

# Creating Semantic Representations of Diagrams

Mark Minas

Lehrstuhl für Programmiersprachen  
Universität Erlangen-Nürnberg  
Martensstr. 3, 91058 Erlangen, Germany  
`minas@informatik.uni-erlangen.de`

**Abstract.** Diagrams that serve as a visual input facility for programming environments have to be translated into some kind of semantic description. This paper describes such a method which is based on a specification of the translation process. The translation process starts with a diagram, which is simply represented as a collection of atomic diagram components, and it ends up with some data structure as a semantic representation of the diagram. The specification of the translation process mainly consists of two parts: the specification of spatial relationships between atomic diagram components in terms of their numeric parameters (e.g., position, size), and an attributed hypergraph grammar that describes the concrete diagram syntax as well as the rules for generating the semantic representation.

## 1 Introduction

Diagram languages which are used for visual programming are formal languages and are thus defined by their syntax, semantics, and pragmatics. Syntax describes atomic components of the language and the rules how they can be arranged to make up valid sentences. Semantics describe the meaning of diagrams, i.e., the behavior of a computer when such diagrams are “executed”, and pragmatics consist of the context where sentences of this language are used. One issue of pragmatics is to “draw” diagrams using a specific graphical editor and then to translate these “drawings” into a representation which is appropriate for some kind of compiler, interpreter, or virtual machine<sup>1</sup>, e.g., diagrams that represent visual programs are first translated into an equivalent textual program which is then translated by a common compiler into machine code. This task requires a graphical editor that “understands” sentences of the specific language which it is designed for. Otherwise it is merely a drawing tool. “Understanding” means that the editor has to be able to check the drawings’ syntax and to transform (“*translate*”) diagrams into a representation which is required by the compiler.

This paper describes a grammar-based method for such a syntax check and translation process: it starts with a diagram (e.g., created with a graphical editor), that consists of a spatial arrangement of atomic components, and ends

---

<sup>1</sup> For brevity, we will use the term “compiler” in the rest of this paper as a representative of all possible further processing steps.

up with a semantic description, e.g., a program text for a common compiler for textual languages, but it is not restricted to strings. The syntax check and translation process for a concrete diagram language is determined by two specifications:

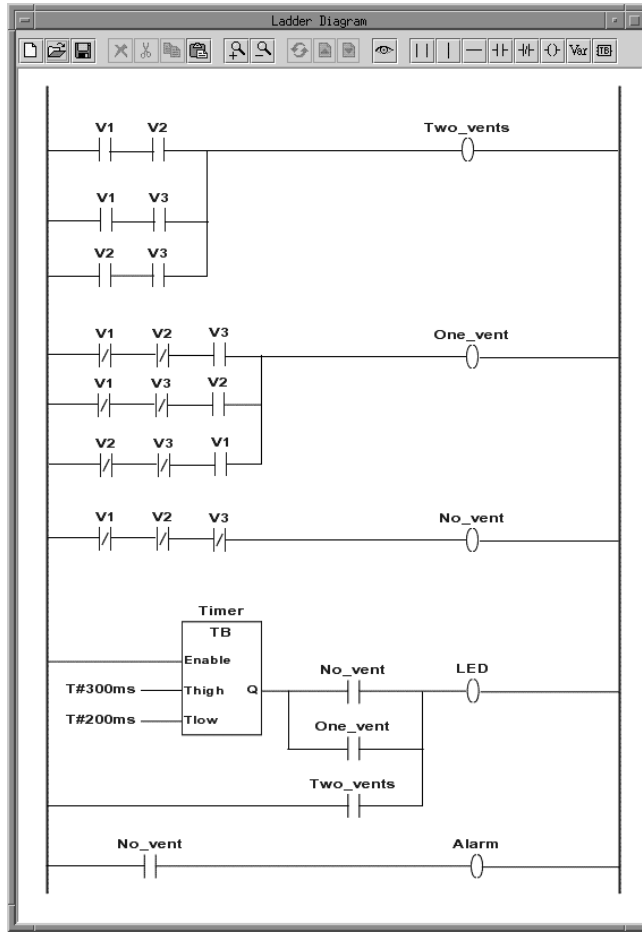
1. The scanning procedure constructs a hypergraph model for the initial diagram. It is controlled by a *spatial relationship specification* which describes meaningful spatial relationships between diagram components.
2. An *attributed hypergraph grammar* specifies the syntax of these hypergraph models and, as a consequence, of the diagram language. The grammar furthermore describes the relationship between hypergraph models and its semantic descriptions. Based on this grammar, a parser checks the diagram syntax and translates the diagram into its semantic representation.

This method is well suited to diagram languages with (hyper) graphs as an appropriate means of diagram representation and (hyper) graph grammars as syntax definition. At least using graphs (and therefore hypergraphs as a generalized form of graphs; see Section 4) as an intermediate representation does not impose a strong restriction on the class of diagram languages which can be processed by this method since graphs can be used as abstract representation for a wide variety of visual languages [7].

