Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice

Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden
Dept. of Computer Science, Christian-Albrechts-Universität zu Kiel, Germany
{chsch,msp,rvh}@informatik.uni-kiel.de

***Abstract*—Node-link-diagrams can effectively communicate information, but their creation and maintenance require a lot of manual effort. Therefore we follow the *transient views approach* that aims at automatically deriving high quality diagrams from arbitrary models. Besides composing diagram structures, this task involves the arrangement of the diagram elements on the canvas, and, on a finer-grained level of detail, the arrangement of the shapes (rectangles, circles, lines, etc.) that form the diagram elements. We show the feasibility of this approach by means of the KIELER Lightweight Diagrams (KLighD) framework that creates diagrams this way. We discuss our overall design objectives in terms of this framework, investigate an alternative way to shape diagram figures, and briefly demonstrate the usage of KLighD in custom modeling environments by means of a case study.**

I. Introduction

In software engineering it is common to use graphical modeling languages to record and document software designs, very often the UML or a derivative. In practice this is usually done by manually drawing diagrams, i. e. by dropping diagram figures onto a canvas, drawing connections, and moving those figures around until a pleasing arrangement is achieved. This procedure is very time consuming and requires appropriate tools. Moreover, hand-made persisted diagrams easily become incorrect if details in the design change over time. A lot of effort is required just for keeping them up to date.

Considering these issues we advocate *transient automatically synthesized representations* [1] that are obtained *on-demand*, and discarded if not required any longer. Given a model adhering to a domain-specific language (DSL) and a function that maps the language constructs to diagram figures, the modeling environment is in charge of creating a corresponding diagram. In contrast to static views on models like hand-drawn diagrams, transient views must be *flexible* to achieve the "Visual Information Seeking Mantra" [2]:

Overview first, zoom and filter, then details-on-demand.

Contributions: We here demonstrate the feasibility of the *transient views* approach by means of the KIELER Lightweight Diagrams (KLighD) framework, which realizes this concept. We address the challenge of designing an appropriate *view model format* for describing diagrams, focusing on the composition of basic shapes, text fields, and bitmaps to diagram figures. In contrast to other formats such as Graphiti's *Pictogram* format[1] or the new Diagram Definition (DD) format[2] standardized by the OMG, our approach supports *relative positioning* of figure components. This flexibility simplifies very much the handling of elements whose size cannot be determined statically, e.g., labels representing variable string values. Furthermore, we propose a method to calculate the required size of compound diagram figures based on concrete string values and such positioning data. This size information is required for calculating the overall diagram layout through graph drawing methods [3], which is an important premise of the transient views approach. Users of Eclipse-based modeling tools may directly employ and benefit from KLighD, however, the concepts proposed here are applicable to visual modeling tools in general.

Outline: Section II discusses the state of practice in creating diagrams and previous works on their automatic synthesis. In Section III, we state our core objectives in the development of KLighD and outline the solution approaches. We investigate the composition of diagram figures by means of primitive shapes and the figure size issue in Section IV. In Section V we turn to practice and demonstrate the usage of KLighD by means of a simple example that aims to encourage readers to try it out themselves. We conclude this work in Section VI with examples of KLighD applications and refer to ongoing activities.

II. Motivation and Related Work

Gotel et al. [4] analyzed characteristics of pure *information visualization* and *software engineering visualization*. They found that "the visual characteristics of all software engineering notations" have been neglected in favor of their semantics. We agree with their emphasis of the potential for synergies between those fields. We do not agree in their focus on requirements engineering, though, since with the success of textual DSL frameworks modeling techniques are easily applicable in further stages of the software life cycle. Since textual formats tend to "hide" semantic mistakes well, especially if models get larger, graphical representations are often a valuable complement.

There are miscellaneous tools and frameworks for documenting structures and processes. In the industry Microsoft Visio[3] is very popular. Its major disadvantages are the manual drawing effort to spend, and the gap between the specified requirements or designs and their take-ups in other tools, e. g. development environments, although Microsoft added some means that address this issue in the latest releases. In many cases changing one artifact does not change or indicate change

This work was funded in part by the German Science Foundation (PRETSY, DFG HA 4407/6-1), the Program for the Future Economy of Schleswig-Holstein and the European Regional Development Fund (ERDF), and ETAS GmbH

[1] http://www.eclipse.org/graphiti/

[2] http://www.omg.org/spec/DD/1.0/

[3] http://office.microsoft.com/visio/

on the others. The tool yEd[4] is an interesting alternative that supports automatic layout of diagrams.

The established graphical modeling approaches in the Eclipse world (GEF, GMF, and Graphiti) as well as DIAMETA [5], GME [6], and VMTS [7] focus on user editing and are conceptually not designed to synthesize diagrams fully automatically. Furthermore, such editors usually support only a limited dedicated set of diagram elements, and their extension is often cumbersome. Besides, there are various diagram tools like PlantUML[5] or UMLet[6] that provide automatic synthesis for pre-defined diagram types (the latter even supports custom figures but requires Java programming). In contrast, KLighD aims to be a more abstract, generic, and lightweight solution supporting arbitrary node-link-diagrams for visualizing domain-specific models in homogeneous modeling environments, as explained in Section III.

