Specifying Graph-like Diagrams with **DiaGen**

Mark Minas

*Institute for Software Technology*
*Department of Computer Science*
*University of the Federal Armed Forces, Munich*
*85577 Neubiberg, Germany*

**Abstract**

This extended abstract demonstrates that creating editors and environments for visual languages becomes considerably easier when restricting the class of visual languages. The presented approach considers graph-like languages whose diagrams consist of nodes and edges with different types. The specification method allows to describe such graphs in terms of their node and edge types and makes use of constraints in order to express syntactic properties. The DiaGen system is used to generate running editors from such specifications.

**1 Introduction**

DiaGen provides an environment for the rapid development of diagram editors. It has been used to create editors for a wide variety of diagram languages, e.g., finite automata, control flow diagrams, Nassi-Shneiderman diagrams, message sequence charts, visual expression diagrams, sequential function charts, ladder diagrams, petri nets, UML class diagrams etc. Actually we are not aware of a diagram language that cannot be specified so that it can be processed with DiaGen.

Generating diagrams editors from a formal specification considerably reduces efforts of creating them. However, when restricting to graph-like diagrams, efforts can be reduced even more. Graph-like diagrams consist of certain kinds of nodes and edges which connect nodes. Sometimes, nodes may be hierarchical nodes, i.e., they may contain other nodes and edges. Edges may possibly cross hierarchy boundaries. Well-known graph like diagram languages are UML class diagrams, statecharts, petri nets etc.

This extended abstract outlines how a very condensed specification can be used to fully specify a graph-like diagram language and how a preprocessor

---

1 minas@acm.org

©2002 Published by Elsevier Science B. V.
may create a “general” specification which is then used by DiaGen in order to create an editor for the specified diagram language. The following section briefly describes DiaGen. Section 3 outlines how graph-like diagrams are specified in a condensed way and generated from such a specification. Section 4 concludes the extended abstract.

2 DiaGen

DiaGen is completely implemented in Java, and it consists of an editor framework and a program generator. In order to create an editor for a specific diagram language, the editor developer supplies a specification for the syntax and semantics of a diagram language. Additional program code which is written “manually” can be supplied too. The specification is then translated into Java classes by the program generator. The generated classes, together with the editor framework and the manually written code, implement an editor for the specified diagram language.

Diagram editors which have been developed using DiaGen ("DiaGen editors") always support free-hand editing so that the editor user can arbitrarily create, delete, and modify diagram components, as with an off-the-shelf drawing tool. After each editing operation, the editor analyzes the “drawing” according to the syntax of the diagram language, and informs the user about syntax errors. The developer of a DiaGen editor may also specify compound operations for syntax-directed editing. Each of these operations is geared to modify the meaning of the diagram. Automatic layout is another optional feature of DiaGen editors. It is obligatory when syntax-directed editing operations are specified. The automatic layout mechanism adjusts the layout of a diagram after any modification. DiaGen offers constraints for specifying the layout mechanism in a declarative way, and a programming interface for plugging in other layout mechanisms.

When the user of a DiaGen editor creates or modifies a diagram by free-hand editing, the editor translates the diagram into a hypergraph model, creates its syntactic structure and thus checks its syntactic correctness with respect to the specified 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: scanning, reduction, parsing, and attribute evaluation. These steps are briefly described in the following paragraphs.

2.1 Scanning step

Diagram components (e.g., circles and arrows in directed graphs) 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

\[\text{DiaGen is available from http://www2.cs.fau.de/DiaGen/}\]
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.

2.2 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 rHGM accordingly. This step is similar to the lexical analysis step of traditional compilers.

2.3 Parsing step

The syntax of the hypergraph models of the diagram language – and thus the syntax of the language – is defined by a context-free hypergraph grammar with embeddings. Such a grammar is similar to string grammars as they define terminal and nonterminal edge types, a starting hypergraph and a set of hypergraph productions. A context-free production allows to replace a hyperedge with a nonterminal type (defined by the left-hand side of the production) by the hypergraph of the right-hand side of the production, whereas an embedding production allows to insert a hyperedge into a context which is defined by the left-hand side of the production. Some restrictions apply to situations where embedding productions may be used (see [1]).

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 rHGM 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.

2.4 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:

---

3 Hypergraphs consist of nodes as well as hyperedges and are similar to directed graphs. Whereas edges of directed graphs connect to two nodes, hyperedges connect to an arbitrary number – which depends on the type of the hyperedge – of nodes.

