Specification of a Drawing Facility for Diagram Editors

Sonja Maier and Mark Minas

Universität der Bundeswehr München, Germany
{sonja.maier,mark.minas}@unibw.de

Abstract. The purpose of this paper is to give an overview of a drawing approach for the visualization of diagrams. The approach is tailored to editors for visual languages, which support structured editing as well as free-hand editing. In this approach, the editor developer visually specifies layout behavior. From this specification a drawing facility is generated. With the generated editor, the user may perform incremental diagram drawing at any time. When visualizing components, taking into account geometric dependencies between different components for layout computation is a challenging task. Therefore, we choose the visual languages Petri nets and GUI forms as running examples. Based on these examples, we show the applicability of our approach to graph-based and hierarchical visual languages.

1 Introduction

When implementing an editor for a visual language, layout is a challenging task. The drawing approach should produce a good-looking result and should support the user. Additionally, the layout specification should be very easy. In the following, we introduce an approach that aims at achieving these concurrent goals.

The approach supports incremental layout computation, which is triggered by user input. As a consequence, layout needs to be computed at runtime. It is tailored to this requirement, which is best known in the graph drawing context as *online layout computation* [10].

With our generic approach, it is possible to compute an initial layout. E.g. for a Petri net editor, a simple graph layout is implemented. In addition, the approach is best suited for incremental layout [7, 8], which starts with an initial layout and performs minor changes to improve it while still preserving the mental map of the original layout.

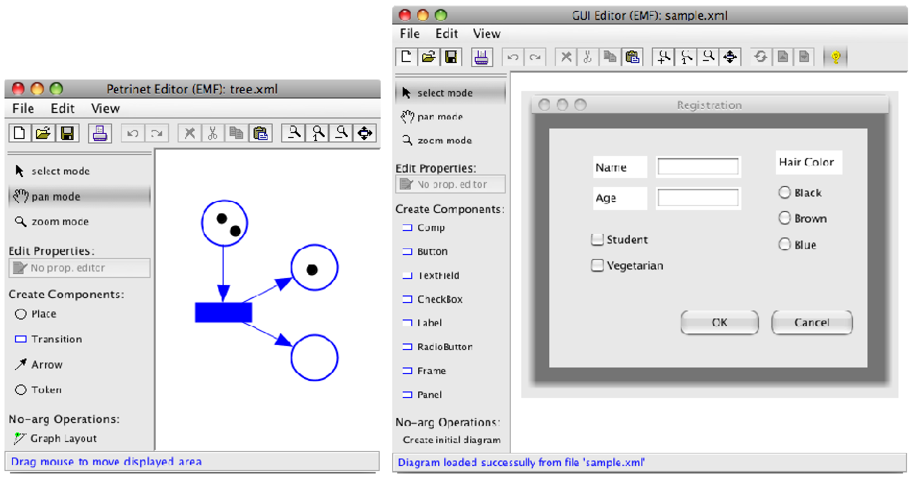
Our approach succeeds in the context of editors that support *structured editing* as well as *free-hand editing*. Structured editors offer the user some operations that transform correct diagrams into (other) correct diagrams. Free-hand editors allow arranging diagram components on the screen without any restrictions.

It was implemented and tested in DiaMeta [4], an editor generation framework. With this tool an *editor developer* may create an editor for a specific visual language, e.g. Petri nets. The result is an editor that can be utilized by the *editor user* to create Petri nets. With DiaMeta, not only editors for graph-based visual languages like Petri nets, but also editors for other visual languages like GUI forms can be created (see Figure 1).

In Section 2 we introduce two visual editors that serve as running examples in this paper. Section 3 gives an overview of the functionality our approach offers to the editor user. The layout algorithm, which is applied after user interaction, is described in Section 4. Section 5 provides an overview of the possibilities the editor developer has when he creates a layout specification. Some details about the implementation of the approach are shown in Section 6. Especially DIAMETa is introduced in this section. In Section 7 future work is summarized and the paper is concluded.

2 Running Examples

In general, we distinguish two categories of visual languages: graph-like visual languages and non-graph-like visual languages. Examples for the first category are Petri nets, mindmaps or business process models. Examples for the second category are Nassi-Shneiderman diagrams, VEX diagrams [3] or GUI forms. In the following we shortly introduce the two visual languages that will serve as running examples.