The work of Storey et al. [8] employs automatic diagram synthesis for program comprehension and architecture recovery. In a follow-up work Bull et al. [9] developed the *Zest*[7] framework enabling visualizations of flat graph structures in Eclipse. Its aim is to provide a graph widget that seamlessly integrates into the existing widget zoo. Their concept of Model Driven Visualization (MDV) leverages MDSE techniques like model to model transformations to perform queries on arbitrary data. The results are visualized by means of fitting widgets such as tables, tree views, and graphs. The publicly available Zest framework, however, does not provide any support on MDSE techniques, but requires classic programming.

Köhnlein presented a very appealing prototype on synthesizing diagrams[8] based on interpreted mappings and style sheet descriptions in a semi-automatic way. Besides manual element placing automatic layout is also available. In his demonstrations the user is, however, required to perform a lot of mouse operations in order to obtain the desired diagram. Similarly to Zest the *Generic Graphical View* supports neither ports nor (arbitrarily) nested hierarchic diagrams.

The work presented here fits into the general context of *modeling pragmatics* [10], which focuses on human productivity aspects during modeling activities, and where the separation of a model from customized, automatically generated views is a key concept. The Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) is a testbed for validation of modeling pragmatics concepts and provides the environment for KLighD, e. g. automatic layout by means of graph drawing methods. However, that aspect is not in focus of this work.

III. KIELER LIGHTWEIGHT DIAGRAMS (KLIGHD)

KLighD is our proposal for on-demand graph visualization in the software engineering context. The key enabler of such a tool is the ability to arrange graph elements to form an *aesthetic* layout [11]. Since different kinds of graphs—and different kinds of data being represented by graphs—demand for different kinds of arrangement, the layout engine must support different layout strategies. For instance, classes depicted in a UML class diagram are usually arranged differently compared to states in a UML statechart or the elements of a Simulink model. With the KIELER Infrastructure for Meta Layout (KIML) there is such a layout facility [1], [10].

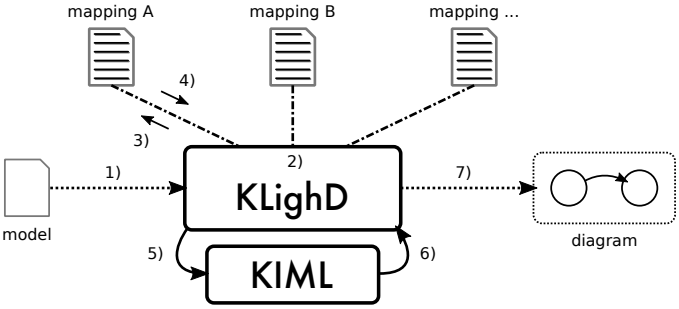


Figure 1. Data flow in KLighD: 1) Request for diagram of semantic model called "model", 2) mapping selection, 3) mapping application, 4) receipt of corresponding KGraph+KRendering data, 5) handover to KIML, 6) receipt of complete diagram data, 7) translation of abstract descriptions into concrete data and handover to a Piccolo2D diagram canvas.

The KLighD framework is supposed to meet the following objectives.

A. Reduction of time to diagram

The time software engineers spend on creating graphical representations of their designs is to be reduced drastically. This applies to the time budget expended for obtaining a single representation, as well as to the training effort that is required to get able to set up an automatic view synthesis.

KLighD achieves that by means of the formal input language *KRendering*, which enables precise and compact formulation of diagrams. Technical details, e. g. the arrangement of the particular figure elements, are specified in a descriptive manner. Such diagram descriptions are usually not composed by users, but created and maintained underneath KLighD's hood. Instead, the framework is provided with mappings of DSL structures to KRendering data. With that knowledge in mind KLighD is just required to create visualizations of given models. This task consists of selecting the required mapping, applying it to the provided model, applying automatic layout to the diagram data, and translating that data into such suitable for drawing the diagram. Figure 1 illustrates the procedure.

For details on the KRendering language we refer to [1], whereas the core concepts are explained below.

B. Integration of established MDSE concepts and tools

Diagram synthesis implementations shall be realizable using model transformations. Modern model transformation languages such as Xtend[9] or ATL [12] provide special support for data translations in form of powerful language constructs and rich convenience function libraries. For example, they allow to formulate translations in a functional programming style, which enables much more condensed implementations than classic programming languages do. The employment of model transformation techniques also paves the way for reusing queries on modeled data, which may be used for model validation and runtime code generation, too.

[4] http://www.yworks.com/de/products_yed_about.html
[5] http://plantuml.sourceforge.net/
[6] http://www.umlet.com/
[7] http://www.eclipse.org/gef/zest/
[8] http://koehnlein.blogspot.de/2012/01/discovery-diagrams-for-generic.html
[9] http://www.xtend-lang.org

The KRendering language has been defined in EMF's metamodeling language Ecore [13]. Hence, in addition to the advantages mentioned above, KLighD benefits from EMF's mature features such as opposite references or adapters. In addition we built up a convenience function library that can be incorporated in diagram synthesis mappings implemented in Xtend or Java. Concrete DSL to KRendering translations can be compacted very much by means of this library, as we show in Section V.

C. Efficiency

Browsing rendered diagrams shall work without any notable delays, e. g. while zooming or panning the visible area. Especially in case of big diagrams the tool has to work as stable as with just a couple of nodes and edges. In contrast to other disciplines dealing with millions of nodes and edges, we assume a magnitude of up to hundreds of diagram elements, an amount that can be mentally handled by humans.

On the one hand we rely on the promising graph drawing framework Piccolo2D [14] that has been ported to the Eclipse Standard Widget Toolkit (SWT). It comes without any sophisticated re-arrangement logic, but with an innovative camera concept making it very fast and responsive.

