INSTITUT FÜR INFORMATIK

Transient View Generation in Eclipse

Christian Schneider, Miro Spönemann,
and Reinhard von Hanxleden

Bericht Nr. 1206
Juni 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Transient View Generation in Eclipse

Christian Schneider, Miro Spönemann,
and Reinhard von Hanxleden

Bericht Nr. 1206
Juni 2012
ISSN 2192-6247

e-mail: {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 report 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.

Contents

List of Figures

1 Introduction

Graphical modeling languages such as the UML and its dialects are established in software development and adjacent disciplines, yielding the key benefit of the abstraction of implementation details [15]. This provides stakeholders of different professions with a common language to communicate their business matters. Graphical languages are characterized by fixed syntaxes and well-defined semantics or at least semantics skeletons. The so achieved advantage of effectiveness, however, is usually bought by the price of less productivity with respect to time to formulate solutions. More precisely, an important amount of productivity is wasted with drawing and beautification activities while working with state-of-the-practice modeling environments.

For this and further reasons domain-specific textual modeling languages gain a lot of popularity in these days.[1] Textual editors support apparently trivial operations such as *inserting by typing* and *automatic line-wrap*, which turn out to be not-so-trivial operations for graphical editors. As most modelers may have experienced, inserting a node into a crowded diagram can be very tedious.

Our objective is to combine the advantages of textual and graphical modeling notations. Building upon automatic layout technology we want to abandon the manual creation of graphical model representations. Instead we follow the concept of Model Driven Visualization (MDV) introduced by Bull et al. [5] by generating representations in a *lightweight* and *transient* fashion, which means that graphical views are generated so seamlessly that there is no need to manually edit or to persist them. This approach facilitates the creation of *multiple views* and the dynamic adaption of the *level of detail*, which are important concepts for the visualization of complex information [18, 19].

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 or dynamically configurable and provides the foundation for transient view synthesis. Both contributions are realized within the KIELER project[2] and are available via an Eclipse update site.[3]

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

[1] Examples are mentioned on `http://www.eclipse.org/Xtext/community/`

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

[3] `http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/`

2 Related Work

The work presented here builds on Model Driven Visualization as proposed by Bull et al. [5, 3], 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[1] [4] 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 [10], 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 for specifying rendering primitives at all, 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

[1] `http://www.eclipse.org/gef/zest/`
[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/`

model, i. e. the abstract syntax of the language, which is not acceptable, since we do not want to impose restrictions on the abstract syntax and aim for miscellaneous views on the same model.

Modeling tools such as GME [14] and VMTS [16], 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 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. Although mapping between EMF and GME models is possible [2], this does not cover the mapping of graphical views between the frameworks.

3 Towards Lightweight Graphical Modeling

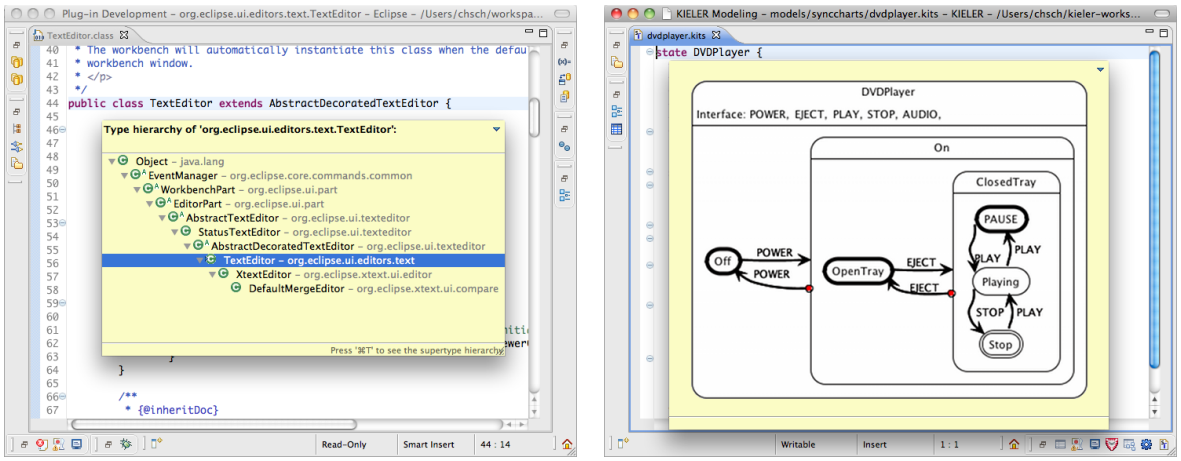
The common graphical modeling approaches require the modeler to manually put each single element on the canvas. If an element shall be characterized with respect to different facets, e. g. a class as part of a software system, multiple diagrams must be drawn, each representing that element from different 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 there is significant advantage 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. 3.1. 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.



