Transient View Generation in Eclipse*

Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden

Real-Time and Embedded Systems Group,
Department of Computer Science, Christian-Albrechts-Universität zu Kiel
{chsch,msp,rvh}@informatik.uni-kiel.de

Abstract. Graph-based model visualizations can effectively communicate information, but their creation and maintenance require a lot of manual effort and hence reduce productivity. In this paper we build on the concept of Model Driven Visualization by presenting a meta model for graphical views and an infrastructure for configurable automatic layout. This enables the *transient views* approach, in which we efficiently derive and draw graph representations from arbitrary models.

1 Introduction

Graphical modeling languages such as the UML and its dialects as well as domain-specific modeling languages (DSMLs) can be effectively employed for a multitude of purposes, including documentation, communication, and code generation. However, an important amount of productivity is wasted with drawing and beautification activities while working with state-of-the-practice modeling environments. Even when automatic layout is available, it is often unable to properly consider domain-specific requirements on the layout. Furthermore, the development of a graphical notation can be unnecessarily tedious, especially when it comes to the details.

Our objective is to allow *lightweight* and *transient* graphical representations, following the concept of Model Driven Visualization (MDV) introduced by Bull et al. [4]. This means that views are completely specified using model transformations, hence no manual user interaction is required. Our main contribution is a meta model for graphs, their layout, and their graphical representation. We employ this meta model both for the view model of generated graphical views and for the interface to layout algorithms. Furthermore, we present an infrastructure for automatic layout that is statically and dynamically configurable and provides the foundation for transient view synthesis. Both contributions are realized within the KIELER project,[1] which provides an Eclipse update site. A more detailed version of this paper is available as technical report [12].

This paper is organized as follows. We discuss previous work on view synthesis and layout integration in Sect. 2. The meta models and basic approaches for transient views are presented in Sect. 3. The integration and configuration of layout algorithms in Eclipse is described in Sect. 4, after which we conclude.

* This work was funded in part by the Program for the Future Economy of Schleswig-Holstein and the European Regional Development Fund (ERDF)

[1] http://www.informatik.uni-kiel.de/rtsys/kieler/

2 Related Work

The work presented here builds on Model Driven Visualization as proposed by Bull et al. [4, 2], which is an extension of the Model Driven Engineering (MDE) approach to the creation of views. This helps to lift the development of graphical tools to a more abstract level. However, Bull et al. focus on view models for different kinds of data visualization, which does not only include graphs, but also tables and charts. The *Zest* toolkit [3] employed in their contribution mainly addresses the SWT integration of graph viewers and offers only few graph layout algorithms. In our approach we go one step further and add rendering specification as well as layout directives to the graph view model and hence allow to express all details of the generated view using MDE methods.

A very simple tool for visualizing EMF models is offered by the *EMF To Graphviz* project.[2] Being restricted to drawing boxes with lists of attributes and using Graphviz as layout engine [8], this tool it is very limited in terms of rendering and layout, which makes it useful for debugging and rapid prototyping, but insufficient for more complex visualizations.

The established graphical modeling frameworks GMF and Graphiti are not well suited for transient model visualization in the sense of this paper. Both are designed for composing models by dragging and dropping figures onto a diagram canvas. They require a fully-fledged editor setup in order to simply *show* diagrams, which is a waste of resources that is felt bitterly for large diagrams. Although Graphiti maintains the description of figures in a view model (the *Pictogram* model), some characteristics, such as the bend point rendering of edges (angular or rounded), are configured in the editor code. GMF's *Notation* model has no means at all for specifying rendering primitives, but points to predefined edit part classes using integer identifiers. The arrangement of figures (*micro layout*) must be realized in Java code in both frameworks. While GMF relies on the *layout manager* concept of Draw2D, Graphiti requires to implement *layout features*. This inconsistent use of view models for the specification of graphics impedes the application of model transformations and other model-based techniques for full-automatic view generation.

