Interactive Diagram Layout

Sonja Maier
Institute for Software Technology
Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany
sonja.maier@unibw.de

Mark Minas
Institute for Software Technology
Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany
mark.minas@unibw.de

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.
ACM 978-1-60558-930-5/10/04.

Abstract
We examine an approach for defining layout algorithms for diagrams. Such an algorithm is specified on an abstract level and may be applied to many kinds of visual languages. It mainly allows for incremental diagram drawing and attaches great importance on mental map preservation. With the approach, it is possible to combine graph drawing algorithms and other layout algorithms. It is capable of defining layout behavior for non-graph-like visual languages like Nassi-Shneiderman diagrams or GUI forms as well as graph-like visual languages such as class diagrams, mindmaps, or business process models. In this paper, we demonstrate that the combination of graph drawing algorithms and other layout algorithms is meaningful by presenting three visual language editors that have been created by students.

Keywords
Graph drawing, visual languages, meta models

ACM Classification Keywords
H.5.2 [User Interfaces]: Interaction styles; D.2.2 [Design Tools and Techniques]: User interfaces

General Terms
Algorithms, design

Introduction

In general, we distinguish two categories of visual languages: graph-like and non-graph-like visual languages. Many visual languages, however, fit into both categories. For instance, some parts of class diagrams show a graph-like structure, e.g., classes together with associations. Other parts of class diagrams show a non-graph-like structure, e.g., lists of attributes or the nesting of packages. Graph drawing algorithms (GA) are specifically tailored to graph-like structures, and rule-based layout algorithms (RA) [2], a variation of constraint-based layout algorithms, are usually used for non-graph-like structures. We present an approach that brings together these drawing strategies, and, hence, the possibilities of both categories are combined.

The goals of our approach are the specification of layout on an abstract level, and the possibility of reusing already defined drawing behavior. Besides, the presented approach is tailored to the interactive nature of diagram editors: the defined layout algorithms run continuously and improve the layout in response to user interaction in real-time. Predictable results that preserve the mental map [4] are favored, instead of high quality layout derived from a standard layout algorithm.

User Study

What kind of layout behavior should be defined via graph drawing algorithms? What kind of layout behavior should be defined via rule-based layout algorithms? To answer these questions, we performed a user study. We asked 7 groups of students, consisting of 2-3 students each, to use DiaMeta [3], an editor generation framework. First, each group had to create a visual language editor to get familiar with the system. Afterwards, each group had to define a layout algorithm for the visual language. They were asked to implement a standard layout algorithm following the descriptions of [7] first. Afterwards, they had to adapt the algorithm to the special requirements of their visual language editor. As a result, some layout behavior was built into the GAs themselves, while other layout behavior was defined outside the GAs. As expected, different groups defined similar layout modules. In addition, different groups defined the same layout behavior outside GAs as other groups. To show some of the students' design decisions, three representative examples are described in the following: *tree drawing* applied to mindmaps, *layered drawing* applied to business process models (BPMs), and *edge routing* applied to class diagrams. For each layout algorithm, we list the layout behavior that was defined via GAs and the layout behavior that was defined outside the GAs.

Tree Drawing applied to Mindmaps

The tree drawing algorithm was applied to the obvious tree structure of a mindmap (Fig. 1). The students decided to implement a circular and a layered layout strategy.

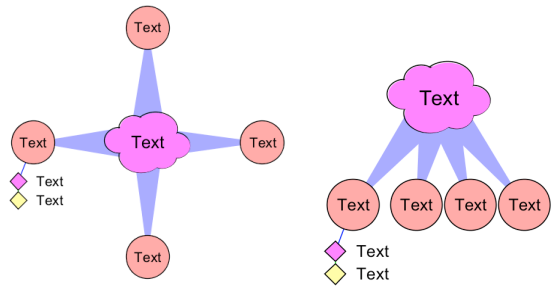

Figure 1. Mindmaps: Circular and layered layout strategy.

