Abstract

Computer programs have to “understand” diagrams if diagrams are going to be used for communications with or through the computer, i.e., the computer has to be supplied with a method that translates a diagram into an abstract internal representation for further processing. This paper describes such a method that 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 a hypergraph as a semantic or abstract syntax 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 a translation grammar that describes the concrete diagram syntax as well as the rules for generating the semantic or abstract syntax representation.

1. Introduction

Diagram languages 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 these sentences, and pragmatics consist of the context where sentences of this language are used. One issue of pragmatics is to communicate with or with the help of computers through sentences of such a language. This requires graphical editors in the case of diagram languages, which have to have well defined syntax and semantics. Moreover, graphical editors have to “understand” sentences of the specific language which they are designed for. Otherwise they are merely a drawing tools. “Understanding” means that the editor has to be able to transform (“translate”) diagrams either directly into a semantic representation or into an abstract representation that allows an easy interpretation (i.e., mapping into semantics). The latter representation typically uses some kind of abstract syntax which describes a diagram’s structure omitting details concerned with the diagram’s concrete appearance. Since in most cases there is no clear distinction between a semantic and an abstract syntax representation, we will use the term “abstract syntax” for both representations throughout this paper.

This paper describes a grammar-based method for such a 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 up with a hypergraph which serves as an abstract syntax representation (“abstract syntax graph”, ASG, in the following) of the diagram. The 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) A translation 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 or abstract syntax representation. Based on this grammar, a parser checks the diagram syntax and translates the diagram into its ASG.

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 a representation of abstract syntax 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 syntax representation for a wide variety of visual languages [5].

The rest of this paper is structured as follows: The next section introduces the visual λ-calculus language VEX, which is used as running example in this paper. Section 3 summarizes related work, and Section 4 briefly introduces

1Semantics are a mapping of yet undefined structures to known structures. Therefore, semantics depend on the user’s viewpoint, i.e., which is considered to be known.
into graphs, hypergraphs, and hypergraph grammars. The translation process is described in Section 5. Section 6 concludes.

2. Example: VEX

Throughout this paper, we will use VEX [4] as running example. VEX is a language for visually representing λ-calculus expressions: in VEX each variable identifier is represented by an empty circle that is connected by a line to a so-called root node. A root node is again an empty circle with one or more lines touching it, leading to all identifiers that represent the same variable. A root node may either be internally tangential to another circle, it then represents a parameter of a λ-abstraction, or it is not included by any other circle, it then denotes a free variable. A circle which represents a λ-abstraction contains its parameter circle and a VEX diagram as its body. An application of two expressions is depicted by two externally tangential circles with an arrow at the tangent point. The head of the arrow lies inside the argument circle. Fig. 1 shows a VEX diagram for \( \lambda y.((\lambda z.yz)x) \). This diagram will be used as a demonstration example throughout the paper. The numbers in Fig. 1 are then used for referencing.

VEX’s semantics are best represented by λ-calculus expressions. We will use term graphs as ASGs which are equivalent to λ-calculus expressions and which are frequently used in compilers and interpreters for functional languages [12]: Term graphs are essentially term trees where common subexpressions are shared. Nodes in term graphs may be labeled by \( \lambda \), \( @ \), and \( \bigcirc \). \( \lambda \) describes functional abstraction. An outgoing body-edge points to the abstraction’s body, a par-edge to its parameter variable. @-nodes represent function application. A fun-edge points to the function that is applied to the argument term which is determined by an arg-edge. \( \bigcirc \) finally represents variables. Fig. 2 shows the term graph of the VEX diagram of Fig. 1.

3. Related Work

Many authors have described semantics of visual languages. In most cases (e.g., [6]), however, they restrict to specific visual languages. Others take an algebraic view of modeling picture semantics [18]. Work that is most closely related to this paper is Erwig’s definition of visual language semantics using abstract syntax graphs [5] and the separation of concrete and abstract syntax proposed in [1, 13]. 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 [1, 13]. 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. These coupled grammars actually have inspired this work on the definition of translation grammars described in Section 5. However, they use plain graph grammars where we use hypergraph grammars. 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 [2, 10] which are not available for plain graph grammars.

