

# KIELER: A Text-First Framework for Automatic Diagramming of Complex Systems

Maximilian Kasperowski<sup>[0000-0002-7509-1678]</sup>,  
Niklas Rentz<sup>[0000-0001-6351-5413]</sup>, Sören Domrös<sup>[0000-0002-8011-8484]</sup>, and  
Reinhard von Hanxleden<sup>[0000-0001-5691-1215]</sup>

Department of Computer Science, Kiel University, Kiel, Germany  
`{mka,nre,sdo,rvh}@informatik.uni-kiel.de`

**Abstract.** In Model-Driven Engineering, editing models is typically not merely a purely textual endeavor, but rather a mix between textual and graphical editors and views. Both have their advantages and use cases where either textual or diagrammatic representations are better suited to edit and understand models. Therefore, a modeling framework offering the best of both worlds can be advantageous.

We define the *text-first* approach to combine the textual and diagrammatic representations by automatically synthesizing the textual model into a diagram. We present the KIELER text-first diagramming framework and its take on current challenges for model visualization and compare it to the diagram-first approach, as exemplified by the GLSP framework.

**Keywords:** Diagramming Framework · Modeling Tools · Automatic Visualization · Diagram Synthesis.

## 1 Introduction

Visualization is a useful tool in Model-Driven Engineering (MDE), especially for communication and documentation purposes during system development. In the context of developing complex, real-world systems, handmade visualizations are often tedious to create and maintain. In the case of ongoing developments and changes they can quickly become a burden [16,10].

We here consider three paradigms for modeling and implementation of complex systems: *text-only*, *diagram-first* and *text-first*. Text-only is the classic process where a system is developed entirely as a collection of textual sources. Diagram-first describes a workflow where a diagram is edited directly and text-first describes a workflow where both a textual source and an automatically generated diagram are available side-by-side. Text-only tools are arguably very established and therefore serve as a good baseline for the requirements that system development tools must provide.

In this paper we introduce the KIELER text-first diagramming framework. We argue that there are significant benefits of a text-first approach, and we compare our approach to the diagram-first approach, as exemplified by the Graphical Language Server Platform (GLSP).

We aim to answer the research question: “How do text-only, diagram-first, and text-first approaches compare in the context of MDE?” In this paper, we lay a stronger focus on the diagrammatic aspects because we argue that text-first solutions to browse and understand complex models, via e. g. finding definitions or showing documentation on hover, are already widely implemented and used. Our contributions are

- a comparison of the concepts, technology, and motivation behind the text-first KIELER framework and the diagram-first approach, as exemplified by GLSP in the context of our assumptions about the MDE requirements of developing complex systems in Sec. 3,
- an overview of the generic APIs offered by the KIELER framework in Sec. 4.1 including the diagram synthesis, options sidebar, and semantic filtering,
- a summary of modern browsing techniques for large, hierarchical diagrams in KIELER such as off-screen element visualization, semantic zooming, and the novel top-down layout approach in Sec. 4.2, and
- our vision of the future of diagram technologies in Sec. 5.

## 2 Related Work

We will first discuss diagramming frameworks other than KIELER and GLSP for a broader overview, as well as some users of such diagramming frameworks.

In addition to KIELER and GLSP there are multiple frameworks and tool-boxes for MDE with support for diagrams. Sirius [25] and Sprotty<sup>1</sup> are examples of low-level diagramming frameworks. Sirius supports customization of elements through palettes and configuration views within Eclipse, which automatically generate diagrams for different models. However, these diagrams do not support advanced browsing techniques for large hierarchical models that we discuss in this paper but only zooming and panning. Sprotty is a fully customizable framework for building diagramming tools on the basis of SVG that KIELER and GLSP use as their foundation and provide generic solutions for common diagram-related tool improvements.

Ptolemy II [5] is an older tool that uses a window-based diagram-first solution to browse complex diagrams of hierarchical models. In contrast to other diagram-first solutions presented here, nested subgraphs are visualized in separate windows and not via compound graphs, which offloads parts of the layout to the user and the operating system’s window management. KIELER and GLSP support the integration of hierarchical layout through interactions such as expand/collapse and with top-down layout (see Sec. 4.2).

The bigER [9] tool is an implementation of a modeling tool for Entity Relationship (ER) diagrams, which offers both text-first editing and diagram-first editing using Sprotty similar to the KIELER approach but without the advanced browsing techniques for large hierarchical models presented here. The bigUML

<sup>1</sup> <https://sprotty.org/>

tool [14] developed later by the same group employs the diagram-first approach for UML diagrams, based on GLSP.

Petzold et al. [18] developed a tool (PASTA) to aid with System-Theoretic Process Analysis (STPA). The tool uses automatic visualization with a text-first approach, which sets PASTA apart from other STPA tools that are either purely textual or purely graphical. In particular, the purely graphical modeling is noted to be particularly tedious, mainly due to the size of the graphs and the non-linear workflow. Petzold et al. build on Sprotty and heavily utilize filtering, showing multiple different views of the underlying model to deal with large graphs.

The coordination language Lingua Franca (LF) also comes with an automatic visualization using the text-first approach [13]. LF programs are developed as textual code in one or more source files, and the diagram serves mainly as an abstract high-level overview over the program. Code snippets in different target languages, so-called detailed reaction code, is usually filtered out of the diagram as it would only distract from the high-level view. Therefore, LF diagrams are always highly filtered. LF uses KIELER for its diagram synthesis and it is possible to utilize the advanced browsing techniques presented here out of the box.