4 Currently, well-formed diagram parts are highlighted. Missing highlighting therefore indicates erroneous diagram parts.
terminal and nonterminal hyperedges are augmented by attributes, and hypergraph grammar productions by evaluation rules.

3 Graph-like diagrams

Graph-like diagrams consist of a set of nodes and edges which connect those nodes. Usually, different kind of nodes and edges are distinguished. Furthermore, nodes may contain other graphs, e.g., packages in UML class diagrams (Fig. 1) which are used as sample language in this extended abstract.

Fig. 2 shows the specification of node and edge types of class diagrams along with predicates which have to be satisfied for valid class diagrams. Class nodes are distinguished from package nodes. Their graphical representation is specified by referring to implementing classes (e.g., uml.Class). Moreover, packages contain (cf. keyword contents) a class diagram for their own (which may be an empty diagram). Two kinds of edges are distinguished: Associations and Generalizations. Their visual appearance is specified by built-in implementations (lines with or without an arrow head). Annotating text (multiplicities etc.) has been omitted for simplicity.

Fig. 3 shows the HGM of the class diagram of Fig. 1. Diagram nodes and
diagram ClassDiagram {}
node Package {
  private String style = "uml.Package";
  ClassDiagram contents;
}
node Class {
  private String style = "uml.Class";
}
edge Assoc {
  private String style = "Line";
}
edge General {
  private String style = "Line";
  private String head = "UnfilledTriangle";
}

predicate 
(forall e in Assoc)
  (src(e) in Class && tgt(e) in Class);

predicate 
(forall e in General)
  (src(e) in Class ==> tgt(e) in Class &&
   src(e) in Package ==> tgt(e) in Package);

predicate 
(forall c in Class + Package)
  (not c -- General+ --> c);
end_diagram

Fig. 2. Specification of the class diagrams

edges are represented by HGM hyperedges which are depicted as gray rectangles being connected by thin lines to black dots, i.e., their visited nodes. Hyperedges visit their nodes in a certain order which is not shown in Fig. 3, but which should be clear from the context. Hyperedge types are the node and edge types. Class and package names have been added in italics in order to visualize the correspondence to Fig. 1. Gray arrows represent spatial relationships: attach connects diagram edges with diagram nodes, contains is used to represent hierarchy. Those hyperedges are used for all kinds of graph-like diagrams and are, therefore, a fixed part of the preprocessor which creates the "general" DiaGen specification from a specification as shown in Fig. 2.

The reducer works almost identically for each graph-like diagram, too. In the resulting rHGM, each node type hyperedge is represented by a node hyperedge, and each edge type hyperedge together with “its” at hyperedges by an edge hyperedge which is immediately connected to the nodes being visited by node hyperedges. A third hyperedge kind of the rHGM is contains which, again, represents hierarchy. Fig. 4 shows the corresponding rHGM for the HGM of Fig. 3.

Figure 5 shows the generic reducer’s reduction rules. Please note that applying a reduction rule \( P \Rightarrow R \) does not mean rewriting of a pattern \( P \) of
Fig. 3. HGM of the class diagram of Fig. 1.

Fig. 4. rHGM of the class diagram of Fig. 1. Letters $s$ and $t$ indicate source resp. target tentacles of the contains hyperedges.

the HGM by $R$. Applying such a rule rather means finding a match of pattern $P$ in the HGM and adding $R$ to the rHGM accordingly [4], i.e., the HGM is not modified by the reducer. The rHGM is created as a second hypergraph instead.

The first rule in Fig. 5 just adds a node hyperedge to the rHGM for each $gNode$ hyperedge of the HGM where $gNode$ is a supertype of each type representing a graph node. In our example, $gNode$ is supertype of the hyperedge
Fig. 5. Generic reducer for graph-like diagrams. The last rule is actually a template with the edge type variables $T_1$ and $T_2$ which is explained in the text.

types class and package.