4. Hypergraphs and Grammars

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

Each graph consists of a set of labeled nodes and a set of labeled edges. Each edge visits two nodes which need not be different. In this paper we will use mixed graphs containing directed edges (connecting a source and a target node) and undirected edges (without priority for either of the visited nodes). \(^3\) Hypergraphs 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 and undirected edges by arrows resp. lines, and hyperedges by boxes containing the hyperedge label. Dotted lines 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 \(P_1 \ldots P_{14}\) 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”) hyperedge on the RHS, i.e., this hyperedge is embedded into the context provided by the LHS when applying such a production (productions \(P_15\) and \(P_16\) of Fig. 7). Parsing algorithms and a more detailed description of both grammar types can be found in [10, 2].

Application of productions can be restricted by requiring the existence or non-existence of certain context. Path expressions [3] can be used to require that a certain path of (hyper) edges exists in the host hypergraph. Negative application context [7, 11], on the other hand, is an additional subgraph of the LHS. The production must not be applied if this subgraph can be matched into the host hypergraph, too. Examples for both kinds application context are described together with Fig. 7.

In the following, we will use (hyper) graphs as diagram representations (as spatial relationship hypergraph SRHG, hypergraph model HGM, and ASG). These graphs can be extended by 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 a rather big representations.

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, circles, lines, and arrows in VEX 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 VEX diagrams have the following attachment areas: circles have their complete area as a single attachment area. Lines and arrows can manifest spatial relationships only at their end points, which have to be their attachment areas. However, only some relationships between different attachment areas make sense. E.g., relationships between two line end points do not make sense in VEX.

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 attach-
Spatial relationship hypergraph

Figure 3. The translation process

Figure 4. The spatial relationship hypergraph of the VEX diagram of Fig. 1.

Relationships between attachment areas are constrained by conditions on parameters of the attachment areas. Table 1 shows these constraints for VEX: the table assigns a constraint to each pair \( n_1, n_2 \) of nodes and each relation between the corresponding attachment areas. The constraint imposes a condition on the attachment areas’ parameters. Accessible parameters are \( p_1 \) and \( p_2 \) for positions of \( n_1 \) resp. \( n_2 \) (circle center resp. positions of line or arrow end point) and radius \( r_1 \) and \( r_2 \) of \( n_1 \) resp. \( n_2 \) if \( n_1 \) resp. \( n_2 \) is a circle.

The scanning procedure essentially works as follows:

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\(^4\) 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 [9].

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., function application is expressed in VEX by 2 “circle” hyperedges, 1 “arrow” hy-

\(^4\)We consider binary relationships in this paper. Since hyperedges are considered, too.
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 step 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 [10, 2].

Additionally to syntax checking, the parser creates the diagram’s ASG 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. semantic actions which compute on the attributes when the production is used in the derivation [8]. We adopt this idea to the situation described in this paper: Instead of attributes we introduce additional tentacles to hyperedges that connect HGM hyperedges (and those in the HGM derivation) to nodes of the ASG. ASGs are created by a hypergraph grammar by their own. Its productions are associated with productions of the HGM grammar and thus play the role of semantic actions in attributed string grammars: Whenever a production is applied in the derivation of the HGM, the corresponding production for the ASG grammar has to be applied. The term "translation grammar" refers to the combination of HGM grammar and ASG grammar which is a hypergraph grammar of its own. Each derivation in the translation grammar creates a valid HGM together with its ASG. This scheme of connected graph grammars is related to the one described in [13, 1] (see Section 3.)

Fig. 7 shows the translation grammar for VEX with term graphs as ASGs. Nonterminally labeled hyperedges are depicted by rectangular boxes, terminally labeled ones by oval boxes. Productions are depicted in the abbreviated form $L ::= R_1 | \cdots | R_n$ if productions $L ::= R_1, \ldots, L ::= R_n$ have the same LHS $L$. LHS and RHS of each production are divided into an upper and a lower part. The upper one represents the HGM share whereas the lower part represents the ASG share of the hypergraph. Node labels $a$, $b$, and $c$

<table>
<thead>
<tr>
<th>$n_1$ relation $n_2$ constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineEnd touch circle $r_2 - \varepsilon \leq |p_1 - p_2| \leq r_2 + \varepsilon$</td>
</tr>
<tr>
<td>arrowEnd inside circle $|p_1 - p_2| &lt; r_2 - \varepsilon$</td>
</tr>
<tr>
<td>circle touch circle $r_1 + r_2 - \varepsilon \leq |p_1 - p_2| \leq r_1 + r_2 + \varepsilon$</td>
</tr>
<tr>
<td>circle iTouch circle $r_2 - r_1 - \varepsilon \leq |p_1 - p_2| \leq r_2 - r_1 + \varepsilon$</td>
</tr>
<tr>
<td>circle inside circle $|p_1 - p_2| &lt; r_2 - r_1 - \varepsilon$</td>
</tr>
</tbody>
</table>