The rest of this paper is structured as follows: The next section introduces *Ladder Diagram*, a widely used programming language for Programmable Logic Controllers (PLCs), which is used as running example in this paper. Section 3 summarizes related work, and Section 4 briefly introduces graphs, hypergraphs, and hypergraph grammars. The translation process is described in Section 5. Section 6 gives a brief survey of *DiaGen*, a framework for creating graphical editors, where the approach of this paper is incorporated to generate front-ends for common execution environments. Section 7 concludes.

## 2 Example: Ladder Diagram

Throughout this paper, we will use ladder diagrams as running example. *Ladder Diagram* is a visual programming language for Programmable Logic Controllers (PLCs) which has become part of the IEC 1131 standard [1]. Ladder diagram has been derived from schematic diagrams of relay controls where each relay is energized by a network of switches, either input or relay switches. Relays have been replaced by boolean values in PLCs, and networks of switches have been replaced by boolean expressions. However, ladder diagrams still allow to program a PLC like a relay control: boolean values which are defined by boolean expressions are drawn as relay coils, boolean input values are drawn as switches. The boolean complement of a value is drawn as a normally closed switch. Ladder diagrams allow to build networks that contain switches that are connected in series (boolean and-operation) and in parallel (boolean or-operation).



**Fig. 1.** A sample ladder diagram with an additional function block of type TB.

Figure 1 shows a sample ladder diagram<sup>2</sup> which controls three vents. An LED shall indicate the states of the vents. A lighting LED indicates at least two working vents. The LED blinks if at least two vents fail. An additional alarm is triggered if all three vents fail. Figure 1 shows the boolean values and their controlling boolean expressions. E.g., the top-most sub-diagram shows that **Two\_vents** is defined as the result of a parallel connection where each branch consists of a series connection of two switches. An equivalent boolean expression for this sub-diagram is  $\text{Two\_vents} := (V1 \wedge V2) \vee (V1 \wedge V3) \vee (V2 \wedge V3)$ . The third sub-diagram, that defines **No\_vent**, uses normally closed switches which represent boolean complements of the represented boolean values. Figure 1 actually uses

<sup>2</sup> This example is taken from [16].

CAL	Timer	(Enable:=TRUE, Thigh:=T#300ms, Tlow:=T#200ms)	LDN	V1	LDN	V1
			ANDN	V2	ANDN	V2
			AND	V3	ANDN	V3
LD	V1		OR(	TRUE	ST	No_vent
AND	V2		ANDN	V1	LD	Timer.Q
OR(	V1		ANDN	V3	AND(	No_vent
AND	V3		AND	V2	OR	One_vent
)			)		)	
OR(	V2		OR(	TRUE	OR	Two_vents
AND	V3		ANDN	V2	ST	LED
)			ANDN	V3		
ST	Two_vents		AND	V1	LD	No_vent
			)		ST	Alarm
			ST	One_vent		

**Fig. 2.** The semantics of the ladder diagram of Fig. 1 written as Instruction List.

an extension of ladder diagrams: As allowed in IEC 1131, ladder diagrams can be combined with function blocks which offer some complex functionality as black boxes with a certain number of inputs and outputs. The inputs may be boolean values, numeric values or any other type of value which is supported by the respective PLC. The fourth sub-diagram of our example makes use of a function block `Timer` of type `TB` which implements an oscillator. The inputs `Thigh` and `Tlow` control the up and down time of the oscillation. The `Enable` input allows to switch the oscillator on and off. The ladder diagram specifies that the LED is lighting continuously if `Two_vents` is *true*, or blinking, if `Two_vents` is *false* and `No_vents` or `One_vent` is *true*.

This paper defines the visual syntax of ladder diagrams with embedded function blocks. Boolean values may be combined by and-operations (parallel connection) and or-operations (series connections). Networks are drawn from left to right. They always start at the left vertical border-line or at a function block output. Networks end either at a coil (boolean output) or a function block input.

Ladder diagram semantics are defined by the behavior of the PLC that is programmed by a specific ladder diagram. In this paper, we will use *Instruction List* (IL), a textual, assembler-like language for PLCs which is also part of IEC 1131. The machine model behind IL is a accumulator-machine with additional stack and one-address commands. Possible commands are “LD”, “AND”, and “OR” which take a boolean variable as operand. Modifiers to these commands are “N”, which negates the operand’s value, and “(” which starts a new sub-expression and pushes the old value onto the stack. The command “)” finishes the subexpression and combines its value with the top of the stack. Function blocks are called by the “CAL” command which also specifies the inputs. Function block outputs are referred to by a dotted name like `Timer.Q` in our example. IL furthermore contains many other commands, but these are not relevant for this paper. Figure 2 shows the IL program that describes the semantics of the ladder diagram of Fig. 1.

