Contents

1 Introduction ...........................................................................................................1

2 Architectural overview .......................................................................................2
  2.1 The diagram model .....................................................................................2
  2.2 Syntactic and semantic analysis ...............................................................4
  2.3 Editing .........................................................................................................5
  2.4 Constraint management ............................................................................5
  2.5 Package structure ......................................................................................5
  2.6 Customization of the framework ...............................................................6

3 The parameter and component model ..............................................................6
  3.1 Components and their graphic representation ..........................................6
  3.2 Parameters and constraint handling .........................................................8
  3.3 Different types of diagrams .....................................................................10
  3.4 Representation of events ........................................................................10

4 Diagram editing ................................................................................................11
  4.1 Displaying the diagram ...........................................................................11
  4.2 Modifying component parameters ...........................................................13
  4.3 Event processing ......................................................................................14
  4.4 Editing concrete component subtypes .....................................................17
  4.5 Selecting multiple components; cut and paste .........................................18

5 The formal syntax layer ....................................................................................19
  5.1 Components and attachment areas ..........................................................19
  5.2 Spatial relationships ................................................................................21
  5.3 The scanning process ...............................................................................22
  5.4 Support for the editing process ...............................................................24

6 Other implementation issues ..........................................................................26
  6.1 Multithreading in the editor ....................................................................26
  6.2 Synchronizing the update pipeline ..........................................................28
  6.3 Serialization of components and diagrams ..............................................29
  6.4 General coding considerations and code performance .............................31
  6.5 Graphic algorithms ..................................................................................31

7 Defining a diagram type ....................................................................................32
  7.1 Concrete components and relations in Java .............................................33
  7.2 Using a diagram type specification language .........................................35
  7.3 Auto-generated source files ....................................................................36
  7.4 Implementation of the generator module ...............................................37

8 Conclusion and ideas for future work .............................................................38
Design and Implementation of a Generic Graphical Editor

Oliver Köth

Abstract

The DIAGEN project aims to provide a general framework for editing and analyzing diagrams. This paper describes the implementation of a generic adaptable diagram editor for this project. The main features of the editor are

• unrestricted manipulation of diagram components,1 which allows the user to create temporarily inconsistent diagrams,
• support for a flexible semantic model, that is built on the concept of a “spatial relationship hypergraph”,
• the possibility to use this semantic information to endorse the editing process,
• portability achieved by the use of the Java language,
• tight integration of the Java 2D library, which provides extensive graphic capabilities, printing support etc.,
• flexible customization for specific diagram types, which are described as a collection of Java classes, and
• a description language and a code generator, which provide additional support for this process.

The work presented here focuses on the design of the underlying class model that provides the basic functionality for representing and manipulating diagrams. It gives a detailed presentation of the involved classes and their interaction and shows how this framework can be used to create editors for specific diagram languages.

1 Introduction

The first implementation of DIAGEN [Minas 95] was written in C++ and used “syntax-directed” editing: diagrams could only be manipulated through pre-determined transformations that preserved their syntactic correctness. This approach turned out to be rather inflexible and difficult to use.

Therefore in [Minas 97] presented a reimplementation that has been switched to “free editing”: This concept allows unrestricted manipulation of diagrams similar to the way drawing programs are used, so it is possible to create intermediate diagrams that are incorrect with respect to the underlying syntax definition. An incremental hypergraph parser was used to analyze the structure of the edited diagrams. Instead of C++, the Java language [Java 2 SDK] was chosen to make the system less platform-

1. as opposed to syntax-directed editing
dependent. This implementation proved that a “free editing”-approach was feasible and that hypergraph parsing could be fast enough to support interactive manipulations.

The hypergraph syntax used in the previous DIAGEN implementations was limited by the way it used overlapping ends or edges of diagram components to detect connections. To address this restriction, [Minas 99] has adapted a more general hypergraph representation for diagrams, which supports arbitrary spatial relations between diagram components. The hypergraph parser of the previous DIAGEN version has been extended with a graph-transformation step to enhance analysis of such representations. This project is a complete reimplementation of the diagram editor part of DIAGEN that supports the new syntactic concepts and interfaces with the hypergraph analysis module.

A second reason for reimplementing the editor is the appearance of the Java2D framework [Java2D], which extends the graphics capabilities of the Java environment enormously. New functionality like user-defined strokes for lines or drawing Bezier curves is now available; operations like zooming, which had to be implemented with considerable effort in the previous editor version (cf [Schmoelz 97]), are now provided by the system libraries.

The next section presents an overview of the re-designed DIAGEN architecture and shows how the editor interfaces with the other architectural entities. In the following three sections the class design of the different model levels and the editor is presented. After that a few particular aspects of the framework will be examined in detail. Finally, the last section explains how an actual editor for a particular diagram type is built from the framework and a source-code generator that aids this process is presented.

2 Architectural overview

Figure 1 presents an overview of the DIAGEN architecture with emphasis on the parts that are presented in this work. The diagram shows the different modules that make up the system and the data flow between them.

2.1 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 and their representation in formal 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:

\[ x_{\text{center}1} = 10, y_{\text{center}1} = 12, \text{radius}_{1} = 5 \]

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

\[ \text{center}_{1} = (x_{\text{center}1}, y_{\text{center}1}) \]
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 a notification mechanism that links parameters and components. 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\(^1\) that connects to all the nodes of its attachment areas.

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\(^2\)

\[
\text{distance}(\text{center}_1, \text{center}_2) = \text{radius}_1 + \text{radius}_2
\]

---

1. This paper consistently deals with hypergraph models, therefore “edge” always stands for “hyperedge” and “graph” for “hypergraph”.
2. For practical purposes, the formulation of the predicates has to account for “near misses”, i.e. the above predicate should be formulated as

\[
|\text{distance}(\text{center}_1, \text{center}_2) - (\text{radius}_1 + \text{radius}_2)| < \epsilon
\]
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.

### 2.2 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 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.

This simplified graph can now be analyzed by an incremental hypergraph parser that uses similar techniques to the standard CYK-algorithm for string grammars. If the diagram is syntactically correct, the graph should reduce to a single starting edge. Otherwise, the parser will find several correct subgraphs and additional incorrect edges.\(^1\)

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. Such a translation could be used in a parallel source-code/graphic editor for programming languages like SDL that support both representations.\(^2\)

The analysis module is not part of the work presented here; a more detailed description of the process with illustrating examples can be found in [Minas 99].

---

1. In this case, the way the edges are grouped into subgraphs is usually nondeterministic.
2. However, the inverse problem of generating a graph with a suitable layout from a semantic representation has not yet been addressed in DIAGEN.
2.3 Editing

To manipulate diagram models, the framework provides a generic editor. The editor is modeled according to the common Model-View-Controller concept [Gamma 94]. It provides 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.

2.4 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.

2.5 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:

---

1. This paper does not use fully qualified class names, as this would affect the readability of the text. All class names in the text are unique across packages and the corresponding package can be found in the index entry for each class.
In addition, the package diagen.editor.lib provides some general-purpose classes and standard implementations that are useful for customizing the framework.

### 2.6 Customization of the framework

Of course, a general editor framework cannot 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 components and relations, the constraints and the hypergraph parser and reducer. The editor itself can also be extended, if more powerful operations are required.

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; this process is presented in more detail in Section 7.3.

### 3 The parameter and component model

The parameter and component levels of the diagram model contain the objects that represent the geometric structure of a diagram at run-time. Figure 2 presents an UML diagram of the classes that constitute these levels and the connections between them.

#### 3.1 Components and their graphic representation

The central point of access for any diagram model is the Model class. An object of this class holds references to all other objects that are parts of the diagram, which are
- the run-time representation of the diagram type,
- the components of the diagram,
- the objects that are responsible for constraint handling and
- worker thread objects that control the execution of updating operations.

The most important operations that a model object can perform are the adding and removing of components. The model hides internal aspects (constraint handling and
The parameter and component model

thread sequencing) from other packages and thus models a Facade pattern [Gamma 94] in object-oriented design terminology.

All the components of a model are referenced from a list, which determines their layering (z-order) in the diagram. Display classes can acquire an unmodifiable copy of this list and they are required to draw them in the correct order. The model class provides a method that can be used to change the z-order of a component.

The components themselves are the most important constituents of the diagram model. Their different aspects are defined by several interfaces; the most basic of these is the DiagramComponent interface, which defines the connection between the components and the mode. There are obviously a lot of aspects of diagram components that cannot be fixed in a general framework, but some generic methods, like the general control-flow for updating a component after manipulations, are provided in the standard implementation class BasicComponent. Any user-defined components will be derived from this class; the customization process is described in detail in Section 7.

Currently BasicComponent is the only class that implements the DiagramComponent interface directly; the reason for separating the two is to prepare the class design for the possible introduction of compound components by defining another implementation of DiagramComponent for them.

FIGURE 2. Classes of the Component Model
Every component must be part of a diagram; most methods of a component may not be called before the component has been added to a model. Many internal initialization functions for a component cannot be executed in the component’s constructor, because they depend on properties of the concrete component type (e.g., parameter values) which are not available to the general abstract class at this point. Only when a component is added to a model, its register method is called to complete the initialization process (e.g., the graphic representation is created at this point).

Methods that concern the graphic appearance of a component are separated into the VisualRepresentation interface which is better than using inheritance, as the graphic appearance can thus be easily exchanged. The design scheme used here is similar to the Bridge pattern [Gamma 94] for decoupling different implementations of an abstraction (the graphic representation of an object) from objects that extend this abstraction in other directions. The visual representation is responsible for drawing the component, calculating the bounding box and hit testing; for a broad spectrum of graphic primitives this functionality can be quite easily implemented using the Java2D library. A standard implementation of the interface for simple geometric shapes is provided by the class SimpleVRep; the library also contains visual representations for shapes with user-defined outline strokes and for text strings.

The method that draws a visual component takes an additional “highlight” parameter. This parameter is used by the editor to indicate selected components and to give visual feedback about the correctness of diagrams. The current implementations of visual representations use different colors to indicate the highlighting mode; the selection is drawn in red and correct diagram parts are colored in blue and green shades.