The variety of frameworks and tools that use KIELER and GLSP highlights the need for diagrammatic representations. However, we also want to highlight the importance of an editable textual source in addition to a diagram.

### 3 Modeling Paradigms for Model-Driven Engineering

Before discussing editing paradigms for diagrams we must first discuss the context in which we consider the application of diagrams. We are specifically interested in diagram tools that can support diagrams of large, complex models that may include some form hierarchy. We propose the requirement that diagrams should always provide a degree of readability, both in overview and detail, that is similar to the readability of small diagrams.

We define two diagram editing approaches by the way users interact with the model behind the diagram.

The *text-first* diagram approach revolves around editing the source model textually, while a diagram is automatically synthesized in real time. A textual editor and diagram can be used side-by-side, as seen in Fig. 1. Moreover, the editor and diagram can interact with each other. E. g., clicking on a node or an edge in the diagram navigates the textual editor to its definition or vice versa. The addition of a textual editor lets this text-first approach inherit all advantages of the text-only approach, since the editor can support any text-based IDE and editing features. In the KIELER framework, the text remains the ground truth of each diagram, and any modifications of the model change the textual source. Changing this textual source will automatically generate a new diagram.

Compared to KIELER’s text-first approach, the popular GLSP framework [21,2] practices the *diagram-first* approach. In the diagram-first approach, a user mainly edits via diagram interaction, which modifies the typically invisible, underlying source model. This is a fundamentally different approach that

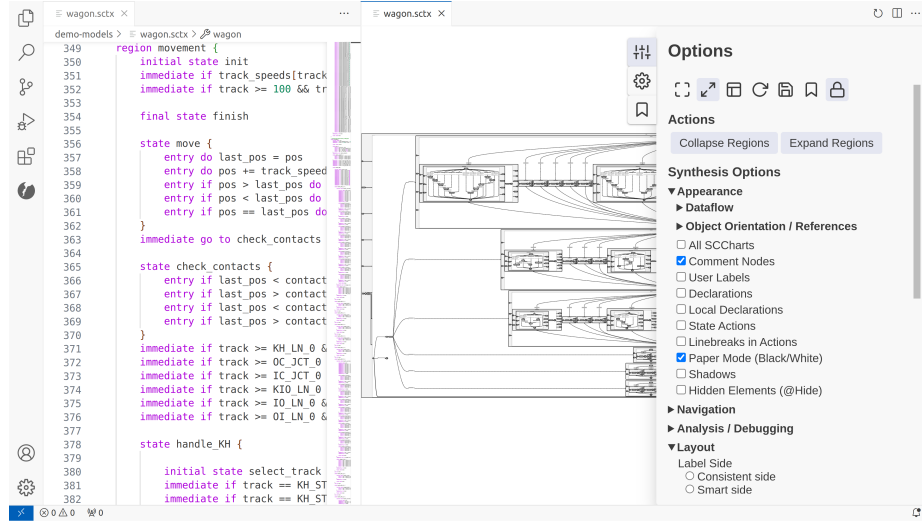


Figure 1: The KIELER SCCharts VS Code extension shows the textual model and the graphical view of the WAGON model side-by-side with opened synthesis option sidebar in the diagram view.

compromises many of the text editing advantages of modeling pragmatics [10], as elaborated on in the following.

### 3.1 Text-First and Diagram-First Editing Paradigms

When developing new tools to support the development of large, complex systems, it is important to keep the strengths of existing technologies and established workflows. Text-only tools are well established and have long been used to model complex systems. Text-first tools employ modeling pragmatics to get the best of a textual and graphical view on the model. E. g., text is easily supported by version management, and it is easy to build tool support in form of content assist, finding references, error and warning markers, and similar standard IDE features. KIELER utilizes all of these features together with a diagram view. The text-first approach utilizes the textual source as a detailed view that can be accurately edited while the diagram can be adjusted to the current needs. GLSP is often configured to have one or more textual models as ground truth. Usually a graphical editor view on the model is built, instead of using a textual editor and a diagram side-by-side, which only allows diagram-first editing. This requires to do all editing operations by interacting with the diagram using palettes, context menus, or input fields.

The text-first framework KIELER was created to relieve the user of the burden of palettes and the need to place everything themselves, by using automatic layout, in the case of KIELER provided by the Eclipse Layout Kernel (ELK) [4]. KIELER is built to use the diagram and text together and is able to map diagram

elements to elements in the textual model and vice versa. This enables users to use the view—textual or diagram—most suitable for the given task. Although GLSP integrates `elkjs`<sup>2</sup> (ELK for JavaScript) for automatic layout, the standard approach involves the user creating and placing everything manually, via diagram interaction either using structure-based editing or palettes.

Diagram-first editing is often argued to be novice-friendly and intuitive. However, this can also be the case for well designed textual languages, as Eumann and Wechselberg reported for railway domain experts than can and do write textual models of SCCharts, a Statecharts dialect, using the KIELER framework without any prior knowledge in programming or SCCharts [6]. Moreover, in real-world models that use diagram-first editing a lot of developer time is spent moving nodes and edges around to manually control the layout [16]. GLSP focuses on such manual control, which is often missing in automatically synthesized diagrams. Automatic layout is rarely able to capture secondary notation [16]. Hence, users are happy with a layout they created themselves and that they can control. However, Domrös et al. [3] show that control over the automatic layout can also be achieved using the order of the textual model. Nevertheless, KIELER partly employs the diagram-first editing paradigm using structure-based editing [11]. Structure-based editing [19] enables users to create a structurally correct model based on diagram interaction using a context menu. In contrast to the WYSIWYG approach employed by GLSP, structure-based editing always employs automatic layout.