(a) Inheritance hierarchy of a class.

(b) Proposal: transient Statechart diagram.

Figure 3.1: Examples of transient views

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 [5]. 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. 3.2, 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.

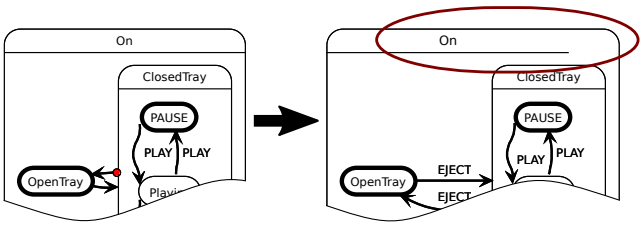


Figure 3.2: Broken figure rendering after an update of the Statechart diagram: inserting labels on the transitions between the OpenTray and ClosedTray states causes the On state to be enlarged; afterwards the state label is not centered anymore and the line below is too short.

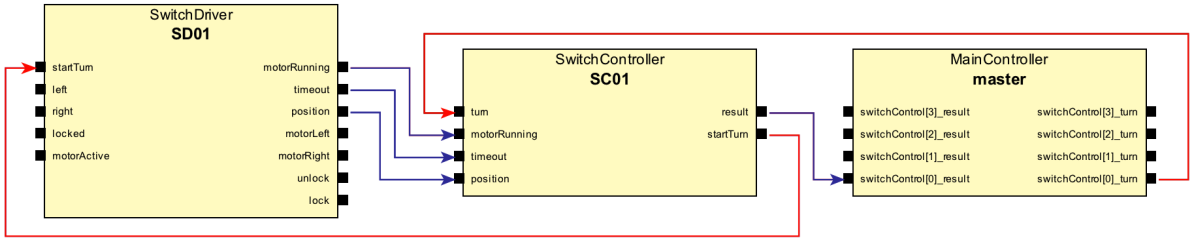


Figure 3.3: Component architectural outline on a specification excerpt of a complex railway signaling system. It has been composed of multiple description parts.

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 a 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 (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.

The third use case of transient graphical views addresses the simulation of DSLs-based specifications. It is evident that highlighting elements in diagrams, e. g. active states, improves humans' ability to perceive the simulated system's progress and to identify mistakes in the execution and, hence, mistakes in the specification. This principle is