3.2 Parameters and constraint handling

The values that describe the shape and position of a component’s visual representation are stored in Parameter objects. A parameter encapsulates a single numeric value of type double, which can be modified through setValue and getValue methods. Whenever this value is altered, the parameter reports to its “owner” (a diagram component), which must implement the ParamOwner interface. The class BasicComponent provides the standard functionality that components need for this purpose and thus links the DiagramComponent interface to the component’s parameters: whenever the component is notified that one of its parameters has changed, the revalidate method is called which then in turn recomputes the component’s visual representation by calling the method computeVRep. Of course, the actual appearance of a component cannot be known to a basic class, so this is an abstract method that must be implemented by concrete component subclasses. The revalidate method can also be overloaded if other aspects of the component need to be adjusted after parameter changes; this mechanism is used in the semantic model to update the hypergraph representation.

Parameters are often used to describe points on the drawing plane. The Java2D classes define a Point2D interface, which is used to pass points to graphic methods. To access those methods conveniently, the class ParametricPoint2D provides a view of two parameters that implements this Point2D interface.

---

1. The ColorHighlight class contains a static method for this purpose.
2. Actually, the revalidation is done in a separate thread, cf Section 6.1.
The purpose of implementing parameters in a separate class rather than just storing their values in the components is to make it possible to intercept value changes and impose constraints on the values: calling a parameter’s `setValue` method does not directly modify the value, but only reports the change request to a `ConstraintHandler` object where it is temporarily stored. When a group of change requests has been issued, the `processChanges` method of the constraint handler must be called. Those requests can be given in any order; they are treated as a group, which is then passed to a constraint manager to do the actual parameter modifications.

The intermediate step of storing the changes and processing them in batches is needed to define the concept of “simultaneous” changes, which might have been processed differently, had they occurred sequentially. For instance, translating a whole group of components will probably not affect constraints that link their parameters, but translating them one after the other might not even be possible, if their relative distance is fixed through constraints.

The `ConstraintManager` interface does not impose any restrictions on how parameter values may be linked. A constraint manager simply receives a batch of change requests and is free to handle them in any desired way (it may completely ignore them), but it should obviously try to set the parameter values as close as possible to those requested. If necessary, it can also modify parameters that were not included in the change requests. The constraint manager must use the `cSetValue` method to actually modify the parameters, as using `setValue` would of course lead to an infinite loop.

A very primitive implementation of this interface is the empty constraint manager provided by the class `NullConstraintMgr`. This type of constraint manager simply executes all change requests as they have been issued without altering other parameters, which means that using an Mokpo constraint manager has basically the same effect as altering the parameter values directly.

Every diagram model has one constraint handler but two constraint managers, only one of which is active at any point: In “intelligent” mode the model uses a “real” constraint manager, but when the user switches to “simple” mode, the model disconnects the constraint manager from the constraint handler and uses an empty constraint manager instead. Here another advantage of separating the constraint handler, which accumulates the modification, from the constraint manager, which processes them, comes into play: While the constraint handler is referenced by all the parameter objects, the constraint manager is only accessed from the single constraint handler object. Therefore switching the active constraint manager does not necessitate modifying the references in all the parameters.

Implementing a constraint solving engine that is sufficiently powerful for our purposes and at the same time fast enough to support interactive editing is obviously a complex task. On the other hand the interface defined by the architecture is very flexible and it is therefore relatively easy to incorporate an existing constraint solving engine into the project with the help of a wrapper class. An experimental integration of the ParCon constraint solver has demonstrated that the design concepts work as expected. Unfortunately ParCon must run as a separate UNIX process, which defies the idea of creating a platform-independent editor framework in pure Java. Presently

---

1. The `NullConstraintManager` class has no instance variables, therefore only one instance of this empty constraint manager exists, which can be shared among all models.
the QOCA constraint solver [QOCA] is being used for the constraint manager module; this engine is available in a Java version as well as in C++.

### 3.3 Different types of diagrams

The DIAGEN framework provides generic support for editing diagrams of different types. For some functions, a run-time representation is needed for all the properties that are common to diagrams of a specific type. At the component level, these common properties are the types of components that can appear in the diagram and the type constraint manager that is needed. Actual diagram types are objects that inherit from the DiagramType class, which provides a standard interface for creating correct model, constraint manager and component instances using the factory pattern [Gamma 94]. To instantiate new components, separate factory classes are employed; those classes also contain information (a descriptive name and icon) that user-interface classes can utilize to create appropriate controls.

As a diagram type is described by a single object, all subclasses of DiagramType should follow the singleton pattern [Gamma 94], i.e. each of those classes has exactly one instance which is accessible as a static class variable, that is named INSTANCE by convention. Creation of new instances is prohibited by having only private constructors for these classes, which cannot be accessed from outside.

### 3.4 Representation of events

The Model class also serves as a central access point for event distribution. Events are represented as actual objects in DIAGEN; other objects that need to be notified of an event have to implement an appropriate event listener interface must be registered for the event distribution. The architecture of DIAGEN required an event handling scheme that makes it possible to separate the event routing from the interpretation of events, because some objects must be able to pass on events without knowing about their specific meaning. The analysis module for example produces events which are delivered to editing classes via the diagram model, which, in spite of this, should not be concerned with their content and representation.

These considerations result in the class design that is shown in Figure 3. All DIAGEN events are derived from the DiagenEvent class which is in turn a subclass of the standard Java EventObject class. Events are delivered to a listener by calling the event’s transmitEvent method and giving the event listener as an argument. The event class itself is then responsible for detecting the correct listener type (using the Java instanceof operator) and activating the appropriate listener method; if the listener is not of the required type, the event superclass is asked to handle it.

The basic DiagenEvent class corresponds to a general listener type EventTransmitter. An event transmitter is a general event routing device that receives DIAGEN events of all types and is able to pass them on to other listeners without knowing their concrete meaning. A useful implementation of this interface is the EventDistributor class. An event distributor is a fan-out point for events; it holds a list of other event listeners and every incoming event is copied to all of them. The Model class uses an event distributor to keep track of its listeners.

---

1. This is automatically enforced by the generator, see Section 7.4.
2. rather than just calls to a callback function
Currently DIAGEN also defines two specific types of events and respective listener and adapter classes:¹ A DiagramModelEvent originates from the high-level model that is generated by the semantic analysis (not from the “diagram model” as the term is otherwise used in this paper) and is spread out by the model’s event distributor. These events inform listeners about the progress of the higher-level analysis. A DiagramComponentEvent is generated by the component model whenever a component has been added, deleted or modified and thus needs to be redisplayed. An example of how the events are broadcast through the editor is given in Section 4.3.

4 Diagram editing

The classes of the DIAGEN framework which are part of the editor module are depicted in Figure 4. The UML diagram also shows their relation to classes of the diagram model that have already been discussed.

4.1 Displaying the diagram

The central class of the editor module is the EditorPane class, which is (indirectly) derived from the JPanel class of the Java Swing Toolkit. An editor pane is a canvas object that displays a diagram model; in order to make it visible on the screen, it must be included in a window (a JFrame), which will usually contain additional elements (panels) such as a menu bar, a button tool bar or a status line.
FIGURE 4. The Editor module
As we have already mentioned, the DIAGEN editor follows the Model-View-Controller paradigm; this means that any editor window must always be linked to a single diagram model, while a model can be displayed and modified in any number of editors. An editor pane is specialized to a concrete diagram type (it holds a reference to a DiagramType object), because it has to know about the types of components that can be inserted into the diagram. The editor pane is registered as an event listener (a DiagramComponentListener for the displayed model, so that it can update the display whenever a component is added, removed or modified. In addition, the editor pane also contains an event distributor, which is linked to the model. This gives other UI objects the possibility to receive events through the editor pane rather than the underlying model so they need not adapt their connections when a different model is loaded into the editor. This is especially important for making the editor conformant to the JavaBeans standard, as we plan to do in the future: The editor pane is a visible screen Component and thus a natural candidate for a Bean; by making it an event source a programmer can connect new listeners to it in a graphical JavaBeans building tool.

The Swing toolkit takes care of handling screen updates and supports double buffering to suppress flicker effects, that appear when the screen is erased and redrawn quickly; this technique had to be implemented by user code in the previous DIAGEN version. The editor pane reacts to repaint notifications by retrieving a list of all components from the model and asking their visual representations to draw themselves; of course only, if they are inside the area that needs to be repainted. After that, other screen elements (the “handles”, which are considered later) must be drawn too.

The highlight parameters used for drawing the components can vary between different editors of the same model (depending on whether a component is selected or not). Therefore they are kept in the editor pane objects. There are two data structures that store highlight parameters: a hash table indexed by the components, which holds the “basic” highlights, and a set that contains all currently selected components. If a component is a member of this set, it will always be drawn with the “select” highlight, otherwise the highlight is determined by the “basic” hashtable. In that way a component can revert to its previous appearance when it is deselected.

The Java2D API supports arbitrary affine transformations from user to screen coordinates, so zooming can be quite easily implemented. The necessary code is contained in the class ZoomPane, which defines a viewport to a user-defined coordinate system. Subclasses can draw into this coordinate system by overriding the abstract paintZoomed method, as the EditorPane class does. The display classes provide method calls for setting the zoom factor and the viewport position, but they do not display UI elements to control them; this must be done by the controller part of the editor. The current implementation features tool buttons for zooming and the viewport can be moved by dragging the mouse while holding down the middle button. For future versions it would be nice to have “ruler”-controls alongside the canvas border that indicate the viewport position and can be used for zooming an panning.

4.2 Modifying component parameters

Components are manipulated by means of so-called “handles”. A handle is a draggable user-interface object whose position, described by its “reference point”, is linked to the parameters of a component: If the handle is moved, the parameters are adjusted; if the parameters change – usually because another handle was moved – the handle’s position is updated. This mechanism is well suited to describing the typical behavior of a drawing tool. Consider for instance the eight small boxes that are used
to manipulate a rectangular frame in many applications. As a DIAGEN component, the frame would be defined by four parameters: x and y coordinates of the upper left corner, width and height. The handles would correspond to boxes at the four corners and in the middle of each side. When the handle in the lower right corner is dragged, the width and height parameters change, this changes the appearance of the frame and also the position of the handles for the right and lower side, which must remain in the middle of their respective sides.