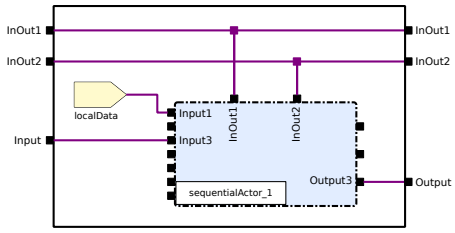
On the other hand we minimize the intermediate data being created during the view synthesis, layout, and update processes. For that reason we built the KRendering language on top of *KGraph*, the input and output language used by KIML for automatic diagram layout. In more detail, the structure of diagrams, i. e. nodes, their ports, edges, and labels, is formulated in terms of KGraph elements. Their look is defined by augmenting those elements with KRendering data. In order to get the diagram data supplemented with positioning information KLighD can immediately hand them over to KIML. This way, we omit costly data extraction for preparing KIML's input and applying the result to the rendered diagram, see steps 5) and 6) in Figure 1.

Furthermore, we omit the employment of any transaction mechanisms like incorporated in GMF-based editors for the sake of realizing undo and redo operations.

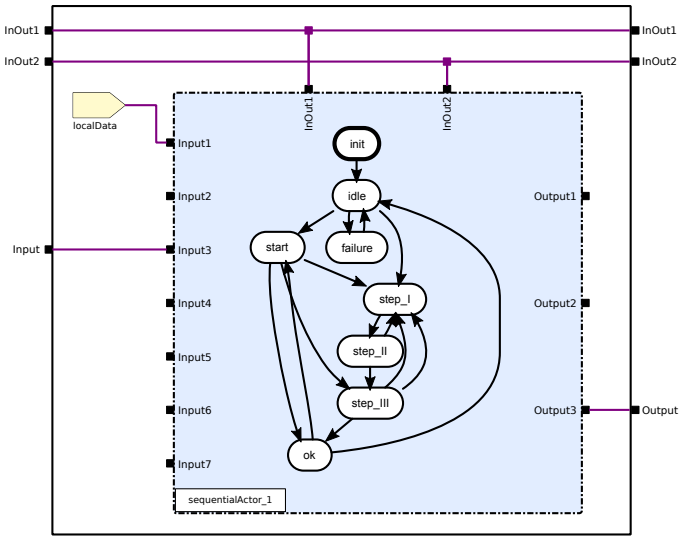
D. Interactivity

For coping with large data and for displaying simulation results in an intuitive fashion, the visualization framework is supposed to support highlighting, *semantic zooming* [15], and further diagram manipulation operations. Semantically zooming diagrams involves the addition of further details to diagram elements (expand), as well as their reduction (collapse). This principle is illustrated by the diagram excerpts depicted in Figure 2. While Figure 2(a) shows a part of a data flow network with collapsed representation of sequentialActor_1, Figure 2(b) shows the same excerpt with sequentialActor_1 in expanded mode. Regarding the highlighting, special attention has to be paid if multiple tool components are in a competition for highlighting particular diagram elements.

We care about these concerns in the design of the KRendering language in the following way: Diagram figures are determined by composing primitive shapes like rounded rectangles, ellipses or polygons. These shapes are configured in terms of colorings, line styles, fonts, etc. by attaching *styles*. Such styles



(a) Collapsed representation of sequentialActor_1



(b) Expanded representation of sequentialActor_1

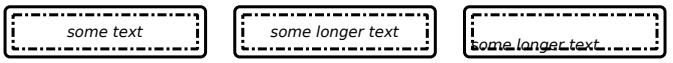
Figure 2. Hierarchic data flow diagram rendering excerpts created by KLighD with different amounts of detail. While the representation of operator sequentialActor_1 is abbreviated by a simple box in the first drawing its sequential behavior is outlined in the second one. Besides, only names of connected ports are shown in the collapsed representation, while all port names are visible in the expanded one. Further refinement may attach transition labels.

can be added and removed by highlighters interchangeably, with each of those highlighters being not allowed to remove or manipulate style objects contributed by others. In case of conflicting style statements the last (youngest) one will make it into the diagram. If it gets removed again the look of the related figure will change immediately according to the previous statement in the row or the default configuration.

The amount of detail of diagrams is influenced by means of *Actions*, which may be related to figure primitives, too. Such an action specifies its trigger (e. g. single click or double click) and refers to an implementation that is to be executed when the trigger is fired. KLighD provides some standard action implementations, e. g. for collapsing and expanding nodes; further actions may be provided via an extension point.

IV. Figure Arrangement in KRendering

As mentioned above, diagram elements like nodes and edges are represented by figures that are described by means of KRendering-based definitions. The figures usually consist of nested rectangles, lines, text fields, etc. In this section we discuss the arrangement of such figures. We refer to that as the *micro layout* of diagram figures. This is distinct from the arrangement of diagram elements, i. e. the layout of the overall diagram, which we refer to as the *macro layout* of diagrams



(a) initial figure (b) with extended text (c) bottom left alignment

Figure 3. Diagram figures with sufficient initial size adhering to similar KRendering-based descriptions; only the label text and alignment varies.

and which is KIML's matter. In order to illustrate that consider sequentialActor_1 in Figure 2(b). While the positions of the state representatives (the rounded rectangles) are matter of the macro layout, the positions of their label strings are matter of the micro layout.