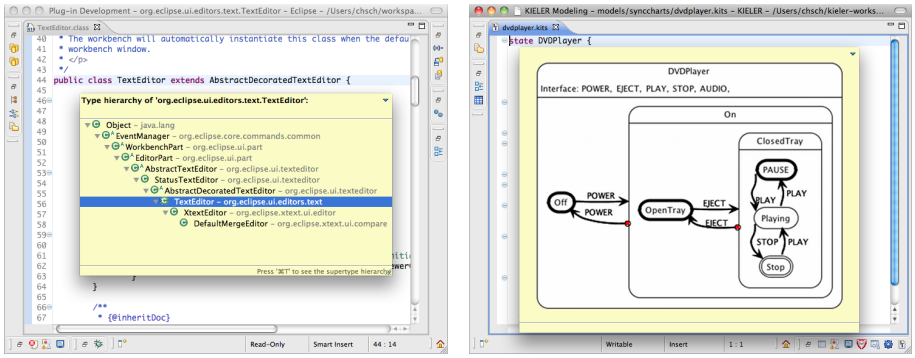
Although editor code generation front-ends such as *GMF Tooling*[3] and *Spray*[4] offer means for model-based view specification, the generated code suffers from the same problems as described above. The additional level of abstraction makes it even harder to customize and fine-tune the views. Furthermore, GMF Tooling requires a tight coupling of model and view. This imposes strong requirements on the structure of the meta model, i. e. the abstract syntax of the language, which is not acceptable, since we aim for multiple views on the same model.

Modeling tools such as GME [10] and VMTS [11], which are built on Windows instead of Eclipse, allow the creation of graphical editors with custom graphics, but the graphics must be created by either using a specific API or editing a

[2] http://sourceforge.net/projects/emf2gv/

[3] http://www.eclipse.org/modeling/gmp/?project=gmf-tooling

[4] http://code.google.com/a/eclipselabs.org/p/spray/



(a) Inheritance hierarchy of a class. (b) Proposal: transient Statechart diagram.

Fig. 1. Examples of transient views

visualization model in the UI. The focus of these projects is on meta-modeling and model transformations, and they do not cover automatic view generation and layout.

3 Towards Lightweight Graphical Modeling

The common graphical modeling approaches, either based on generic modeling environments or customized editing tools, require the modeler to manually put each single element on the canvas and determine its position. If an element shall be characterized with respect to different facets, e. g. a class as part of a software system, multiple diagrams must be drawn, each representing that element from different points of view. Those diagrams are often persisted in separate files, which may lead to consistency issues if elements are reordered or deleted. Regarding this dissatisfying situation we believe that it can be improved by exploiting the ability to automatically arrange diagram elements.

3.1 Transient Graphical Views

We propose to employ the *transient views* approach, which consists of the direct synthesis of graphical views out of existing models. This inverts the traditional *graphical editing* approach, in which a model is constructed using a graphical view. In our vision a modeler works with an arbitrary editor, e. g. based on a textual DSL, and requests and dismisses graphical views like Java programmers hit *ctrl+T* to see the inheritance hierarchy of a class, see Fig. 1. This way the benefits of graphical modeling are preserved, while disadvantages such as time consuming composition are avoided.

The transient graphical view synthesis process comprises the following steps.

1. Select models to be represented, possibly with manual or automatic filtering.

2. Construct a view model according to mapping rules from the domain model.
   a. Identify the essential graph elements (nodes, edges, labels, ports).
   b. Create each element's graphics by composing rendering primitives.
   c. Arrange the rendering primitives (*micro layout*).
3. Arrange the graph structure of the view model (*macro layout*).
   a. Analyze the view model and derive a layout graph.
   b. Configure the layout by choosing layout algorithms and setting options.
   c. Execute the layout algorithms.
   d. Transfer the computed layout back to the view model.
4. Render the view model by means of a 2D graphics framework.

In our approach we aim to optimize the synthesis process in terms of performance and simplicity in order to justify the predicate "lightweight". This would enable truly transient views, eliminating the necessity of persisting the view models. We achieve this by employing the same meta model for layout algorithms and for the view model, and extending it with annotations for expression of rendering primitives and their arrangement. This yields the following benefits:

- The view model is based on EMF and thus allows to use model transformation as well as other model-based techniques, which is the basic idea of MDV [4]. For instance, this enables the employment of interpreted transformations that could be formulated by the tool user. This advantage applies to all three parts of Step 2 in the view synthesis process.
- In common graphical editors the micro layout (Step 2.c) is implemented in Java (see Sect. 2). By including the micro layout specification in the rendering model we are able to express it on an abstract level. While this may seem like a trivial matter, it turns out to be crucial for the consistent use of automatic layout: as illustrated in Fig. 2, changes of the macro layout may require recomputation of the micro layout. Therefore a close coupling of both levels of layout is beneficial.
- Step 3.a is often an intricate task, which concerns the extraction of the graph structure as well as the initial macro layout. We obtain the simplest possible solution by using the same graph structure for the view model and the layout process, hence no transformation or adaption is needed. This also applies to Step 3.d, since the concrete layout attached to the graph instance during execution of layout algorithms directly affects the view model.