Obviously, if a component is displayed in two editor windows, it could have handles in one window but not in the other. Consequently, handles are part of the editor view and not of the editor model.

The abstract class Handle provides the basic implementation for all handles in DIAGEN. This class registers itself as an event listener for the model and filters out change notifications from the associated component. Subclasses only need to implement the graphic rendering and the connection to the parameters. Usually, a handle is only linked to a single component, but the Handle class also supports handles that manipulate several components simultaneously; this is needed for the handles that make it possible to move a selection that consists of multiple components. A handle is registered with the editor pane as a listener for component events so it can react to changes of the associated component(s).

The UML message sequence diagram in Figure 5 presents the message flow that takes place when the user drags a handle and the mouse handler calls the moveHandle method of the editor pane:

First the graphic display of the handle is moved: it is erased (using XOR-mode drawing), the reference point is updated and then it is drawn again. Then the effects on the parameters are evaluated by calling the updateComponent method. This method first sets the values of a number of parameters according to the handle’s new reference point\(^1\) and then notifies the constraint handler that this set of simultaneous changes is complete, so the changes are passed to the constraint manager (see Section 3.2).

As a result of its computations, the constraint manager will then execute the actual modifications of the parameters through csetValue method calls and the component(s) that own the parameters are notified. When a component learns that one of its parameters has changed, it schedules itself to be revalidated. During this revalidation process, the visual representation of the component is recomputed and the aspects that are relevant for the scanner (positions and size of the attachment areas) are updated. The incremental scanning process, which then further updates the SRHG representation, is discussed in Section 5.3.

The whole process that has been presented here occurs in different stages that are processed by different threads. Those multithreading aspects of the DIAGEN editor will be covered in Section 6.1.

### 4.3 Event processing

After the model has been modified by editing, the event handling mechanism must be invoked so that the change is reflected in all displays of the model. The message dia-

---

1. This step is obviously dependent on the concrete function of the handle, therefore handle subclasses must implement the abstract updateParams method which executes this.
gram in Section 6 illustrates the event flow that takes place after a component has been revalidated as shown above.¹

The component notifies the model object – the central source for event distribution – and as a consequence a DiagramComponentEvent object is generated, which stores the information that is relevant to describe this event, that is, the modified component and the affected display area; the display area is computed as the union of the compo-

¹. This event flow is similar to that which occurs after a component has been added or deleted.
FIGURE 6. Event processing after a model update
Diagram editing

Before and after the modification. This event is given to the model’s event distributor, which broadcasts it to all editor panes that display this model. Actually, the event is not given directly to the editor pane but to the event distributor that is associated with it.

As has been mentioned in Section 3.4, the event is itself responsible for detecting the correct listener type. The editor pane’s event distributor does not implement the correct listener interface for diagram component events, so the transmission of the event is delegated to the superclass (DiagenEvent). Because the event distributor implements the general EventTransmitter interface (and no specific listener interface), the event is handed to it through its fireEvent method and is then in turn passed on to all listeners that connect to this event distributor. In this manner, several event distributors can be chained together to provide multiple access points for the event flow.

The event listeners that finally connect to the editor pane’s event distributor are the editor pane itself and its handles. Those classes implement the specific listener interfaces for component events and are therefore notified by calling the componentModified method. Upon receiving the event, the editor pane reacts by asking the Swing window system to repaint the respective screen area. When a handle that belongs to the modified object gets the event, it recomputes its position (and possibly its shape) from the new parameter values and initiates a repaint of the affected screen area.

It can be seen from the above explanations that the information flow between handles and parameters goes both ways. Dragging a handle (the “selected” handle) on the screen modifies the parameters; this turn causes an event to be sent and leads in turn to the updating of other handles. Obviously this, last step must be omitted for the selected handle itself to avoid an infinite loop. Instead, the selected handle always follows the drag pointer. Only when the dragging action is completed and the handle is released, its position is again updated from the parameters. To give an example of the behavior that results from this mechanism, consider the case of a handle that is used to manipulate the radius of a circle:

The position of this handle is calculated from the parameters as (xcenter+radius, ycenter), i.e. the handle will be positioned at the “3 o’clock” point on the circle outline. When the handle is now dragged on the screen, the radius is recalculated as dist(ref-point(handle), center). Therefore the circle outline will always be adjusted to intersect the handle, which can be moved to any position. When the handle is finally released, its position is recalculated from the new value of the radius, so it will “snap back” to the right of the circle’s center. The selected handle should react to user actions immediately to give good feedback. That is why drawing is not achieved by the standard Java AWT repaint mechanism, which employs a separate repaint thread, that updates a whole portion of the window area; instead this is done by XOR-mode painting, so that the underlying diagram parts do not have to be repainted as well.

4.4 Editing concrete component subtypes

Apparently, the specific behavior and appearance of a handle depends on the concrete type of component that it is linked to, therefore the handle class does not define how a handle is to be drawn and how it should interact with parameters. For some common cases, however, the DIAGEN framework provides specialized library classes: These

---

1. This is done by giving the handle object a field that indicates its “active” state.
implement a standard graphic appearance as a small black square (the BoxHandle class), the behavior of a handle whose position is directly linked to a point object in the parameter model (the class PointHandle) and also a handle that is used to translate a component or a group of components (the MoveHandle class).

As it is to be used in an editor, a diagram component is required to implement a few more methods than merely those required for the interaction with the model; the EditableComponent interface extends the component with those methods that are required for manipulation: When selected, the component of course needs to create suitable handles; moreover it may also make other component-specific editing facilities available; for this purpose the hasFocus and lostFocus hooks are provided. The current editor implementation only supports insertion of additional menu items to give the user access to component-specific manipulations. A more convenient way of making them accessible future versions would be to have modeless dialog boxes, e.g. for text properties (font, size etc.), that are activated by the mentioned hook methods when a component of the corresponding type is selected.

A last requirement for all editable components is that they support translation as a basic editing operation; i.e. they can be moved on the drawing plane without their shape or other properties being affected, except for possible adjustments caused by the constraint manager. This behavior is captured in the Movable interface, which defines an object having a virtual “reference point” that can be read and set. A typical component has parameters that either define a point or a distance on the drawing surface, so the moving operation can usually be implemented by translating all the point parameters while keeping their relative positions\(^1\) and leaving the distance parameters unchanged.

### 4.5 Selecting multiple components; cut and paste

By defining the Movable interface as a separate superinterface of EditableComponent, it is easy to support the selection and moving of a whole group of components: All selected components are kept in a Selection object, which is itself a movable object. If the selection contains only a single component, the editor displays the handles that correspond to its specific type. However, if multiple components are selected at the same time, then only a single MoveHandle is displayed. This handle reacts to changes in any of the selected components and dragging it moves the whole selection object, which in turn translates all the contained components accordingly.

The selection object is also responsible for cut and paste actions using the system clipboard: The selected components are wrapped in a special object of type ComponentSelection to indicate that they are intended to be used only in the DIAGEN framework; the general utility class ClipData then takes care of wrapping this object in a format that conforms to the requirements of the Java Data-Transfer API [Java 2 SDK]. The cut and paste mechanism should be able to transfer objects between different Java virtual machines running on the same computer, which makes it impossible to simply pass object references. Instead of this, the ClipData class uses the Java serialization mechanism to convert the component selection being transferred into a general format that can be transferred across different platforms. As a desirable side effect, the deserialization process produces a copy of the affected objects instead of

---

1. For this purpose, the StandardComponent class implements a convenience method movePoints.
returning the originals, so that copying process does not have to be coded explicitly. Unfortunately, the Swing implementation does not yet enable data transfer across multiple virtual machines, but this ability is expected in the future.

In the current implementation, components can be exchanged between diagrams of different types. Unfortunately this also permits insertion of such components into a diagram that are not supported by the edited diagram type and thus cause runtime errors during the analysis. Therefore it would probably be desirable to include the diagram type in the clipboard data and forbid cut and paste across different types.

5 The formal syntax layer

Figure 7 depicts the structure of the main classes of the formal syntax layer and their relation to the component layer. The diagram shows that subclassing is used to extend the key parts of the component layer: All the components must implement the GraphComponent interface, which inherits from the DiagramComponent interface. The class BasicGraphComponent subclasses the standard implementation BasicComponent and implements this interface. The corresponding model object is provided by the GraphModel class which extends the Model class. Finally, the diagram type for such a graph model has to be a subclass of GraphDiagramType, the class that extends the DiagramType with the properties that are required for the SRHG creation.

The purpose of the formal syntax layer is to interface with the analysis module of DIAGEN. The input required for the semantic analysis is a spatial relationship hypergraph consisting of attachment nodes, which are connected by component and relationship edges (cf Section 2.2). This connection is defined by the SRHypergraph interface, which must be implemented in the hypergraph analysis. This interface contains methods for adding edges to or removing them from the SRHG. Nodes are added and removed implicitly along with their connecting edges. As several changes often occur at the same time, the parsing structures are not automatically updated after each of these operations; instead a method is defined that explicitly initiates the reduction and parsing process after changes have been made.

5.1 Components and attachment areas

The GraphComponent interface does not add much to a diagram component; it only serves as provider of an access point for the corresponding component edge, the attachment areas and possibly also for user-defined attributes that might be needed in the high-level model (such as an editable text label).

A graph component must react correctly to any update events to keep the SRHG in accordance with the diagram model. This means that the revalidation process must be extended to propagate updates to the formal syntax layer; this is done by the BasicGraphComponent implementation. Concrete components (subclasses) should always be derived from this class and only need to implement two methods, one that defines the edge label and a second that creates the attachment areas.

Attachment areas are necessary to connect components and to form larger structures; they serve two purposes:

1. Section 7.1 gives a complete overview of all the classes and interfaces that are involved in defining a diagram component.
A component can have multiple attachment areas, possibly with different names, as different regions of interaction convey different semantic meanings: for instance it makes a difference whether an arrow is connected to a box at its head or at its base.