### 3 Related Work

Many authors have described semantics of visual languages. In most cases (e.g., [9]), however, they restrict to specific visual languages. Others take an algebraic view of modeling picture semantics [25]. Work that is most closely related to this paper is Erwig’s definition of visual language semantics using abstract syntax graphs [7] and the separation of concrete and abstract syntax proposed in [2, 17].

Erwig uses abstract syntax graphs that abstract from representation details of concrete diagrams. He does not restrict semantic definition to this representation, but uses different schemes, e.g., denotational semantics, to define diagram semantics based on abstract syntax. However, he does not offer a method for translating a concrete diagram into its abstract syntax representation.

Rekers et al. have proposed to use spatial relationship graphs (SRGs) to represent a diagram’s concrete syntax and an abstract syntax graph (ASG) for its abstract syntax [2, 17]. The syntax of each of the graphs is represented by a graph grammar. By coupling both grammars, they are able to translate SRGs into ASGs and vice versa. The correspondence between ASG and SRG is represented by special edges connecting ASG nodes by corresponding SRG nodes. The approach which is described in this paper uses hypergraphs and hypergraph grammars to describe concrete syntax and arbitrary data structures for semantic representations. Hypergraphs seem to offer a more natural representation of diagram components that have different “attachment areas” which link to other diagram components (consider connection points of a transistor symbol in schematic diagrams of electric circuits). Moreover there are restricted, yet powerful types of hypergraph grammars that allow for efficient parsing [3, 12] which are not available for plain graph grammars. Arbitrary data structures for semantic representations, e.g., strings, that are used in this paper, have the advantage that they can be customized for common compilers.

VLCC (Visual Language Compiler-Compiler) [6] is a tool whose approach is related to the approach in this paper. VLCC creates parsers for visual languages. Similar as with textual parsers (e.g., generated by `yacc`), semantic actions are used to create semantic representations of visual sentences. VLCC depends on *positional grammars* which basically extend string grammars by 2D-relationships. Its parser is based on LR-parsers used for context-free string grammars which is rather restricted compared to hypergraph parsers which are used in this paper’s approach.

### 4 Hypergraphs and Grammars

Before the translation process is described in the next section, we will briefly introduce the notion of graphs, hypergraphs, and hypergraph grammars as used in this paper.

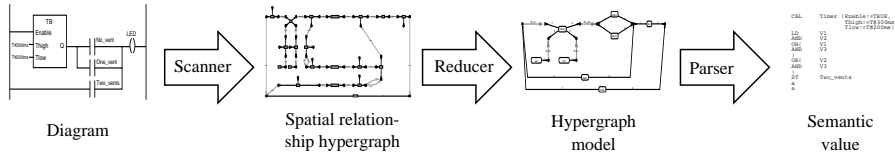
Each (*directed*) *graph* consists of a set of labeled nodes and a set of labeled (directed) edges. Each edge visits two nodes which need not be different. *Hyper-*

*graphs* are generalizations of directed graphs: they have a set of labeled hyperedges instead of edges. Each hyperedge has a fixed number of labeled tentacles which is determined by the hyperedge’s label. Tentacles connect the hyperedge with nodes visited by the hyperedge. A regular directed graph is a hypergraph where each hyperedge has two tentacles with labels *source* and *target*. Nodes will be represented by black dots, (directed) edges by arrows, and hyperedges by boxes containing the hyperedge label. Thin lines or arrows are used to represent tentacles connecting the hyperedge with visited nodes. Tentacle labels are omitted where possible.

Hypergraph grammars are similar to string grammars. Each hypergraph grammar consists of two sets of *terminal* and *nonterminal* hyperedge labels and a *starting hypergraph* which contains nonterminally labeled hyperedges only. Syntax is described by a set of *productions* of the form  $L ::= R$  with  $L$  (left-hand side, LHS) and  $R$  (right-hand side, RHS) being hypergraphs. A production  $L ::= R$  is applied to a (host) hypergraph  $H$  by finding  $L$  as a subgraph of  $H$  and replacing this match by  $R$  obtaining hypergraph  $H'$ . We say,  $H'$  is derived from  $H$  (written  $H \rightarrow H'$ ) in one step. The grammar’s language is then defined by the set of terminally labeled hypergraphs which can be derived from the starting hypergraph in a finite number of steps.

There are different types of hypergraph grammars which impose restrictions on a production’s LHS and RHS as well as the allowed sequence of derivation steps. *Context-free* hypergraph grammars are the simplest ones: each LHS has to consist of a single nonterminally labeled hyperedge together with the appropriate number of nodes. Application of such a production removes the LHS hyperedge and replaces it by the RHS. Matching node labels of LHS and RHS determine how the RHS has to fit in after removing the LHS hyperedge. Productions  $P_1 \dots P_{24}$  of Fig. 7 are context-free ones. Context-free hypergraph grammars *with embeddings* are more expressive than context-free ones. They additionally allow *embedding productions* which consist of the same LHS and RHS, but with an additional (“embedded”) (sub-) hypergraph on the RHS, i.e., this hypergraph is embedded into the context provided by the LHS when applying such a production (production  $P_{25}$  of Fig. 7; the gray edges represent the embedding context). Parsing algorithms and a more detailed description of both grammar types can be found in [12, 3].

