A Top-Down Approach on Automatic Graph Visualization

Maximilian Kasperowski

Master's Thesis
October 2021

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
M. Sc. Niklas Rentz

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Automatic visualization of hierarchical graphs is a complex process involving several distinct phases. A common approach is to perform the entire process in an all-in-one bottom-up approach. This is in part because a typical requirement is that the sizes of innermost elements should determine the sizes of the outer elements, in order to allow drawings that are readable on a printed medium.

The KIELER and KEITH projects from Kiel University contain such a bottom-up graph visualization framework to generate diagrams for specific languages. For very large diagrams the existing process does not scale well, motivating the search for alternative approaches to graph visualization.

Top-down graph visualization is one such alternative idea, which promises better scalability and responsiveness for interacting with large diagrams by moving from a monolithic process to a more incremental approach. This allows the user to view parts of the diagram before everything has been loaded. Furthermore, top-down graph visualization lets us rethink some of the requirements of the diagrams produced by automatic layout. In particular, top-down graph layout can help to produce diagrams that are more suitable for viewing on a zoomable computer screen.

This thesis approaches the topic of top-down graph visualization in two aspects. We discuss the necessary architectural changes and apply the iterative concept to server-client communication of KEITH. We also present a concept for top-down layout of graphs. The ideas presented are implemented for KIELER and KEITH, but serve as a more general guide to how top-down visualization can be achieved.

Acknowledgements

First of all, I would like to thank Prof. Dr. Reinhard von Hanxleden as head of the Real-Time and Embedded Systems Group for providing me with the opportunity to work on this interesting and innovative topic, and for the helpful and constructive feedback he gave to me.

I would also like to thank Niklas Rentz, who proposed the initial idea of top-down graph visualization, for supervising my thesis and helping me prioritize the right areas. He was always available for quick discussion when I needed to decide the next steps.

I would also like to thank the whole working group for providing a welcoming and friendly virtual office environment, especially considering the challenges related to the ongoing pandemic.

A special thanks goes to Sören Domrös for helping me get started with the development in ELK, which ended up becoming an important part of my thesis.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DOM	Document Object Model
ELK	Eclipse Layout Kernel
GPU	Graphics Processing Unit
HCI	Human-Computer Interaction
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KEITH	Kiel Environment Integrated in Theia
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
KLighD	KIELER Lightweight Diagrams
LoC	Lines of Code
LSP	Language Server Protocol

QUIM Quality in Use Integrated Measurement

SCCharts Sequentially Constructive Charts

SVG Scalable Vector Graphics

VS Code Visual Studio Code

Chapter 1

Introduction

Visual media are a useful tool in conveying complex information with many connected components concisely. Therefore, visualization of complex systems lends itself naturally to all areas of system design and engineering. Traditionally these diagrams had to be created manually, which is a very time-consuming process and requires a lot of extra work when changes occur.

With the growing use of computers as tools much of this work could be automated. Specifically for this thesis, *automatic graph layout* is of special interest. This means taking a description of the components of a graph as input and computing a suitable layout of these components in order to create a drawing of the graph.

Stemming from the times when these diagrams were often printed out on large sheets of paper, a typical constraint has been that the smallest components of a graph dictate how much space larger components require. Therefore, the typical approach for laying out deep hierarchical graphs is working bottom-up, laying out the small internal parts first and then the surrounding pieces. The result is that drawings of large graphs will use a large area, which can make it difficult to find details or even to get an overview. A way this clutter of information can be alleviated is filtering parts of the diagram.

While this is a necessary method for printed diagrams, nowadays most work is done on computer screens, where users can enlarge or shrink visualizations as much as they want. So instead of sticking to the bottom-up approach, it might be beneficial to explore the advantages of a top-down perspective. What this means more specifically is that instead of making outer components larger so that their internals have enough space, we just give them a certain amount of space and make the internals smaller to fit.

Sequentially Constructive Charts (SCCharts) is a visual programming language developed and maintained by the Real-Time and Embedded Systems Group of the Department of Computer Science at Kiel University [HDM+13]. Currently two tools exist which implement the language. The first is the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER), an extended Eclipse client, and the second is the Kiel Environment Integrated in Theia (KEITH), which aims to bring the KIELER Integrated Development Environment (IDE) experience to the web [Dom18; Ren18]. These tools implement automatic graph generation from textual descriptions and the subsequent automatic layout computation necessary to render visualizations of these models. These systems are where the theoretical ideas of this thesis are put into practice.

A particular challenge posed by the bottom-up approaches implemented in both systems concerns responsive performance, especially as graphs get bigger. This is particularly pronounced in KEITH, since it reuses a large part of the existing architecture also used by KIELER and then adds its own overhead in several places.