The spatial extent of an attachment area serves as a hint to speed up the detection of spatial relations: only attachment areas whose extents intersect are tested for possible relationships.

The relation test remains the critical criterion for establishing a relationship between two attachment areas. This means that the diagram representation should remain unchanged if the spatial extents were expanded and even if every pair of attachment areas was tested. Therefore we can restrict the form of spatial extents to rectangles (i.e. bounding boxes), which allow for fast intersection testing. It is not
required that the extent of an attachment area lies inside the component’s visual representation, although this would usually be the case.

Attachment areas are represented by objects that are derived from the abstract class AttachmentArea. The position of an attachment area on the drawing plane is made available to the parser by the associated node, because this information might be needed for specialized productions of the hypergraph grammar. A component’s attachment areas are recomputed whenever the component is revalidated, after its visual representation has been updated. The type of an attachment area (with respect to the scanner) is defined by a string label rather than its Java class so that a single class can map to several types for the purpose of relation detection. For performance reasons, these labels are internally translated into integer numbers using a global symbol lookup-table (AttachmentArea.names).

While the label of an attachment area can be given as a constructor argument, the extent of the bounding box must be defined in subclasses by implementing the computeBounds method. Two general-purpose types of attachment areas with bounding boxes can be found in the library package:

- A PointArea corresponds to Point2D object (usually a ParametricPoint2D defined by two parameters) and its extent is computed as a small square around that point. This type of attachment area is used e.g. for the endpoints of a line or arrow.

- A ShapeArea corresponds to an entire component and its extent is computed as the component’s bounding box and extended slightly to account for “near misses”, e.g. components that should touch but are a few pixels apart.

All the attachment areas of a model are referenced by an AttachSet object. This class is responsible for detecting spatial relations and keeping the SRHG updated. The scanning process that creates the SRHG is described in detail in Section 5.3.

5.2 Spatial relationships

A spatial relation is defined by a predicate on a tuple (pair) of attachment areas which tests their labels and the parameter values of their corresponding components.¹ For instance, a relation “ArrowInside” between an arrow head and a circle area could be described as:

\[
\text{ArrowInside}(a1, a2) := \\
a1.\text{type} = \text{“CircleArea”} \text{ and } a2.\text{type} = \text{“ArrowHead”} \text{ and } \\
distance(a1.\text{component}.\text{center}, a2.\text{component}.\text{end}) < a1.\text{component}.\text{radius}
\]

This notion is captured in the abstract class SpatialRelation that comprises two attachment area names and a boolean test function, which only needs to be defined on attachment areas of the required labels. Labels are tested automatically, but the test function has to be defined by subclassing and may contain arbitrary Java expressions. Accessing the component parameters obviously requires typecasts, so the programmer should make sure that the attachment labels determine the type of the components sufficiently to prevent casting errors. Also, when defining a relation test, it should be taken into account that real diagrams satisfy mathematical properties only up to a certain degree; e.g. two parameter values should be tested for equality with \(p1-p2 < \text{epsilon}\)

¹ Note on terminology: This section distinguishes between “relations” as defined here and “relationships” which denote single tuples (pairs) of related objects.
rather than
p1=p2.
To give relations access to component parameters, it is usually required that the parameter fields of a component class are declared as public.

All spatial relations that are meaningful in a given diagram type are kept in a SpatialRelationSet object. The scanning process requires quick retrieval of all relations that are possible between two attachment areas with given labels, therefore the spatial relation set organizes the relations into a hashtable, which is indexed by a combination of the two corresponding attachment labels. The hashtable contains lists of relations, since there may be several relations that can occur between a given pair of attachment areas. The GraphDiagramType class extends the basic diagram type with such a spatial relation set and also with an (abstract) function for creating a suitable reducer/parser that implements the SRHypergraph interface.

5.3 The scanning process

All the classes presented above have been introduced with the purpose of creating a higher-level representation of the diagram in the form of a spatial relationship hypergraph. We will now turn to the description of the scanning process that generates and updates this representation. Figure 8 shows another view of all the classes that are directly involved in this process.

FIGURE 8. Representation of the SRHG
The basic parts that make up the SRHG are nodes and edges; their basic properties are defined in the `diagen.hypergraph` package and are of interest only to the hypergraph analysis module. Nodes have a one-to-one correspondence with attachment areas in the diagram; they are created by a factory in the hypergraph package and their representation class is unknown to the editor. Edges come in two flavors: component edges, which directly correspond to visual components on the drawing surface and relation edges, which describe the spatial layout of components. Although the hypergraph analysis does not distinguish between component and relationship edges, the editor represents them as specialized classes `ComponentEdge` and `RelationEdge`, which contain additional information about the diagram components or attachment areas they correspond to.\footnote{This makes it possible, for example, to find the components that correspond to a set of semantically related edges and display them in the same color.}

While the creation of a component edge for every component is straightforward, the interesting part of the scanning process is the detection of relationships between attachment areas that lead to the creation of relation edges. Spatial relationships are detected in a three-step process:

1. Find all pairs of intersecting attachment areas.
2. For each of these pairs, check the attachment area types and find all relation definitions that match this combination.
3. For each of those relation definitions, perform the relation test on the pair; if the test succeeds, a relationship has been found.

As editing actions typically change only a small portion of a diagram, this process should be performed in an incremental fashion. It is encapsulated in the `AttachSet` class which manages all attachment areas and relationship edges and is responsible for updating the SRHG.

A `RectIndexedSet` data structure holds all attachment areas and provides the functionality for detecting intersections. Whenever an attachment area is added to the `AttachSet`, all intersecting set members are retrieved and tested for possible relationships using the three-step process mentioned above. When an attachment area is removed from the set (because the corresponding component has been deleted), all relation edges are checked and those that were attached to it are deleted. This process could be accelerated by using a hash index or by keeping links to all associated edges in the `AttachmentArea` class itself, but care must be taken to update the index (resp. the links) correctly, because every relationship edge would be referenced twice.

Modification of a component and its attachment areas is currently handled by removing all associated edges and starting the detection process again. This step could also be optimized; however, even if relationships persist through component modifications, the hypergraph analysis needs to be revalidated, because it might depend on node position information that has changed. The current approach avoids this problem because those relationships are deleted and recreated and thus also parsed again.

If required, the `AttachSet` can also rescan all relationships from scratch. This is done whenever the user switches from simple mode back to intelligent mode and whenever a new model is loaded into the editor.

The scanning process takes place as part of the update process that propagates parameter changes upwards in the model. However, due to the fact that the parsing process,
for which the SRHG is generated, is a relatively time-consuming procedure, updates to the hypergraph model are handled somewhat differently from updates to the visual representation: The visual representation should reflect the effects of ongoing editing actions. That means, while a handle is being dragged on the drawing plane, constraints and parameter values are continuously being adjusted and the component shapes with them. In contrast to this, the incremental scanning and parsing process is only executed after an editing action has been completed, i.e. a handle has been dragged to its final position and the mouse button has been released. This improves the interactive response time, and at the same time it seems desirable that spatial relations appearing temporarily during an editing action do not immediately cause parameter constraints; otherwise for example dragging one component over another might cause them to stick together as the editor tries to preserve such transient relations.

Therefore, whenever changes occur on the lower layers, the GraphModel class remembers the modified objects and only initiates the scanning when it is notified that the editing action is complete. The revalidation of a component causes a notification message to be sent to the model (see the message diagram in Figure 5). The GraphModel class extends the basic model’s reaction and adds the component to a “modified”-set. Similar sets are kept for added and removed components.

Figure 9 summarizes the message flow that takes place after an editing action has been completed: The editor pane notifies the model by calling the changeComplete method. The graph model then makes a copy of the “modified”-set to minimize code that needs to be synchronized\(^1\) and calls the changed method of the attach set for all attachment areas of modified components.\(^2\) The attach set removes all relations that reference the attachment area, reports the new bounding box to the RectIndexedSet and asks it for all elements that intersect with this new box. For every one found,\(^3\) all the spatial relations that conform to the two area type labels are retrieved from the spatial relation set and their test method is called; if the test returns true, a relation edge is created and added to the SRHG. Finally, after the graph model has processed all modifications, the update method of the SRHG is called to trigger the reducing and parsing.

5.4 Support for the editing process

As has been mentioned in Section 2.1, the editor module only interacts with the parameter and component layers of the model and it is mostly independent of the formal syntax and parsing techniques. The analysis results are indirectly linked back to the editor by the generated constraints. However, the discovered structure can also be used directly in the editor to simplify the user’s task:

The SRHG interface defines a getSubgraphs method, which returns the recognized top-level structure of the diagram. The result of this method is a collection of sets of SRHG edges that make up different parts of the diagram. If the parser was able to reduce the SRHG to a starting symbol, then the whole diagram forms a single entity. Otherwise the returned edge sets represent disjoint correct subdiagrams that could be

---
\(^1\) Multithreading aspects are covered in the following section.
\(^2\) Adding and removing of components is handled analogously except for the fact that corresponding component edges would be added to or removed from the SRHG at this stage.
\(^3\) This excludes the modified attachment area itself; reflexive relations would not make sense in this context.
FIGURE 9. Updating the SRHG

The formal syntax layer
found during the parsing process. Indicating those parts of the diagram should help the user to find the modifications that are necessary to transform the incorrect diagram into a correct one.

The SubgraphSupport class of the editor module uses this parsing information. It listens for events that are sent out from the SRHG and passed along by the model and the editor pane. Whenever an event indicates that the parsing structure has been updated, it retrieves the new structure information from the SRHG and highlights the components (which are accessible from the returned component edges) accordingly.

For future extensions, it would also be conceivable to use structural information from the analysis process to implement “smart” selection of groups of components that belong together in a syntactic sense.