- Layout behavior defined via GAs: Nodes should stay near the position where the user has placed them. Besides, the different node shapes (e.g., a cloud) and sizes need to to be considered.
- Layout behavior defined outside GAs: Lists are required to remain attached to their owner nodes, and the order of list entries should be preserved. Links between different branches need to remain attached, and must be routed without crossing other nodes.

Layered Drawing applied to BPMs

The layered drawing algorithm was applied to business process models (Fig. 2). Here, flow objects serve as nodes and connecting objects as edges. The start activity is used as source and the end activity as target of an edge. To alter the drawing, the students have provided many options, e.g., the horizontal or vertical alignment of components.

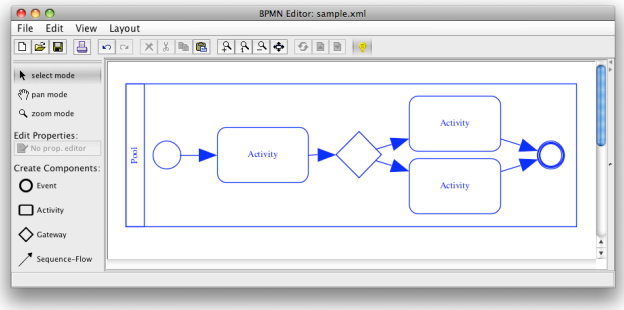

Figure 2. Business Process Models: Visual language editor.

- Layout behavior defined via GAs: Changing the diagram should not result in flow objects being moved to a (completely) different layer.
- Layout behavior defined outside GAs: To cope with nodes of different sizes, a special edge router is used. Swimlanes allow for node nesting, and the layouter should preserve the order of them.

Edge Routing applied to Class Diagrams

Edge routers are a somewhat different category of drawing algorithms, as node positions are fixed. The students applied them to class diagrams (Fig. 3). They have implemented two edge routers, which may be combined.

- Layout behavior defined outside GAs: Nodes should not overlap. Besides, attributes need to be aligned and the nesting of packages and classes needs to be preserved.

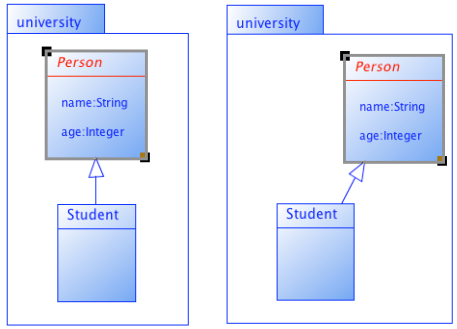

Figure 3. Class diagrams: The same class diagram before and after moving class *Person*.

Edge Follower

The first implemented strategy is an edge follower that makes sure that edges follow a component and exactly start at the contour of this component.

- Layout behavior defined via GAs: For class diagrams, edges need to follow classes, which are visualized as rectangles, or packages, whose shape is a bit more complex.

Edge Positioner
The second implemented strategy is an edge positioner, whose purpose is to route edges, e.g., to introduce bend points. The algorithm especially avoids crossing of nodes. For class diagrams, the edge positioner is applied to associations and generalizations.

- Layout behavior defined via GAs: Here again, the shape of nodes is important. To achieve a more pleasant result, the bounding box of nodes is used as the basis of the computation. Other important requirements are that edge crossings are avoided and that two edges do not start or end at the same point.

Summary
Each drawing algorithm is either called explicitly by clicking a button, or it is called automatically after each diagram change. The students who have implemented the algorithms *tree drawing* and *layered drawing* have chosen the first option, whereas the students who have implemented the *edge routing algorithms* have provided both options. This was reasonable as the edge routers perform no major structural changes.

Some requirements have been solved by changing the GAs, others have been solved outside the GAs. Most students decided to define the following layout behavior outside the GAs: preserving the *size* of nodes, the *containment* of nodes and the order of nodes (*lists*). To define such layout behavior, it is reasonable to use RAs.