In common editor-based diagram drawing tools, the micro layout is either performed by *layout managers* (e. g. StackLayout, FlowLayout, ToolbarLayout, XYLayout, etc.), or by full-custom handwritten implementations, like in Graphiti-based editors. They are part of the editing tools in both cases and not reflected in the diagram description at all. KGraph/KRendering-based diagrams, however, are supposed to be fully self-contained on the one hand, and robust with respect to changes of size constraints on the other hand. Such size constraints are imposed by text fields of arbitrary length, or, in case of hierarchic nodes, by the size of the content, as seen in Figure 2(b). Both requirements are derived from the top level objective *Reduction of time to diagram*. Thus, we propose to specify micro layout concerns in an *abstract* fashion.

A. Smart Figure Layout

We address the above requirements by providing three figure layout strategies in the KRendering language.

1) Reference-point-based child placement: The child figure is located at a reference position within the parent figure, and aligned according to horizontal and vertical alignment settings. The size of the child figure may be determined by related minimal height and width parameters; additional horizontal and vertical margins may be configured.

2) Sub-area-based child placement: The child figure is laid out within a contained area of the parent that is determined by *top left* and *bottom right* positions.[10] These positions form a bounding box and, thus, the bottom right position is never located left or above the top left position.

3) Grid-based child placement: All children of the parent figure are positioned along a grid with a given number of columns. The size of the particular colums and rows is derived from the size of the contained child figures, the size of the parent figure, and minimal width/height constraints.

To achieve robustness against size changes, we introduce a more abstract notion of *position*. As usual a position consists of a horizontal and a vertical component. These components, however, consist of an absolute part in dots/pixels, and a relative part in range of zero to one, which is applied to the width or height of the parent figure. In addition, *the horizontal position components refer either to the left or right bound of the parent, and the vertical ones to the top or bottom bound*. The given length is applied from the respective side towards

[10]In the previous version of KRendering this placement strategy was called "direct placement" [1].

```
KNode {
 width=200 height=50
 KRoundedRectangle {
  lineWidth=3!
  KRectangle {
   lineStyle=DASH_DOT
   KAreaPlacementData
    topLeft TOP abs=8px rel=0, LEFT abs=8px rel=0
    bottomRight BOTTOM abs=8px rel=0, RIGHT abs=8px rel=0
   KText "some text" {
    italic
    KPointPlacementData
     refPoint TOP abs=0px rel=0.5, LEFT abs=0px rel=0.5
     horMargin=8px
     horAlignment=CENTER
     vertAlignment=CENTER
    }}}}
```

Listing 1. Description of the diagram figure in Figure 3(a) in the KRendering format (pseudo code). Modifying the label text to "some longer text" leads to Figure 3(b). The label position and alignment were changed for Figure 3(c).

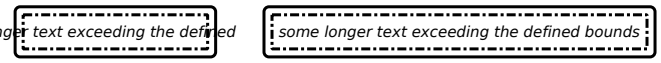
the center of the figure. Note that the freedom w. r. t. the position reference sides is not restricted by the sub-area-based child placement condition. That condition refers to concrete coordinates being calculated based on the abstract position data. This way we can express the positioning of a child at a fixed ratio w. r. t. the parent's size rather than at a fixed absolute position. For example, we can place text labels at the center of their parents. We can also direct a figure to occupy a certain quadrant or corner of its parent, or prescribe a fixed margin surrounding the child regardless of the parent's concrete size.

Example: Imagine a diagram figure that consists of two nested rectangles and a text label like in Figure 3(a). There shall be a margin of 5px between the inner and outer rectangle, the label shall be aligned centrally in both horizontal and vertical directions. In addition, the label shall preserve a margin of at least 5px to the inner rectangle on its left and right side.

Listing 1 shows the KRendering-based description of Figure 3(a). It starts with the definition of a **`KNode`** that denotes the structural diagram element. The node is configured with an initial size and a composed KRendering defining its look. The root figure is a rounded rectangle of line width 3. This line width is to be applied to its children, too, indicated by the exclamation mark. The rounded rectangle contains a regular rectangle with a special line style. This child rendering definition is augmented with **`KAreaPlacementData`** that define the area of the rounded rectangle to be covered by this inner rectangle. This definition determines insets of 8px on each side w. r. t. the bounds of the rounded rectangle. The value of 8px is chosen to comply with the line width of 3 and the required margin of 5px mentioned above.

The inner rectangle is equipped with a text field, indicated by the keyword **`KText`** followed by the string to be displayed. It is augmented with **`KPointPlacementData`** that define the reference position of the text figure to be at the half width and half height of the containing rectangle. This is indicated by the position components' relative portions of 0.5. The text figure is to be aligned centrally for both dimensions meaning it grows equally to the four directions starting from the reference point. The **`horMargin`** of 8px reflects the required minimal margin of the label (5px) adjusted by the parent's line width of 3px.

If we modify the label string to "some longer text", we will obtain the diagram figure depicted in Figure 3(b). If we, in



(a) broken micro layout (b) increased figure satisfying the micro layout

Figure 4. Diagram figure drawings with insufficient given size values.

contrast, would like to let the text start on the lower left corner, we would simply define the reference point to be at BOTTOM **abs**=0px **rel**=0, LEFT **abs**=0px **rel**=0, set the horizontal alignment to LEFT, and the vertical alignment to BOTTOM. A related drawing is depicted in Figure 3(c). Additional margins on the left and bottom side might be configured by means of the absolute position parameters. Aligning the text, e. g. on the lower right corner, works analogously with the reference point on the RIGHT and the horizontal alignment to RIGHT.