Drag mouse to move displayed area

Diagram loaded successfully from file 'sample.xml'

Fig. 1. Petri net editor & GUI forms editor

2.1 Petri Net Editor

As a representative of the first category, we choose Petri nets. In Figure 1 (left side), a Petri net editor, which was created with DIAMETa, is shown. Internally, the concrete syntax of the visual language is represented by a graph. More precisely, it is represented by a hypergraph [4], a generalization of graphs. In contrast to graphs, edges of hypergraphs may visit more than two nodes. E.g., a Petri net, as shown on the left side of Figure 2, is represented by the graph shown on the right side of Figure 2. Here, nodes are visualized as circles, *and edges are visualized as arrows and as rectangles*. Places and transitions have one attachment area (the edge "connects" one node), whereas arrows have two attachment areas (the edge "connects" two nodes).

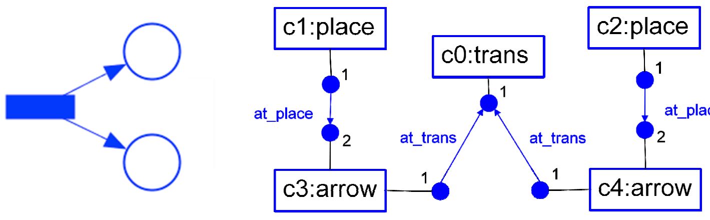


Fig. 2. A Petri net consisting of a transition and two places, and its graph model

In the graph, the edges that are visualized as rectangles represent the components contained in the diagram: one transition, two places and two arrows. The edges `at_place` and `at_trans` represent connections between arrow and place and connections between arrow and transition.

2.2 GUI Forms Editor

As a second example, GUI forms are chosen as a representative of the second category. In Figure 1 (right side), a GUI forms editor is shown. A sample GUI is shown on the left side of Figure 3, which is represented by the graph shown on the right side of Figure 3. The graph stores the information that the diagram consists of two buttons that are aligned horizontally, and of a panel that contains these two buttons. Nodes 1 and 2 express horizontal connections. Similarly, nodes 3 and 4 express vertical connections. An inside relation is expressed by the nodes 5 and 6. Buttons do not have a node 6, as they may not contain other components.

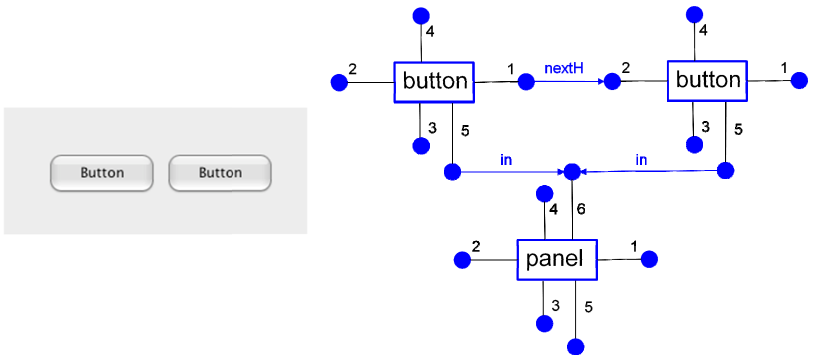


Fig. 3. A GUI consisting of two buttons and a panel, and its graph model

3 Layout Modes

After introducing the running examples, we now present an overview of the functionality our approach offers to the editor user.

The layout engine may either be called automatically each time the diagram is changed by the user, or the layout engine may be called explicitly, e.g. by clicking a

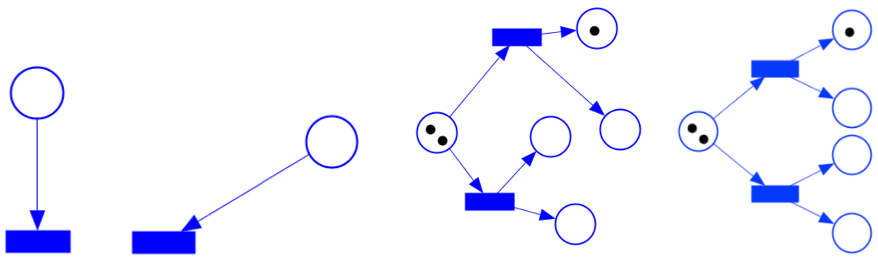


Fig. 4. Automatic & static layout

button. The first strategy is called *automatic layout* in the following, and the second strategy *static layout*. Furthermore, it may either be that a diagram is visualized the first time, or that a diagram was visualized already. The first usually happens in structured editing mode, the second in free-hand editing mode. In the first case, the layout engine does not take into account any previous layout information, and hence computes an *initial layout*. In the second case, the layout engine takes into account previous layout information, which means that an *incremental layout* is computed. In the next paragraph, we give an example of each of the four categories.