3.2 Use Cases

Applications of transient views are manifold. As motivated in Sect. 3.1, modelers may want to get certain information on their system under development. Similarly, modelers continuously want to check the correctness of their work, e. g. the reachability of states in Statecharts, by reviewing it in an alternative notation. In case of an error the fix shall be performable directly in the view.

In practice, specifications of large systems are usually created in a component-based way. This often occurs in form of declaring and referencing elements separately. An example, found in the railway signaling domain, looks as follows

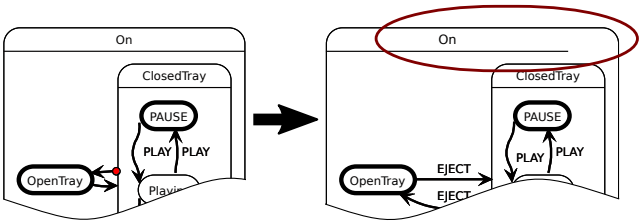


Fig. 2. Broken figure rendering after an update of the Statechart diagram: inserting labels on the transitions between the OpenTray and ClosedTray states causes the On state to be enlarged; afterwards the state label is not centered anymore and the line below is too short.

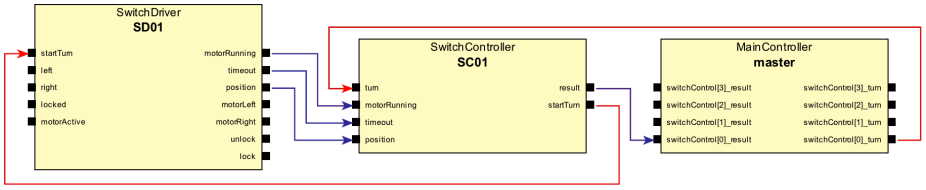


Fig. 3. Component architectural outline on a specification excerpt of a complex railway signaling system. It has been composed based on various single description parts.

(simplified). There are three types of components, each of them specified in a separate document: a MainController, SwitchControllers, and SwitchDrivers. The components communicate via dedicated interfaces described in further specification parts. Finally, instances of the components are introduced and connected in an additional statement. Although those particular descriptions may be simple, the resulting networks can become quite complex and difficult to browse, understand, and maintain. By means of transient views the tool can offer specific compound representations, which are built upon multiple parts of the specification, and provide a component architectural outline as shown in Fig. 3.

3.3 The KGraph and KRendering Meta Models

The *KGraph* meta model describes the graph as used in steps 2.a and 3.a of the view synthesis process. Its class diagram, derived from an EMF Ecore model, is shown in Fig. 4. The graph structure is represented by the classes KNode, KEdge, KPort, and KLabel. Each instance of these graph elements contains an attached KEdgeLayout (for edges) or KShapeLayout (for other elements), which are both able to hold concrete layout data as well as abstract layout data represented by *layout options* (see Sect. 4). A graph is represented by a KNode instance with its content stored in the children reference.

Fig. 5 depicts an excerpt of the *KRendering* notation meta model, which is an extension of KGraph. Basic figure shapes are instances of KRendering, which inherits from KGraphData and thus can be attached to KGraphElements. KRen-