In the following, we will use (hyper) graphs as diagram representations (as spatial relationship hypergraph SRHG and hypergraph model HGM). These graphs can be extended by geometric attributes, e.g., representing exact positions in the plane. This additional information is omitted here, but it is clear that using graphs does not impose any loss of information. Using graphs has the advantage (e.g., compared to relational structures) that they explicitly represent items and relationships between items which makes this information readily available. Furthermore, graphs offer a wide variety of graph algorithms for further processing and graph grammars for defining graph classes and their structure. However, using graphs also has the disadvantage that making relationships explicit can lead to rather big representations.



**Fig. 3.** Translating a diagram into a semantic representation.

## 5 The Translation Process

Fig. 3 shows the three steps of the translation process and the resulting hypergraphs with increasing abstraction level. These steps are described in the following.

### 5.1 Scanning

A diagram consists of a set of diagram components (transistor and resistor symbols etc. for schematic diagrams of electronic circuits, switches, coils, function blocks, and lines in ladder diagrams) with spatial relationships between them. In general, each component has a certain number of *attachment areas* which are somehow linked to attachment areas of other components. The way how these areas may be linked depends on the types of related components. For schematic diagrams of electronic circuits, each symbol has its connectors as attachment areas. Actually each connector can be linked to any other connector. Components of ladder diagrams have the following attachment areas: switches and coils have their left and right contacts as their attachment areas. Lines can manifest spatial relationships at their end points as well as at the lines itself; lines have these three attachment areas. Function blocks have as many attachment areas as they have inputs and outputs. Finally, the left and right lines of a ladder diagram are considered as a special chassis component with the lines as its attachment areas. However, only some relationships between different attachment areas make sense. E.g., direct relationships between a function block output and a coil contact does not make sense in ladder diagrams.

A *spatial relationship hypergraph* (SRHG) is used to explicitly represent components and their relationships: Each component together with its attachment areas is represented by a hyperedge and some nodes that are visited by the hyperedge through its tentacles, which thus identify the attachment areas. Spatial relationships are represented by hyperedges (in general regular edges), too. Nodes are connected by such edges if the corresponding attachment areas are appropriately linked.

For ladder diagrams, we have component hyperedges for the chassis consisting of the left and right vertical line (type “chassis”, 2 tentacles “Left” and “Right”) and vertical and horizontal lines (type “vline” and “hline” with 2 tentacles “LineEnd” and 1 tentacle “Line”). Furthermore, we have normally open



$n_1$	relation	$n_2$	constraint
LineEnd	conn	Left	$ y_1 - y_2  < \varepsilon$
LineEnd	conn	Right	$ y_1 - y_2  < \varepsilon$
LineEnd	conn	Contact	$\ \mathbf{p}_1 - \mathbf{p}_2\  < \varepsilon$
LineEnd	input	Input	$\ \mathbf{p}_1 - \mathbf{p}_2\  < \varepsilon$
Output	output	LineEnd	$\ \mathbf{p}_1 - \mathbf{p}_2\  < \varepsilon$
LineEnd	conn	Text	$\ \mathbf{p}_1 - \mathbf{p}_2\  < \varepsilon$
LineEnd	conn	LineEnd	$\ \mathbf{p}_1 - \mathbf{p}_2\  < \varepsilon$
Line	conn	Line	$l_1.\text{intersects}(l_2)$
LineEnd	conn	Line	$l_2.\text{intersects}(\mathbf{p}_1.\text{area}())$

**Table 1.** Spatial relationships for ladder diagrams

1. For each diagram component, create an appropriate hyperedge together with its visited nodes, which are labeled according to the component’s attachment areas.
2. Check for any pair of nodes<sup>3</sup> and any possible relationship type between those nodes whether the nodes’ parameters satisfy the constraints for this relation. If the constraint is satisfied, add a corresponding relationship edge.

Checking each pair of nodes is quite inefficient ( $O(n^2)$  where  $n$  is the number of nodes). Attachment areas which do not intersect in the plane are generally not related. A more efficient solution is to consider the rectangular bounding box of the attachment area of each node and to check only those pairs of nodes with intersecting bounding boxes. The complexity of this search is  $O(n \log n + k)$  where  $k$  is the number of intersections [11].