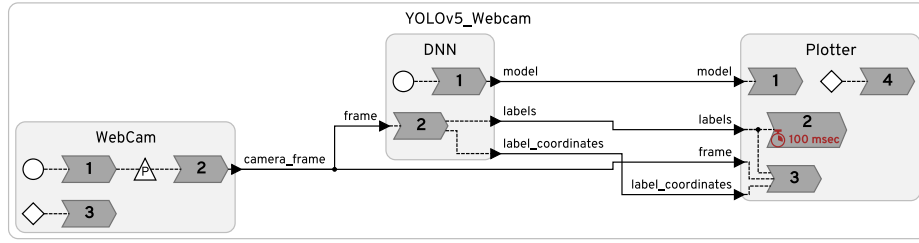
Additionally, filtering does not integrate well into diagram-first editing while it is a first-class citizen in the KIELER framework to handle complex models using transient views [22]. E.g., SCCharts provides an *induced data-flow* view that shows a different graph structure than the regular *state-and-transition* view [26]. GLSP can also filter a diagram. However, if the diagram view is filtered to create a smaller model, the usual approach is to use automatic layout to place the remaining elements to close gaps that would appear between elements and to make the filtered view more readable. Additionally, graphical programming has the disadvantage that coordinates for filtered elements cannot easily be inferred from the unfiltered view, for which they would be necessary. This issue is not inherent to the diagram-first approach if an editing approach that continuously employ automatic layout is used. The issue occurs since manual coordinates and coordinates determined by automatic layout cannot trivially be merged.

Generally, GLSP implements many interesting concepts and both KIELER and GLSP use a very similar technology stack, which might make the two approaches compatible, as detailed in Sec. 5.

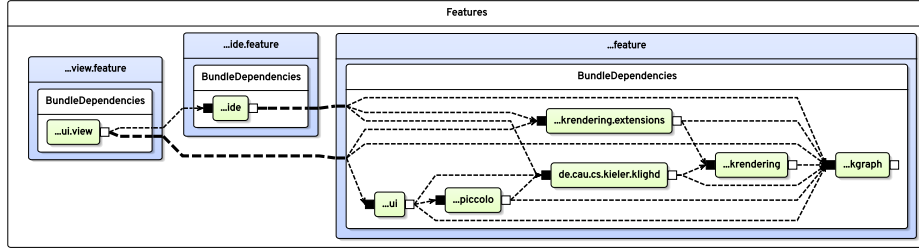
### 3.2 Infrastructure of KIELER and GLSP

KIELER is intended to be a framework mainly for Domain Specific Languages (DSLs) and provides a set of configurable standard features. Fig. 2 shows several diagrams from projects that utilize KIELER. They demonstrate the versatility

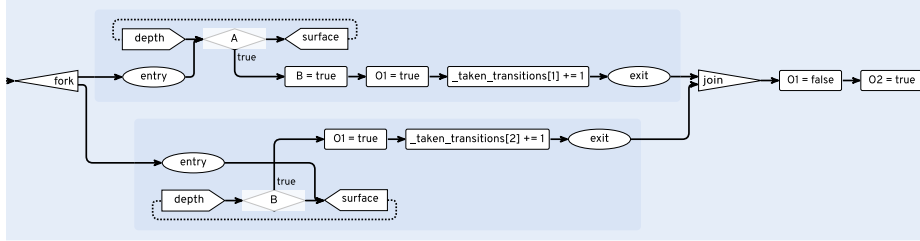
<sup>2</sup> <https://github.com/kieler/elkjs>



(a) A diagram generated from a Lingua Franca program taken from the Lingua Franca playground repository<sup>3</sup>.



(b) A diagram generated using the Software Project Visualization (SPViz) tool presented by Rentz [20].



(c) An excerpt from a Sequentially Constructive Graph (SCG) created during the SC-Charts compilation process [24].

Figure 2: Different diagram applications that use the KIELER framework.

of visualizations that KIELER can create as they have very different underlying models and come from very different domains. In comparison, GLSP is more of a sandbox than a framework by extending Sprotty’s functionality without offering default implementations as provided by KIELER. Instead, GLSP provides a set of tools that enables adopters to build the features they require.

Both KIELER and GLSP utilize Sprotty and its extension to the Language Server Protocol (LSP) to create interactive diagrams. GLSP additionally provides a standard set of diagram interaction mechanisms by extending the LSP. Petzold et al. extended the LSP for their diagram interaction constraint frame-

<sup>3</sup> [https://github.com/lf-lang/playground-lingua-franca/blob/2ea55cc6a35/examples/Python/src/YOLOv5/YOLOv5\\_Webcam.lf](https://github.com/lf-lang/playground-lingua-franca/blob/2ea55cc6a35/examples/Python/src/YOLOv5/YOLOv5_Webcam.lf)

