Visual specification of visual editors with DiaGen

Mark Minas

Institute for Software Technology
Department of Computer Science
University of the Federal Armed Forces, Munich
85577 Neubiberg, Germany
minas@acm.org

Abstract. DiaGen is a tool for specifying a visual language and generating a graphical editor from such a specification. The specification is mainly based on graph transformation and graph grammars. This paper demonstrates DiaGen and its features by briefly describing how DiaGen can be used to generate a simplified version of an application which is used in education courses for electrical engineers.

1 Introduction

Programming graphical editors for visual languages like UML class diagrams, Petri nets, Statecharts etc. is hard work. Therefore, various generators for diagram editors have been proposed and developed that create diagram editors from specifications. DiaGen [1] is one of such tools which has been described in detail in [3]. Continued work on DiaGen is also presented in another paper of this volume [2].

This paper briefly outlines a system demonstration of DiaGen which gives a step-by-step description how DiaGen is used to generate a simplified version of a visual programming environment (VPE) for Programmable Logic Controllers (PLCs). The VPE is used as component of a commercial programming environment for PLCs and also in education courses for electrical engineers at the University of Kaiserslautern [5]. The visual programming language (SIPN, Signal Interpreted Petri Net) is based on a special type of Petri nets. This paper focuses on the process of generating a plain Petri net editor which – after visually editing a Petri net – provides the net’s structure, i.e., the set of its places and transitions as well as they are interconnected. Fig. 1 shows an example. The VPE extends such editors by hierarchical Petri nets, code generation etc. [4].

2 DiaGen

DiaGen comes with the Designer, a visual specification tool that is described in more detail in another paper of this volume [2]. The Designer allows for specifying all aspects of a visual language, e.g., the set of visual components
Fig. 1. Screenshot of a Petri net editor and a textual description of the depicted net.

and how they are drawn on the screen, and the language structure. Fig. 2 shows
a screenshot of the DESIGNER while editing a reduction rule for the Petri neteditor (see below.) The DESIGNER translates such a specification to Java code
that, together with the DiaGen framework, implements an editor for the visual
language. Generated editors can be easily plugged into other systems as the VPE
for SIPNs has shown.

The following sections briefly survey the main concepts of DiaGen and the
Petri editor which has been specified and generated with the DiaGen DESIGNER.

3 Hypergraphs and Grammars

DiaGen editors are based on hypergraphs as internal diagram models and hy-
pergraph grammars as a means for syntax specification. This section briefly
surveys these concepts.

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. Hypergraphs are generaliza-
tions 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 (cf. Fig. 4)
by filled circles, directed edges by arrows, and hyperedges by boxes containing
the hyperedge label. Small circles at the border of a box together with tiny
arrows starting at those circles are used to represent tentacles connecting the

Places: [p1*, p2, p3, p4]
Transitions:
t1:
  pre: [p1*]
  post: [p2, p3]
t2:
  pre: [p2, p3]
  post: [p4]
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. $P_1 \ldots P_6$ of Fig. 5 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 ($P_7$ of Fig. 5). Parsing algorithms and a more detailed description of both grammar types can be found in [3].

DiaGen uses hypergraphs as diagram representations and hypergraph grammars for specifying syntactically correct diagrams. The following section de-
Fig. 3. Reduction rules for the Petri net editor. Rule $R_5$, which is similar to $R_4$ with reverse arrow direction, has been omitted. The rules’ graphical representation has been created by the DESIGNER and its printing functionality.

4 Petri Net Editing

The Petri net editor mainly consists of a free hand diagram editor which translates drawings into a hypergraph model, creates its syntactic structure and thus checks its syntactic correctness with respect to the Petri net syntax. As a result of this process, the editor has to provide visual feedback to the editor user if the drawing contains errors. The editor performs this task in a sequence of four steps after each editing operation: the scanning, the reduction, the parsing, and the attribute evaluation step. These steps are illustrated for the Petri net in Fig. 1.

**Scanning step:** Diagram components (e.g., places, transitions, tokens, and arrows) have *attachment areas*, i.e., the parts of the components that are allowed to connect to other components (e.g., start and end of an arrow). The most general and yet simple formal description of such a component is a hyperedge which connects to the nodes which represent the *attachment areas* of the diagram components. These nodes and hyperedges first make up an 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 hypergraph model (HGM) of the diagram.

**Reduction step:** HGMs tend to be quite large even for small diagrams. In order to allow for efficient parsing, a reduced hypergraph model (rHGM) is created from the HGM first. The reducer is specified by some transformations that identify those sub-hypergraphs of the HGM which carry the information of the diagram and build the HGM accordingly. This step is similar to the lexical analysis step of traditional compilers. Fig. 3 shows the reduction rules for the Petri net language whereas Fig. 4 shows the rHGM for the Petri net of Fig. 1. Unary edges of type $t_{\text{place}}$, $t_{\text{trans}}$, and $t_{\text{token}}$ represent places, transitions, and tokens, resp. Binary edges of type $t_{\text{arrow}}$ represent arrows.
**Parsing step:** The syntax of the hypergraph models of the diagram language—and thus the syntax of the language—is defined by a hypergraph grammar. Fig. 5 shows a context-free hypergraph grammar with embeddings. The starting hypergraph of the grammar consists of a `Net` hyperedge which does not visit any node.

Similar to compilers for (textual) programming languages, a hypergraph parser which is built-in into each DiaGen editor is used for creating the syntactic structure of the HGM of the diagram, i.e., for finding a derivation sequence from the starting hypergraph to the rHGM. The parser is capable of identifying syntax errors which are then visualized to the editor user.

**Attribute evaluation step:** The final step of the translation process creates the semantic representation of the diagram by some kind of syntax-directed translation based on an attribute grammar as it is also used in compilers for (textual) programming languages: terminal and nonterminal hyperedges are augmented by attributes, and hypergraph grammar productions by evaluation rules. Initial attribute values are defined by assignment rules being attached to reduction rules. Fig. 3 shows such assignments. Attributes are specified by `e.a` where `e` is the name of the edge (i.e., the string in front of a colon in a hyperedge label), and `a` is the attribute label. Each terminal hyperedge has an attribute whose value is a reference to the corresponding visual component as a Java object. Method `self()` provides such a reference. Evaluation rules of the Petri net grammar productions

---

1 Please note that productions $P_2$ and $P_3$ are so-called *set productions* which are indicated by the “stacked” edges on the RHS. This more readable notation can be regarded as a shorthand for recursive productions. However, this is actually a special kind of production which allows for more efficient parsing.

2 Currently, correct diagram parts are highlighted. Missing highlighting therefore indicates syntax errors.
Fig. 5. Hypergraph grammar productions of the visual language of Petri nets. Production $P_8$ which is similar to $P_7$ has been omitted.

(cf. Fig. 5) are implemented by methods of the Java class $\text{Sem}$. These methods are invoked as soon as all attributes that are required as parameters of such a method have been computed. Please note the special syntax <<...>> in $P_2$ and $P_3$. As these productions are set productions, <<r.place>> resp. <<r.trans>> describe arrays of all place resp. trans attributes of the corresponding Place resp. Trans edges.

The implementation of the attribute evaluation methods and their required data structures is omitted here. Their task is generating an object structure that represents the analyzed Petri net. This object structure is referenced by the net attribute of the resulting Net edge of the parsing process. Fig. 1 shows the textual representation of this value for the depicted Petri net.

References

1. DIAGEN homepage. www2.cs.fau.de/DiaGen/.