## 5.2 Reducing

The SRHG which has been produced by the scanning step can now be used for syntax analysis. However, the situation is similar as for compilers for textual languages: the parser does not operate on the stream of characters directly. For efficiency reasons, this stream is preprocessed by the lexical analysis that removes unnecessary characters (e.g., comments) and combines elementary character sequences to larger components (e.g., keywords). The same holds for the SRHG. Many spatial relationship edges are necessary to represent simple concepts. E.g., intersecting and connected lines in ladder diagrams represent the same boolean value of a boolean expression (or electric potential in the term of relay controls). Therefore, it makes sense to reduce all connected lines with their representing

<sup>3</sup> We consider binary relationships in this paper. Since hyperedges are allowed as relationship edges, arbitrary  $n$ -ary relationships could be considered, too.

nodes to a single node. In order to make parsing—the following step—more efficient, we take a *reducing* preprocessing step that creates a *hypergraph model* (HGM) from the SRHG by representing “essential” SRHG-subgraphs by more abstract hyperedges and/or combined (“unified”) nodes. Each SRHG node and hyperedge that is not member of one of these “essential” subgraphs will not become member of the HGM.

The essential diagram situations in ladder diagrams consist of single SRHG hyperedges only. Figure 5 shows these situations together with their SRHG representation and how these situations are represented in the HGM. Only “hline”, “vline”, and “conn” edges of the SRHG are really reduced; all the other edges are simply translated into equivalent HGM ones. For other diagram types (e.g., visual  $\lambda$ -calculus VEX [5]) much more complicated reducing steps have to take into account subgraphs which consist of several SRHG edges and negative application conditions, i.e., context that must not occur when taking a specific reduction action.

Fig. 6 shows the result of reducing the SRHG by the rules depicted in Fig. 5. Please note how much the SRHG (Fig. 4) has been reduced.

### 5.3 Parsing

The HGM which has been produced by first scanning the diagram and then reducing the obtained SRHG describes the diagram’s concrete structure. Syntax analysis of the diagram can thus be performed on the HGM. This step of the translation process checks the HGM according to a specified hypergraph grammar. As usual, syntax checking is performed by parsing, i.e., searching for a derivation sequence from the starting hypergraph to the HGM using grammar productions. For a survey of parsers which may be used in the context of visual languages, see [12, 3].

Additionally to syntax checking, the parser has to create a semantic description in the process of constructing a derivation sequence. The situation is similar to compilers for textual languages where nonterminal symbols and productions are extended by attributes resp. attribute evaluation rules which compute on the attributes when the production is used in the derivation [10]. This idea has already been adopted to graph grammars (e.g., [4, 8]), and we make use of this idea of semantics definition together with syntax description: Each hyperedge may carry attributes, and productions are extended by attribute evaluation rules which compute attribute values when the corresponding production is used in the derivation. The term “*attributed hypergraph grammar*” refers to a hypergraph grammar which has been extended by attributes and attribute evaluation rules.

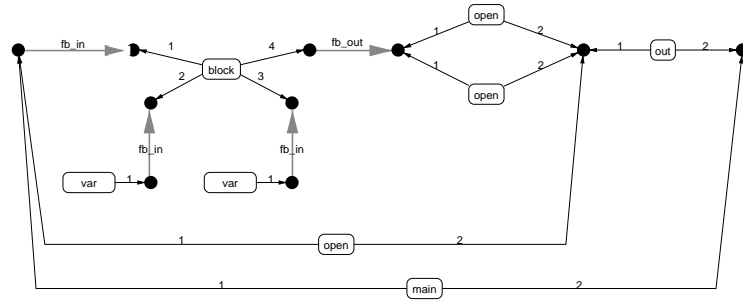
Figure 7 shows the attributed hypergraph grammar for ladder diagrams. Nonterminally labeled hyperedges are depicted by rectangular boxes, terminally labeled ones by oval boxes. Productions are depicted in the abbreviated form  $L ::= R_1 | \dots | R_n$  if productions  $L ::= R_1, \dots, L ::= R_n$  have the same LHS  $L$ . The upper part of Fig. 7 shows the hypergraph grammar only. Node labels a, b, etc. describe how the RHS has to fit into the host hypergraph when the LHS has

	Diagram situations			
Diagram situation				
Spatial relationship hypergraph				
Hypergraph model				
Diagram situation	T# 300ms			
Spatial relationship hypergraph				
Hypergraph model				
Diagram situation				
Spatial relationship hypergraph				
Hypergraph model				

**Fig. 5.** Reduction rules for translating spatial relationship hypergraphs of ladder diagrams into their hypergraph models.

been removed. Figure 7 omits the productions' application conditions that use positional attributes of the affected hyperedges in order to guarantee processing of boolean expressions from top to bottom.

The lower part of Fig. 7 assigns program code as attribute evaluation rules to productions. Each hyperedge carries, depending on its label, an attribute  $\sigma$  which contains an intermediate semantic description of that sub-hypergraph that is represented by the hyperedge,  $\gamma$  for generated IL program code, and 'name', 'in', etc. which are defined by the diagram components itself. For readability, attributes are written in a more "mathematical notation" as  $edge_{index}.\alpha$  where  $edge$  is the label of a hyperedge,  $index$  the index of the hyperedge (0 means the LHS hyperedge, 1 and 2 RHS hyperedges), and  $\alpha$  the attribute of this hyperedge. Functions **and**, **or**, etc. have straight-forward implementations which are omitted in this paper.