Table 1. Spatial relationships for VEX

Figure 6. The hypergraph model of the VEX diagram of Fig. 1.
Figure 5. Reduction rules for translating spatial relationship hypergraphs of VEX diagrams into their hypergraph models.

describe how the RHS has to fit into the host hypergraph when the LHS has been removed. ASG nodes are either λ, @, ⊙ as used in term graphs (see Section 2), or ξ which represents an unspecified ASG node type that matches with λ, @, and ⊙. Fig. 8 shows parts of the derivation of the term graph (Fig. 2) of the VEX diagram of Fig. 1.

The grammar is a context-free hypergraph grammar with embeddings. Productions $P_1 \ldots P_{14}$ are context-free productions, $P_{15}$ and $P_{16}$ are embedding productions: A def-edge is added to connect nodes of a bound variable and a parameter resp. free variable. Consider the path expressions as additional application conditions. They are regular expressions on edge labels describing paths in the HGM. The path expression $a [\text{def}]$ is satisfied for all paths starting at $a$ and following a def-edge, i.e., $P_{15}$ must not be applied if there is already a def-edge starting at node $a$ which means that the corresponding variable is already bound. The path expression $a [\text{body'}] [\text{par}] b$ describes all paths starting at node $a$ and ending at $b$. The last edge of the path has to be a par-edge, the other have to be body-edges. However, the path has to follow body-edge in the reversed direction, expressed by body'. The existence of such a path means that the corresponding bound variable lies in the abstraction circle of the corresponding parameter, i.e., in the scope of this parameter. Please note that using non-context-free productions to describe variable bindings is again similar to the situation for compilers for textual languages: checking variables against their declarations cannot be described by context-free string grammars.

The translation step from a HGM to its ASG is performed as follows: The hypergraph parser (see [10, 2]) searches for a derivation of the HGM from the starting hypergraph using the HGM grammar. Tentacles connecting HGM hyperedges with ASG nodes are neglected in this step. The derivation consists of a sequence of HGM productions which uniquely induces a sequence of ASG productions, i.e., a derivation of the ASG using the translation grammar’s ASG share. The tentacles that connect HGM hyperedges with ASG nodes and which have been neglected in the previous step are now used to keep HGM and ASG connected in this derivation.

The task of creating an HGM’s ASG by first parsing with the HGM grammar and then generating with the ASG grammar is simple as long as both steps can be performed independently. This is the case for the VEX translation grammar (Fig. 7): An ASG always can be generated if a HGM derivation of the HGM has been found. This is so because the ASG productions have discrete LHSs, i.e., without any edges. Further edges in the LHSs might prevent an ASG production from being applied even if the HGM derivation requires its application. A more complicated translation process is necessary which interweaves deriving with the HGM grammar and generating with the ASG grammar. It is not yet clear whether such ASG grammars are really needed for the purpose of translation. However, this still is work in progress. Triple graph grammars [14] are a starting point for a solution of those problems.

6. Conclusions and Future Work

This paper has presented a grammar based method for translating diagrams into a hypergraph which serves as an abstract representation of the diagram. 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
Figure 7. Translation grammar translating hypergraph models of VEX diagrams into their corresponding term graphs.

the plane. Typical abstract representations which are the result of the translation process are semantic descriptions of translated diagrams or some kinds of abstract syntax representations which are then easily transformed into corresponding semantic descriptions. This method makes use of a specification of meaningful spatial relationships between diagram components, how diagrams are represented by hypergraphs, and a translation grammar which specifies the diagram syntax as well as the way how the final translation result is created. The concepts which have been described in the paper have been demonstrated for VEX, a visual λ-calculus language. VEX diagrams are translated into their
corresponding term graphs.

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, i.e., at the concrete level of diagram components that are spatially related, and at the abstract level of (hyper)graphs which represent the semantic structure of the diagram only. Furthermore, hypergraph grammars provide a powerful tool for describing diagram syntax as well as the translation process from the diagram into its abstract representation.

As already pointed out, this is not finished work. Open questions concern the structure of translation grammars where the grammar share that is responsible for generating the abstract representation imposes additional application conditions on the grammar share that describes the diagram syntax. Furthermore it is conceivable to reverse the process of parsing and generating, i.e., starting at an abstract representation and generating a corresponding diagram. This unparsing problem leads to further questions on the structure of translation grammars.

References