B. Estimating the required node size

So far we are able to place primitive figures, described in form of rendering definitions, within a compound diagram figure of given size. Often, however, it is required to adapt the size of the whole diagram node in order to meet the requirements imposed by the attached figures' placement strategies. Especially in case of text fields of variable length, e. g. those representing the name or another value of an object, an adjustment of the node size is usually necessary or, at least, desired. To illustrate, consider the example in Figure 3/Listing 1 again. If we extend the text string to be displayed to "some longer text exceeding the defined bounds", we will expect a result as depicted in Figure 4(b) rather than a figure like Figure 4(a) with the text violating the figure bounds.

To this end, we developed a heuristic estimation of node sizes based on the attached KRendering data. It is related to the calculation of the minimal and preferred sizes of figures by the layout managers of Eclipse GEF. Since the node size must be known while computing the macro layout, we situate it as a macro-layout-preprocessing activity that is performed between steps 4) and 5) according to Figure 1. Since the macro layout might also affect the size of nodes, this estimation will not work in general, because with the relative position portions we would have to search for fix points. Regarding our objective *Efficiency* we avoid fix point iterations. Our approach works quite well in common cases, though.

Algorithm 1 illustrates the starting point of the size estimation function that distinguishes three major cases. If the current rendering definition is a text field, the required space of the string's drawing is to be determined, e. g. by means of the platform's window system. That calculation must incorporate the selected font, font size, and font style (normal, bold, italic). If the rendering definition is an instance of **KChildArea**, a required size of $(0, 0)$ will be assumed. Child areas are virtual rendering definitions acting as placeholders for the children of **KNode**s in hierarchic diagrams. Since the placement of those children is determined during the macro layout phase, the required space is allocated at that time, too. Thus, a required size of zero is returned by the node size estimation function.

Both the **KText** and **KChildArea** cases serve as axioms of this function. In contrast, the size of **KContainerRendering**s, i. e. those that may contain child rendering definitions, must be derived from the sizes of the contained children recursively. In

Algorithm 1 Size estimation of KRendering-based diagram figure definitions

```
1  function estimateSize(rendering, (float, float) initialSize) returns (float, float):
2  switch (type(rendering))
3    case KText:
4      return estimateTextSize(rendering)
5    case KChildArea:
6      return (0, 0) // the size of child areas is determined by the macro layout
7    case KContainerRendering:
8      if (requiresGridChildPlacement(rendering))
9        return estimateGridSize(rendering, initialSize)
10     else
11       (float, float) maxSize ← initialSize
12       for (KRendering r in children(rendering))
13         if (child has KPointPlacementData)
14           maxSize ← max(maxSize, estimatePointPlacedChildSize(child))
15         elsif (child has KAreaPlacementData)
16           maxSize ← max(maxSize, estimateAreaPlacedChildSize(child,
                 initialSize))
17         end if
18       end for
19       return maxSize
20     end if
21 end switch
```

order to comply with the size constraints imposed by the child figures' placement strategies, these strategies must be applied in the inverse way. Thus, further cases are to be distinguished. In case the children are to be arranged in a grid all children are treated at once, since they impose interdependencies in terms of heights and widths of the grid's rows and columns. In contrast, children being arranged according to the reference-point and sub-area-based strategies are considered independently. Otherwise we would have to perform further fix point iterations. Therefore, the size estimation is performed for each child separately, and the final required size is the maximum of the sizes required by the contained children (the max() function in line 14 and line 16 is applied componentwise).

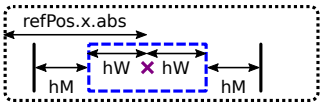
In the following, we explain the size estimation w. r. t. the particular placement strategies. For the sake of brevity we omit the grid-based one. A basic precondition is assumed to hold for all child rendering definitions: Each child's bounds (adjusted by required margins) are equal to its parent's bounds or fully-enclosed by them. This implies that children are never larger than their parents and do not overlap their parents bounds.

Algorithm 2 illustrates the process for rendering definitions augmented with reference-point-based placing information. First, the maximum of the minimal size defined in the placement data and the actually required size is calculated, see Algorithm 2 line 4 (again, max() is applied componentwise). In case of LEFTward horizontal alignment the child figure's left bound is to be put on the reference position. Thus, we estimate the child figures' required width by the sum of the leftward indentation defined by the reference position's absolute part, the child's actual width, and the horizontal margin, which is to be added on the child's right side in this case (Algorithm 2 line 10). The reference position's relative part is ignored as its proper incorporation would require the aforementioned fix point iteration. The RIGHTward alignment case is symmetric.

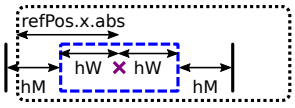
In case of CENTRAL alignment the horizontal margin prescribed by the placement data is to be applied to both sides,

Algorithm 2 Estimation of space required by a rendering definition placed according to the point-based placing strategy.

```
1  function estimatePointPlacedChildSize(child) returns (float, float):
2  ppd ← placementData(child)
3  (float h, float w) minimalSize = (ppd.minHeight, ppd.minWidth)
4  (float h, float w) cSize ← max(estimateSize(child, minimalSize), minimalSize)
5  float requiredHeight, requiredWidth
6
7  switch (ppd.horAlignment)
8    case LEFT:
9    case RIGHT:
10     requiredWidth ← ppd.refPos.x.abs + cSize.w + ppd.horMargin
11     break
12   case CENTRAL:
13     float halfWidth ← cSize.w / 2
14     if (ppd.refPos.x.abs > halfWidth + ppd.horMargin)
15       requiredWidth ← ppd.refPos.x.abs + halfWidth + ppd.horMargin
16     else
17       requiredWidth ← 2 × halfWidth + 2 × ppd.horMargin
18     end if
19 end switch
20
21 ... // vertical dimension analogously
22 return (requiredHeight, requiredWidth)
```



(a) The hor. position's absolute part exceeds the sum of hM and hW.