6 Other implementation issues

6.1 Multithreading in the editor

The preceding sections have shown at several points that the propagation of model updates from the parameter layer to the hypergraph analysis is quite a complex process. To guarantee an acceptable interactive response time for the editor, this process needs to be separated into several stages with a decreasing priority for recomputation from the visual level to the hypergraph analysis. Consequently this recomputation has to be carried out by a pipeline of multiple threads with queues between them to pass on the jobs. The main goal of this design is to cut down interactive response time, but an additional benefit of using such a pipeline structure is that it is sometimes possible to coalesce multiple jobs in the waiting queues and thus avoid repeated recomputations. The multithreading approach followed here was strongly influenced by the classic paper from Xerox PARC on multithreading in interactive environments [Hauser 93].

To conveniently express this pipeline paradigm, the DIAGEN framework includes some general utility classes. The abstraction that other classes usually deal with is a “worker”, which is given “jobs” (executable objects, which must implement the standard Java Runnable interface) and processes them asynchronously. The Worker interface defines methods to start the worker and tell it to terminate; it also provides a synchronization method completeJobs, which can be called if another thread wants to wait until the worker has completed all the currently assigned jobs.

The SequentialWorker class implements this interface: It contains a waiting queue that stores incoming jobs and a single worker thread that loops forever trying to get jobs from the queue and execute them. The completeJobs method is simply implemented by appending a special “synchronization job” to the queue and waiting on a flag in this job to be set. The flag is initially off and is turned on by the synchronization job when it executes; at this point all other jobs must have been completed. Currently there is no other implementation of the Worker interface, but it is conceivable, for example to define a worker class that starts a new thread for every assigned task.

The JobQueue class implements the waiting queue; it holds a sequence of jobs and synchronizes access to it. A thread that tries to retrieve a job from an empty job queue is blocked until a new job has been added to the queue. A new job is only appended to a job queue if it is not “equal” (in the sense defined by the corresponding object) to any job that is already in the queue. To make it easy to use this equality concept, a util-
ity class `AbstractJob` is provided: this class inherits from the `Runnable` interface and can thus be used for defining a job, but it delegates equality tests to another referenced object. As an example, a job that recomputes the visual representation of a component takes this component as an equality reference.

The typical coding pattern for delegating work to another thread looks as follows:

- create and start an appropriate worker
- encapsulate the work to be delegated in a "run" method of an anonymous class.
- if jobs can be merged, make the anonymous class a subclass of `AbstractJob` and give it a suitable reference object for the equality test
- tell the worker to process the job object.

Using these general utility classes, the update pipeline for the model works as follows:

The main thread of any Java application with a graphical UI is the event handler thread. This thread receives events from the UI, for example, when the user drags the mouse to move a handle. Screen refreshing is usually done by a special high-priority repaint thread inside the UI toolkit. But to give immediate user feedback without having to redisplay an entire portion of the application window, the event handler moves the currently selected handle by directly accessing the screen using XOR-mode painting. The only remaining task of the event handler is to notify the constraint manager about parameter change requests resulting from the new handle position.

The inner workings of the constraint manager are not part of the editor framework, but it should be implemented in a separate thread which computes the constraints and modifies parameters accordingly. The constraint manager is called from the event handler thread, therefore it is essential that it returns quickly and delegates complex computations to a worker thread. Otherwise the UI would freeze while the constraint solving takes place. The diagram components are automatically notified about parameter changes and their revalidation is given as a job to a "revalidator" worker for the model.

Here the work-saving effect of the job queue inside this worker comes into play: In most cases a user action does not change just one parameter of a component; for example the x and y coordinates that define a point are usually modified together. A simple single-threaded notification mechanism would thus cause multiple subsequent recomputations of the visual representation. In contrast to this, the update pipeline combines all requests with equal jobs (i.e. jobs that have the same diagram component as an equality reference) when they are entered into the queue for the revalidator.

The revalidator thread, which is part of the `Model` class, gets components (jobs) from the queue and computes the new shape. It then notifies and updates other handles belonging to the component and finally events are sent out, which cause the editor pane to repaint the affected area on the screen.

As has been mentioned above, updates on the formal syntax model occur less frequently than on the component model; the incremental scanning only takes place after an editing operation has been completed. To this end, the `GraphModel` class

---

1. The equality test actually requires that the two jobs are of the same Java class in addition to referencing “equal” objects. This distinguishes different jobs that are performed on equal objects.
keeps three sets, where components are entered when they are created, deleted or modified. When the model is notified that an editing operation is complete, those three sets are copied in a critical section, so subsequent editing operations do not interfere with their processing. The modifications are then given as a single job to the incremental scanner described in Section 5.3, which runs as a worker thread inside every GraphModel object. In this case merging of jobs does not occur in the worker’s queue, because a single job consists of a lot of modifications, but implementing the modification stores as sets automatically merges identical requests. The parser, which is not part of this implementation, is also run inside the scanner thread, as both scanning and parsing take place in the background and do not need to be decoupled.

6.2 Synchronizing the update pipeline

The use of such an update pipeline makes it necessary to give special consideration to the way information is shared among the different threads.

For one thing, the definition of concurrency in Java (cf. [Java Language]) allows threads to maintain local variable caches: The propagation of memory writes between different threads is guaranteed only after the writing thread has released a lock and the reading thread has acquired one. In the presented implementation this condition is always assured by the locks on a worker’s job queue, which synchronize thread interaction.

Still the problem remains to guarantee consistent and well-behaved access to variables. For the presented implementation we have adopted a rather relaxed approach to synchronization; in borderline use cases, some temporary inconsistencies may occur, as long as they do not lead to runtime errors and are quickly superseded by subsequent updates.

In contrast to the previous implementation [Schmoelz 97], no complex locking schemes like reader-writer locks have been utilized to augment the standard Java object-monitor mechanisms; this might restrict possible parallelism in some rare instances but we believe that those cases do not justify slowing down the common case through the additional complexity. Several objects are accessed by multiple threads and require locking considerations:

Parameters are written by the constraint manager and read by the revalidator thread. The underlying double values are declared as volatile to ensure atomic access. Usually it should not be necessary to ensure atomicity of a set of parameter changes. If changes occur in quick sequence, it may be possible that the revalidator thread reads intermediate states of a component’s parameters, i.e. some of them have already been updated again and some have not. Conforming to the requirements stated above, this inconsistency is temporary, because the additional pending parameter changes will quickly cause another revalidation of the component. The incorrect visual representation should not be apparent to the user, especially since the editing actions cause only small incremental changes to parameter values.

1. Both locks need not be identical, neither need they refer to the modified object. Different semantics apply for values that are declared as volatile.
2. for example editing a diagram while a view on the same model is being updated by the window system for external reasons
3. The specification in [Java Language] permits variables of type double to be accessed nonatomically with two memory reads.
If it should be possible\(^1\) that reading inconsistent parameter values could lead to run-time errors (e.g. division by zero) or other unwanted behavior, then it is the responsibility of the constraint manager to lock the parameter owners (the component) and execute atomic changes on a set of parameters. The recomputation of the visual representations, which reads the parameters, is also synchronized on the corresponding component. In addition, any other access to the visual representation also requires locking the component. This ensures consistency between the revalidator and the repainting mechanism. Of course the main component list of the model requires synchronized access, too.

It is currently assumed that interdependent changes to a model are not being made simultaneously in two editor windows. While the program tolerates this behavior, the constraint manager will process an undefined mixture of both parameter changes and the result will possibly not reflect any desired modification.

The last set of shared objects are the node and edge attributes, which are modified by the revalidator thread and read by the scanning and analysis process. The attachment bounding boxes as well as the node positions\(^2\) are changed atomically; again there is the possibility that higher stages of the pipeline might read some attributes that have already been changed again and some that are about to be updated. As in the case of the parameter values, this can lead to temporary inconsistencies but those inconsistencies are again of a very short duration.

If consistency on a larger level is required, the only solution would be to prevent the user from making modifications until the analysis and constraint updating cycle is complete. It would probably be useful to introduce an operation mode for the model that enforces this alternative behavior; the operation mode could then be selected depending on the properties of the concrete diagram type and the preferences of the user.\(^3\) Also, it would be desirable to give the user an indication whether a modification has already been processed “full-cycle”\(^4\).

### 6.3 Serialization of components and diagrams

A useful diagram editor must be able to save diagrams to permanent storage and retrieve them again later. It would be possible to define a general file format for diagrams and implement methods that convert the runtime representation (the diagram model) into and from this format. An easier and more natural solution is to use the serialization mechanism of the Java language, which converts complete object graphs like the diagram model into a permanent representation with almost no additional programming effort. Serialization can also be used to exchange objects across applications, networks and platforms and the diagram editor uses it to implement the clipboard data exchange (cf. Section 4.5).

The main drawback of this approach is that the file structure is very tightly coupled to the implementation of the stored classes. Every small change to those classes that might be potentially incompatible leads to a change in the serialized format and makes previously saved files unusable. The Java Serialization Specification [Java Seri-

---

1. This depends on the concrete component implementation.
2. Those are of type float, therefore atomic access is always guaranteed.
3. One mode would force occasional interaction delays; in the other mode the editor might not always react “intelligently” to quick sequences of changes.
4. The analysis module generates events to indicate this.
alization] presents a versioning approach that can be used to circumvent those problems to a certain extent at the expense of increased effort on the side of the programmer, who is responsible for detecting incompatible class changes and convert between different versions. At the present experimental stage of the project, those mechanisms are not yet implemented.

To reduce incompatibility problems, the serialization process for diagram objects has been customized so that as few classes and fields as possible need to be serialized. As a side effect, this keeps the file sizes small. A diagram is completely defined by its type, the component types and their parameter values. Therefore the parameters are the only fields of a component that are serialized; all other member variables are marked as transient and are recomputed during the registration process. Visual representations, for example, need never be stored, so those classes are not serializable – currently most of the Java2D classes cannot be serialized anyway.

Components also do not serialize the link to their model; this link is restored when a deserialized component is registered with a model. (This is done automatically when a complete model is loaded.) This makes it possible to link them to a different model than they originally belonged to and thus provides an easy way to copy components between different models.