**Fig. 6.** The hypergraph model of the ladder sub-diagram of Fig. 1 that defines LED.

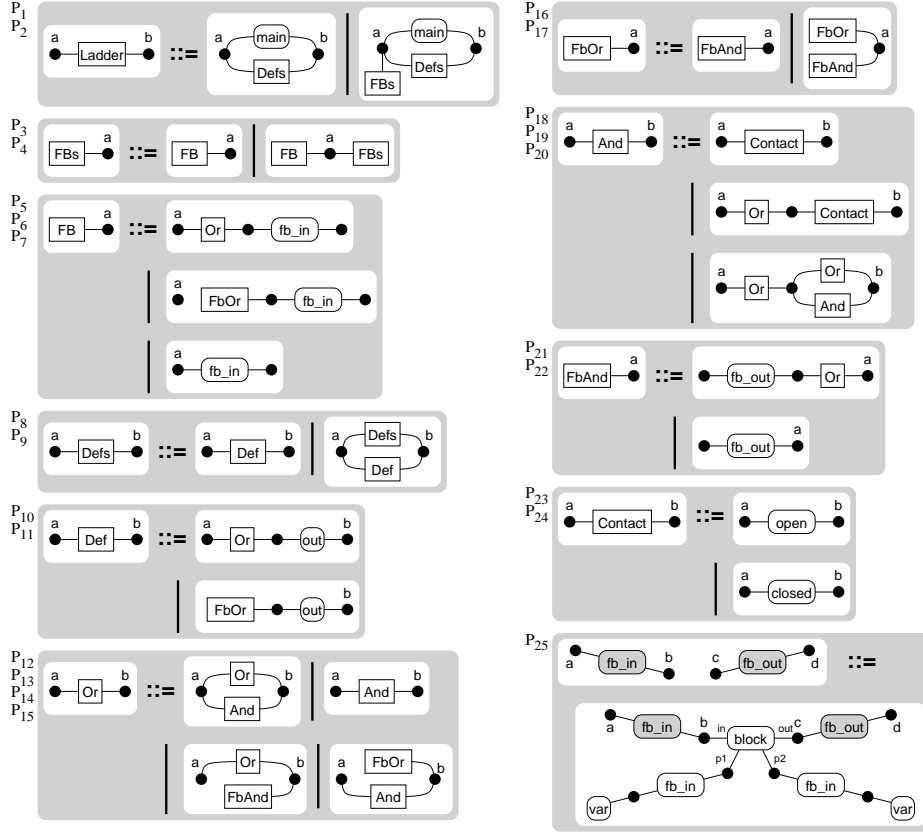
The grammar is a context-free hypergraph grammar with embeddings. Productions  $P_1 \dots P_{24}$  are context-free productions,  $P_{25}$  is an embedding production: A *block*-edge is added between corresponding *fb\_in* and *fb\_out* edges. A plain context-free hypergraph grammar without embedding production would have been sufficient for the restricted ladder diagram language of this paper with only this simple function block type. However, function blocks with more than one output cannot be described by a context-free grammar. The grammar of Fig. 7 is easily extended for such function blocks with new productions similar to  $P_{25}$ .

The translation step from a HGM to its semantic description is performed as follows: The hypergraph parser (see [12, 3]) searches for a derivation of the HGM from the starting hypergraph using the HGM grammar. Attributes and semantic actions are neglected in this step. The derivation consists of a sequence of HGM productions which uniquely induces functional dependencies among the attributes of hyperedges that occur in the derivation. As for attributed string grammars, these dependencies have to be non-circular, i.e., there has to be a total ordering on all instances of dependencies such that each attribute can be computed from known values determined by earlier dependencies. This is the case in the grammar of Fig. 7.

## 6 *DiaGen*

This translation process is used in *DiaGen* for creating visual programming front-ends for further processing steps, e.g., for compilers from PLC Instruction List to machine code of specific PLCs.

*DiaGen* consists of an editor framework and a generator. A formal specification of a diagram language serves as input for the generator which creates custom components that build—together with the framework—a graphical editor customized for the specified diagram language. Main features, which have been described in [12, 14, 19, 13], are:



- $P_1 : \text{Ladder}_0.\gamma = \text{Defs}_1.\gamma$   
 $P_2 : \text{Ladder}_0.\gamma = \text{FBs}_1.\gamma \cdot \text{Defs}_1.\gamma$   
 $P_3 : \text{FBs}_0.\gamma = \text{FB}_1.\gamma$   
 $P_4 : \text{FBs}_0.\gamma = \text{FBs}_1.\gamma \cdot \text{FB}_1.\gamma$   
 $P_5 : \text{fb\_in}_1.\sigma = \text{Or}_1.\sigma;$   
 $\text{FB}_0.\gamma = \text{fb\_in}_1.\gamma$   
 $P_6 : \text{fb\_in}_1.\sigma = \text{FbOr}_1.\sigma;$   
 $\text{FB}_0.\gamma = \text{fb\_in}_1.\gamma$   
 $P_7 : \text{fb\_in}_1.\sigma = \text{alwaysTrue}();$   
 $\text{FB}_0.\gamma = \text{fb\_in}_1.\gamma$   
 $P_8 : \text{Defs}_0.\gamma = \text{Def}_1.\gamma$   
 $P_9 : \text{Defs}_0.\gamma = \text{Defs}_1.\gamma \cdot \text{Def}_1.\gamma$   
 $P_{10} : \text{Def}_0.\gamma = \text{output}(\text{Or}_1.\sigma, \text{out}_1.\text{name})$   
 $P_{11} : \text{Def}_0.\gamma = \text{output}(\text{FbOr}_1.\sigma, \text{out}_1.\text{name})$   
 $P_{12} : \text{Or}_0.\sigma = \text{or}(\text{Or}_1.\sigma, \text{And}_1.\sigma)$   
 $P_{13} : \text{Or}_0.\sigma = \text{And}_1.\sigma$   
 $P_{14} : \text{Or}_0.\sigma = \text{or}(\text{Or}_1.\sigma, \text{FbAnd}_1.\sigma)$   
 $P_{15} : \text{Or}_0.\sigma = \text{or}(\text{FbOr}_1.\sigma, \text{And}_1.\sigma)$   
 $P_{16} : \text{FbOr}_0.\sigma = \text{FbAnd}_1.\sigma$   
 $P_{17} : \text{FbOr}_0.\sigma = \text{or}(\text{FbOr}_1.\sigma, \text{FbAnd}_1.\sigma)$   
 $P_{18} : \text{And}_0.\sigma = \text{Contact}_1.\sigma$   
 $P_{19} : \text{And}_0.\sigma = \text{and}(\text{Or}_1.\sigma, \text{Concat}_1.\sigma)$   
 $P_{20} : \text{And}_0.\sigma = \text{and}(\text{Or}_1.\sigma,$   
 $\text{or}(\text{Or}_2.\sigma, \text{And}_1.\sigma))$   
 $P_{21} : \text{FbAnd}_0.\sigma = \text{and}(\text{fb\_out}_1.\sigma, \text{Or}_1.\sigma)$   
 $P_{22} : \text{FbAnd}_0.\sigma = \text{fb\_out}_1.\sigma$   
 $P_{23} : \text{Concat}_0.\sigma = \text{open}(\text{open}_1.\text{name})$   
 $P_{24} : \text{Concat}_0.\sigma = \text{closed}(\text{closed}_1.\text{name})$   
 $P_{25} : \text{fb\_in}_0.\gamma = \text{fbCall}(\text{block}_1.\text{name}, \text{block}_1.\text{in}, \text{fb\_in}_0.\sigma,$   
 $\text{block}_1.\text{p1}, \text{var}_1.\text{value}, \text{block}_1.\text{p2}, \text{var}_2.\text{value});$   
 $\text{fb\_out}_0.\sigma = \text{block}_1.\text{name} . \text{block}_1.\text{out}$

**Fig. 7.** Attributed hypergraph grammar translating hypergraph models of ladder diagrams into their semantic representation.

- Diagrams are internally represented by hypergraphs; a diagram language is thus a hypergraph language together with a mapping from hypergraphs to their visual representation as diagrams.
- Nodes and hyperedges carry attributes, and each grammar production is augmented by layout constraints on attributes accessible in the production. A constraint-solver provides automatic, user-adjustable layout of diagrams [15].
- Diagrams can be edited in a syntax-directed manner. For the diagrams' context-free share, transformations on derivation trees are used. Further transformations may modify the diagrams' hypergraphs directly. To hide those details from the user, interactions of the user and the editor are described by certain interaction automata.
- Free-hand editing is also supported. The user can arbitrarily add, delete, move, or modify parts of the diagram. The underlying hypergraph model is modified accordingly, a hypergraph parser distinguishes correct diagrams from incorrect ones by keeping the underlying hypergraph's syntactic meta-structure up-to-date. Free-hand editing with parser support relaxes the need to specify a full set of transformations on diagrams for syntax-directed editing since free-hand editing can be used for (yet) unspecified diagram operations. Therefore, this editing mode enhances usability of editors and also makes rapid prototyping of diagram editors possible because—as an extreme case—specification of diagram operations can be omitted completely.

The translation approach which has been described in this paper allows free-hand editing of diagrams which are then translated into its semantic description. *DiaGen* has been used to generate a ladder diagram editor from the specification which has been outlined in this paper. Figure 1 shows a screenshot of this editor, Fig. 2 the equivalent IL program that has been created as a semantic description for the depicted ladder diagram. This description could be further processed, e.g., in a compiler that compiles the IL program into the machine code for a specific PLC. IL would then be the intermediate language in the processing of a ladder diagram into PLC machine code; the diagram editor with its semantic translation process acts as a compiler front-end, the translator from IL into machine code as the compiler back-end.