The second rule searches for any edge of the graph. Hyperedge type gEdge is supertype of each hyperedge type representing a graph edge, i.e., assoc and general in our example. The rule requires that the graph edge is attached to some graph nodes which is indicated by $at$ edges to nodes $a$ and $b$. The crossed out parts of the rule’s LHS indicate a negative context: The second rule must not be applied if the graph edge is attached to other nodes at its source and/or target. Such an ambiguous situation may occur if the user draws two overlapping nodes on the screen and adds an edge to the intersection of their borders as shown in the following example of plain graphs:

The last reduction rule in Fig. 5 finally is responsible for adding contains hyperedges to the rHGM representing hierarchy. The depicted rule is actually a template which has to be instantiated multiply. The instantiations have to reflect the hierarchy specification which is described by the contents keyword (cf. Fig. 2). In our example, the template has to be instantiated for $(T_1, T_2) = (package, class)$ and $(T_1, T_2) = (package, package)$ since packages may contain
packages as well as classes.

The rHGMs are parsed with respect to a hypergraph grammar which is identical for all graph-like diagram languages. It is shown in Fig. 6. The grammar is a context-free hypergraph grammar with embeddings. Ovals represent nonterminal hyperedges, whereas rectangles represent terminal hyperedges, i.e., hyperedges of the rHGM. All productions but the last one are context-free productions, i.e., the LHS consists of a nonterminal hyperedge with its visited nodes only. The “...” indicate repetition which could be represented by recursive productions, too. However, DiAGEN uses these set productions instead which allows for more efficient parsing. The last production is an embedding production which allows to connect any two nodes by an edge.

The remainder of the specification (Fig. 7) defines consistency predicates quite similar to GTDL [3]: Each of the predicates describes a property which must hold for the diagram. The first one describes that associations connect classes only. The next one tells that classes (packages) may generalize classes (packages) only. The last one finally requires that generalization edges do not create cycles. -- General+ --> matches any path which consists of General edges only.

The preprocessor translates those predicates into Java code which is executed during attribute evaluation, i.e., during semantic analysis. The editor is able to display quite detailed error messages when predicates are violated. E.g., if the user connects a class to a package with an association edge, the editor will highlight this edge and show the violated predicate. Future versions will display a more readable text, too, which will be part of the specification.
An alternative approach would have been to translate such predicates into reduction rules. This would have been possible for the first predicate and the second one. *Association* and *generalization* hyperedges of the rHGM would not be reduced to *edge* hyperedges of the rHGM if the predicates are violated. However, the editor would have to use its generic ability to deal with erroneous diagrams, i.e., it would highlight wrong edges, but the editor could not display a detailed error as described above.

Fig. 2 contains the complete representation of class diagrams except the Java classes *uml.Class* and *uml.Package* which implement class and package symbols. The current implementation requires to implement these classes manually, but we are currently building a tool for specifying them graphically. These some 30 lines of textual specification of Fig. 2 are translated to some 100 lines of textual specification of UML class diagrams with DiaGen. Additionally, a layout module is created from a generic graph layouter. It is not the layouter’s task to avoid or minimize edge intersections on the screen. Instead, it spreads the nodes on the screen such that they do not overlap except hierarchically contained ones. Of course, the layouter takes into account the previous layout, i.e., it tries to minimize layout modifications.

4 Conclusions

This extended abstract has briefly outlined a simplified method for specifying a certain class of visual languages that consists of graphs with different node and edge types. Hierarchical graphs where nodes may contain other graphs can be specified, too. Syntactic properties of the visual languages have to be specified by predicates that have to be satisfied by the graph in order to be valid. Given such a specification, a running graphical editor is generated in two steps. In the first step, this specification is transformed into another one which is suited for the DiaGen system whose generator finally generates the running editor in the second step.

This specification method has been inspired by GTDL [3]. However, since the DiaGen approach is used in the end, many of DiaGen’s features like automatic layout, generating Java Beans etc. are features of this described approach, too.

The described approach is still work in progress. It shows that specifications can be considerably smaller when visual languages are restricted to certain classes of languages. For the UML class diagram example of this extended abstract, the specification was reduced from some 100 to some 30 lines. Moreover, the generated editor comes with an automatic layouter. This layouter would have to be created manually from the generic layouter if we had not used this approach. And finally, editors which are created using this approach are able to show quite detailed error messages in the case of erroneous diagrams without additional programming efforts which would be required when not using this approach.
However, this approach is still using graph grammars and graph parsers for specifying and checking some parts of the visual syntax after translating the specification into a DiaGen-conforming one. But we can also avoid this additional effort: The specification as it has been described in this paper is actually a graph schema together with some constraints. Syntax checking, therefore, can be performed by a graph schema and constraint checker. This approach will be followed in the future. Graph schemas and predicates will then be similar to extended ER diagrams and Graph Specification Language constraints as they are used in the Kogge-system [3].

References


