Generating Diagram Editors with *DiaGen*

Mark Minas and Oliver Köth

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

Abstract. *DiaGen* is a specification method, which is primarily based on a hypergraph grammar, and a tool that allows to automatically generate diagram editors from such a specification. Generated editors are free-hand editors, but with an automatic, constraint-based layout for correct diagrams. A hypergraph parser checks diagram correctness and makes it possible to translate diagrams into some user-defined semantic representation. This paper briefly outlines *DiaGen* and the process of creating diagram editors with *DiaGen*.

1 Introduction

Today many systems communicate information through diagrams and therefore have to contain diagram editors, i.e., graphical editors that are tailored to the corresponding class of diagrams. Examples are current UML tools, which offer editors for class diagrams, sequence diagrams and others [8], or visual programming tools for programmable logic controllers, which allow to edit ladder diagrams and sequential function charts [3]. When implementing such a diagram editor, two main problems have to be tackled: First, the diagram language must be exactly specified. The specification has to describe its syntax and its semantics, i.e., the rules how to build correct diagrams and the meaning of diagrams. The voluminous documentation on UML [8] shows that this is a nontrivial task. The second problem consists of implementing an editor which conforms to this specification. Such an editor supports either syntax-directed editing or free-hand editing. Syntax-directed editors offer a restricted set of editing operations which can be used to create and edit diagrams. On the other hand, free-hand editors provide no specific editing operations but allow to modify diagrams in any way and may thus produce incorrect diagrams. A parser is responsible for checking the diagrams' syntax and for distinguishing correct from incorrect ones.

The *Dia*gram editor *Gen*erator *DiaGen* is a tool that offers support for the two problems which have been explained in the previous paragraph: *DiaGen* primarily uses attributed hypergraph grammars as a powerful means to specify diagram languages. Main parts of the editor are then automatically generated from this specification. In its current state, the tool creates editors with the following main features:

– *DiaGen* currently generates only free-hand editors.[1] A hypergraph parser checks the correctness of diagrams, creates their syntactic structure, and controls the semantic analysis.
– Generated editors provide an automatic layout based on constraint solving. Currently, *DiaGen* uses as constraint solver either QOCA [6] or PARCON [4].
– *DiaGen*'s generator as well as generated editors run under JAVA 2 and are thus portable to many computer platforms.
– The current *DiaGen* version generates a stand-alone editor. Manual modifications of this editor are necessary if such an editor has to be part of another system. Work in progress aims at solving this problem by generating the editor as a software component which conforms to the JAVABEANS standard [5] and can then be used with RAD tools.

The concept of the generated editors and how they perform syntactic and semantic analysis are described in [7] within this volume. This paper presents *DiaGen* as a tool which is based on these concepts. Rooted trees as shown in Fig. 1a are used as a running example. We have chosen this very simple example because its complete specification fits on a single page (see Fig. 2). But we do not want to give the impression that *DiaGen* can only be applied to trivial examples. Real-world examples, which have been generated with *DiaGen*, are ladder diagrams (the running example of [7]) and sequential function charts, which are used to program programmable logic controllers.

The next section briefly outlines how free-hand editors of *DiaGen* perform syntactic and semantic analysis of diagrams. Section 3 then shows the process of creating a diagram editor with *DiaGen* and explains the specification for rooted trees. Behavior and features of generated editors are discussed in Sec. 4. Section 5 concludes the paper.

2 Diagram Analysis

Free-hand editors, which are part of a larger system, have to translate a diagram into some semantic representation. This section outlines the translation process as it is used by diagram editors which are generated with *DiaGen* and as it is described in more detail in [7]. It consists of four steps which are performed after each modification of the diagram: scanning, reducing, parsing, and semantic analysis. A *DiaGen* specification describes these steps for the specified diagram language. Section 3 discusses the specification for rooted trees.