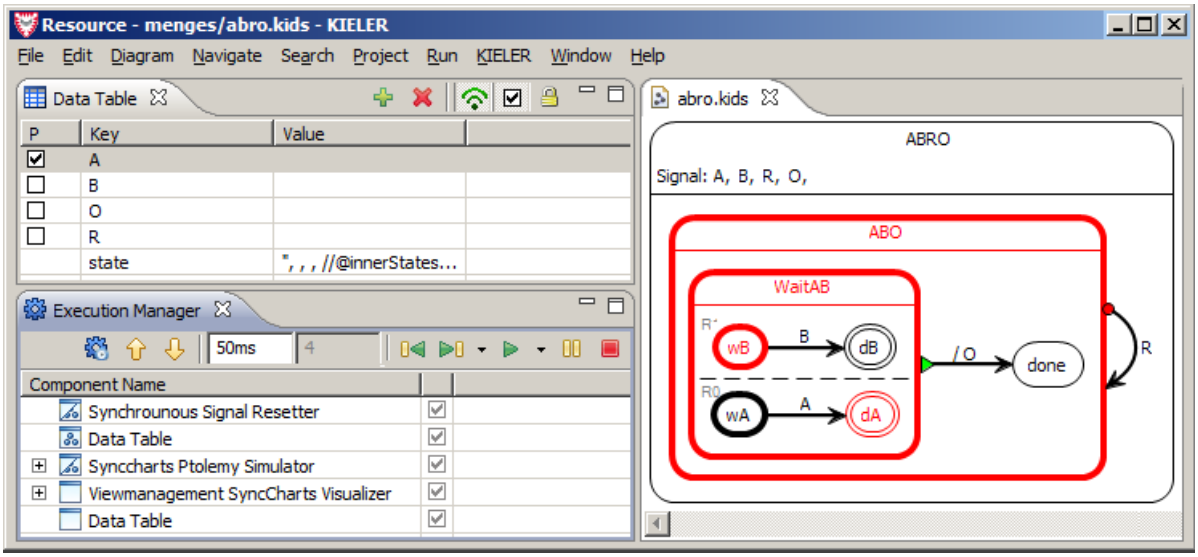


Figure 3.4: Simulation of a Statechart in KIELER (from Motika et al. [17]).

outlined by means of Fig. 3.4, showing a Statechart being simulated on the right, and tables with input and output data as well as the simulation steering on the left. Besides marking active states, assignments of variables or metrics of statements can also be visualized in diagrams. This enables the feedback of results of complex analyses, e. g. performance estimations discovering shortest and longest execution paths.

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. 3.5. 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. 3.6 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. KRenderings can be configured in terms of KStyles and KPlacementData for specification of properties such as line width or foreground and background color and the micro layout (see Fig. 3.7). Given KRenderings can be reused by means of KRenderingRefs, which refer to other rendering definitions, templates of KRenderings may be stored in a KRenderingLibrary.

As indicated by Fig. 3.8 properties of KRendering figures are not determined by means of attributes of the KRendering class, but by attaching KStyles on them. This allows us to define the propagateToChildren flag meaning the style with the flag set to true will

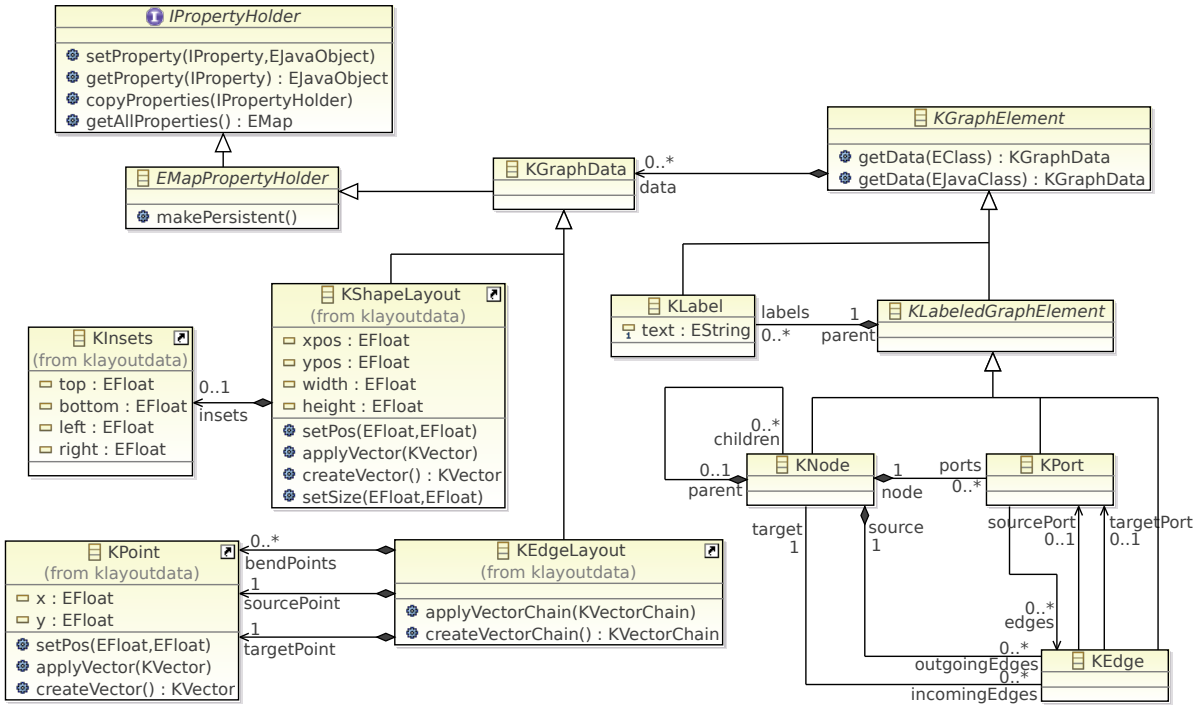


Figure 3.5: KGraph meta model with basic graph structures and layout data.

