DiaGen

Overview

http://www2.informatik.uni-erlangen.de/DiaGen/
diagen@immd2.informatik.uni-erlangen.de

February 24, 2000

DiaGen is a system for easy developing of powerful diagram editors. It consists of two main parts:

- A framework of Java classes that provide generic functionality for editing and analyzing diagrams.
- A generator program that can produce Java source code for most of the functionaliy that depends on the concrete diagram language.

This document describes the idea of *DiaGen* and the prominent features of the system. It gives an overview of the system's architecture and explains some basic concepts and terms that are used throughout the documentation.

1 Main features of the system

To define a specific diagram language and create an editor for it, the language implementor needs to give a formal specification of the diagram language. This specification is processed by the generator, which produces classes that extend the basic class framework and specify the syntactic structure of the diagram language. To complete the editor, the language implementor needs to add some extra Java code that mostly relates to the graphic display and manipulation of diagrams.

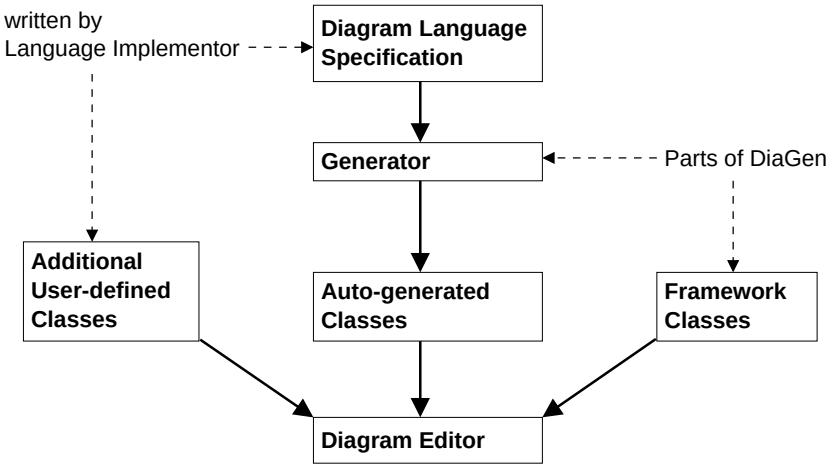

Figure 1: The *DiaGen* system

The combination of the following main features distinguishes *DiaGen* from other existing diagram editing/analysis systems:

- *DiaGen* editors include an analysis module to recognize the structure and syntactic correctness of diagrams on-line during the editing process. The structural analysis is based on hypergraph transformations and grammars, which provide a flexible syntactic model and allow for efficient parsing. *DiaGen* has been specially designed for fault-tolerant parsing and handling of diagrams that are only partially correct.

- *DiaGen* uses the structural analysis results to provide syntactic highlighting and an interactive automatic layout facility. The layout mechanism is based on flexible geometric constraints and relies on an external constraint-solving engine.

- *DiaGen* combines free-hand editing in the manner of a drawing-program with syntax-directed editing for major structural modifications of the diagram. The language implementor can therfore easily supply powerful syntax-oriented operations to support frequent editing tasks, but she does not have to worry about explicitly considering every editing requirement that may arise.

- *DiaGen* is entirely written in Java and is based on the new *Java 2 SDK*. It is therefore platform-independent and can take full advantage of all the features of the new Java2D graphics API: For example, *DiaGen* supports unrestricted zooming, and rendering quality is adjusted automatically during user interactions.

2 Architectural overview

Figure 2 sketches the architecture of an editor which has been generated by the *DiaGen* generator and which is based on the *DiaGen* framework. The diagram shows the different modules that make up the editor and the data flow between them. The following sections will explain the structure presented in this figure and the specific terminology. This explanation first surveys the structure imposed by the *DiaGen* framework which supplies major parts of the editor. Thereafter, customizations which are necessary for creating a specific editor from the framework are briefly described. The figure indicates these customized framework components along with the *DiaGen* source code generator which carries out major parts of the customization process.