Layout Patterns

Layout patterns combine GAs and RAs. They allow for defining the layout on an abstract level and for reusing these layout algorithms.

Execution of a Layout Algorithm
When the editor user moves a class, or more generally, changes the diagram, the layout engine is called and the diagram is updated. In our example, the user has moved class `Person` right (Fig. 3). During movement, the position of the attributes `name` and `age` are updated, the package `university` is resized, and the generalization follows the class `Person`. Each of these changes is performed by a different layout pattern and hence by a different layout algorithm.

Specification of Layout Patterns
Each layout pattern encapsulates certain layout behavior. It is based on a language-independant, but pattern-specific meta model (PMM). This way, reuse of layout behavior is possible.

The term *pattern* is already known in the context of layout specification [6] based on tree grammars, while we use meta models.

Internally, a diagram is represented by an instance of a language-specific meta model (LMM). In order to apply a layout pattern to a certain visual language, i.e., in order to instantiate the pattern, a mapping between the PMM and the LMM needs to be defined.

A layout pattern consists of a meta model *MM* and a layout algorithm *Alg*, for instance a GA or a RA. When instantiating a layout pattern, a list of options *opt* may be provided. Besides, a history *hist* is created, which stores the previous layout of the diagram, the changes

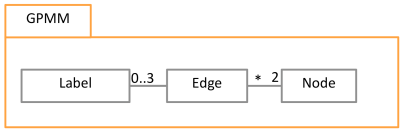

Figure 4. Graph layout meta model.

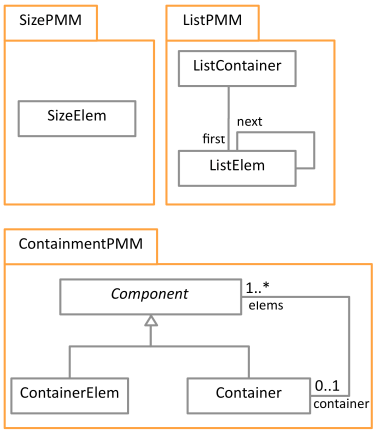

Figure 5. PMM for the patterns Size, List and Containment.

done by the user, and the changes already performed by other layout patterns.

Meta Model(s) for Graph Drawing Algorithms
To allow for reuse, GAs are integrated as certain patterns, and hence, are based on a pattern-specific meta model GPMM (graph pattern meta model). The GPMM is shown in Fig. 4: A graph consists of several components. A component is either an edge or a node. An edge has up to three labels and connects two nodes.

Meta Models for Rule-based Layout Algorithms
The meta models SizePMM, ListPMM and ContainmentPMM, on which the patterns *Size*, *List* and *Containment* are based on, are shown in Fig. 6. These patterns are responsible for preserving the correct size, alignment and nesting of components. The meta model SizePMM consists of a `SizeElem`, which stands for the resizable component. The meta model ListPMM describes a list in terms of many `ListElems`. The meta model ContainmentPMM consists of a `Container` which may contain one or more `Components`. A `Component` is either a `ContainerElem` or a `Container`.

Mappings
A pattern-specific meta model is an abstraction of the situation in the LMM, meaning that concrete classes in the PMM correspond to classes in the LMM. Usually a pattern is not exactly represented in the meta model of a diagram language. Instead, some variation can be found. Hence, the correspondence between LMM and PMM must be defined for each pattern. The mapping between the instances of different meta models can be seen in Fig. 6. Here, instances of the meta models of Fig. 4 and Fig. 5 are ListPM, SizePM, ContainmentPM and GPM. The dashed arrows indicate the transformations between different models. Every ellipse connected with a rectangle forms a pattern instance, e.g., *List* together with ListPM, or *Edge Follower* together with GPM.