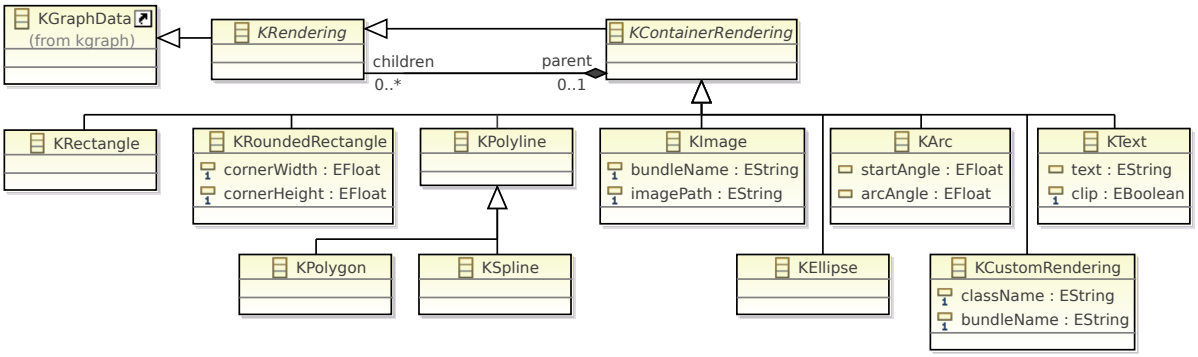


Figure 3.6: KRendering primitives to be composed to diagram figures.

be applied deeply to all children of the current figure. This, in turn, is beneficial for highlighting purposes, e.g. during simulations as motivated in Sec. 3.2, and other *view management* effects [9].

The placement of KRenderings can be defined by means of KDirectPlacementData or KPolylinePlacementData (see Fig. 3.9), 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 exem-

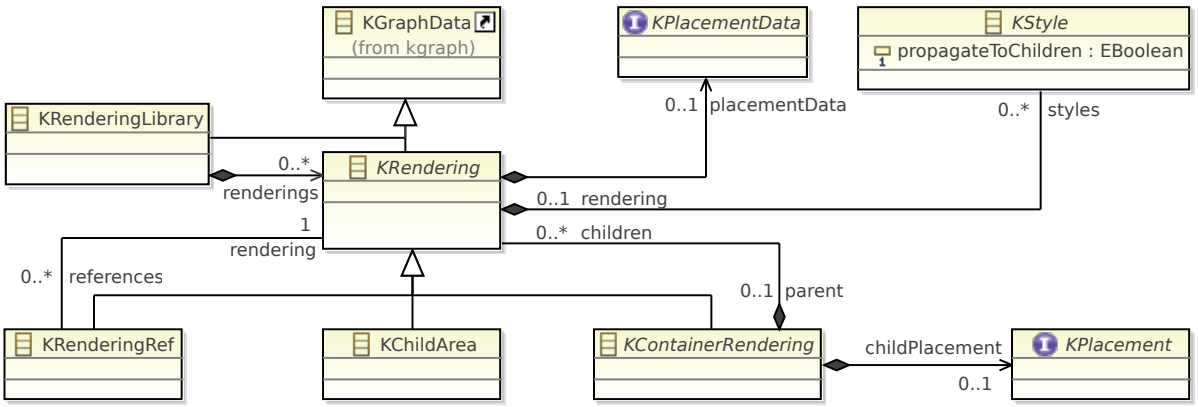


Figure 3.7: KRendering core elements.