1. *Scanning step*
   Diagram components (e.g., circles and arrows in trees) are represented by hyperedges. Nodes represent the diagram components' *attachment areas*, i.e., the parts of the components that are allowed to connect to other components (e.g., the end points of an arrow). These nodes and hyperedges make up an

[1] However, syntax-directed editing as an additional mode has been designed and is currently being implemented.

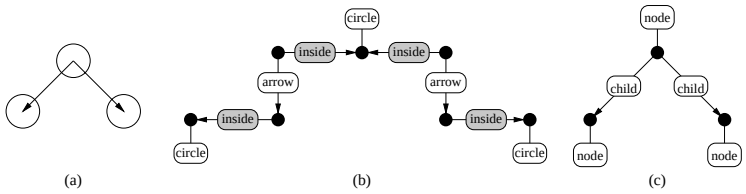

Fig. 1. Representations of a rooted tree in an editor which has been generated with *DiaGen*: diagram (a), spatial relationship hypergraph (b), and reduced hypergraph model (c).

unconnected hypergraph. The *scanner* connects nodes by additional edges if the corresponding attachment areas are related in a specified way, which is described in the specification. The result of this scanning step is the *spatial relationship hypergraph* (SRHG) of the diagram. Figure 1b shows the SRHG of the rooted tree in Fig. 1a. Nodes are represented by black dots, hyperedges by ovals that are connected to their nodes. The inside-relation between an arrow's end point and a circle's area holds if the arrow starts resp. ends within the circle.

2. *Reducing step*
   SRHGs tend to be quite large even for small diagrams (see Fig. 1b). In order to allow for efficient parsing, the SRHG is reduced first. This step is similar to the lexical analysis done by compilers. The result of this *reducing* step is the actual *hypergraph model* (HGM) of the diagram. The reducer is specified by some transformations that identify sub-hypergraphs of the SRHG and build the HGM. Figure 1c shows the HGM of the rooted tree in Fig. 1a.
3. *Parsing step*
   The syntax of the hypergraph models of the diagram language—and thus the main part of the diagram language's syntax—is defined by a hypergraph grammar. *DiaGen* supports context-free hypergraph grammars with embeddings, i.e., productions are either context-free ones with a single nonterminal hyperedge on the production's left-hand side (LHS), or they are embeddings where the production's right-hand side (RHS) is the same as its LHS, but with an additional hyperedge. The second type of productions is not necessary for trees (see Sec. 3).
4. *Semantic analysis step*
   The diagram's syntax structure, which has been identified by the parser, is used to create the diagram's semantic representation. The hypergraph grammar which is used for parsing is actually an attributed one. Similar to textual grammars, semantics are represented as attribute values of terminal and nonterminal symbols (hyperedges); semantic analysis is specified by attribute evaluation rules, which are assigned to grammar productions.

Layout is specified in a very similar way: the diagram's syntactic structure is used to set up a constraint system on the diagram components' parameters.

Diagrams which have been recognized as correct therefore try to preserve their syntax and semantics even when modified by the user.

3 Specifying diagram editors

The process of creating a graphical editor for a specific diagram language with *DiaGen* consists of three steps:

1. *Specify the diagram language*
   The diagram language is defined in the *DiaGen* specification language. The specification has to contain complete descriptions of the reducer, the parser, and the layout constraints (however, constraints are optional). The different diagram components, the scanner, and the semantic analysis are specified partially. In the following, this is demonstrated by means of a specification of rooted trees.
   The specification language is currently a textual language. As soon as this language will have settled, it is planned to build a diagrammatic one on top. *DiaGen* can then be used to generate a visual specification tool based on that language.
2. *Run the* DiaGen *generator*
   The *DiaGen* generator reads the specification and creates some JAVA classes, which are ready-to-use and which implement the reducer, the parser, the core for semantic analysis, and the optional constraint solver for automatic layout. Furthermore, some JAVA class skeletons are generated which have to be fleshed out by the user. The skeletons ensure that these classes conform to the *DiaGen* API and runtime system.
3. *Flesh out class skeletons and add additional* JAVA *classes*
   These classes implement the different diagram components of the diagram language (the user, e.g., has to add code for the visual appearance of the components), the detection of relationships between such components, and those parts of the semantic analysis which create user-defined data structures. The *DiaGen* API provides easily customizable standard classes to simplify this process.