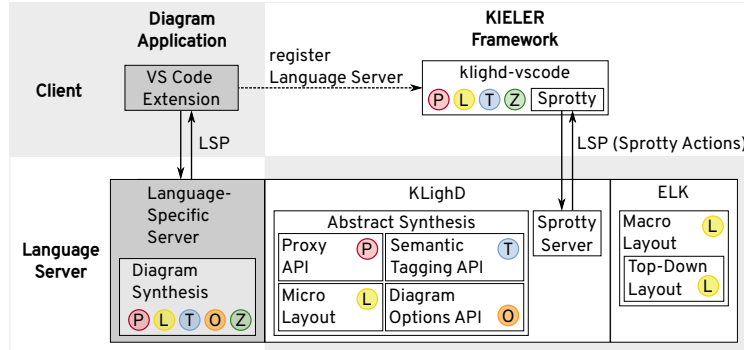


Figure 3: The general architecture of an application built with the KIELER framework. The icons indicate where features discussed in this paper are implemented and configured: proxies, layout, i. e. positioning of shapes and graph elements, semantic tagging, diagram options, smart zoom.

work [17], and we built structure-based editing for KIELER [11] in the same way. Similarly, both utilize and extend the Sprotty server component to create a one-way interaction between diagram and model, i. e. changes are applied to the model and the diagram is updated based on these changes. The technical details were explained by De Carlo et al. [2]. In the following, we discuss the differences between KIELER and GLSP and how they affect diagram interaction.

## 4 KIELER Features

Most of KIELER’s features are agnostic to the diagram type, meaning they work out of the box for general diagrams. The KIELER API directly includes tooling for a diagram synthesis, diagram options including a sidebar, diagram interaction methods, and advanced browsing techniques as presented below. This enables the designers of domain-specific diagrams to quickly configure the diagram interaction, configuration, and visualization.


### 4.1 The KIELER Framework

Fig. 3 is divided into four different sections by two orthogonal areas and illustrates the typical architecture when working with the framework. The first area, the area of responsibility, divides between the *diagram application*, i. e. the part that the diagram designer implements themselves, and the *KIELER framework*, i. e. the technology and its public API. The second area divides between the *client* represented by the two top boxes, i. e. the parts directly executed in the web environment implemented in TypeScript, and the *language server* represented by the bottom box, i. e. the parts implemented in Java or Xtend, connecting to the client via the LSP. Fig. 3 outlines the individual features and highlights the API to be used for individual diagram applications, further discussed below.

**Diagram Synthesis** The diagram synthesis is the point of entry for users of the KIELER framework designing their own diagram application. It is located on the server and connects the configuration of all other features. The main goal is to let the users define how their model is translated to a diagram. Specifically, KIELER Lightweight Diagrams (KLighD) [23] defines a graph and rendering model named *KGraph* and *KRendering*, respectively. It also defines an API to synthesize a model to this graph structure with attached rendering and styling information. The diagram application uses this API to write a diagram synthesis that defines the translation from their DSL model to a *KGraph*. The rendering and styling information define the visual appearance of all graph elements. For example, the simple states from multiple figures in this paper have a rounded rectangle with a certain background color and a centered text inside, and some specific transitions are dashed lines with arrow heads at the end.

Next to the visual appearance of, e. g., lines and boxes, styles can also modify the size of graph elements. Therefore, the diagram generation needs to first estimate the graph element sizes via their styles (called *micro layout*), then do the automatic layout with ELK (*macro layout*), and finally render the diagram. We currently execute both layout steps on the server.

While GLSP configures the micro layout on the server, the absolute element sizes are calculated on the client<sup>4</sup>. This comes at the cost of a larger communication overhead during layouting, as it requires an additional roundtrip between the client and the server because they execute the macro layout on the server. KIELER utilizes KLighD for more advanced server-side positioning [23], while GLSP only supports basic server-side micro layout configuration (padding, horizontal/vertical gap, minimal width/height) [2].

**Diagram Options**  There can be multiple graphical representations of models with varying levels of detail and different focuses. Therefore, we want to give the user options to configure the diagram, so that different views can be shown for each model. As options should be easily accessible to the user, we provide them via an options sidebar in KIELER. Our implementation can be seen in Fig. 1 for SCCharts. The sidebar consists of two parts, the synthesis options depicted in Fig. 1 and the render options, both hosting language-specific options.

The server-side synthesis options filter the model, can configure the layout, or even change what the graphical model representation is in its entirety. The diagram synthesis allows modular configuration of the sidebar for the respective language. E. g., the SCCharts language has buttons to collapse and expand all hierarchical elements, different categories such as **Layout**, and a wide range of check, choice, text, and select boxes to configure the diagram.

The client-side render options configure how the client-side view model interaction works. E. g., whether selecting a diagram element selects the corresponding text or vice versa, whether and how movements are animated, the visualization of layout constraints [17], or what the size threshold of smart-zoom (see Sec. 4.2) is configured to be.

<sup>4</sup> <https://eclipse.dev/glsp/documentation/clientlayouting/>



The sidebar is implemented as a UI extension for Sprotty, so the concept is not limited to text-first frameworks such as KIELER and could also be provided directly in Sprotty so that any diagram-first framework can use it.

**Semantic Tags and Filters** ⓘ The diagram should not only be a static representation of the textual model, but offer the user and the system further information about the model and the semantic elements represented by the diagram. KIELER supports to add such information to the graph by using semantic tags and filters. The KGraph elements that are produced during the diagram synthesis can be tagged to retain semantic information about the original model that would otherwise be lost in the graph and rendering structure. Additional semantic filters can be created and attached to the graph, which may later be used on the diagram client. This allows the KIELER rendering framework to perform client-side diagram-type-specific interactions and overlays, e.g. popups or filtered proxies (see Sec. 4.2). Sprotty and GLSP usually attach such semantic information as properties on the graph elements to be able to work with that information in similar ways.