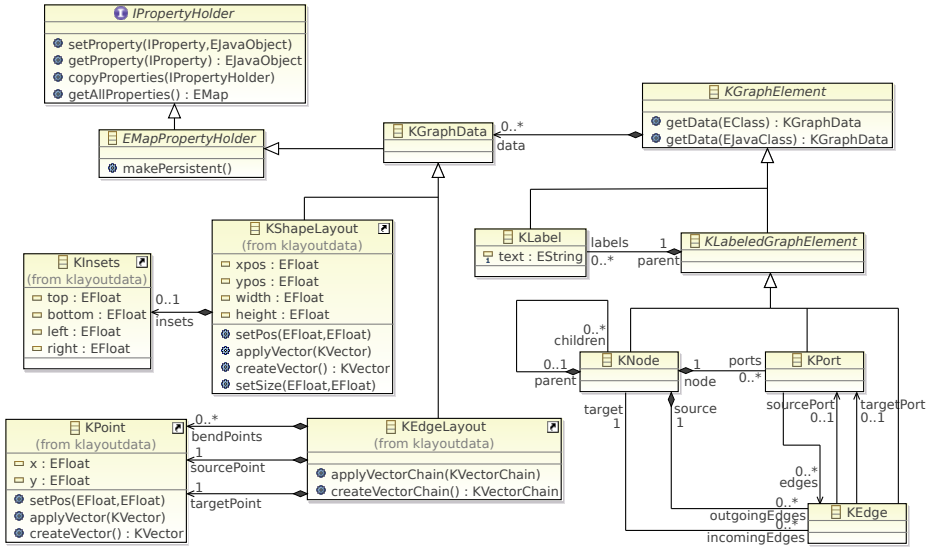


Fig. 4. KGraph meta model with basic graph structures and layout data.

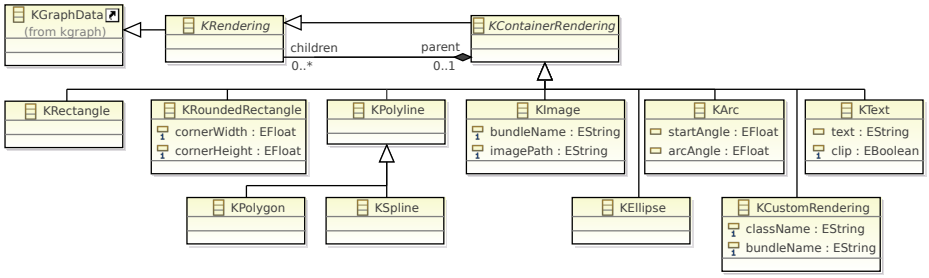


Fig. 5. KRendering basic shapes to be composed to diagram figures.

derings can be configured in terms of KStyles for specification of properties such as line width or foreground and background color (see Fig. 6). KRenderingRefs refer to other rendering definitions, and templates of KRenderings may be stored in a KRenderingLibrary. The placement of KRenderings can be defined by means of KDirectPlacementData or KPolylinePlacementData (see Fig. 7), or by attaching them to grid- or stack-based micro layouts, which are omitted here. The direct placement definitions consist of two points, given by KX-/KYPositions, that are related to the borders of the parent KRendering. Polyline placements consist of a number of bend positions, respectively. These implicit coordinates are resolved in Step 4 of the view synthesis process.

The composition of diagram figures by means of the KRendering primitives is exemplified in Fig. 8, showing a description excerpt of the SwitchController diagram node from Fig. 3. The figure consists of a KNode comprising a KShapeLay-

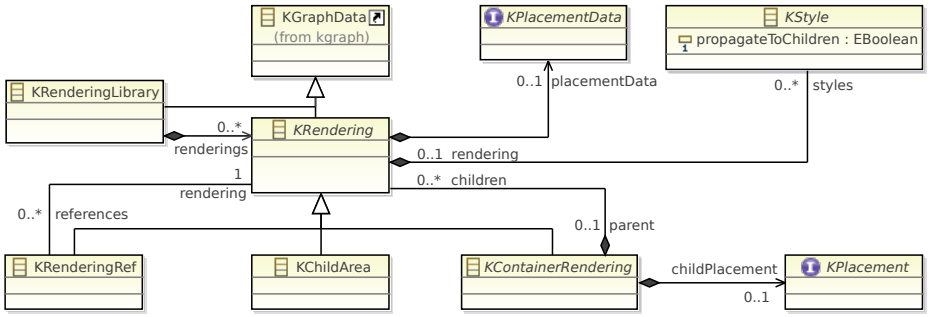


Fig. 6. KRendering core elements.