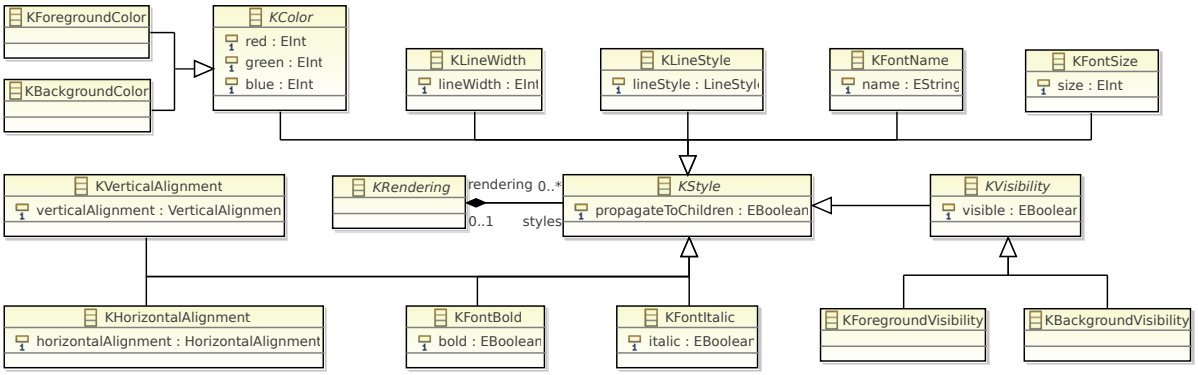


Figure 3.8: KRendering styles for configuring diagram figures.

plified in Fig. 3.10, showing a description of the SwitchController diagram node from Fig. 3.3. The figure consists of a KNode comprising a KShapeLayout defining its size and a layouter hint, a KRectangle, and a bunch of KPorts. 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. They are modified in terms of the text's vertical alignment and the transparency of the their bounding

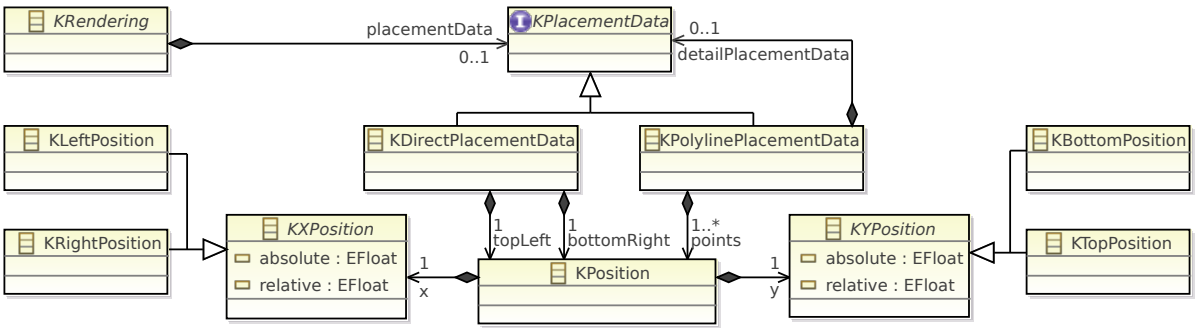


Figure 3.9: KRendering placement elements for specifying micro layout directives.

```
KNode {
 KShapeLayout { /* size constraint,  layout  data */
  width 200 height 100
  "de.cau.cs.kieler.portConstraints"
      = "FIXED_POS"
 }
 KRectangle { /* box figure */
  KBackgroundColor 255 250 205
  KText "SwitchController" { /* label 1 */
   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 12.0 rel 0.0
   }
  }
  KText "SC01" { /* label 2 */
   KFontBold true
   KVerticalAlignment TOP
   KBackgroundVisibility  false
   KDirectPlacementData {
     topLeft  KLeftPosition  abs 0.0 rel 0.0
            / KTopPosition abs 12.0 rel 0.0
     bottomRight KRightPosition abs 0.0 rel 0.0
                 / KTopPosition abs 24.0 rel 0.0
   }
  }
 }
 KPort {
  KShapeLayout { /* size constraint,  layout  data */
   xpos -8 ypos 31 width 9 height 9
  }
  KRectangle { /* port figure (black)  */
   KBackgroundColor 0 0 0
   KText "turn" { /* port label */
    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
    }
   }
  }
 }
```

```
 KPort {
  KShapeLayout { /* size constraint,  layout  data */
   xpos 200 ypos 31 width 9 height 9
  }
  KRectangle { /* port figure (black)  */
   KBackgroundColor 0 0 0
   KText "result" { /* port label */
    KFontSize 9,
    KHorizontalAlignment RIGHT
    KBackgroundVisibility  false
    KDirectPlacementData {
     topLeft  KLeftPosition  abs 0.0 rel 0.0
            / KTopPosition abs 0.0 rel 0.0,
     bottomRight KLeftPosition abs -3.0 rel 0.0
              / KBottomPosition abs 0.0 rel 0.0
    }
   }
  }
 }
 ...
} --> "<<the master node>>" { /* 1st edge */
 sourcePort  "<<SC01::result>>"
 targetPort  "<<master::switchControl[0]_result>>"
 KEdgeLayout {} /* layout data place holder */
 KPolyline {
  KLineWidth 2
  KForegroundColor 0 0 255! /* propagated to children */
  KPolygon { /* arrow decorator */
   KBackgroundColor 0 0 255
   KPolylinePlacementData {
    points
      KLeftPosition  abs 0.0 rel 0.0
     / KTopPosition abs 0.0 rel 0.0,
      KRightPosition abs 0.0 rel 0.0
     / KTopPosition abs 0.0 rel 0.5,
      KLeftPosition  abs 0.0 rel 0.0
     / KBottomPosition abs 0.0 rel 0.0,
      KLeftPosition  abs 0.0 rel 0.3
     / KTopPosition abs 0.0 rel 0.0
    KDecoratorPlacementData {
     relative
     location  1.0
     xOffset  -9 yOffset -4 width 8 height 7
    }
   }
  }
 }
} --> "<<the SD01 node>>" { /* 2nd edge */
 ...
}
```