Semantic tags are user-defined strings that may optionally contain a number value. A tag itself is an atomic filter rule expression. Complex filter rules can be constructed by combining other filter rules with numeric or logical operators.

When applying filters to a set of graph elements, an element is retained if the applied filter rule evaluates to true for that element. The filter rule `#someTag` returns true for an element if it contains the tag `someTag`. Using `$someTag` we can get the number associated with the tag—if the tag does not exist, then 0 is returned instead—and use it for further evaluation. E.g. `#state && $children > 4` evaluates to true for nodes tagged with `state` and `children`, where the `children` tag has a value larger than 4.

## 4.2 Browsing Techniques for Large Diagrams

In the following, we present several browsing techniques that are particularly useful for large, hierarchical diagrams available in KIELER, which are made possible by the infrastructure illustrated in Fig. 3 and outlined in the previous sections. Fig. 4 shows a view of a large, hierarchical model with disabled browsing techniques that we use to compare our techniques to.

**Proxies for Off-Screen Elements** ⓘ *Proxies* have been used in many settings and different strategies exist to place, merge, and interact with them effectively [1,7]. They are a useful visual aid for diagrams that are too large to be easily viewed in their entirety on a computer screen. Our implementation of proxies works on generic KGraphs.

The proxies are realized as an overlay on top of the diagram rendering. By default, a proxy is created automatically from the existing node rendering so that proxies are always available independent of the diagram type. It is, however, also possible to define a custom proxy rendering for each node. In Fig. 5 we see an

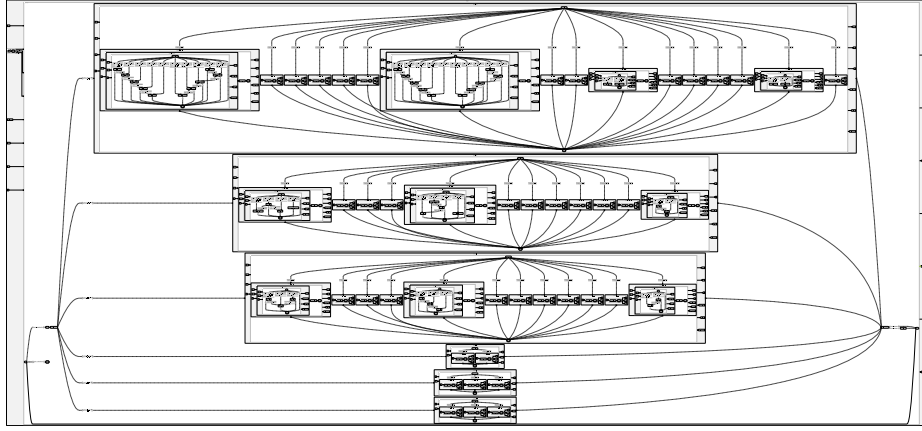


Figure 4: The WAGON SCChart is a large model that was created as part of a model railway project. The examples shown in this paper use this model.

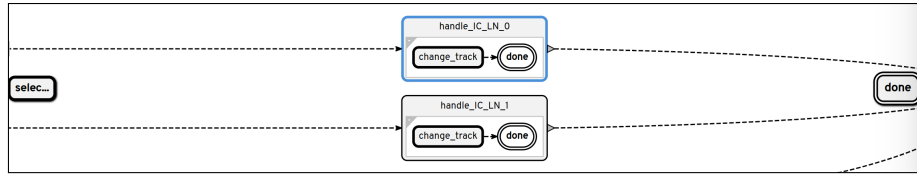


Figure 5: The WAGON model zoomed in with proxies enabled. The renderings on the left and right side are proxies of off-screen nodes. They may be selected to automatically pan the viewport to focus on their respective nodes.

example of synthesis-defined proxy renderings used for SCCharts. Instead of showing the entire state with all information, just the typical state shape and styling with a shortened text label is shown.

When there are many nodes in a diagram, decisions must be made about which proxies to show, otherwise the view becomes cluttered and the benefit of the proxies is diminished. We filter proxies based on their nodes' adjacency and hierarchical inclusion in relation to on-screen nodes.

Filtering based purely on the structure of the graph is often insufficient. If a language defines different types of nodes then we may want to create proxies for one type but not the other. Semantic information is necessary to make this distinction and this is one occurrence of where the semantic filtering API, discussed in Sec. 4.1, is used. Filter rules defined by the synthesis are automatically inserted as toggleable options into the sidebar. This gives a synthesis developer control over client-side rendering behavior.

A common challenge with the introduction of proxies for off-screen elements is visual clutter at the edge of the viewport due to many, potentially overlapping proxies [7,2]. The primary information proxies provide is the direction in which

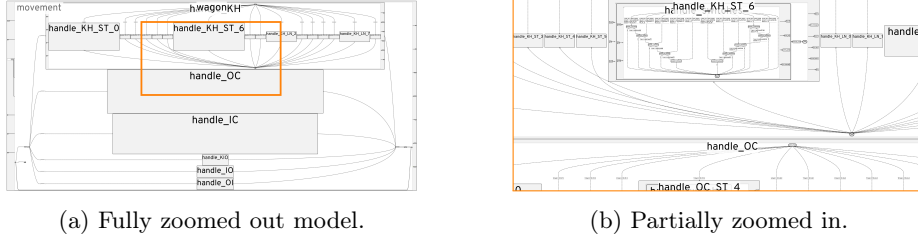



Figure 6: The WAGON model at different zoom levels with smart zoom enabled. Even when zoomed out, the titles remain readable and connections between elements remain visible, while hiding inner behavior to reduce clutter. In Fig. 6a the view is fully zoomed out and in Fig. 6b the view is partially zoomed in on the state labeled `handle_KH_ST_6`.

their nodes lie or the target node of an edge leaving the viewport. Further useful information is the distance to the nodes. When proxies overlap, we draw them with closer nodes' proxies placed on top. Additionally, we decrease the opacity as the distance increases. This technique aims to reduce visual clutter and add helpful visual cues for navigating complex models.