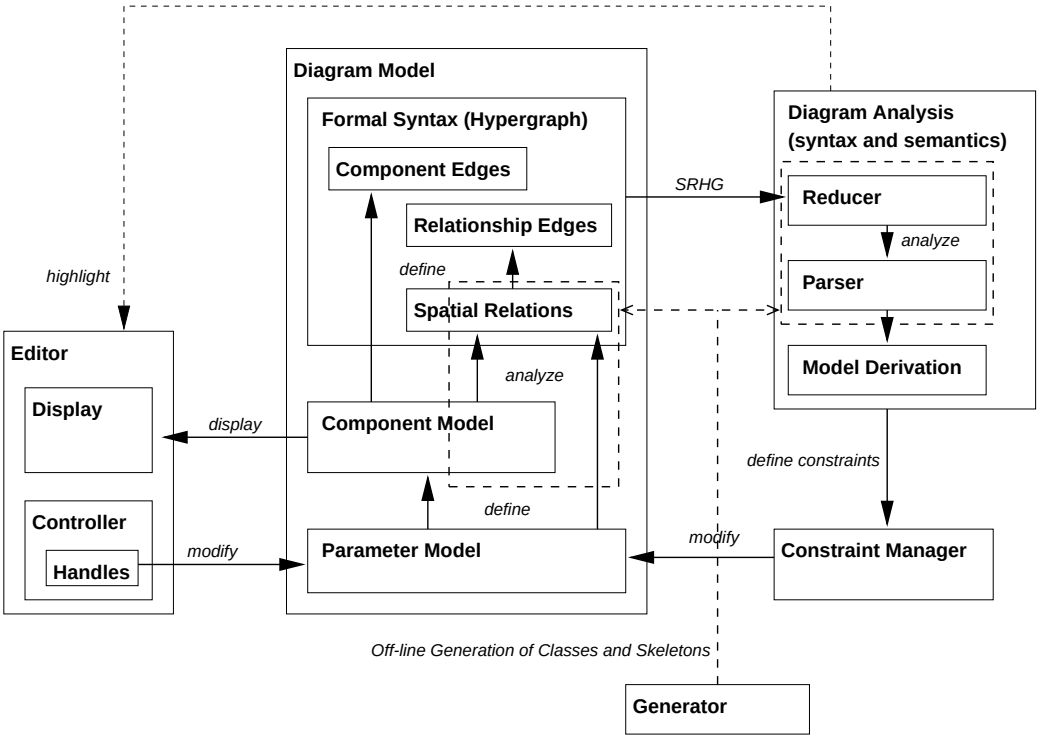

Figure 2: Architecture of an editor generated by and based on *DiaGen*

3 The diagram model

The architectural center of the framework is the *diagram model*, the run-time representation of the diagram as objects in memory. The term "model" is used here in the sense of the Model-View-Controller paradigm and must be distinguished from abstract models that can be derived by higher-level analysis of the diagram. The diagram model consists of a set of components that make up the diagram (e.g. circles, lines or complex objects like UML class definitions) and their representation in the formal hypergraph syntax; it also includes a run-time description of the type of diagrams that can be manipulated. The diagram model is divided into three layers:

At the lowest level, the model is represented as a collection of "parameters", simple real numbers that determine the properties of the diagram components. For example, at this level the representation of a circle might be:

```
xcenter1 = 10, ycenter1 = 12, radius1 = 5
```

A pair of parameters often forms a point, e.g.

```
center1 = (xcenter1, ycenter1)
```

Each parameter belongs to exactly one component; changes to the model always occur at the parameter level first and are then propagated upwards.

The *component model* describes how the graphic representation of the diagram is computed from the parameters. It also implements an event-flow that updates the graphic representation when the parameters change. The graphic representation provided by this layer uses the classes and concepts of the Java2D API.