Automatic Layout

On the left side of Figure 4, a place is moved to the right. During movement, the arrow stays attached to the outlines of the components at any time.

Static Layout

After creating the diagram in image 3 of Figure 4, the user may click the button "Graph Layout" (see Figure 1). The layout engine considers the previous layout information, and draws the diagram as shown in image 4.

Initial Layout

For the GUI forms editor, we defined a structured editing operation that creates the graph shown on the left side of Figure 5. From this graph, the visualization that is shown on the right side of Figure 5 is automatically created.

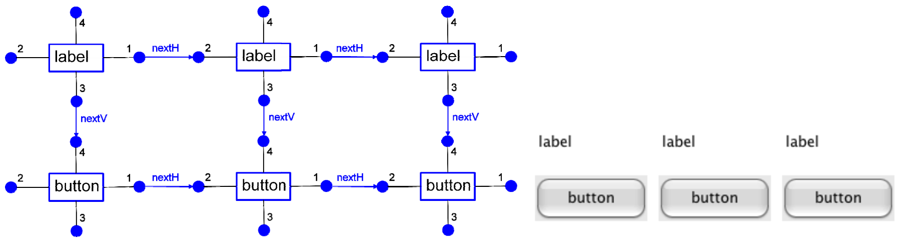


Fig. 5. Initial layout

Incremental Layout
In Figure 6 on the left, the user moves the checkbox “Red” to the right. As a consequence, the layout engine moves the other checkboxes and the surrounding panel, taking into account previous layout information.

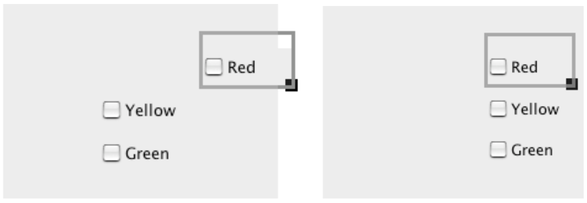


Fig. 6. Incremental layout

4 Layout Algorithm

After describing the different layout modes, we now explain what happens if the layout engine is called. Roughly speaking, several *layout rules* are applied, following an application recipe (called *application control*).

The algorithm is based on ideas described in [1, 2]. In the new approach, layout rules are defined on the concrete syntax level, whereas the application control is described on the abstract syntax level. As layout rules are defined on the concrete syntax level, it is now possible to specify them visually in the application domain.

Layout Rule
The layout algorithm consists of several layout rules. Examples for layout rules are “align buttons horizontally” or “keep arrows attached to places”. For each rule, one diagram component is given as input. This is usually a component, the user changed, or a component the layout engine changed. E.g., for the rule “keep arrows attached to places”, this is the place. Then other components that are necessary for the layout rule are identified. In case of the rule “keep arrows attached to places”, this is the arrow. The set of required components is called *pattern*. Then, if a certain *condition* is fulfilled, a particular *action* is applied. For the example rule, the condition checks if the “arrow is correctly attached”, and the action executes “attach arrow correctly”. This means that one (or more) layout attribute(s) of one (ore more) component(s) is (are) changed. In conditions and actions, attribute values may be accessed. Here, several values are available: *old values* (values before user interaction), *new values* (values after user interaction), and several *intermediate values* (values during layout computation).

Application Control
The layout rules, which are defined by the editor developer, are applied in an order, which is determined by the editor developer. The application control defines this sequence of applied layout rules. E.g., first places and transitions are aligned in a tree-like structure, and afterwards, the arrows connecting these places and transitions are updated. In addition, the order in which the layout rule is applied to different parts of the diagram usually follows an exact plan. For instance, buttons are aligned

horizontally from left to right. The layout creator can choose whether a rule is applied once, or iteratively until a certain condition is fulfilled. Besides, a rule may either be applied to just one match, or to all matches that occur.