A benefit of a top-down paradigm is being able to process a graph bit by bit. Implementing this fully would require re-designing the entire diagram visualization process, which is a significant undertaking. As part of this thesis this idea is applied to the part of the process which was the most significant bottleneck. This proof-of-concept helps demonstrate feasibility and highlights the challenges that come with the idea.

The other part of this thesis applies the top-down idea directly to the layout process itself. Several variations and approaches of how to do this are explored and the benefits regarding usability and aesthetics criteria are evaluated as well as general implementation feasibility within the existing system architecture.

1.1 Outline

The thesis is structured as follows. After covering the necessary background knowledge and state of the art in Chapter 1, Chapter 2 will take a deeper look at the architectural design used in KIELER and KEITH. This is used as the foundation for analyzing how the process could be modified to support an incremental top-down data flow. This idea is applied to the communication between KIELER Lightweight Diagrams (KLighD) and the KEITH client. Finally, the performance and usability impact is evaluated.

Chapter 3 delves into the other end of the system, the Eclipse Layout Kernel (ELK), where we look into using a top-down perspective to create new types of graph layout. The results are evaluated according to aesthetics criteria.

In Chapter 4 the concepts are brought back together to return to the overall theme of top-down visualization. How well do the developed ideas work in the existing system? How well can they be expanded further and what immediate benefits do we have?

In Chapter 5 the findings are summarized and further areas that need work or are of research interest are highlighted.

1.2 Background

Before going into the details of top-down visualization, we need to lay foundations and define the terminology used throughout this thesis. In Section 1.2.1 necessary graph theoretical concepts are introduced. Section 1.2.2 covers the general concept of graph drawing, Section 1.2.3 introduces the measures used to evaluate the quality of these drawings, and in Section 1.2.4 the concept of a mental map and why it is important is introduced. Because part of the motivation of this entire topic is the question of how to improve usability of the KEITH web platform, some of the relevant core principles of building responsive web pages are discussed in Section 1.2.5. How all these concepts are tied together and implemented within KIELER and KEITH is explained in Section 1.2.6.

1.2.1 Graphs

1.1 Definition. (Graph). A *graph* G is a pair (V, E), where V is a set of *vertices* (or *nodes*) and E is a set of *edges* with $E \subseteq V^2$.

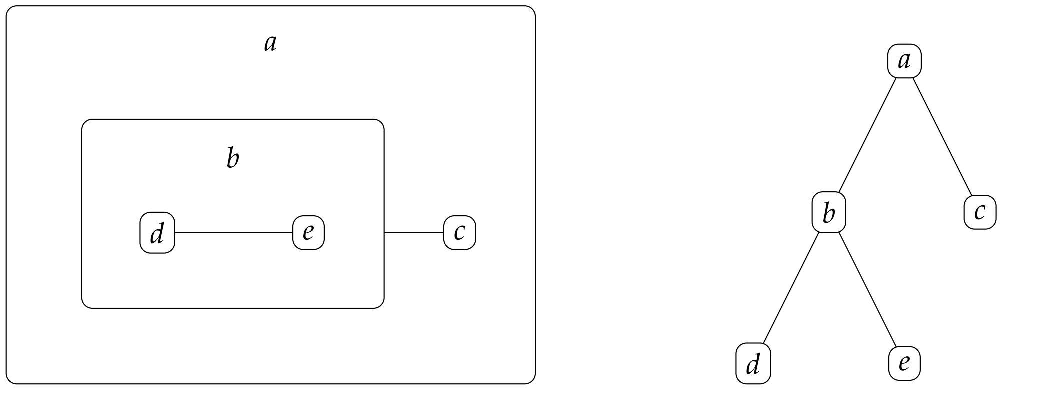
▷ A pair of vertices $(u, v) \in V$ is called *adjacent* if and only if $\exists e \in E$ with $e = (u, v)$ or $e = (v, u)$.

▷ A vertex $v \in V$ is *incident* to an edge $e \in E$ if and only if $\exists e \in E, \exists x \in V$ with $e = (v, x)$ or $e = (x, v)$.

Our main topic of interest is drawing diagrams. These diagrams are represented more abstractly by graphs. A *drawing* of a graph is an embedding of the graph in a plane, where nodes have specific positions and dimensions and edges are drawn along a path between their incident nodes.

1.2 Definition. (Hierarchy function). Let $G = (V, E)$ be a graph. The *hierarchy function* $\tau : V \to V \cup \{\bot\}$ maps each node $v \in V$ to its *parent* node or to $\bot$ if it does not have a parent node. If $\tau(v) = w$ is the parent, then v is known as the *child* of w. A valid hierarchy function τ has the requirement that there is no sequence of nodes $v_1, \ldots, v_n$ with $\tau(v_{i+1}) = v_i$ for all $i < n$ and $\tau(v_1) = v_n$, i.e., the parent-child relation is acyclic.