The rest of this section describes the specification of rooted trees which is shown in Fig. 2. Numbers are line numbers of this specification.

A specification starts with a declaration part (1–11) and then contains a specification of the reducer rules (13–28) and grammar productions (30–54). The specification declares the JAVA package (1) which will contain the generated classes and class skeletons. Declarations of diagram components (3–4) and relationships between components (5) specify the corresponding hyperedge types along with the number of tentacles in brackets. The specification also contains the names of generated class skeletons for those components in curly braces. Additionally, the types of attachment areas for each component have to be specified. An arrow, e.g., has two attachment areas of type `ArrowEnd` and is implemented in a class `Arrow`. The generated class will contain this information. Code that

```
1  package editor.tree;
2
3  component   circle [1]     { Circle[CircleArea] },
4              arrow[2]       { Arrow[ArrowEnd,ArrowEnd] };
5  relation    inside [2]     { ArrowInside[ArrowEnd,CircleArea] };
6  terminal    node[1]        { Variable x, y, radius; },
7              child [2];
8  nonterminal Tree[1]        { Semantics.Node root; Variable x, y, l, r;},
9              Subtrees[2] { java.util.List sub; Variable y, l, r, lRoot, rRoot;};
10
11 constraintmanager diagen.editor.param.QocaLayoutConstraintMgr;
12
13 reducer {
14    circle (a) ==> node(a) {
15      rhs[0].x = lhs[0].param(0); rhs[0].y = lhs[0].param(1);
16      rhs[0].radius = lhs[0].param(2);
17      radius = lhs[0].param(2);
18      constraints: radius <= 30;
19    };
20
21    arrow(b,c) inside(b,a) inside(c,d) circle(a) circle(d) ==> child(a,d) {
22       ax1 = lhs[0].param(0); ay1 = lhs[0].param(1);
23       ax2 = lhs[0].param(2); ay2 = lhs[0].param(3);
24       c1x = lhs[3].param(0); c1y = lhs[3].param(1);
25       c2x = lhs[4].param(0); c2y = lhs[4].param(1);
26       constraints: ax1 == c1x; ay1 == c1y; ax2 == c2x; ay2 == c2y;
27    };
28 }
29
30 grammar { start Tree;
31    Tree(a) ::= node(a) {
32          $$.root = createNode();
33          constraints: $$.x == $0.x; $$.y == $0.y; $$.l == $0.x; $$.r == $0.x;
34       };
35
36    Tree(a) ::=node(a) Subtrees(a,b) {
37          $$.root = tree($1.sub);
38          constraints: $$.x == $0.x; $$.y == $0.y; $$.l == $1.l; $$.r == $1.r;
39                       $0.y <= $1.y−50; $0.x >= $1.lRoot; $0.x <= $1.rRoot;
40       };
41
42    Subtrees(a,b) ::= child(a,b) Tree(b)  {
43          $$.sub = subtrees($1.root);
44          constraints: $$.l == $1.l; $$.r == $1.r; $$.y == $1.y;
45                       $$.lRoot == $1.x; $$.rRoot == $1.x;
46       };
47
48    Subtrees(a,b ) ::= [ [  child(a,b) Tree(b) ] Subtrees(a,c ) ] !
49       if b.x < c.x || b.x == c.x && b.y < c.y {
50          $$.sub = subtrees($1.root,$2.sub);
51          constraints: $$.l == $1.l; $$.r == $2.r; $$.y == $1.y; $$.y == $2.y;
52                       $1.r <= $2.l−50; $$.lRoot == $1.x; $$.rRoot == $2.rRoot;
53       };
54 }
```

Fig. 2. *DiaGen* specification for rooted trees

actually paints an arrow on the screen and that allows user interaction with an arrow (modifying its size etc.) has to be added by the user later on. Relations have to be specified and coded similarly. This can typically be achieved by applying some straightforward coding patterns and requires about 30 hand-written lines of code for a simple component (e.g., a circle or an arrow) and less than 5 for a relation.