5 Layouter Definition

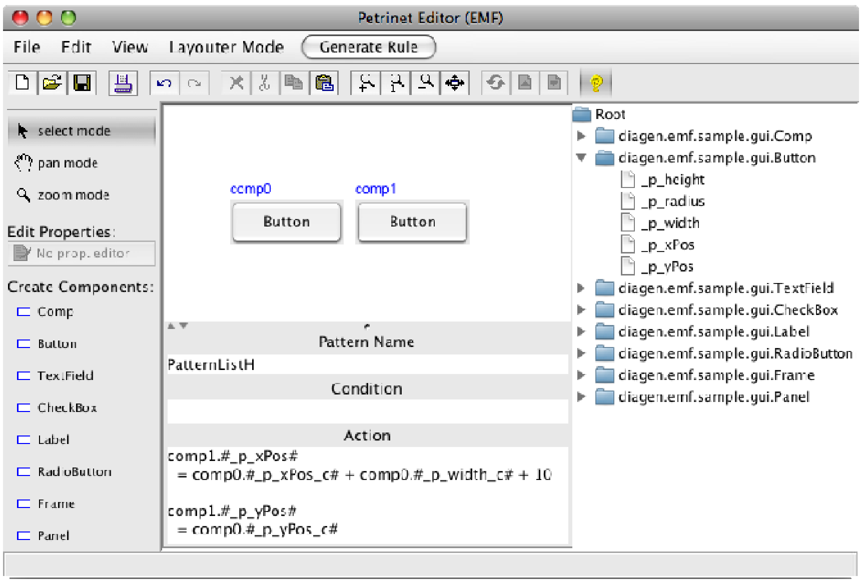
In this section we will describe, how layout rules may be defined, and what concepts are used when implementing the application control. Rules are defined visually, whereas the application control has to be written by hand. From the visual specification of rules, Java classes are generated. The generated classes, together with some handwritten classes and some framework classes form the desired layout engine, which may be included in the editor.

5.1 Definition of Layout Rule

To allow a more intuitive description of layout, we introduced a visual language for layout rule definition, which is based on the concrete syntax of the diagram language.

A generic editor for specifying these rules is provided. This language-specific editor is based on a diagram editor for a certain visual language, which does not support layout yet (as shown in Figure 10).

In Figure 7, a screenshot of such an editor for GUI forms is shown. In order to create a layout rule, the editor developer draws a pattern, and enters the pattern name, a condition and a rule in the middle of the editor (from top to bottom). For specifying the pattern, he may use the components available on the left side of the editor. In the


Petrinet Editor (EMF)
File Edit View Layouter Mode Generate Rule
select mode
pan mode
zoom mode
Edit Properties:
No prop. editor
Create Components:
Comp
Button
TextField
CheckBox
Label
RadioButton
Frame
Panel
comp0
comp1
Button
Button
Pattern Name
Condition
Action
Root
diagen.emf.sample.gui.Comp
diagen.emf.sample.gui.Button
_p_height
_p_radius
_p_width
_p_xPos
_p_yPos
diagen.emf.sample.gui.TextField
diagen.emf.sample.gui.CheckBox
diagen.emf.sample.gui.Label
diagen.emf.sample.gui.RadioButton
diagen.emf.sample.gui.Frame
diagen.emf.sample.gui.Panel

PatternListH

```
comp1.#_p_xPos#
  = comp0.#_p_xPos_c# + comp0.#_p_width_c# + 10

comp1.#_p_yPos#
  = comp0.#_p_yPos_c#
```

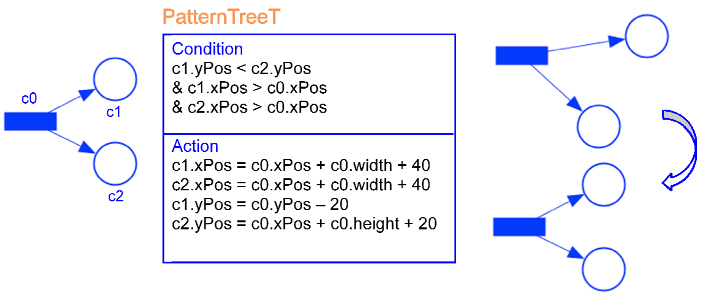
Fig. 7. Layout editor for specifying layout rules

example, he uses the component `button` twice. For specifying conditions and rules, he may access the attributes shown on the right side of the editor. In the example, he uses the attributes `xPos`, `yPos` and `width`.

Incremental Layout

To define a layout rule usable for incremental layout, we need to proceed as follows: He has to provide a pattern, a condition (which is optional) and a rule. In the example (Figure 7), the pattern consists of two buttons. The left button (`comp0`) is the component that was changed by the user. In the example, no condition is provided, which means that the rule is applied each time. The rule that is defined here updates the attributes `xPos` and `yPos` of the right component (`comp1`). Here, the new values of the attributes are used. ("`_c`" stands for changed.) If we want to use old values, we may access an attribute via `#_p_xPos#`.