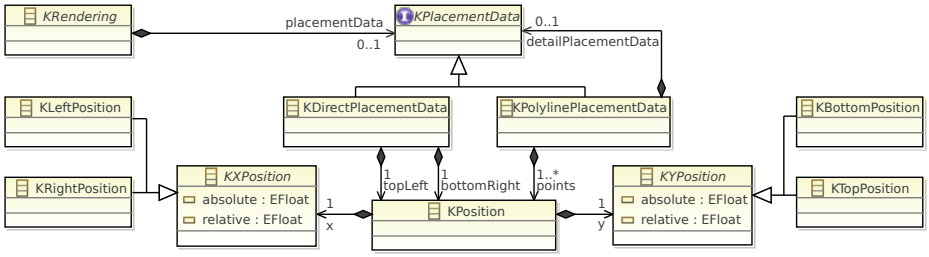


Fig. 7. KRendering placement elements for specifying micro layout directives.

out defining its size and a layouter hint, a KRectangle, and a bunch of KPorts (only one is shown for the sake of brevity). The rectangle covers the whole figure, since no placement data are given, and contains two horizontally centered text fields showing the type and instance name of the depicted element (the second text field's description is omitted, too). They are modified in terms of the text's vertical alignment, the transparency of the background color, and their size and position in the figure that is characterized by the KDirectPlacementData entry. Thus, the first text field spans a rectangle ranging from the left to the right and from the top to 14pt below the top border of the diagram figure. The horizontally centered alignment is set by default.

The figures of the ports are basically constructed in the same way, apart from horizontal alignment of the port label text field. In addition, one observes that KText elements determine their minimal size by their font size and text length. Hence, the horizontal part of the bottomRight position does not matter for the left aligned port label. Observe furthermore that child KRenderings need not to be placed within the bounds of its parent, which is the case for the port label.

The view synthesis process is realized in the KIELER Lightweight Diagrams (KLighD) project, our test bed for investigating the topic of transient graphical representations, which is integrated in the KIELER view management [7]. KLighD provides infrastructure to manage the mapping rules needed in Step 2, which currently have to be provided by the tool developer. Step 3, the macro layout, is

```
KNode {
 KShapeLayout {
  width 200 height 100
  mapProperties:
  "de.cau.cs.kieler.portConstraints"
      = "FIXED_POS"
 },
 KRectangle {
  KBackgroundColor 255 250 205
  KText "SwitchController" {
   KVerticalAlignment TOP
   KBackgroundVisibility false
   KDirectPlacementData {
     topLeft KLeftPosition abs 0.0 rel 0.0
           / KTopPosition abs 0.0 rel 0.0
     bottomRight KRightPosition abs 0.0 rel 0.0
              / KTopPosition abs 14.0 rel 0.0
   }
  },
  ...
 }
```

```
KPort {
 KShapeLayout {
  xpos -8 ypos 31 width 9 height 9
 },
 KRectangle {
  KBackgroundColor 0 0 0
  KText "turn" {
   KFontSize 9,
   KHorizontalAlignment LEFT
   KBackgroundVisibility false
   KDirectPlacementData {
    topLeft KLeftPosition abs 12.0 rel 0.0
          / KTopPosition abs 0.0 rel 0.0
    bottomRight KLeftPosition abs 0.0 rel 0.0
           / KBottomPosition abs 0.0 rel 0.0
   }
  }
 },
 ...
}
```

Fig. 8. Excerpt of the KRendering-based specification of the SwitchController diagram element depicted in Fig. 3.

delegated to the KIELER Infrastructure for Meta Layout (KIML), see Sec. 4, and Step 4, the rendering of the representation, is performed by the graphics framework *Piccolo2D* [1]. Since the view model (KGraph + KRendering) does not rely on any specific graphics framework, we will add support for other frameworks such as Draw2D in the future.

4 Configurable Automatic Layout

Research on graph drawing algorithms has led to a rich variety of methods over the past 30 years [6, 9]. In theory, these layout methods should equip users of graphical modeling tools adequately to satisfy their need for automatic diagram layout. However, today's modeling tools are still quite far from the point where diagram layout would be available with the same flexibility as textual formatting such as the Java code formatter of Eclipse JDT. The problem is not the lack of appropriate algorithms or libraries for graph layout, but rather their integration.

Two examples of excellent libraries are OGDF [5] and Graphviz [8], both of which offer several layout methods with plenty of options for customization. The former offers a C++ API and support for GML and OGML graph formats, while the latter offers a C API and support for the DOT graph format. Connecting one of these tools to a Java application is a costly task, since it consists either in the intricacy of directly executing native code or in the communication with a separate process using one of the supported graph formats.

The KIELER Infrastructure for Meta Layout (KIML) provides a bridge between diagram viewers and layout algorithms and offers interfaces for layout configuration. Graphviz, OGDF, and a collection of Java-based algorithms are

included, thus offering a wide variety of layouts to Eclipse based diagram editors and viewers. Graphiti and GMF have been connected generically such that layout can be done in many editors that are based on these frameworks without the need of adding or changing any code. However, as explained in Sect. 3.1, the layout process is a lot more efficient for KLighD views.