## 7 Conclusions and Future Work

This paper has presented a grammar based method for translating diagrams into a semantic description which then can be interpreted by a common interpreter. Diagrams that are translated by this method have to be represented as a collection of atomic diagram components with appropriate numeric parameters representing their size, position, etc. in the plane. This method makes use of a specification of meaningful spatial relationships between diagram components, how diagrams are represented by hypergraphs, and an attributed hypergraph which specifies the diagram syntax as well as the way how the semantic description is created. The concepts which have been described in the paper have been demonstrated for ladder diagrams, a widely used visual programming language

for Programmable Logic Controllers (PLCs). Ladder diagrams are translated into their corresponding Instruction List programs, a textual programming language for PLCs.

The method that has been described on the previous pages is based on representation of diagrams by hypergraphs, which are a generalization of graphs. (Hyper) graphs appear to be an appropriate way to represent diagrams on different levels of abstraction. Furthermore, hypergraph grammars provide a powerful tool for describing diagram syntax as well as the translation process from the diagram into its abstract representation.

This is not finished work. So far, it has been used “one-way” for translating diagrams into a representation which can be further processed by an execution environment (e.g., a PLC runtime system). This might be sufficient for this example. For diagrams that are translated into a semantic representation, which is then interpreted and creates results, that have to be translated back into the diagram language, the unparsing problem has to be solved. This unparsing problem requires parsing of the interpreter results and creating a diagram in correspondence with the derivation of the interpreter result. *Triple graph grammars* [18] appear to be a starting point for solving this problem.

## References

1. Deutsche Norm DIN EN 61131 Teil 3 “Speicherprogrammierbare Steuerungen – Programmiersprachen”. Beuth Verlag, Berlin, 1994. in German.
2. M. Andries, G. Engels, and J. Rekers. How to represent a visual specification. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 245–260. Springer Verlag, 1998.
3. R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of graph transformation to visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume II: Applications, Languages and Tools, pages 105–180. World Scientific, 1999.
4. H. Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE pattern analysis and machine intelligence*, 4(6):574–582, 1982.
5. W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *VL’95 [22]*, pages 294–301, 1995.
6. G. Costagliola, G. Tortora, S. Orefice, and A. D. Lucia. Automatic generation of visual programming environments. *IEEE Computer*, 28(3):56–66, Mar. 1995.
7. M. Erwig. Semantics of visual languages. In *VL’97 [24]*, pages 304–311, 1997.
8. H. Göttler. Attributed graph grammars for graphics. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 130–142, 1983.
9. V. Haarslev. Formal semantics of visual languages using spatial reasoning. In *VL’95 [22]*, pages 156–163, 1995.
10. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Errata 5:1 (1971) 95–96.
11. K. Mehlhorn. *Data Structures and Algorithms 3, Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.

12. M. Minas. Diagram editing with hypergraph parser support. In *VL'97 [24]*, pages 230–237, 1997.
13. M. Minas. Hypergraphs as a uniform diagram representation model. In *Preliminary Proc. 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98), Paderborn, Germany*, pages 24–31. University of Paderborn, Technical Report tr-ri-98-201, Nov. 1998.
14. M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *VL'95 [22]*, pages 203–210.
15. M. Minas and G. Viehstaedt. Specification of diagram editors providing layout adjustment with minimal change. In *VL'93 [20]*, pages 324–329.
16. P. Neumann, E. E. Grötsch, C. Lubkoll, and R. Simon. *SPS-Standard: IEC 1131*. Oldenbourg, 1995. in German.
17. J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *VL'96 [23]*, pages 148–155, 1996.
18. A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, number 904 in Lecture Notes in Computer Science, pages 151–163, Berlin, 1994. Springer Verlag.
19. G. Viehstaedt and M. Minas. Interaction in really graphical user interfaces. In *VL'94 [21]*, pages 270–277.
20. *1993 IEEE Symp. on Visual Languages, Bergen, Norway*. IEEE Computer Society Press, Aug. 1993.
21. *1994 IEEE Symp. on Visual Languages, St. Louis, Missouri*. IEEE Computer Society Press, Oct. 1994.
22. *1995 IEEE Symp. on Visual Languages (VL'95), Darmstadt, Germany*. IEEE Computer Society Press, Sept. 1995.
23. *1996 IEEE Symp. on Visual Languages (VL'96), Boulder, Colorado*. IEEE Computer Society Press, Sept. 1996.
24. *1997 IEEE Symp. on Visual Languages (VL'97), Capri, Italy*. IEEE Computer Society Press, Sept. 1997.
25. D. Wang and H. Zeevat. A syntax directed approach to picture semantics. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 307–324. Springer Verlag, 1998.