KIELER improves the utility of proxies with semantic tags and filters, which makes it possible to define proxies and a semantic context in the synthesis for further filtering. De Carlo et al. [2] note that this could be done on the client only, but they want to move it partly to the server. In any case, it is important to utilize semantic information about the model for filtering. If proxies are not sufficiently filtered, one cannot use them at all and this is not a matter of merging or overlaying proxies with clever algorithms but a general problem that occurs in complex models.

**Smart Zoom**  Proxies of off-screen nodes provide context when the view is zoomed in on the details of the diagram. However, they do not provide a good overview over a large diagram in the zoomed out state. When viewing large diagrams as a whole it becomes difficult to discern any details. This makes it challenging to build a mental model of the diagram, and it is difficult to determine where to navigate to.

An approach to help with this is semantic zooming, which is an overview-and-detail technique that has seen many iterations and applications. The key idea is to provide different views depending on the current zoom level according to the required level of detail [15,8,2].

We here propose a variant of semantic zooming that we refer to as *smart zoom*. We approach the readability of diagram elements using only zoom and pan operations based on a *static base diagram*. The diagram application can define a part of the rendering of graph elements as its *key rendering* via a semantic tag. Such a tag will only be used within the framework on the client to make such a rendering more legible, while keeping the layout and overall appearance

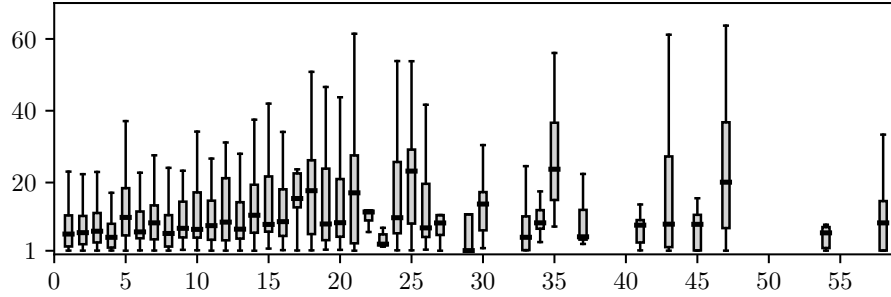


Figure 7: The y-axis represents the node-key width ratio, i. e. the maximum factor by which smart zoom can upscale a key rendering. The x-axis shows the number of scalable key renderings in a graph. The box plots show the distributions of node-to-key ratios for different graph sizes. This data stems from an analysis of 1250 SCCharts collected over approximately ten years from different projects and teaching. We removed outlier ratios that were larger than 75 in order to not hide the ratios in the denser part of the small to medium-sized graphs, which represents the majority of the graphs.

of all graph elements stable based on the layout of the static base diagram. For example, Fig. 6a shows the large model from Fig. 4 using smart zoom, scaled down by approximately 6000%, yet with legibly represented titles. Here we scale up and overlay the titles as the key renderings within the pre-laid-out bounding box of the graph element they represent to a constant 100% scaling, if the space permits, to keep the key renderings legible at many zoom levels. Compared to Fig. 4, this shows better readability.

The framework allows any part of the rendering (names, icons, shapes, etc.) to be the key rendering, for this example it is the text rendering of name of the region. Scaled up key renderings may also overlap other parts of the diagram that are lower down in the hierarchy making the key rendering more prominent. While zooming in, the key rendering will remain at that size until the diagram around it has been scaled up to match its size and any overlaps are then resolved. Furthermore, the client simplifies elements with content that would be too small by hiding the content entirely. Compare the state labeled `handle_KH_ST_6` between Figs. 6a and 6b, where zooming in reveals further inner behavior.

GLSP follows a different concept. De Carlo et al. [2] point out that their semantic zoom re-computes the layout and, therefore, requires extra roundtrips to the server, limiting the performance of their implementation. In KIELER we decided to always base such zoom and pan operations on the layout of the static base diagram to keep the performance high and even increase it by omitting too small elements, while also maintaining a mental map of the model by having a consistent layout where no elements move around due to a new layout.

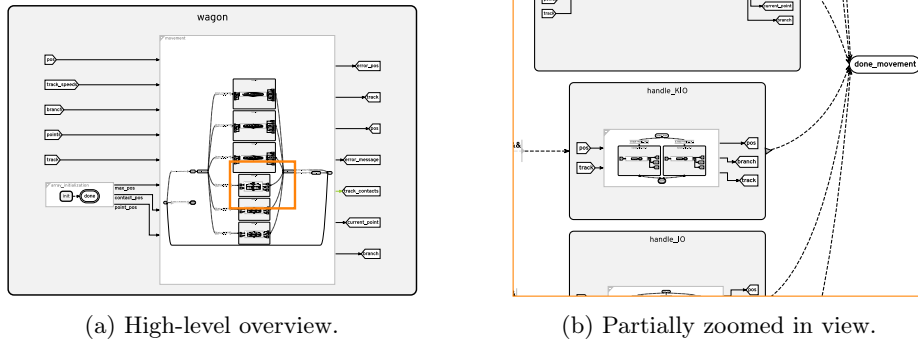



Figure 8: Top-down drawing of the WAGON model used also in the other examples. Fig. 8a shows the entire diagram. Node titles are readable similar to the smart zoom effect. The structure can also be seen without zooming or panning. Fig. 8b shows a zoomed in excerpt. Details of the current layer are readable and the general structure of the model can easily be recognized.