Figure 8 shows another rule that modifies attributes. On the right side of Figure 8, a Petri net before and after applying a rule that updates the attributes `xPos` and `yPos` of the places `c1` and `c2` is shown. On the left side of Figure 8, the pattern is presented. Here, component `c0` is the component, which the user has changed. A simplified version of the condition as well as the rule is shown in the middle of Figure 8. The rule introduces a (rather simple) tree-like structure to Petri nets.



```
Condition
c1.yPos < c2.yPos
& c1.xPos > c0.xPos
& c2.xPos > c0.xPos

Action
c1.xPos = c0.xPos + c0.width + 40
c2.xPos = c0.xPos + c0.width + 40
c1.yPos = c0.yPos – 20
c2.yPos = c0.xPos + c0.height + 20
```

Fig. 8. Layout rule

Initial Layout

Defining a layout rule usable for initial layout is a special case of defining a layout rule usable for incremental layout: all components are marked as changed. Then the layout is computed as usual, without having initial values for attributes available. In case an attribute is required in a computation, this attribute is set to zero, and the computation proceeds.

5.2 Definition of Application Control

The definition of the application control is based on the abstract syntax of the diagram language. Up to now, the whole definition needs to be hand-coded. In future, there will be provided some standard traversal strategies. Usually these strategies follow the structure of the meta model of the diagram language. After the editor developer chooses the appropriate strategy, it would be possible to generate the control program automatically. Then, only the choice which action is applied to what part of the diagram needs to be defined.

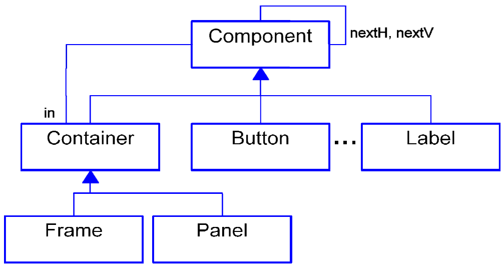


Fig. 9. Excerpt of the meta model of GUI forms

For instance, an excerpt of the meta model of GUI forms is shown in Figure 9. If a button is resized, we follow the links `nextH` and `nextV`. Following the link `nextH`, the next components are moved and resized. When following the link `nextV`, the next components are also moved and resized.

Another example is moving a panel. Then, the components, which are contained in the surrounding panel, are moved. Surrounding components (e.g. another panel or a frame) are also moved, if the moved panel would leave the surrounding component, otherwise.

6 Implementation

The approach was implemented and tested in DiaMeta [4], an editor generation framework. The editor generation framework DiaMeta provides an environment for rapidly developing diagram editors based on meta modeling. Figure 1, for instance, shows a DiaMeta editor for Petri nets (left) and a DiaMeta editor for GUI forms (right). Each DiaMeta editor is based on the same editor architecture, which is adjusted to the specific diagram language. DiaMeta's tool support for specification and code generation, primarily the DiaMeta Designer is described in this section.

6.1 DiaMeta Framework

The DiaMeta environment consists of a framework (consisting of layout editor framework and editor framework) and the DiaMeta Designer (Figure 10). The framework is basically a collection of Java classes and provides the dynamic editor functionality, necessary for editing and analyzing diagrams.

In order to create an editor for a specific diagram language, the editor developer has to enter two specifications: First, the *ECore Specification* [5], and second the *Editor Specification*. Besides, the editor developer has to provide a *Layout Specification*, which was described in Section 5.

A language's class diagram is defined by an ECore specification. The EMF compiler is used to create Java code that represents this model. The editor developer uses the DiaMeta Designer for specifying the concrete syntax and the visual appearance of diagram components, e.g., places (Petri nets) are drawn as circles. The DiaMeta Designer generates Java code from this specification.