A certain type of graph that we are particularly interested in are *hierarchical graphs*. A hierarchical graph is a graph that is extended by a hierarchy function. A hierarchy function defines a parent node for each node in a graph. This corresponds to a drawing where a child node is drawn within the bounds of its parent node. Figure 1.1 shows an example visualization of a hierarchical graph as well as a visual representation of its hierarchy function. If there is only one root node the hierarchy graph is also called an *inclusion tree*.



(a) Visualization of hierarchical graph G

(b) Hierarchy graph visualizing τ.

Figure 1.1. Hierarchical graph $G = (V, E)$ with $V = \{a, b, c, d, e\}, E = \{(d, f), (b, c)\}$ and hierarchy function $\tau = \{a \to \bot, b \to a, c \to a, d \to b, f \to b\}$.

1.2.2 Automatic Graph Layout

1.3 Definition. (Graph layout). Let $G = (V, E)$ be a graph. A *layout* Γ (or *drawing*) of G is a function that maps each vertex $v \in V$ to a *point* $\Gamma(v) \in \mathbb{R}^2$ in a plane and each edge $(u, v) \in E$ to a mapping $\Gamma(u, v) \in (\mathbb{R}^2)^*$ between two *endpoints* $\Gamma(u), \Gamma(v)$ with no repeated points.

The problem of automatic graph drawing aims to find algorithmic solutions for a specific class of data presentation problems of modeled data such as circuit schematics and software engineering diagrams or also data analysis results. In general, a graph drawing algorithm takes a graph G as input and produces a layout Γ such that the drawing may be rendered on a screen or printed on physical media. As stated in Definition 1.3, each vertex is mapped to a single point. In practice, however, each node will usually be drawn as some shape with two-dimensional bounds. The details of this are up to the algorithm.

There is a wide array of different algorithms which aim for a certain look or expect certain constraints on their input. Some algorithms are designed for trees while others consider hierarchical graphs [DET+94; Sch19]. Several examples of graph visualizations created using different layout algorithms can be seen in Figure 1.2.

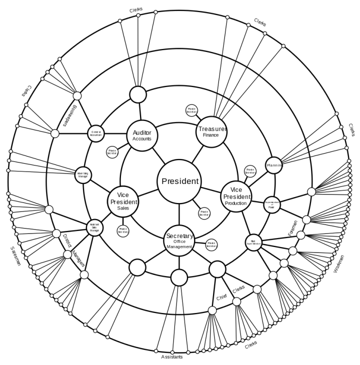
We will mainly be concerned with hierarchical graphs, as these are where top-down concepts may be applied. The typical bottom-up approach to dealing with layout on multiple hierarchy levels is to take a layout algorithm that works on a normal graph, i.e., a graph with a hierarchy function where $\tau(v) = \perp$ for all $v \in V$, and recursively apply that algorithm to each set of nodes V with a common parent $\tau(v) = w$ for all $v \in V$. The space taken up by the layout of these child nodes then determines the drawing size of the parent node, which can then again be used in its own layout step. The benefit of this approach is that *atomic* elements of a diagram, that is, elements with no own children, can all have a similar scale. Before dealing with these innermost parts of a hierarchical graph we have no way of knowing how much space they will require.

1.2.3 Aesthetics

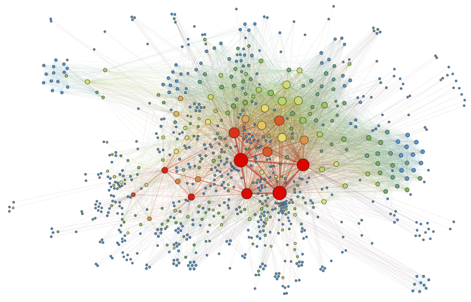
In order to evaluate the quality of an automatic layout algorithm, we need quantitative measures that can be applied to the resulting drawing. These metrics are called *aesthetics criteria* as they attempt to objectively measure how pleasing a visualization is.

There are many different metrics which can be grouped into several categories such as node metrics, edge metrics and overall layout metrics. Some of the metrics can be used in conjunction with each other while others contradict each other [BRS+07]. An interesting observation we can make right away is that SCCharts have so far had the goal of being readable at one zoom level, meaning texts generally have the same size and node sizes are consistent. This ensures that they can be read on printed media.

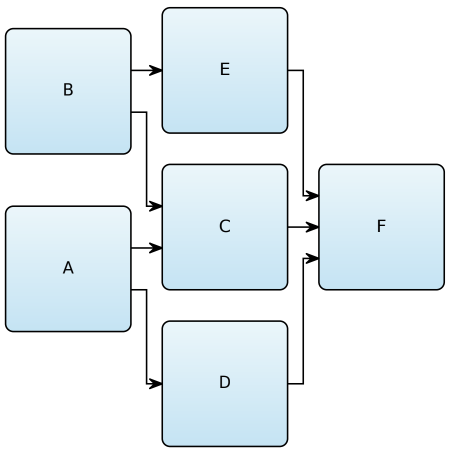
In this thesis a new method of handling layout for hierarchical graphs building upon the already existing layout algorithms is considered. What this means is the underlying layout algorithms remain the same, but they are applied differently or their outputs are modified.