For the Model class, the list of components must be serialized, other fields are again marked as transient. While transient fields of components can be initialized by the registration process, because a component may not be used before it has been registered, the Model class requires a customized deserialization method that recreates transient fields. No information about the higher levels is stored; the SRHG is recreated from scratch for the whole diagram when it is loaded, and is then passed to the semantic analysis process.

For the purpose of saving edited diagrams to permanent storage and loading them again it would not be necessary to store diagram types: Diagram editors are always specialized for exactly one diagram type and cannot use diagrams of other types. Nevertheless we decided to make the DiagramType class serializable as well, so the type can be stored along with diagram models. For one thing, this makes it possible to type-check diagram files when they are loaded, so diagrams cannot be loaded into an editor for a different type. In addition, the deserialized models immediately have a link to a correct type object and are thus completely usable for other purposes without additional information.

Actually diagram types are singleton objects so it is neither necessary nor desirable to save and restore them as objects (because the deserialized DiagramType object would have a different identity from the existing singleton). Instead they contain only transient fields, so the only data that gets written out is the Java class marker.¹ The deserialization of diagram types is customized using the Java read-resolve protocol, which allows to substitute an object that is read from a stream with a different object of the same type. Using this mechanism, every diagram type object that is deserialized is immediately substituted by the singleton instance of its class and it is not possible to create two objects of the same diagram type class.

The Java system does not allow classes to inherit this sort of customization; therefore every concrete subclass of DiagramType must contain the respective code, but usually those classes are not written by hand but created by the DIAGEN generator module.

¹. This also means that saving the diagram type along with the model adds almost nothing to the diagram file sizes.
which will be described in Section 7.3, so programmers that use the framework do not have to take care of that.

6.4 General coding considerations and code performance

Like most programs written in a consequently object-oriented style, the DIAGEN editor uses a lot of temporary objects that are used only to pass around data values. This pertains particularly to points and rectangles, which are used very frequently in the code; those objects usually do not have a meaningful “identity” but are simply containers for coordinate values. As consequence of the fact that Java uses only object references, all those objects have to be allocated and later garbage-collected. The employed coding style tries to minimize the load that this places on the runtime system by avoiding allocation of new temporary objects and sharing references to those point and rectangle objects as far as possible. To prevent unwanted side effects caused by modifying such objects, the following conventions were adopted for information objects (e.g. bounding boxes or point coordinates) that are passed around in the code:

- they may not be modified by receiver
- they may not be modified by source after they have been handed out

In case the code should be extended or modified, those conventions should be observed; otherwise unwanted side effects might occur.

Unfortunately, our experience with this project showed that it is not always easy to keep in mind whether references to an object have been handed out. During the development, several subtle bugs occurred as a consequence of modifying such information objects that were referenced from other places in the code. In hindsight it would have been better to either not modify those objects at all (as in a functional programming style) or to always copy them before references are given out: The little performance gains that could result from the employed coding style are not worth the extra effort.

With regard to code performance it must be said that the class framework that has been presented in the preceding sections was not generally written with a lot of concern about efficiency. We have not done any profiling measurements so far, but one would expect that the hypergraph parser and the constraint manager should by far dominate the runtime cost of the editor and the code that was developed in the context of the presented work should have relatively little impact on the processor load.

6.5 Graphic algorithms

In case it should still be necessary to speed up the editor, specialized graphic algorithms – presented for example in [Mehlhorn 84] – could play a role in two central aspects: the refreshing of the screen display and the search for overlapping attachment areas that might be associated by a spatial relation. The important operation in both cases is an orthogonal range query:

operation 1

given

s: a set of rectangles on the plane and
r: a an additional rectangle

find all elements of s that intersect with r

1
Another important operation occurs when the diagram is not modified incrementally but the entire semantic representation needs to be rebuilt after loading a new diagram or after returning from simple to intelligent mode. In this case, all pairs of overlapping attachment areas must be determined:

operation 2

given

s: a set of rectangles on the plane
find all pairs of intersecting elements in s

Fast execution of both algorithms depends on a suitable spatial data structure for the set s.

This data structure, along with operations 1 and 2, is encapsulated in the class RectIndexedSet which holds a set of objects and their indexing rectangles. The current implementation uses a simple linear list with resulting runtime orders of \(O(n)\) for operation 1 and \(O(n^2)\) for operation 2. So far this has not shown to be a performance bottleneck and for reasonable diagram sizes (some 100 components) there should be no problems, especially since the runtime cost of the hypergraph parser dominates that of the relationship search. Nevertheless a potential for considerable improvements exists here.

Two algorithms with runtime \(O(n \log n + k)\), where \(k\) is the number of intersections, are described in the literature (for example in [Mehlhorn 84]) for operation 2. These are a line sweep algorithm and a divide-and-conquer algorithm that both use specialized geometric data structures for intermediate results. Unfortunately, a brief literature survey showed no ideal data structure for operation 1: The attachment set and the set of component bounding boxes are highly dynamic (about one insertion/deletion per query) and the possible coordinate values are not known in advance but span the entire range of floating point coordinates in the drawing plane. Both of these properties are unfavorable for the standard data structure to hold rectangles on the plane, the range-interval-tree. But considering the fact that most components of any typical diagram have a relatively equal size and even distribution on the drawing area it should still be possible to implement at least an ad-hoc solution that runs in polylogarithmic time for operation 1.

All the advanced spatial data structures are also rather complex to implement, therefore a general Java library for geometric algorithms and data structures, similar to the Java collections framework for general-purpose data structures, would be rather helpful.

7 Defining a diagram type

One of the main goals of the presented work was to provide an easy way of specifying different concrete diagram types. The design that has been described so far provides a basic framework, which contains all the functionality that is independent of the specific diagram type. This framework can now be used to create editors for a specific kind of diagrams.

1. The screen refreshing application imposes the additional difficulty that the layering order of components has to be observed in the returned list.
1. The parsing takes exponential runtime in the general case, but through careful definition of the grammar, diagram languages in practical use can usually be parsed in polynomial time.
A concrete diagram type in DiaGEN is defined by the following properties:

- the types of components that can appear in the diagram
- the spatial relations between those components that are of interest for the syntactic and semantic analysis
- constraints that preserve the semantic structure across editing actions

The specifications of the semantic analysis and constraint handling parts are not part of the work presented here. At this point it suffices to state that a generator exists (and is still being extended) that allows to specify these aspects in a specialized description language and translates that language into Java source code for the analysis module.

The following subsection explains how the framework is extended with Java classes that describe concrete components and relations; after that it is shown how parts of this Java code can be generated automatically from a specification file.

### 7.1 Concrete components and relations in Java

A component type in DiaGEN corresponds to a Java class that must implement a number of interfaces that are shown in Figure 10; a typical component class will

**FIGURE 10. Classes and interfaces for diagram components**
inherit from the abstract convenience class StandardComponent, which combines all
the necessary interfaces.

The following aspects must be defined in a concrete component class:

- the parameters that characterize the component geometry
- the visual representation that is displayed in the editor
- the information that is needed to build the SRHG: the hypergraph edge label and
  the attachment areas
- means of manipulating the component: handles and possibly component-specific
  editing actions (property dialogs etc.)

To implement a relation class, the programmer must specify

- the hypergraph edge label and the types of the related attachment areas and
- the relation test predicate.

Most of these aspects are specified using polymorphic inheritance: The concrete com-
ponent class has to implement or override methods of the base classes and interfaces.
The two basic interfaces DiagramComponent and GraphComponent serve only to
define the connection between the model and the components; as has been mentioned
before, user classes should not implement these interfaces directly but rather inherit
from the standard implementation BasicGraphComponent (and thus implicitly from
BasicComponent), as these classes contain the internal mechanism that directs the
update flow. These abstract classes then call polymorphic methods of the concrete
subclass for those parts of the process that depend on aspects of the concrete compo-
nent. A concrete component must also implement the methods defined in the
EditableComponent interface so that is can be manipulated in an editor. The
StandardComponent class does not supply any functionality but just serves as a con-
venience point for inheritance as it brings the different branches together in a single
class.

Note that not all of the aspects mentioned above must be implemented in a single
Java class; it is possible to build inheritance hierarchies of components where those
aspects are defined in several steps, shared among similar components or redefined
through overriding. In particular, it is generally useful to separate the scanner informa-
tion from the specification of parameters and manipulation properties: The latter
can often be used for different diagram types (visual structures like circles, lines and
boxes are useful for a lot of diagram types), while the former aspects (edge labels and
attachment areas) usually depend heavily on the type of diagram in question – for
example, in some diagram types components may attach to a line anywhere, while in
others only the endpoints of a line are relevant for connections.

When a concrete component is implemented by way of subclassing, some consider-
ation must be given to serialization of its member variables (cf Section 6.3) Parameters
should always be serializable; fields that cannot be recreated automatically, for exam-
ple text labels that can be edited by the user, should be serializable as well. Member
variables should be marked as transient, if it is possible to regenerate them when the
component is loaded. This refers for example to fields that hold cached values from
expensive computations. Transient fields can be initialized by overriding and extend-
ing the component’s register method.

Definition of concrete relations is comparatively easy; the programmer only needs to
write a subclass of SpatialRelation which defines the related attachment labels and
the test predicate.
7.2 Using a diagram type specification language

The previous section demonstrated how a concrete diagram type is implemented at the Java level. A main objective of DIAGEN is to permit an easy specification of concrete diagram types. Originally, the idea was to provide a complete “diagram specification language” and a generator that would take a textual specification and produce Java source code to customize the editor framework for the given diagram type. This approach worked rather well for the reducer and parser; the Java code for these parts is usually quite lengthy and schematic and lends itself well to automatic generation.

For the specification of components and relations on the other hand, it was not clear that defining an entire new language with a syntax for graphic representations, spatial relationships etc. would be a suitable approach: The customization of these classes by way of subclassing and polymorphic overriding that has been outlined in the last subsection turned out to work well enough that the eventual Java source code was almost as clear and concise as a suitable specification that could be used to generate it.