4.1 Layout Configuration

We consider two levels of automatic layout: concrete layout and abstract layout. A concrete layout determines the exact position and size of all elements of a graph, including nodes, labels, and edge bend points, whereas an abstract layout consists of options for the selection and configuration of layout algorithms. When an algorithm is executed on an input graph, it reads these options and considers them in the calculation of graph element positions.

Layout options can be set for each graph element independently. This allows to modify general settings of an algorithm, to set constraints for specific graph elements, or even to apply different layout algorithms for different hierarchy levels of a compound graph. These layout options are set in Step 3.b of the view synthesis process described in Sect. 3.1 by executing a set of *layout configurators* on each graph element. Some of the currently defined layout configurators are described in the following.

- The **Default** configurator has the lowest priority and returns default layout settings that are acceptable for most graphs.
- The **Eclipse** configurator manages an extension point and a preference page, which can both be used to override default values for specific element types. The edit part class or the domain model class can be used to specify the type of a graph element.
- The **Semantic** configurator is an extensible mechanism for deriving layout settings from the domain model. This is used when different layout option values are chosen depending on properties of the domain model instance.
- The **GMF / Graphiti** configurators allow to customize the layout for a single diagram, which can be done through an Eclipse view named *Layout*. For GMF diagrams the options are stored as Style annotations in the *Notation* model, while for Graphiti the options are stored as Property annotations in the *Pictogram* model.

5 Conclusion and Future Work

We presented a continuation of the MDV approach by allowing the view model to express graph structure as well as rendering and layout directives. The KGraph meta model includes structural information that is relevant for layout algorithms, properties for abstract layout specification, and concrete layout data calculated by algorithms. The KRendering meta model adds rendering primitives with style and micro layout annotations. The system is backed by a flexible and configurable

automatic layout infrastructure. Putting these building blocks together yields a tool that is well suited for the visualization of complex models.

Our future work will target various aspects. Diagram specifications shall be expressed in a textual language, similarly to Köhnlein's *Generic Graph View*.[5] This involves describing the rendering of elements, as well as composing diagrams, which may be understood as *queries* on a model base. The synthesized diagrams shall be equipped with *semantic zoom*, i. e. the ability to change their amount of detail according to the user's focus. Finally, intuitive means for modifying the abstract layout, e. g. in form of sliders or gestures, shall be investigated.

References

1. Bederson, B.B., Grosjean, J., Meyer, J.: Toolkit design for interactive structured graphics. IEEE Transactions on Software Engineering 30(8), 535–546 (Aug 2004)
2. Bull, R.I.: Model Driven Visualization: Towards A Model Driven Engineering Approach For Information Visualization. Ph.D. thesis, University of Victoria, BC, Canada (2008)
3. Bull, R.I., Best, C., Storey, M.A.: Advanced widgets for Eclipse. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange. pp. 6–11. ACM, New York, NY, USA (2004)
4. Bull, R.I., Storey, M.A., Litoiu, M., Favre, J.M.: An architecture to support model driven software visualization. In: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06). pp. 100–106. IEEE (2006)
5. Chimani, M., Gutwenger, C., Jünger, M., Klein, K., Mutzel, P., Schulz, M.: The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing (GD07) (2007)
6. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall (1998)
7. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). LNCS, vol. 6394, pp. 196–210. Springer (Oct 2010)
8. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software—Practice and Experience 30(11), 1203–1234 (2000)
9. Kaufmann, M., Wagner, D. (eds.): Drawing Graphs: Methods and Models. No. 2025 in LNCS, Springer-Verlag, Berlin, Germany (2001)
10. Lédeczi, Á., Maróti, M., Bakay, Á., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Völgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing (2001)
11. Mezei, G., Levendovszky, T., Charaf, H.: Visual presentation solutions for domain specific languages. In: Proceedings of the IASTED International Conference on Software Engineering. Innsbruck, Austria (2006)
12. Schneider, C., Spönemann, M., von Hanxleden, R.: Transient view generation in Eclipse. Technical Report 1206, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (Jun 2012)

[5] http://koehnlein.blogspot.de/2012/01/discovery-diagrams-for-generic.html