The *formal syntax layer* is built on top of the component model. *DiaGen* uses hypergraphs as a higher-level representation of diagrams: Each component has one or more "attachment areas", sensitive areas that it can interact with other components. For an arrow, those areas would usually be its endpoints, while for a circle it could be its entire interior. Each attachment area maps to a node in the hypergraph model; a component is represented by a hyperedge that connects to all the nodes of its attachment areas. Component hyperedges thus represent the same concept as "N-attaching point entities" (NAPEs), which have been used in other diagram parsing approaches.

The way components are combined to form a diagram is represented by "spatial relationships". These are relations on the attachment areas which are defined by predicates on the corresponding components' parameters. For example, two circle areas could have the relationship "touch" if the parameters of their components satisfy the equation

```
distance(center1, center2) = radius1 + radius2
```

Presently, the editor supports only binary relationships which should in fact be sufficient for all practical purposes. To avoid the checking every pair of attachment areas, we impose the restriction that spatial relationships can only exist between overlapping attachment areas. This means that the spatial extents of the attachment areas can serve as hints for the relation detection.

Each of those spatial relationships maps to an edge in the hypergraph model which connects the nodes corresponding to the related attachment areas. We call the resulting formal syntax representation of the diagram its *spatial relationship hypergraph* (SRHG); it consists of a set of separate component edges and distinct nodes attached to them, which are then connected by relationship edges.

The model layers form a hierarchy in the sense that each layer can only depend on lower layers; for example, it would be possible to change the formal syntax representation without affecting the component and parameter model.

4 Syntactic and semantic analysis

The SRHG provided by the formal syntax model is processed by the syntactic and semantic analysis module. The *DiaGen* architecture splits the diagram analysis into three stages:

The “reducer” simplifies the SRHG by replacing subgraphs that match certain patterns with simpler structures. The reduction rules can include negative context and additional conditions can be introduced that need to hold before a reduction step is performed. In the resulting simplified *hypergraph model* (HGM) all component edges that are connected by meaningful spatial relations should be transformed and all “random” relations, that do not convey a semantic meaning, should be eliminated.

The HGM can then be analyzed by an incremental hypergraph parser; context-free hypergraph grammars with embeddings are used as a formalism for the definition of the syntactic structure: The use of hypergraphs instead of standard graphs makes it usually possible to express most of the structural requirements of a typical diagram language in a context-free form. Additional “embedding” productions can be used to express connections that cannot be described in a context-free form.

The hypergraph parser uses similar techniques to the standard CYK-algorithm for string grammars. The CYK parsing algorithm constructs all partial syntax trees that can be found in the input and is therefore well suited to handle diagrams that are only partially correct.

The resulting information about the diagram structure can then be used to build a high-level semantic model of the diagram content. The concrete form of this semantic representation is highly application dependent and can therefore not be defined in the framework. One possibility for using this semantic information is to translate the content of the diagram into another (e.g. textual) representation; this has already been implemented in sample editors for the new *DiaGen* system.

5 Editing

To manipulate diagrams, the framework provides a generic editor. The editor is modeled according to the common Model-View-Controller concept. It implements the functionality for displaying the model at the component layer (view) and of manipulating it at the parameter level (controller) with the help of several types of “handles“. A handle is a user-interface object whose position is linked to certain parameter values in the parameter model by specific equations. Moving

the handles changes those parameters and modifying the parameters in turn causes the handle to adjust its position and appearance. The editor also allows the selection of multiple components, which can then be relocated together, and it supports standard cut & paste operations. The use of the MVC-pattern makes it possible to have a 1-to-n relationship between model and editor, which means that a model can be edited by any number of editors – including none – at the same time.

The editor mainly interacts with the component level of the diagram model (to display the diagram) and the parameter level (to modify components). Although the editor is fully integrated in the current *DiaGen* system, it would be possible to switch to a different formal syntax model (e.g. attributed multisets) and another parsing technique with hardly any changes to the editor module. The results of the analysis module can still be used to support the editing process directly: The current implementation provides a highlighting mechanism that gives the user visual feedback about the semantic correctness of the diagram and indicates diagram parts that cannot be parsed correctly.