Also, as the main framework evolved, it became apparent that it would not be possible to shield the programmer completely from the underlying Java code (e.g. the event-flow and updating mechanism for handles). And even in the cases where it would have been possible to define a specification language (e.g. expressions for the relation predicates), using a new syntax that gets translated into Java expressions would have sacrificed expressiveness without offering apparent benefits. Therefore we abandoned the idea of defining a complete stand-alone diagram specification language.

Still it seemed useful to integrate at least those aspects of the specification of components and relations that are relevant for the analysis process with the already existing generator for the parser and reducer. This would allow the generator to do some checks on the consistency of the syntactic information; e.g. it could make sure that the edge labels used in the hypergraph transformations were consistent with those returned by the actual component classes.

These considerations resulted in the following eventual design for the generator:

The diagram type specification, which the generator processes, consists of

- all component and relation edge labels
- the Java class names for the component and relation classes that correspond to those edges
- the type names for the attachment areas that are associated with the components and relations
- the transformation rules for the reducer and parser (which are not relevant in the context of this discussion)

An example of the relevant parts of a specification file in pseudo-syntax looks as follows:

```
package <qualified.package>;

cOMPONENT <EdgeLabel>[<#Attachs>] { 
  <ImplementingClass>[<AttachLabel>,...], ...
};

cOMPARTMENT <EdgeLabel>[2] { 
  <ImplementingClass>[<AttachLabel1>,<AttachLabel2>], ...
};
```
... constraintmanager <fully.qualified.Classname>;
reducer { ... }
grammar { ... }

The corresponding parts of the BNF grammar that the generator uses to parse the specification can be found in the appendix.

7.3 Auto-generated source files

From such a specification the generator produces two kinds of Java source files:

- The Java code for the reducer, parser and diagram type definition classes can be completely created from the information given in the specification file.
- For component and relation classes, only the scanning aspect of the definitions (edge labels and attachment area types) can be generated from the specification. All other aspects (see Section 7.1) must be coded in Java. Therefore the generator creates only skeleton source files for those classes, which have to be hand-edited later to fill in the remaining parts.

If source files for skeleton classes already exist, they are first read in by the generator and the hand-coded changes in those files are preserved. Only the auto-generated scanning definitions are overwritten every time the generator is run, to make sure they are consistent with the reducer and parser code. These parts of the Java sources are included in special marker comments to distinguish them from code that should be preserved. This way of integrating a special definition language with hand-edited Java code also offers the programmer the possibility to take advantage of future extensions of the Java libraries like Java2D which are still evolving.

A deliberate restriction of the generator is the fact that all of the above-mentioned Java files must belong to the same package. Thus a specific diagram type corresponds to a Java package and creating it proceeds in the following steps:

4. Write the diagram type specification
5. Use the generator to process it; complete sources and skeleton files are created.
6. Hand-edit the component and relation skeletons to complete the definition
7. Compile the entire package into Java bytecode
8. The `DiagramType` class of this package can now be used to generate diagram models and editors.

Instead of hand-coding the parameters, visual representations and manipulation properties of components in step 6, these aspects can be inherited from existing component classes, so that the class in the diagram package defines only the scanning aspect and a factory. Our eventual vision is a library of abstract component classes that define all typically needed primitive components and that are only slightly customized by subclassing, so that a programmer who wants to define new diagram types can concentrate on the structural aspects. The component classes in the `diagen.editor.lib` package show examples of such general-purpose components.

All implementations of components as well as the code produced by the generator follow certain coding patterns that define relevant properties (e.g. the attachment types for the component) as class constants. By using these coding patterns it is rela-
Defining a diagram type

tively easy to obtain information about the properties of a component directly from
the compiled class-file by means of the introspection and reflection mechanisms that
the Java language offers. Diagram components thus become a sort of self-contained
and self-documenting software building blocks, similar to JavaBeans, which employ
the same mechanisms. This feature is not used in the current implementation, but it
might be beneficial for the implementation of future tools that further automate the
process of defining diagram types and diagram editors.

7.4 Implementation of the generator module

As the DIAGEN generator is a separate program that runs off-line and shares no code
with the other modules, its internal structure is not linked tightly to the rest of the
framework and is of no importance to programmers who want to use the framework.
Therefore the implementation of the generator will be sketched only briefly. The part
of the generator that is concerned with the reducer and parser specification will not be
considered at all; it is considerably more complex than the parts mentioned here.

The generator was built using the JavaCC parser generator [JavaCC]. This tool takes a
LL(k) grammar specification with embedded Java code blocks and generates a top-
down parser for the respective language. The main class Generator contains this gen-
erated parser along with the code that controls the execution and reads command-line
arguments. The parser methods that represent the various nonterminals in the recur-
sive-descent algorithm do not build an explicit syntax tree but create an object struc-
ture in memory that holds the information necessary for the code generation step. The
root of this temporary structure is a Grammar object, which references substructures
for the different parts of the specification. Information about components and relations
is stored in objects of type ComponentClass and RelationClass respectively; one of these objects is created for every implementing class mentioned in a compo-
nent or relation edge declaration. The handling of component and relation classes dif-
fers only in the code-generation step, therefore both information classes are derived
from a common ancestor ImplClass.

When the entire specification file has been parsed successfully, the grammar object is
asked to generate the output files that define the diagram type. The code responsible
for this consists mainly of a load of println statements that write out the Java source
code and fill in the information from the specification in the correct places. For com-
ponent and relation implementations, the ImplClass base class takes care of reading
in potentially existing source files and preserving the parts outside the marker com-
ments. The subclasses simply implement abstract methods that write out the actual
code for the auto-generated part and also the standard skeletons for the hand-coded
part in case the source file did not exist prior to the generator run.

Before the output files are created, a number of checks are performed to catch incon-
sistencies in the specification. Some possible inconsistencies are prevented by the
specification syntax and the fact that all of the code that defines the syntactic structure
and analysis is auto-generated. For example edge names defined by the implement-
ing classes must necessarily match those used by the parser, because they are created
from the same specification statements. Likewise, the symbol tables for the different
labels are initialized automatically and cannot be incorrect.

Some possible specification errors that are detected by the generator include:
• relation edges must have arity 2, since only binary relations are currently sup-
ported
• the number of attachments defined in an implementation class must correspond to
  edge arity that the hypergraph parser expects
• every attachment label that can take part in a possible relation must actually be
  defined by at least one component type that could occur in the diagram
• the constraint manager class exists, implements the required interface and contains
  the member fields that are needed to link it with the other modules

Of course the generator has to rely on the application programmer for correctly fleshing out the skeleton classes; the system can still be broken if, for example, the programmer does not use the auto-generated constants as attachment labels in the constructor calls that create the actual attachment objects.

8 Conclusion and ideas for future work

The revised DIAGEN architecture – including the editor and class framework presented in this paper – has already been used to generate several sample editors, among others an editor for a standardized graphical programming language for logic circuits [Minas 99]. These applications demonstrate that the concept of generating free-hand editors from formal specifications and augmenting them with a parser and a constraint-based layout system is suited for practical use. In spite of the processing costs of the parser and the constraint solver, the generated editors exhibit a reasonably fast reaction to interactive manipulations, at least for the diagram sizes that we have created so far (a few dozen components). The “intelligent” reaction to diagram manipulations proved to be very helpful for the editing process and makes our approach clearly superior to systems that only provide a separate off-line analyzing stage, such as VLCC [Costagiola 97]. At the same time, specifying a new type of diagram editor requires a lot less effort than would programming a comparable editor “by hand”, even if we assume that a suitable framework for general drawing programs is available.

For the near future, we plan to enhance the system in two main directions: One short-term goal is to transform the editor into a JavaBeans software component, which can be integrated into larger systems using graphical interface-building tools. At the same time, the interface controls (menus and toolbars) should be revised and extended to make them more user-friendly.

Besides that, the free-hand manipulation capabilities of the editor will be extended with syntax-directed editing operations.1 The two kinds of manipulation will complement each other nicely: Syntax-directed operations can be used to execute major transformations on the diagram, which would require a whole lot of elementary editing actions. On the other hand, the programmer who writes a specification does not need to worry about providing a complete set of transformations because of the availability of free-hand manipulation. It is even possible to specify transformations that deliberately produce incorrect diagrams, because the diagram syntax is analyzed by the parser in a separate step. By contrast, in editors that allow only syntax-directed editing, the diagram language is typically defined implicitly as the set of all diagrams that can be generated using the editor, so that it becomes impossible to create incorrect temporary diagrams.

Of course there are several further extensions to the system that one could think of:

---
1. The basis for this extension has already been outlined in [Minas 00].
An idea for improving the editor performance would be to take advantage the constraint solving mechanism to save scanning and parsing effort: If constraints guarantee that a specific spatial relation will be preserved then, if the “intelligent” mode is active, the scanner does not need to constantly delete and redetect that relation (and trigger parsing updates as a consequence) whenever this component is manipulated. As has been mentioned in Section 5.3, unfortunately, even if the spatial relations remain unchanged, attribute changes could still affect the scanning and parsing results because they might influence application conditions. This possibility would have to be excluded explicitly, if necessary by including additional constraints.

To make the editor usable in practice, an “undo” operation is a must: The interference of the constraint solving system can sometimes make it hard for the user to anticipate the exact effects that a local change will have on the entire diagram. This problem will be exacerbated with the introduction of syntax-directed operations, which cause even more extended transformations of the diagram. Therefore the user must be able to revert the diagram to a previous state so that she can easily try out the effect of an editing operation. The undo facility would require that the model keeps track of the changing parameter values and the added and removed components; similar to the deserialization process, the complete diagram state can be recovered from this information.

Another step that should increase the usability of the editor would be to animate the effects of the constraint propagation. Research in the visual languages field indicates that abrupt changes of the display can cause disorientation of the user. Again, the introduction of complex syntax-directed operations will necessitate that more consideration is given to those concerns. Smoothly animated transitions could make it easier for the user to see and understand all the effects of editing operations. An obvious approach would be to interpolate intermediate values between the old and the modified parameter set (without using the constraint solver) and to compute intermediate visual representations. The necessary trade-off between editing speed and animation smoothness should be controllable by the user.