Smart zoom can only work effectively when there is sufficient space to upscale the key renderings, i. e. when the node containing the key rendering is significantly larger than the key rendering itself. At the same time smart zoom is only necessary when diagrams reach a certain size, since there is simply no need for upscaling in small diagrams.

In Fig. 7 we show how *node to key rendering* ratios (*node-key* ratios) are typically distributed in graphs of varying sizes. The examples analyzed are all nested graphs, i. e. graphs with hierarchical containment. This means that nodes can be relatively large since they have to accommodate their children. In these types of graphs the node-key ratios generally tend to become larger in larger graphs. Smart zoom is effective in all cases except for very large graphs without any nesting where each node has roughly the same size as its key rendering.

**Top-Down Layout**  Dynamic adjustments of the displayed diagram are limited in what accessibility features can be realized with them while remaining agnostic to the concrete diagram type. For large diagrams that visualize nested graphs we can also use *top-down layout* [12]. We obtain a similar benefit to smart zoom, i. e. names of nodes are readable when zoomed out but, in contrast to smart zoom, the diagram of one hierarchy level remains quite compact when zooming in. This makes reading and navigating large diagrams a lot easier. Fig. 8 shows the effect of zooming in a top-down layout. The main advantage of top-down layout is that entire hierarchy levels can be read at once. The layout stays consistent and does not depend on the layout of the inner graph in a deeper hierarchy level. Here, zooming is the main control interaction to view details.

We previously examined the effect of top-down layout on the readability of diagrams [12]. The key takeaway from that evaluation was that top-down layout does indeed increase readability of labels in large diagrams across different zoom

levels. GLSP currently does not support top-down layout since it is a relatively new feature of ELK, but there are no technical limitations to supporting it.

The raw data used for the smart zoom and top-down layout analysis is available for download<sup>5</sup>.

## 5 Conclusion

We presented the KIELER framework that utilizes both the textual and the graphical model and provides general solutions for common problems of languages that aim to have a graphical view.

A text-only approach for developing complex systems misses the advantages that filtered, graphical representations provide for some languages. Furthermore, compared to a diagram-first approach on the example of GLSP, the text-first approach, as used by KIELER framework, still provides all the advantages of textual editing, while being able to utilize the best of a graphical view. Filtering is a first-class citizen in KIELER and it is common to hide parts of the model that are better edited textually or that are irrelevant for the current use case. In a diagram-first approach, however, filtering does not integrate well with editing. A well implemented diagram-first approach should not sacrifice the benefits of textual editing but rather emulate and enhance them. A text-first approach naturally does this by building on top of text-only tooling. Linking text and diagram lets users utilize the representation that helps with their current problem.

We presented the KIELER synthesis, options sidebar, proxies, smart-zoom, and top-down layout approach to efficiently work with diagrams of complex systems and help to increase the readability of large human-made hierarchical models and compared each to its GLSP pendant if it exists.

Some features presented in this paper are experimental and have not yet found their way into the official releases of the open-source projects. Nevertheless, we provide a tool demo<sup>5</sup> that is a pair of Visual Studio Code extensions that can be installed locally to test smart zoom, top-down layout, and proxies. The download also includes several example models and a usage guide.

## Future Work and Outlook

The KLighD component currently implements many features of the KIELER framework. In the long run, they would serve better as modules within the Sprotty framework for languages that fit the KIELER approach to modeling. Hence, we plan to migrate part of the KLighD micro layout capabilities as well as proxies and the synthesis option sidebar into Sprotty itself. This would not only allow more users to easily configure diagrams for their textual language, but specifically allow using them together with GLSP. Furthermore, we plan to make more of our framework available on the client-side such as the micro and macro layout. Currently, our synthesis API is quite restrictive as a synthesis must be

<sup>5</sup> <https://doi.org/10.6084/m9.figshare.25304083>

written in Java — this could be expanded with a more open API, e. g., a JSON schema, allowing syntheses in other languages.