(b) The hor. absolute position part is below the sum of hM and hW.

Figure 5. Placing constellations of a point-based placed child shape (dashed rectangle) in a parent shape (dotted rectangle) that is to be aligned centrally, limited to the horizontal dimension. The reference point is marked by the cross, hW denotes the child's half width, hM refers to the horizontal margin.

i. e. on the left and right. This entails a distinction of two cases, which are illustrated in Figure 5. In case the reference position's absolute part exceeds the sum of half width and horizontal margin (Figure 5(a)), the required width is estimated by the sum of the absolute part, the child's half width, and the horizontal margin, see Algorithm 2 line 15. Otherwise, the basic precondition stated above appears to be violated, since the child adjusted by its margins seems to overlap the parent's bound (Figure 5(b)). This violation, however, may be fixed by a valid relative positioning part. Since the parent size is still unknown, we cannot compute the distance determined by the relative part. We, thus, assume the good case, i. e. the satisfaction of our precondition, and estimate the required width by the sum of the child's width and twice the margin. The estimation of the required height works similarly.

The required space-estimation of rendering definitions arranged according to the sub-area-based placement strategy is outlined in Algorithm 3. Due to the similar treatment of horizontal and vertical dimension we skip to the vertical one. Again, the actual minimal size of the child rendering definition is calculated first, and the size gain due to the placement parameters is determined subsequently. The gain is affected by the share of the parent's width to be covered by the child, as well as by the absolute amount the child is smaller than the parent. The share is derived from the positions' relative parts, the absolute amount from the positions' absolute parts,

Algorithm 3 Estimation of space required by a rendering definition placed according to the area-based placing strategy.

```
1  function estimateAreaPlacedChildSize(child, initialSize) returns (float, float):
2  apd ← placementData(child)
3  (float h, float w) cSize ← estimateSize(child, applyAreaPlacing(initialSize, apd))
4  float absAmount, share;
5
6  Position left ← apd.topLeft.x
7  Position right ← apd.bottomRight.x
8  switch ((left.referenceSide, right.referenceSide))
9    case (LEFT, RIGHT): // topLeft refers to the left side, bottomRight to the right
10     share ← 1.0 − left.rel − right.rel
11     absAmount ← left.abs + right.abs
12     break
13   case (LEFT, LEFT): // topLeft refers to the left side, bottomRight to the left
14     share ← right.rel − left.rel
15     absAmount ← −right.abs + left.abs
16     break
17   case (RIGHT, RIGHT): // topLeft refers to the right side, bottomRight to right
18     share ← left.rel − right.rel
19     absAmount ← −left.abs + right.abs
20     break
21   case (RIGHT, LEFT): // topLeft refers to the right side, bottomRight to the left
22     share ← right.rel − (1.0 − left.rel)
23     absAmount ← −right.abs − left.abs
24 end switch
25 float requiredWidth ← (share == 0.0 ? 0.0 : cSize.w / share) + absAmount
26
27 ... // vertical dimension analogously
28 float requiredHeight ← (share == 0.0 ? 0.0 : cSize.h / share) + absAmount
29 return (requiredHeight, requiredWidth)
```

respectively. Due to the flexibility in defining the positions of the sub area's topLeft and bottomRight corners in terms of the horizontal reference sides (see Section IV-A), four cases are to be distinguished for computing these values, see lines 9 to 21. The child's required width is then obtained by dividing the child's actual width by the share and adding the absolute amount. Note the special treatment in case the share is equal to zero. This will occur if the width of the child is adjusted only by the positions' absolute parts, like in Listing 1 for example.

V. Case Study: UML Use Case Diagrams

To demonstrate the effectiveness of our approach, we now present an exemplary application of KLighD for synthesizing UML *use case diagrams*. We utilize the Xtend language and KRendering-related convenience functions that form an internal DSL. It abstracts, e. g., factory and constructor calls, and the style definitions via separate objects by simple setter methods.

Use case diagrams show actors by means of stick figures and use cases in form of labeled ellipses. Those figures are usually connected by simple straight lines that represent relations between actors and use cases. We used the Papyrus[11] tool to create a simple example, see Figure 6, that contains two actors named Actor 1 & Actor 2. Actor 1 is related to use case Simple UseCase A, Actor 2 to use case Simple UseCase B. Both actors are related to a third use case named Very important UseCase requiring lots of attention. Papyrus structures this semantic information on actors, use cases, and relations according to the UML2 meta model provided by the Eclipse MDT project. Thus,