Declarations of terminal (6–7) and nonterminal hyperedge types (8–9) also describe the attributes that are assigned to instances of those hyperedges. Attribute type `Variable` represents a constraint variable which is used by the constraint solver for automatic layout. `Semantics.Node` is a user-defined data structure which is used by semantic analysis. Finally, the constraint solver engine, which will be used by the generated editor, has to be specified (11). *DiaGen* currently supports QOCA [6] and PARCON [4].

The reducer specification describes the reducing rules. Each rule specifies a pattern of the SRHG and the corresponding pattern of the hypergraph model. The LHS of each rule thus consists of component and relationship hyperedges while the RHS consists of terminal hyperedges only. Each hyperedge is textually represented by its type and the visited nodes in parenthesis. Lines 14–19 specify that circles in the SRHG map directly to node hyperedges in the HGM. Edges from parent to child nodes are more complicated; they consist of an arrow which starts and ends in corresponding circles (21–27). Furthermore, rules specify attribute values of terminal edges in terms of the diagram components' parameters (15–16) and constraints on the components' parameters, which must be satisfied for those patterns (17–18, 22–26). The constraints in this example define that a circle has a maximum radius of 30 units[2] and that arrows have to start and end at circle centers.

The grammar of the hypergraph model is specified by the type of the starting hyperedge (30) and by grammar productions. For trees, context-free productions are sufficient. Each production has a nonterminal hyperedge on the LHS and an arbitrary hypergraph of terminal and nonterminal hyperedges on its RHS. The parser is similar to the Cocke-Younger-Kasami parser for textual languages and thus actually requires a grammar in Chomsky normal-form (CNF) [1], i.e., each production's RHS has to consist either of a single terminal or of two nonterminal hyperedges. *DiaGen*'s generator transforms the specified grammar into CNF. Since this transformation is crucial for parsing efficiency, hints can be given to the generator. E.g., the fourth production (48–53) uses brackets to give a hint how to split up the original RHS. The "`!`" is a further efficiency improving hint to the generator.[3] Line 49 is an application condition for this production. The condition on the nodes' positions defines an ordering on the tree nodes to make the grammar unambiguous.

[2] The default unit is a screen point when using a zoom factor of 1.

[3] As a default behavior, the generated parser deliberately disregards the gluing condition in order be able to deal with (partially) incorrect diagrams. However, this may result in a less efficient parser. Efficiency can then be improved by forcing the parser to obey the gluing condition for selected productions which are marked by "`!`".

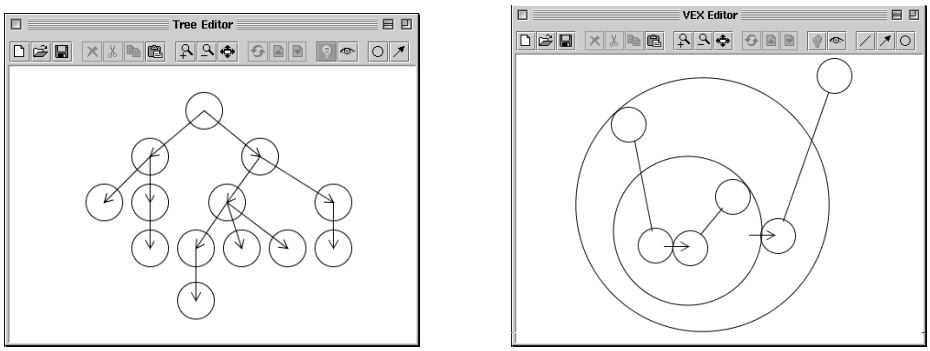

Fig. 3. Two sample editors which have been generated with *DiaGen*: A tree editor and an editor for the visual λ-calculus VEX [2].

