Approach, in J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Vol. B*, Elsevier, pp. 193-242, 1990.

[Erw96] Erwig, M.: Active Patterns, *8th Int. Workshop on Implementation of Functional Languages*, LNCS 1268, pp. 21-40, 1996.

[Erw97a] Erwig, M.: Functional Programming with Graphs, *2nd ACM SIGPLAN Int. Conf. on Functional Programming*, pp. 52-65, 1997.

[Erw97b] Erwig, M.: Semantics of Visual Languages, *IEEE Symp. on Visual Languages*, 1997. http://voss.fernuni-hagen.de/pi4/erwig/abstracts.html#VL97

[Har88] Harel, D.: On Visual Formalisms, *Communications of the ACM, Vol. 31*, No. 5, pp. 514-530, 1988.

[MM96] Marriott, K., Meyer, B. & Wittenburg, K.: A Survey of Visual Language Specification and Recognition, *Workshop on Theory of Visual Languages*, 1996.

[MMW96] Marriott, K. & Meyer, B.: Towards a Hierarchy of Visual Languages, *IEEE Symp. on Visual Languages*, 1996.

[Mos90] Mosses, P.D.: Denotational Semantics, in J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Vol. B*, Elsevier, pp. 575-631, 1990.

[RS95] Rekers, J. & Schürr, A.: A Graph Grammar Approach to Graphical Parsing, *IEEE Symp. on Visual Languages*, pp. 195-202, 1995.

[RS96] Rekers, J. & Schürr, A.: A Graph Based Framework for the Implementation of Visual Environments, *IEEE Symp. on Visual Languages*, 1996.

[WL93] Wang, D. & Lee, J.R.: Visual Reasoning: its Formal Semantics and Applications, *Journal of Visual Languages and Computing 4*, pp. 327-356, 1993.