[11] http://www.eclipse.org/papyrus/

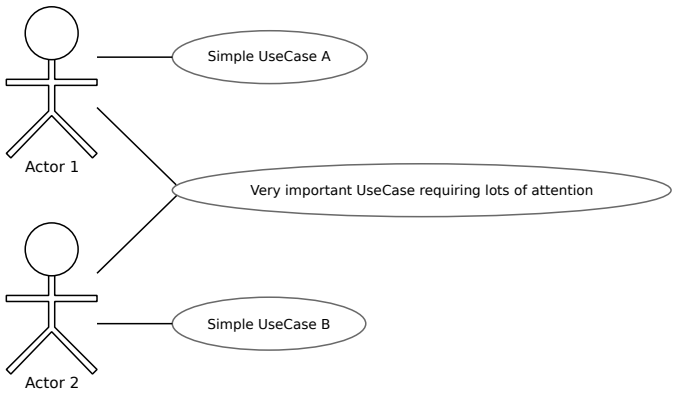


Figure 6. UML use case diagram automatically created & arranged by KLighD.

```
def KNode createUseCaseNode(UseCase useCase) {
  return useCase.createNode().putToLookUpWith(useCase) => [
    it.addEllipse() => [
      it.foreground = "darkGray".color
      it.addText(useCase.name).putToLookUpWith(useCase) => [
        it.setSurroundingSpace(10, 0.1f) // absolute part, relative part
      ] ] ] }
```

Listing 2. Synthesis of UML use case figure definitions, formulated in Xtend.

our diagram synthesis must translate data complying with that meta model into KGraph/KRendering data.

Listing 2 shows the translation of UseCase objects into related diagram objects. Each one is represented by a dedicated diagram figure, so a related KNode is created by means of the createNode() function. In order to easily access the node object while translating the relations later on, an explicit mapping of the UseCase object and the node is kept in mind by that function. Since this mapping is maintained in one direction only, there is the function putToLookUpWith(...) that builds up a bidirectional map. These data are available as long as the diagram exists and, thus, enable to reveal and focus semantic elements, which are related to selected diagram figures, e. g. in an editor or another viewer. Thanks to Xtend we can write these calls in suffix notation as shown in the listing.

Once the required node is created, it needs to be configured. This can be done elegantly with the ... => [...] construct. It applies the procedure in square brackets to the object on the left. Within the procedure that object is accessible via the built-in identifier **it** and returned finally. We augment the node with an ellipse, whose foreground/stroke color is set to dark gray, and which is equipped with the required text field displaying the name of the UseCase object. The size of the whole figure is not defined, since it is to be determined automatically by the text field's string according to the principles discussed in Section IV-B.

The actor figures are composed as depicted in Listing 3. The required KNodes are created similarly to those of the use cases; the node size, however, is defined explicitly, since it is fixed rather than derived from text strings. The actors' names are displayed by explicit node labels (a construct of the KGraph format), which are attached and configured by the addOuterCentralBottomNodeLabel(...) method. The root rendering definitions of actor figures are invisible rectangles that act as

```
def KNode createActorNode(Actor actor) {
  return actor.createNode().putToLookUpWith(actor) => [
    it.setNodeSize(60, 100);
    it.addOuterCentralBottomNodeLabel(actor.name).putToLookUpWith(actor)
    it.addRectangle() => [
      it.invisible = true
      it.addPolyline() => [
        it.points += createKPosition(LEFT, −2, 0.5f, TOP, 34, 0)
        it.points += createKPosition(LEFT, −2, 0.5f, TOP, −2, 0.5f)
        ...
        it.points += createKPosition(RIGHT,−2, 0.5f, TOP, −2, 0.5f)
        it.points += createKPosition(RIGHT,−2, 0.5f, TOP, 34, 0)
      ]
      it.addEllipse().setPointPlacementData(
          createKPosition(LEFT, 0, 0.5f, TOP, 0, 0), // ref side, abs part, rel part
          H_CENTRAL, V_TOP, 0, 0, 35, 35 // alignment, margins, width, height
      ).background = "white".color
    ] ] }
```

Listing 3. Synthesis of UML actor figure descriptions, formulated in Xtend.

```
override KNode transform(Model input) {
  return input.createNode() => [ // the root node representing the diagram canvas
    it.addLayoutParam(LayoutOptions::SPACING, 40f)
    // ... and add the diagram elements
    it.children += input.packagedElements.map[
      switch(it) {
        Actor: it.createActorNode()
        UseCase: it.createUseCaseNode()
        default: null
      }
    ].filterNull() // drops null values
    // create representations of the associations (edges)
    // they are attached to the diagram implicitly by setting their source node
    input.packagedElements.filter(typeof(Association)).forEach[ association |
      association.createEdge().putToLookUpWith(association) => [
        it.source = association?.ownedEnds.head?.type?.node
        it.target = association?.ownedEnds.last?.type?.node
        it.addPolyline()
      ]
    ] ] }
```

Listing 4. Main mapping function building descriptions of UML use case diagrams based on model data conforming to the UML2 meta model (Xtend).

containers of the polylines forming the bodies of the stick figures and ellipses forming the heads. The start, bend, and end points of the body polylines are determined in KRendering's flexible position notation, too. They, besides, determine the positions of the polylines within their parent rendering definitions, and here therefore within the whole diagram figures. In the listing the enumeration of the 17 points is abbreviated. The ellipse rendering definitions are placed according to the reference point–based placing strategy with a horizontal and vertical margin of zero, the (minimal) width and height equal to 35px yield the desired circles. The circles' explicit background color avoids the polylines' ends to project into the circles.