Specifying diagrammatic languages and editors in a purely textual form appears to be a mismatch.1 The graph grammars and transformations that describe the syntactic model of DIAGEN are probably easier to specify in a visual editor. Using the DIAGEN system itself to implement this editor would give us more experience and insight into possible shortcomings and it would obviously increase the credibility of our approach. It seems that the “visual” aspect of a diagram language, the diagram components, would also best be defined in a graphical system. While it is relatively easy to specify simple components like circles or boxes textually, the definition of complex components with a composite visual representation (like class descriptions in an UML diagram) requires considerably more effort. A graphical component editor could be helpful here. On the other hand it is not apparent how the dependence of the component appearance on the parameter values could easily be described in a graphical way; this holds even more for a visual specification of the interaction of handles and parameters.

---

1. However, if we believe that visual programming is a useful approach to many problems, this does not necessarily imply that the definition of a visual language is itself one of these. It could still be argued that a textual description is more suitable for parts of that process, especially when dealing with dynamic and parametrizable aspects such as the diagram components in DIAGEN.
A concept that might be helpful in addressing those problems is the idea of “compound components”: General drawing programs typically support “group” objects that are treated as a unit.\textsuperscript{1} To make such a grouping process useful in the context of a diagram editor that also includes structural analysis, it should be restricted so that a component group always corresponds to a complete higher-level entity in the structural analysis. In this way, complex components can be built from elementary parts – by graphical manipulations instead of textual specification – and treated like simple components by the editor. The user does not have to care about manipulating those low-level entities, as she is only concerned with the larger building blocks. If this grouping facility is available to the end user (and not to the designer of a diagram language only), then she could also create a library of the special building blocks that she uses frequently. However, the idea also poses some problems that require further consideration:

• The elementary parts of a complex component must be linked together by constraints. Some provision must be made so that they do not simply fall apart when the editor is switched to “simple” mode. Besides, the additional work for the constraint manager could slow down the editing process compared to hand-programmed complex components, which calculate their layout directly in Java code.

• Determining which components are part of a higher-level structural entity is a straightforward task in a context-free syntactical model.\textsuperscript{2} But the reducing step of the revised diagram analysis module as well as the extension of the grammar with “embedding productions” introduce context dependencies, which have to be observed.

• When defining a compound component, there should be some way to indicate which aspects of the elementary parts (especially handles and attachment areas) should be visible to the outside. Yet the end user should not have to be concerned with internal aspects of the internal mechanisms of the diagram analysis.

It is not clear whether the user of a special-purpose diagram editor really needs the possibility of “grouping” components in that manner (syntax-directed editing will facilitate complex operations that are specially tailored to the specific diagram languages). A better approach might be to create a special meta-editor that can use grouping to define complex components for diagram languages.

\textsuperscript{1} The previous implementation of DIAGEN [Schmoelz 97] also provided such “group components”.

\textsuperscript{2} such as the previous DiaGen implementation
Conclusion and ideas for future work

References


[Gamma 94] E. Gamma et al: *Design Patterns: Elements of Reusable Object-Oriented Software*
Addison-Wesley, Reading Mass. 1994

ACM Special Interest Group on Operating Systems 1993, p. 94-104

[Java 2 SDK] Sun Microsystems: *The Java 2 SDK, Standard Edition Documentation*
http://java.sun.com/products/jdk/1.2/docs/

[Java 2D] Sun Microsystems: *Java2D Documentation*
http://java.sun.com/products/jdk/1.2/docs/guide/2d/

http://java.sun.com/docs/books/jls/html/

[Java Serialization] Sun Microsystems: *Java Object Serialization Specification*

[JavaCC] Metamata Inc.: *JavaCC Documentation*
http://www.metamata.com/JavaCC/docs/

Springer Verlag, Berlin 1984

[Minas 00] O. Köth, M. Minas: *Generating Diagram Editors Providing Free-Hand Editing as well as Syntax-Directed Editing*
submitted to Joint APPLIGRAPH / GETGRATS Workshop on Graph Transformation Systems 2000

[Minas 95] M. Minas, G.Viehstaedt: *A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams*

[Minas 97] M. Minas: *Diagram Editing with Hypergraph Parser Support*

[Minas 99] M.Minas: *Generating Semantic Representations of Diagrams*

[QOCA] S. Chok: *QOCA Project Homepage*

[Schmoelz 97] H.J. Schmödl: *Einsatz eines inkrementellen Hypergraph-Parsers in Diagrammeditoren*
Diplomarbeit at the IMMD2, University of Erlangen 1997
BNF grammar of specification files

The specification of the Parser and Reducer is ommitted

```
Specification := PackageDef Declarations ( ConstraintMgr )? Reducer
Grammar <EOF>

PackageDef := \texttt{package} QualifiedName ";"
Declarations := ( SRHGDeclaration | GrammarDeclaration )* 
SRHGDeclaration := SRHGType SRHGEdgeDeclaration 
( "," SRHGEdgeDeclaration ")" 
SRHGType := \texttt{component}
| \texttt{relation}
SRHGEdgeDeclaration := Identifier \{ Integer \} 
(\{ ImplementationDeclaration 
( "," ImplementationDeclaration ")"\})
ImplementationDeclaration := QualifiedName \{ Identifier \} ( Identifier \} Identifier "}? \}"
GrammarDeclaration := GrammarType GrammarEdgeDeclaration 
( "," GrammarEdgeDeclaration ")" 
GrammarType := \texttt{terminal}
| \texttt{nonterminal}
GrammarEdgeDeclaration := ...
ConstraintMgr := \texttt{constraintmanager} QualifiedName ";"
Reducer := ...
Grammar := ...
QualifiedName := Identifier ( "." Identifier )* 
Identifier := <ID>
```
Index of concepts and keywords

A
attachment areas 3, 19–21

B
bounding box 8, 20, 24, 32

C
clipboard 18
constraint solving engines 9
constraints 5, 9, 27, 39, 40

D
DiaGen
architecture 2–4
current state 38
evolution of the system 1–2
modules 2
packages 6
possible extensions 38–40
diagram components 2, 7
class hierarchy 33
displaying 8, 13
editing specific properties 18
formal syntax representation 3, 19
modifying 13
moving 18
revalidating 8, 14, 19, 27
serialization 30
specification of 34
diagram model 2–4
component layer 6
formal syntax layer 19
layers 2
parameter layer 8
serialization 30
diagrams
correct and incorrect parts 4
internal representation. see diagram model
saving and loading 29
semantics 4
types 10, 19, 22
types, defining 32
types, serialization 30

E
editor
cut & paste 18
features 1
modes 5, 9
overview 5, 11
usability improvements 39
events 10–11, 14–17

F
free editing 1

G
generator 6, 35
consistency checks 37
generated files 36
implementation 37

H
input syntax 35
geometric data structures 32
graph. see hypergraph
graphic algorithms 31
handles 5, 13–14, 17
hypergraph 3
component edges 4, 23
nodes 3, 21
parsing 4, 19, 24, 28
relationship edges 4, 23
scanning 22–24, 28

I
inheritance, specifying components by use of 34, 36
intelligent mode. see editor modes

J
Java2D 2, 8
JavaCC 37

M
model-view-controller pattern 5
multithreading 26–29

P
packages 6
parameters 2, 17
synchronized access 28
parser module 4

R
reducer module 4

S
serialization 18, 29–31, 34
simple mode. see editor modes
skeleton classes 36, 37
spatial relation
 specification of 34
spatial relationship hypergraph 4, 19
spatial relationships 3, 21
detection process 23
specification language 6, 35
contents and syntax 35
graphical specification 39
srhg. see spatial relationship hypergraph
Swing Toolkit 11
synchronization 28
syntax-directed editing 38

W
worker threads 26

X
xor-mode painting 17, 27

Z
zooming 13
Index of framework classes

A
AbstractJob (diagen.editor.util) 27
AttachmentArea (diagen.editor.graph) 21
AttachSet (diagen.editor.graph) 21, 23

B
BasicComponent (diagen.editor.model) 7
BasicGraphComponent (diagen.editor.graph) 19
BoxHandle (diagen.editor.lib) 18

C
ClipData (diagen.editor.ui) 18
ComponentEdge (diagen.editor.graph) 23
ComponentSelection (diagen.editor.ui) 18
ConstraintHandler (diagen.editor.param) 9
ConstraintManager (diagen.editor.param) 9

D
DiagenEvent (diagen.util) 10
DiagramComponent (diagen.editor.model) 7
DiagramComponentEvent (diagen.editor.model) 11
DiagramModelEvent (diagen.model) 11
DiagramType (diagen.editor.model) 10

E
EditableComponent (diagen.editor.ui) 18
EditorPane (diagen.editor.ui) 11
EventDistributor (diagen.util) 10
EventTransmitter (diagen.util) 10

G
GraphComponent (diagen.editor.graph) 19
GraphDiagramType (diagen.editor.graph) 19
GraphModel (diagen.editor.graph) 19

H
Handle (diagen.editor.ui) 14

J
JobQueue (diagen.editor.util) 26

M
Model (diagen.editor.model) 6
Movable (diagen.editor.ui) 18
MoveHandle (diagen.editor.ui) 18

N
NullConstraintMgr (diagen.editor.param) 9

P
Parameter (diagen.editor.param) 8
ParametricPoint2D (diagen.editor.param) 8
PointArea (diagen.editor.lib) 21
PointHandle (diagen.editor.lib) 18

R
RectIndexedSet (diagen.editor.util) 23, 32
RelationshipEdge (diagen.editor.graph) 23

S
Selection (diagen.editor.ui) 18
SequentialWorker (diagen.editor.util) 26
ShapeArea (diagen.editor.lib) 21
SimpleVRep (diagen.editor.model) 8
SpatialRelation (diagen.editor.graph) 21
SpatialRelationSet (diagen.editor.graph) 22
SRHypergraph (diagen.editor.graph) 19
StandardComponent (diagen.editor.lib) 34
SubgraphSupport (diagen.editor.ui) 26

V
VisualRepresentation (diagen.editor.model) 8

W
Worker (diagen.editor.util) 26

Z
ZoomPane (diagen.editor.ui) 13