Figure 3.10: Shortened KRendering-based specification of the SwitchController diagram element depicted in Fig. 3.3.

boxes' backgrounds. Their size and position in the figure 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 12pt below the top border of the diagram figure, the second one is placed below in the same way. The horizontally centered alignment is set by default.

The ports and their figures are basically constructed in the same way, apart from the

horizontal alignment of the port label text field. Due to their similarity the definitions of the second highest ones et seq. are omitted on both sides. 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 label "turn" of first-mentioned port. The same holds for the topLeft position of the second port's right aligned "result" label, respectively. Observe furthermore that child KRenderings need not to be placed within the bounds of its parent, which is the case for the port labels.

Connections are related to the KNodes they are leaving. They are declared by KEdge statements, in Fig. 3.10 abbreviated with -->. Such KEdges have to be equipped with KEdgeLayouts and a rendering, e. g., KPolyline. Decorators, such as arrow heads, are added as usual child renderings. In the current example the decorator is formed by a KPolygon, a specialization of KPolyline whose only difference is the consideration of the path given by a list of KPositions as a circuit. Since KEdges are not characterized by any covered area, the KPolylinePlacementData statement defining the path of our arrow has to be augmented with special a KDecoratorPlacementData element that defines the size of the decorator's bounding box, its location on the edge in terms of a value of $[0.0, 1.0]$, and the flag relative indicating whether the decorator shall be rotated according to the layout of the edge.

3.4 KIELER Lightweight Diagrams (KLighD)

The view synthesis process is realized in the KLighD project, our test bed for investigating the topic of transient graphical representations. It 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 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], which is based on the Java2D API but also offers an SWT wrapper. The resulting diagrams can be displayed in Eclipse views or other UI elements. 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.

The prototype is employed in MENGES[1] [11], a conjoint research project of industry and academia that aims at improving the software development process for safety-critical railway signaling systems.

Besides being able to draw view models, KLighD provides infrastructure to register implementations of mapping that synthesize view descriptions based on arbitrary input.

KLighD integrates with the KIELER Infrastructure for View Management (KIVi) [9], which triggers the view synthesis either full-automatically or on the modelers' demand. In addition to the pure extraction of certain model content, views obtained this way maintain links between model elements and their figures and form a navigation means for exploring models.

[1] `http://menges.informatik.uni-kiel.de/`

4 Configurable Automatic Layout

Research on graph drawing algorithms has led to a rich variety of methods over the past 30 years [7, 12]. 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 [6] and Graphviz [10], 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. Even if both libraries are connected to a modeling environment, further questions arise on the details of their integration. How could a suitable layout algorithm be selected and configured? How could such a configuration be linked to a specific diagram or application, either statically or dynamically? What if users would like to get a good layout for their specific diagram without the need to understand the concepts of the available algorithms? The KIELER Infrastructure for Meta Layout (KIML) addresses these questions by providing a bridge between diagram viewers and layout algorithms and by offering interfaces for layout configuration. The concepts behind this infrastructure are described in the following.