The main function of the diagram synthesis is shown in Listing 4. Its signature is prescribed by the required super class AbstractDiagramSynthesis. The function creates a root KNode and configures some macro layout parameters. Afterwards, the switch statement in the square brackets is applied to each of the entries in the input object's list named packagedElements, indicated by the map() function. The resulting KNodes and

null values are combined to a new list returned by map(). The nulls are dropped by the filterNull() function applied to the aforementioned list, and the remaining collection of KNodes is added to the root node's children.

In order to synthesize the links representing the relations between actors and use cases, the packagedElements list is traversed again, whereby only elements of type Association are regarded, indicated by the filter() function call. Similar to the creation of nodes, a KEdge is created for each Association object and linked for tracing purposes. Source and target of these edges are set by revealing the ends of the association, i. e. the actors and use cases related to each other. They are returned by the expressions association?.ownedEnds.{head|tail}?.type? in this case, the question marks avoid potential null pointer faults. The expression *node* returns the KNode object that has been created while applying createNode() to the actor or use case object beforehand. Besides, setting the source of edges automatically attaches those edges to the networks of KNodes via EMF's opposite reference mechanism. Finally, polyline rendering definitions are attached. The concrete start, bend, and end points are matter of the macro layout and, thus, not regarded here.

Our diagram synthesis mapping is contributed to KLighD by registering it via an extension point. If KLighD is demanded to show a diagram of our example model by executing DiagramViewManager.getInstance().createView(<*ourExample*>), e. g. via a menu entry, a new Eclipse view will be opened showing the diagram in Figure 6. Thus, regarding this case study an effort of writing down about 60 lines of code is required to come to simple use case diagrams rather than implementing a custom editor or viewer by means of the established technologies. The source of this exemplary diagram synthesis is available in the KIELER source code repository.[12]

VI. Conclusion and Future Work

With the work of this paper we tackle the problem of wasting lots of productivity while creating graphical representations of system designs. Instead of drawing diagrams manually, we propose their automatic generation from a specification (a model) at the time they are needed. This *transient view approach* is an advancement of Model Driven Visualization (MDV). The KLighD framework demonstrates the feasibility of this concept and is our proposal for employing on-demand synthesized node-link-diagrams. Besides the required ability to construct appealing diagram layouts, we draw attention to the issue of composing the diagram figures, which appears to be almost equally important in practice since diagram figures are usually labeled with at least a name.

The KLighD framework was conceived within the MENGES[13] project, which aimed at employing MDSE methods in the development of safety-critical railway signaling systems. Currently, we are working on its integration in an advanced model browser that is developed by an automotive software supplier. It is supposed to equip engineers applying software in the field with a tool realizing state of the art browsing techniques like semantic zooming in an intuitive fashion, and to display results of queries on the model base.

[12] http://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/pragmatics/browse/plugins/de.cau.cs.kieler.klighd.examples/

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

Beyond the visualization of data perceived as models, KLighD can be employed for visualizing arbitrary data in form of node-link-diagrams. For example, it is currently used in a project for representing Java collection data and other data structures in a human-friendly form while debugging a piece of Java software. In addition to diagram drawings in the tool we are also working on a Scalable Vector Graphics (SVG) export.

Other future work will address the specification of diagram synthesis mappings by means of a DSL rather than implementing it in model to model transformation or program code. We intend to achieve a further step of abstraction with such a DSL. This, in turn, might pave the way for allowing domain experts, which are often programming non-experts, to extend their modeling tools by further specific diagram syntheses.

References

[1] C. Schneider, M. Spönemann, and R. von Hanxleden, "Transient view generation in Eclipse," in *Proceedings of the First Workshop on Academics Modeling with Eclipse*, Kgs. Lyngby, Denmark, Jul. 2012.

[2] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL'96)*. IEEE Computer Society, Sep. 1996, pp. 336–343.

[3] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[4] O. Gotel, F. T. Marchese, and S. J. Morris, "The potential for synergy between information visualization and software engineering visualization," in *Proceedings of 12th International Conference on Information Visualisation (IV'08)*. IEEE Computer Society, 2008, pp. 547–552.

[5] M. Minas, "Generating meta-model-based freehand editors," in *Proceedings of the 3rd International Workshop on Graph Based Tools (GraBaTs'06)*, ser. Electronic Communications of the EASST, vol. 1, Berlin, Germany, 2006.

[6] Á. Lédeczi, M. Maróti, Á. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Völgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing*, 2001.

[7] G. Mezei, T. Levendovszky, and H. Charaf, "Visual presentation solutions for domain specific languages," in *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, 2006.

[8] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Müller, "On integrating visualization techniques for effective software exploration," in *Proceedings of the IEEE Symposium on Information Visualization*. IEEE, 1997, pp. 38–45.

[9] R. I. Bull, M.-A. Storey, M. Litoiu, and J.-M. Favre, "An architecture to support model driven software visualization," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 100–106.

[10] H. Fuhrmann and R. von Hanxleden, "Taming graphical modeling," in *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, ser. LNCS, vol. 6394. Springer, Oct. 2010, pp. 196–210.

[11] H. C. Purchase, "Metrics for graph drawing aesthetics," *Journal of Visual Languages and Computing*, vol. 13, no. 5, pp. 501–516, 2002.

[12] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF Eclipse Modeling Framework*, 2nd ed., ser. Eclipse Series. Addison-Wesley, Pearson Education, 2009.

[14] B. B. Bederson, J. Grosjean, and J. Meyer, "Toolkit design for interactive structured graphics," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 535–546, Aug. 2004.

[15] K. Perlin and D. Fox, "Pad: An Alternative Approach to the Computer Interface," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 57–64.