Grammar productions optionally carry evaluation rules and constraints. Attribute access has been inspired by YACC: the LHS hyperedge is referred to by `$$`, the RHS hyperedges by `$0`, `$1`, etc. E.g., `$$.root` (line 32) means attribute `root` of the LHS. Evaluation rules are always value assignments where values are computed by user code. It is this user code which is the part of the semantic analysis which is not automatically created by the generator. The constraint solver engine, which has been specified in line 11, restricts the set of possible constraints here: QOCA, which is written in JAVA and which is thus as portable as the rest of the editor, only supports linear constraints, PARCON additionally allows some non-linear constraints. But unfortunately, PARCON is only available under SOLARIS and LINUX[4].

4 Editor Usage

Figure 3 shows two screenshots of the current (preliminary) editor user interface (UI): It consists of a drawing canvas and a toolbar. Using either the toolbar or popup-menus, the user can place diagram components on the canvas. When a component is selected, it creates '*handles*'. These are UI objects whose position is linked to the component's geometric attributes (which in turn define the visual appearance). The component can thus be moved or reshaped by dragging the handles.

When a component is modified, the constraint solving mechanism tries to preserve the syntactic and semantic structure of the diagram by adjusting the geometric attributes of other components. E.g., if a node is moved in the tree editor, its connections are adjusted and, if necessary, entire subtrees are moved to preserve the vertical node ordering. This "intelligent" mode can be switched off if the user wants to execute a modification that changes the diagram's syntactic structure, e.g., moving a subtree from one node to another.

[4] Due to the inefficiency of the current JAVA 2 implementation under LINUX, PARCON is not really usable by *DiaGen* on LINUX.

The recognized diagram structure is indicated by coloring the components: A substructure that could parsed correctly (or an entire correct diagram) is displayed in blue-green shades while incorrect diagram parts remain black.

In addition, the editor currently provides the following features:

- Selection of multiple components, cut & paste operations.
- Saving and loading of diagrams.
- Displaying and editing at arbitrary zoom factors.
- Multiple editor windows for the same diagram.

5 Conclusions

The paper has given a brief outline of the current state of *DiaGen*, a specification method and generator for diagram editors, and how diagram editors are generated with this tool. The tool and sample editors (e.g., the tree editor, which has been used in this paper) are available on the web (`http://www2.informatik.uni-erlangen.de/DiaGen`) .

DiaGen can only generate free-hand editors so far. Current work adds syntax-directed editing as an additional editing mode. *DiaGen* is furthermore going to generate diagram editors that are not stand-alone programs, but software components conformable with the JavaBeans standard. Hopefully, users will then be able to easily customize generated diagram editors and integrate them into larger systems.

References

1. 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, pp. 105–180. World Scientific, 1999.
2. W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *Proc. 1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*, pp. 294–301. 1995.
3. Deutsche Norm DIN EN 61131 Teil 3 "Speicherprogrammierbare Steuerungen – Programmiersprachen". Beuth Verlag, Berlin, 1994. [In German].
4. P. Griebel. *Parcon – Paralleles Lösen von grafischen Constraints.* Phd thesis, University of Paderborn, 1996. [In German].
5. JavaBeans specification. Sun Microsystems, 1997. Available on `http://java.sun.com/beans`.
6. K. Marriott, S. S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical application. In *4th International Conference on Principles and Practice of Constraint Programming, Pisa, Italy*, pp. 340–354, Oct. 1998.
7. M. Minas. Creating semantic representations of diagrams. This volume.
8. UML documentation. Rational Software Corporation. Available on `http://www.rational.com/uml`.