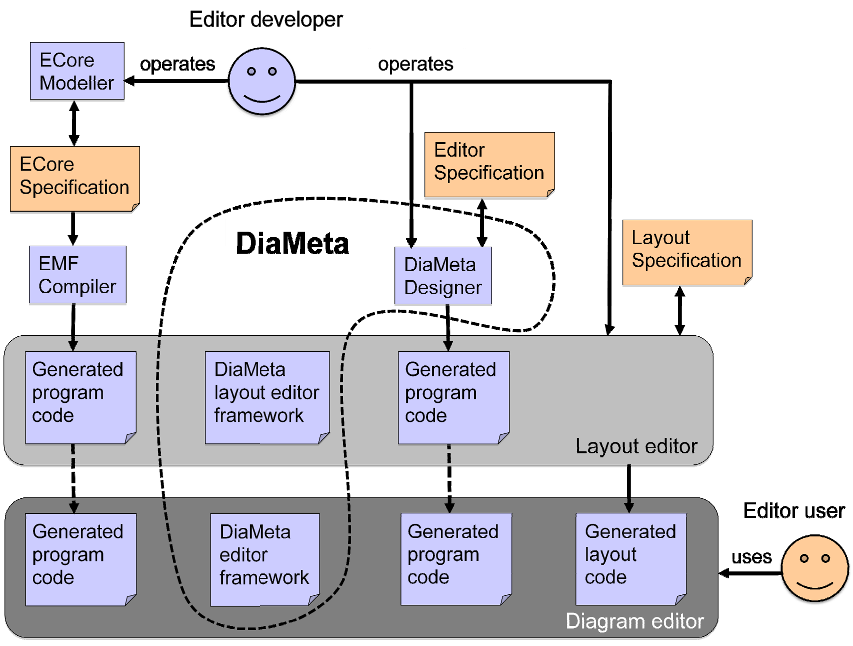


Fig. 10. DiaMeta framework

The Java code generated by the DiaMeta Designer, the Java code created by the EMF compiler, and the layout editor framework, implement a *Layout Editor* for the specified diagram language. The editor developer uses this Layout Editor for specifying the layout engine. From this specification, Java code is generated.

The Java code generated by the DiaMeta Designer, the Java code created by the EMF compiler, the Java code generated by the Layout Editor and the editor framework, finally implement a *Diagram Editor* for the specified diagram language.

7 Future Work and Conclusions

In this paper, we gave an overview of a drawing approach for the visualization of diagrams. The approach is tailored to an editor for visual languages, which supports structured editing as well as free-hand editing. Here, the editor developer visually specifies layout behavior. From this specification a drawing facility is generated. With the generated editor, the user may perform incremental diagram drawing at any time. As running examples, we chose the visual languages Petri nets and GUI forms, as representatives for graph-like and non-graph-like visual languages.

An open issue is the reusability of layout rules and application controls. At the moment, they are rewritten each time they are needed. To cope with this challenge, we plan to apply the concepts used in [11].

In this regard, we also plan to provide a way of integrating hand-written graph drawing algorithms. Right now, we are developing the graph drawing algorithms tree layout, force-directed layout, edge routing and layered drawing [6], which are tailored to the special needs of dynamic graph drawing.

Currently, we also work on a three-dimensional visualization of visual languages. In this context, we extend our layouting approach to a third dimension.

Up to now, we primarily focus on achieving "nice" results. In future, we also plan to take care of easy diagram manipulation. In this context, some user studies will need to be performed. To evaluate the "degree of mental map preservation", some metrics will be helpful [9].

References

1. Maier, S., Minas, M.: A Static Layout Algorithm for DiaMeta. In: Proc. of the 7th Intl. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008). ECEASST (2008)
2. Maier, S., Mazanek, S., Minas, M.: Layout Specification on the Concrete and Abstract Syntax Level of a Diagram Language. In: Proc. of the 2nd Intl. Workshop on Layout of (Software) Engineering Diagrams (LED 2008). ECEASST (2008)
3. Citrin, W., Hall, R., Zorn, B.: Programming with visual expressions. In: Proc. of IEEE Symposium on Visual Languages (VL 1995). IEEE Computer Society Press, Los Alamitos (1995)
4. Minas, M.: Generating Meta-Model-Based Freehand Editors. In: Proc. of 3rd Intl. Workshop on Graph Based Tools. ECEASST (2006)
5. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education, London (2003)
6. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing. Prentice Hall, Englewood Cliffs (1999)
7. Purchase, H.C., Samra, A.: Extremes are better: Investigating mental map preservation in dynamic graphs. In: Proceedings Diagrams. LNCS(LNAI) (2008)
8. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. Journal of Visual Languages and Computing (1995)
9. Bridgeman, S., Tamassia, R.: Difference metrics for interactive orthogonal graph drawing algorithms. In: Whitesides, S.H. (ed.) GD 1998. LNCS, vol. 1547, p. 57. Springer, Heidelberg (1999)
10. Branke, J.: Dynamic Graph Drawing. In: Drawing Graphs (1999)
11. Schmidt, C., Cramer, B., Kastens, U.: Usability Evaluation of a System for Implementation of Visual Languages. In: Proceedings VL/HCC 2007 (2007)