4.1 The Diagram Layout Process

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 hints for the selection and configuration of layout algorithms. When an algorithm is executed on an input graph, it reads these hints and considers them in the calculation of graph element positions, therefore layout algorithms *convert* abstract layouts into concrete layouts.

The main class for coordination of the layout process in KIML is DiagramLayoutEngine. This process involves the following steps, as illustrated in Fig. 4.1.

1. The selected diagram is analyzed by a specialized DiagramLayoutManager. Depending on the type of diagram viewer, the required information is drawn from

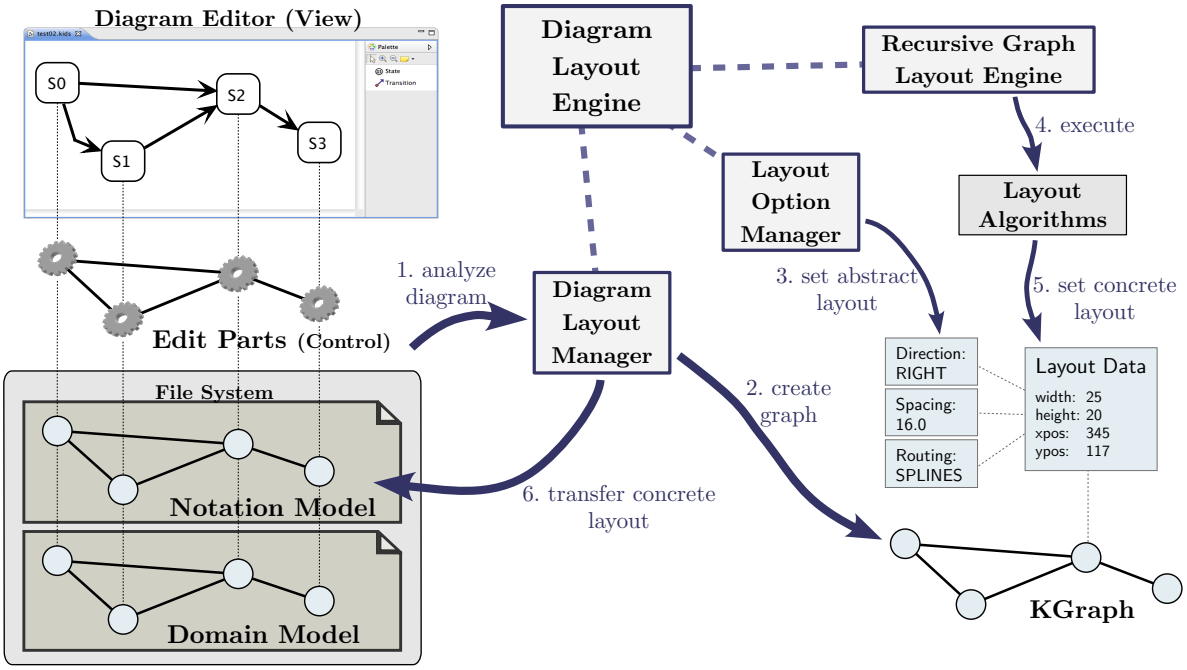


Figure 4.1: Overview of the layout process for typical diagram editors.

the viewer's model, control, or view component. Each of these components typically represents the structure of the diagram by a specialized data structure. For instance, the GMF uses Draw2D figures as views, *edit parts* as controllers, a *notational* model for storing the concrete layout and customized styles, and a *domain* model defined by the DSL.

2. The DiagramLayoutManager creates an instance of the KGraph data structure (see Fig. 3.5 on p. 9 and step 3.a on p. 5) based on the analysis of Step 1.

3. The class LayoutOptionManager configures an abstract layout and attaches this information to the elements of the graph in form of a mapping of layout options to specific values (see step 3.b on p. 5). There is a special layout option used to select the layout algorithm that shall be executed.

4. The selected algorithm is executed on the input graph (see step 3.c on p. 5). The graph is represented by a KNode instance with its content stored in the children reference. Each child node may itself contain a nested subgraph, thus allowing hierarchically structured graphs, which are also called *compound graphs*. The RecursiveGraphLayoutEngine, which is responsible for executing layout algorithms, processes each hierarchy level of a compound graph separately, starting with the innermost nodes.

5. The executed algorithms set concrete layout information on the input graph.

6. The computed layout information is transferred back to the diagram by the DiagramLayoutManager (see step 3.d on p. 5). Usually this information is written to the notational model, so it is stored persistently with the diagram.