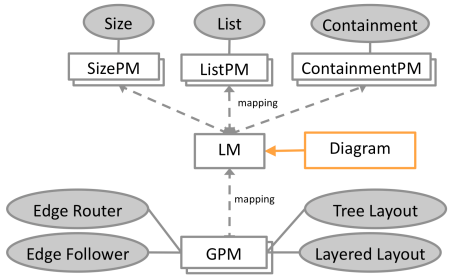

Figure 6. Correlation of diagram, LM, PMs and patterns.

Graph Drawing Algorithms
A GA gets as input an instance I_{GPMM} of the meta model GPMM, the list of options *opt* and the history *hist*. The algorithm is either self-coded or provided by a graph drawing library. The GAs *edge router*, *edge follower*, *tree layout* and *layered layout* all operate on the same GPMM (Fig. 6). GAs are usually quite complex, and performance is crucial in an interactive environment. Hence, it is reasonable to implement them, not to define them on an abstract level.

Rule-Based Layout Algorithms
A RA gets as input an instance I_{MM} of the corresponding meta model MM, a list of options *opt*, and a history *hist*. The specification of layout rules as well as strategies is based on the corresponding PMM (Fig. 5). The RAs *Size*, *List* and *Containment* all operate on different meta models. RAs are a variation of constraint-based

layout algorithms, and have been introduced in [2]. Constraint-based layout algorithms are used in many tools for drawing graphs, e.g., in GLIDE [5] or Dunnart [1]. In contrast to standard constraint-based algorithms, a more predictable layout behavior may be defined via RAs: When the layout engine is called, roughly speaking, several *layout rules* are applied to different parts of the diagram and predictably update the values of certain attributes.

Combination of Layout Patterns
Different layout patterns are combined via the application control, a language-specific control program. It decides in which order different patterns are applied and it instantiates each pattern.

Conclusions

In this paper, we have examined an approach for defining layout algorithms for diagrams. With the approach, it is possible to combine graph drawing algorithms and rule-based layout algorithms. The approach is capable of defining layout behavior for various visual languages: non-graph-like visual languages like Nassi-Shneiderman diagrams or GUI forms and graph-like visual languages, such as class diagrams, mindmaps or business process models.

We have demonstrated that the combination of graph drawing algorithms and rule-based algorithms is meaningful by outlining three visual language editors that have been created by students. Besides, the reuse of certain layout behavior is motivated. As a next step, we plan to enhance the framework to allow for the easy specification of layout behavior for a visual language editor. Then, we will ask students to define both, GAs as well as RAs, using the framework.

Our overall goal is to create a platform on which new language-specific layout algorithms that are tailored to interactive diagram drawing may be created and tested.

References

[1] Dwyer, T., Marriott, K., Wybrow, M.: *Dunnart: A constraint-based network diagram authoring tool*. In: Graph Drawing: 16th Intl. Symp. (GD'08). Volume 5417 of LNCS, Springer-Verlag (2009) 420–431.

[2] Maier, S., Mazanek, S., Minas, M.: *Visual specification of layout*. In: Graph Drawing: 16th Intl. Symp. (GD'08). Volume 5417 of LNCS, Springer-Verlag (2009) 443–444.

[3] Minas, M.: *Generating meta-model-based freehand editors*. In: Proc. of the 3rd Intl. Workshop on Graph Based Tools (GraBaTs'06). Volume 1 of ECEASST. (2006).

[4] Purchase, H.C., Samra, A.: *Extremes are better: Investigating mental map preservation in dynamic graphs*. In: Proc. of the 5th Intl. Conference on Diagrammatic Representation and Inference (Diagrams '08), Springer-Verlag (2008) 60–73.

[5] Ryall, K., Marks, J., Shieber, S.: *An interactive constraint-based system for drawing graphs*. In: Proc. of the 10th ACM Symp. on User Interface Software and Technology (UIST'97), ACM (1997) 97–104.

[6] Schmidt, C., Kastens, U.: *Implementation of visual languages using pattern-based specifications*. Software: Practice and Experience 33(15) (2003) 1471–1505.

[7] di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall (1998).