## References

1. Burigat, S., Chittaro, L., Gabrielli, S.: Visualizing locations of off-screen objects on mobile devices: A comparative evaluation of three approaches. In: Proc. 8th Conf. on Human-Computer Interaction with Mobile Devices and Services. pp. 239–246. ACM (2006). <https://doi.org/10.1145/1152215.1152266>
2. De Carlo, G., Langer, P., Bork, D.: Advanced visualization and interaction in GLSP-based web modeling: Realizing semantic zoom and off-screen elements. In: Proc. 25th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 221–231. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3550355.3552412>
3. Domrös, S., Riepe, M., von Hanxleden, R.: Model order in Sugiyama layouts. In: Proc. 18th Int. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP. pp. 77–88. INSTICC, SciTePress (2023). <https://doi.org/10.5220/0011656700003417>
4. Domrös, S., von Hanxleden, R., Spöemann, M., Rüegg, U., Schulze, C.D.: The Eclipse Layout Kernel. arXiv preprint, arXiv:2311.00533 [cs.DS] (2023). <https://doi.org/10.48550/arXiv.2311.00533>
5. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (Jan 2003). <https://doi.org/10.1109/JPROC.2002.805829>
6. Eumann, P., Wechselberg, N.: Application of SCCharts in the railway domain (2023), <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/Synchron23/Day1/Day1-0900-Eumann-SCChartInRailway.pdf>, Int. Open Workshop on Synchronous Programming
7. Frisch, M., Dachsel, R.: Visualizing offscreen elements of node-link diagrams. Information Visualization **12**(2), 133–162 (2013). <https://doi.org/10.1177/1473871612473589>
8. Frisch, M., Dachsel, R., Brückmann, T.: Towards seamless semantic zooming techniques for UML diagrams. In: Proc. 4th ACM Symp. on Software Visualization. pp. 207–208 (2008)
9. Glaser, P.L., Bork, D.: The bigER tool - hybrid textual and graphical modeling of entity relationships in VS Code. In: 25th Int. Enterprise Distributed Object Computing Workshop. pp. 337–340. IEEE (2021)
10. von Hanxleden, R., Lee, E.A., Fuhrmann, H., Schulz-Rosengarten, A., Domrös, S., Lohstroh, M., Bateni, S., Menard, C.: Pragmatics twelve years later: A report on Lingua Franca. In: 11th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation. LNCS, vol. 13702, pp. 60–89. Springer, Rhodes, Greece (Oct 2022). [https://doi.org/10.1007/978-3-031-19756-7\\_5](https://doi.org/10.1007/978-3-031-19756-7_5)
11. Jöhnk, F.: Structure-based editing for SCCharts. Master thesis, Kiel University, Department of Computer Science (May 2022), <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fej-mt.pdf>
12. Kasperowski, M., von Hanxleden, R.: Top-down drawings of compound graphs. arXiv preprint, arXiv:2312.07319 [cs.DS] (December 2023). <https://doi.org/10.48550/arXiv.2312.07319>

13. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a Lingua Franca for deterministic concurrent systems. *ACM Trans. on Embedded Computing Systems (TECS)* **20**(4) (May 2021). <https://doi.org/10.1145/3448128>
14. Metin, H., Bork, D.: On developing and operating GLSP-based web modeling tools: Lessons learned from bigUML. In: 26th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 129–139. IEEE (2023). <https://doi.org/10.1109/MODELS58315.2023.00031>
15. Perlin, K., Fox, D.: Pad: An alternative approach to the computer interface. In: Proc. 20th annual Conf. on Computer Graphics and Interactive Techniques. pp. 57–64. ACM, New York, NY, USA (1993). <https://doi.org/10.1145/166117.166125>
16. Petre, M.: Why looking isn’t always seeing: Readership skills and graphical programming. *Comm. ACM* **38**(6), 33–44 (Jun 1995). <https://doi.org/10.1145/203241.203251>
17. Petzold, J., Domrös, S., Schönberner, C., von Hanxleden, R.: An interactive graph layout constraint framework. In: Proc. 18th Int. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP. pp. 240–247. INSTICC, SciTePress (2023). <https://doi.org/10.5220/0011803000003417>, with accompanying poster
18. Petzold, J., Kreiß, J., von Hanxleden, R.: PASTA: Pragmatic Automated System-Theoretic Process Analysis. In: 53rd Int. Conf. on Dependable Systems and Network. pp. 559–567. IEEE (2023). <https://doi.org/10.1109/DSN58367.2023.00058>
19. Prochnow, S., von Hanxleden, R.: Statechart development beyond WYSIWYG. In: Proc. 10th Int. Conf. on Model Driven Engineering Languages and Systems. LNCS, vol. 4735, pp. 635–649. IEEE, Nashville, TN, USA (Oct 2007). <https://doi.org/10.1007/978-3-540-75209-7>
20. Rentz, N., von Hanxleden, R.: SPViz: A DSL-driven approach for software project visualization tooling. arXiv preprint, arXiv:2401.17063 [cs.SE] (Jan 2024). <https://doi.org/10.48550/arXiv.2401.17063>
21. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: Towards a Language Server Protocol infrastructure for graphical modeling. In: Proc. 21th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 370–380. ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3239372.3239383>
22. Schneider, C., Spönemann, M., von Hanxleden, R.: Transient view generation in Eclipse. In: Proc. First Workshop on Academics Modeling with Eclipse. Kgs. Lyngby, Denmark (Jul 2012)
23. Schneider, C., Spönemann, M., von Hanxleden, R.: Just model! – Putting automatic synthesis of node-link-diagrams into practice. In: Proc. Symp. on Visual Languages and Human-Centric Computing. pp. 75–82. IEEE, San Jose, CA, USA (Sep 2013). <https://doi.org/10.1109/VLHCC.2013.6645246>
24. Smyth, S., Motika, C., von Hanxleden, R.: A data-flow approach for compiling the sequentially constructive language (SCL). In: 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung. Pörtlach, Austria (Oct 2015)
25. Vujović, V., Maksimović, M., Perišić, B.: Sirius: A rapid development of DSM graphical editor. In: 18th Int. Conf. on Intelligent Engineering Systems. pp. 233–238 (2014). <https://doi.org/10.1109/INES.2014.6909375>
26. Wechselberg, N., Schulz-Rosengarten, A., Smyth, S., von Hanxleden, R.: Augmenting state models with data flow. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of his 60th Birthday. pp. 504–523. LNCS 10760, Springer International Publishing (2018). [https://doi.org/10.1007/978-3-319-95246-8\\_28](https://doi.org/10.1007/978-3-319-95246-8_28)