The KIELER project provides integrations of the OGDF and Graphviz layout algorithm libraries. Their configuration options are made available in KIML such that both libraries can be configured using the same interface. Furthermore, KIELER contains new layout algorithms written in Java, which makes their integration more efficient compared to C or C++ libraries. The most important of these Java algorithms is *KLay Layered* [13], which is able to process normal graphs as well as port based graphs such as data flow diagrams and has a special mode for compound graphs.

Connecting the KIML layout process to Eclipse based diagram editors or viewers requires the implementation of specialized DiagramLayoutManagers. This has already been done generically for GMF and Graphiti such that layout can be done in most editors that are based on these frameworks without the need of adding or changing any code.

4.2 Layout Configuration

Layout options, which are used to control how layouts are computed, 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 of Fig. 4.1 by iterating over all graph elements and executing a set of *layout configurators* on each element. Such configurators analyze the context in which a graph element was created and derive specific layout option values from that context. If multiple configurators affect the same layout option, the one of highest priority is taken. The most important layout configurators are described in the following, with ascending priority.

- 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.

- The **Volatile** configurator applies a key-value mapping for one single layout computation. This is very useful when the layout process is triggered from some application specific control logic and needs to be adapted to the current state of the application, e. g. by using a *view management* system [9].

The KIML allows to plug in other layout configurators, which may employ more complex methods for finding an appropriate abstract layout. There are two approaches in the focus of current research which are both based on analysis of the input graph. The first approach uses *graph structure analysis* to determine which type of layout algorithm is most suitable for a graph. This decision depends on structural properties such as planarity, connectivity, or number of cycles. The second approach uses *graph drawing analysis* to assess the layout produced by the previous configuration and derive a new configuration with the aim of improving measurable criteria such as the number of edge crossings and bends or the total area and aspect ratio. Deriving new configurations from existing ones may involve evolutionary methods [8], which may be driven by automatic analysis, user feedback, or a mixture of both.

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*.[1] 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.

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

Bibliography

[1] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering*, 30(8):535–546, August 2004.

[2] Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéric Jouault, and Ivan Kurtev. Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework. In *Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development*, 2005.

[3] Robert Ian Bull. *Model Driven Visualization: Towards A Model Driven Engineering Approach For Information Visualization*. PhD thesis, University of Victoria, BC, Canada, 2008.

[4] Robert Ian Bull, Casey Best, and Margaret-Anne Storey. Advanced widgets for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange*, pages 6–11, New York, NY, USA, 2004. ACM.

[5] Robert Ian Bull, Margaret-Anne Storey, Marin Litoiu, and Jean-Marie Favre. An architecture to support model driven software visualization. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 100–106. IEEE, 2006.

[6] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing (GD07), 2007.

[7] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[8] Björn Duderstadt. Evolutionary meta layout for KIELER. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2011.

[9] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 196–210. Springer, October 2010.

[10] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1234, 2000.

[11] Wolfgang Goerigk, Reinhard von Hanxleden, Wilhelm Hasselbring, Gregor Hennings, Reiner Jung, Holger Neustock, Heiko Schaefer, Christian Schneider, Elferik Schultz, Thomas Stahl, Steffen Weik, and Stefan Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In *Software Engineering 2012 – Lecture Notes in Informatics*. Gesellschaft für Informatik, 2012.

[12] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001.

[13] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Improved layout for data flow diagrams with port constraints. In *Proceedings of the 7th International Conference on Diagrammatic Representation and Inference (DIAGRAMS'12)*, volume 7352 of *LNAI*, pages 65–79. Springer, 2012.

[14] Ákos Lédeczi, Miklós Maróti, Árpád Bakay, Gábor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Péter Völgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, 2001.

[15] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[16] Gergely Mezei, Tihamér Levendovszky, and Hassan Charaf. Visual presentation solutions for domain specific languages. In *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, 2006.

[17] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), pages 891–896, Leipzig, Germany, September 2010. Bonner Köllen Verlag.

[18] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 83–91. ACM, 1992.

[19] Doug Schaffer, Zhengping Zuo, Saul Greenberg, Lyn Bartram, John Dill, Shelli Dubs, and Mark Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Transactions on Computer-Human Interaction*, 3:162–188, 1996.