6 Constraint management

The information gained from the diagram analysis can then be used to extend the model with constraints between the parameters. Such constraints try to propagate changes “intelligently” through editing actions. This means that, for example, related diagram parts move together, an arrow adjusts itself if its destination is moved or a text box widens to fit the contained text. For the implementation of the constraint manager module we rely on an existing constraint solving engine, which is wrapped into adapting code to interface it with the other *DiaGen* modules.

The *DiaGen* editor operates in two modes: The “intelligent” mode provides the full functionality that has been outlined above to support the editing process. In “simple” mode, the information flow is cut off between constraint manager and parameters. This enables the user to generate temporarily inconsistent diagrams and also gives her a means to control the editor if it should react “too smart” and execute unintended changes. When the editor is switched back to “intelligent” mode, the semantic representation is recomputed and changes are propagated.

7 Syntax-directed editing

The fact that *DiaGen* uses an internal formal model of the diagram (the SRHG) makes it possible to support high-level editing operation that operate on this abstract level: Complex language-specific diagram modifications can be described as hypergraph transformations on the SRHG. The user can select such an operation together with components that designate a partial application context in the SRHG; the system can then automatically extend the context as required for the transformation and executes it. The modified SRHG is then analyzed again and, with the help of the constraint-based automatic layout mechanism, it is transformed back into a visual representation.

Such high-level operations give the user a means to easily execute larger structural changes on the diagram. For example, substructures can be disconnected, moved to another place and reconnected there almost automatically in a single step. As the syntax-directed editing facilities only supplement the free-hand editing, the editing operations need not allow all necessary modifications. They only serve as shorthands for some common cases while free-hand editing can still be used to achieve diagram modifications that the programmer of the diagram editor has not foreseen.

8 Package structure

The modular structure of the *DiaGen* system that has been outlined above is reflected in the package structure of the framework. Table 1 shows the correspondence between modules and Java packages.

In addition, the package `diagen.editor.lib` provides some general-purpose classes and standard implementations that are useful for customizing the framework.

9 Customization of the framework

Of course, a general editor framework can not provide all the functionality that will be required to implement a concrete editor for a specific diagram type. The main parts that still need to be specified are the concrete component types and relations, the constraints and the hypergraph parser and reducer. The editor itself can also be extended, if more powerful operations are required.

Table 1: Packages of the *DiaGen* architecture

Module	**Package**
parameter model	`diagen.editor.param`
component model	`diagen.editor.model`
(concrete) formal syntax	`diagen.editor.graph`
editor	`diagen.editor.ui`
abstract syntactic and semantic analysis	`diagen.model`, `diagen.hypergraph` and subpackages
constraint manager	`diagen.editor.param`, `diagen.constraints` and subpackages
generator	`diagen.generator`

As usual in object-oriented environments, customization is accomplished by subclassing the general-purpose classes of the framework: Component subclasses, for example, define concrete graphic representations, attachment areas and specific handles for manipulation. This customization process is supported by a generator module that takes a diagram specification in a special-purpose language and produces Java source files that extend the framework to form an editor for diagrams of the respective type. The generator creates the complete reducer and parser modules and part of the Java code that implements components and relations. The component and relation implementations cannot be auto-generated completely, because some aspects like the graphical representations and the specific handles are best described at the Java level. Therefore the generator creates skeleton source files for these parts, which have to be hand-edited later.

10 Current state of development

At its current state, *DiaGen* is a purely academic “proof-of-concept” system. That means that it contains enough features and is sufficiently stable to write academic demonstrations for non-trivial diagram languages (the system includes several examples) and probably educational tools as well (perhaps an editor/simulator for FSAs or Petri Nets). Still, the system probably contains major bugs that can result in occasional hangs or crashes, the user interface is quite primitive and the included library of pre-defined diagram elements (components) is very restricted.

We strongly discourage using the system for any serious application, and, of course, we carry no responsibility whatsoever for any direct or indirect damages of any sort that may arise from running the system.