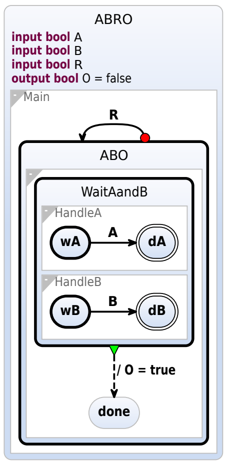
(a) A radial tree drawing of an organization chart [Oys14].



(b) A visualization of a social network graph drawn using a force-based layout algorithm [Gra15].



(c) Drawing of a Directed Acyclic Graph (DAG) laid out using a layered algorithm in KEITH.



(d) SCCharts diagram drawing, an example of a hierarchical graph, drawn by KEITH. Several layout algorithms are combined to create the final layout.

Figure 1.2. Collection of example visualizations of several graphs created using different layout algorithms.

For this reason we are specifically interested in how the resulting visualizations are different and can be interacted with in new ways.

1.2.4 Mental Map

Another important point of consideration will be the preservation of the *mental map* [MEL+95], also known as *dynamic stability* [Nor96]. The mental map is the internal abstract model a person viewing a diagram has. A viewer's mental map contains information such as the relative positions of nodes. For example, if two nodes A and B are drawn so that A is to the left of B, the viewer will expect A to also be left of B after viewing or when making changes to other parts of the diagram.

When updating a diagram, for example when the underlying model is updated, it is important to help the viewer maintain their mental map. Large changes, such as mirroring parts of the diagram or changing the ordering of elements, can disrupt the viewing experience and require the viewer to completely re-parse the diagram [ELM+91].

It is important not to confuse the term mental map with the concept of a *mental model*. A mental model describes internal visualizations as tools which must be functional whereas mental maps are a viewer's abstract representation of the layout of the graph being viewed [LS10].

For the case of top-down layout it is interesting to determine whether updating the new top-down layout preserves the mental map just as well or better than the existing bottom-up approach. Of particular interest will be the effects of scaling parts of the diagram.

1.2.5 Principles of Responsive Web Design

Since part of this thesis will be looking into how to make KEITH more user friendly by becoming more responsive when handling very large graphs, we also need to take a look at some of the important principles of responsive web design. Some of this is more generally true for any user interface, but web technology poses its specific set of challenges.

Before looking at the specific case of web pages, we can look at the more general case of Human-Computer Interaction (HCI). There have been investigations of what constitutes a reasonable delay for different types of interactions such as responses to activation, identification, simple and complex inquiries and more. For tasks that should be perceived to be instantaneous the response time should be less than 0.1 seconds. Delays up to about one second are acceptable when interaction does not have to feel instantaneous, but continuity of thought should not be interrupted. Once delays start taking longer than about 10 to 15 seconds users may start doing secondary tasks, leaving their problem-solving state of mind [Mil68].

We can easily apply these concepts to web pages in general and more specifically to KEITH. Interactions with user interface controls should feel instantaneous, meaning for example when a checkbox is ticked, there should be no noticeable delay. Similarly opening menus and typing in text areas should give an immediate visual update. When editing a model in KEITH the diagram for that model is updated whenever a change is made. In an ideal world this update should take no longer than one second. In certain cases it could be argued that up to 10 seconds might also still be acceptable. Since for large models this is currently not possible, we need to find a way to reduce the perceived delay.

A way to improve responsiveness when loading a lot of data that needs to be displayed is to split the data and load it in parts or, to go a step further, to decrease the data density so that coarse data can be shown quickly and details can be fetched afterwards. This way the user can already see and potentially interact with some of the requested data. Specifically in online map services this is a typically necessary practice to obtain a fast and responsive service [GKW14; Kar11]. Splitting up a single large request into many small requests introduces an overhead and a compromise has to be found for which method is better in which circumstance.

1.2.6 Layout in KIELER and KEITH

Before diving into new ideas we need to first take a closer look at how the whole visualization process currently works in KIELER and KEITH. Since the focus of this thesis is on SCCharts we will go through the transformation process, from model to rendered diagram, they go through. Other types of supported graphs do, however, generally go through mostly the same steps.

After some pre-processing SCCharts models are transformed into a *KGraph* structure using the KLighD framework [SSH13]. This KGraph is then transformed into an *ElkGraph* so that ELK can compute a layout for the graph, which is then applied to the KGraph. SCCharts are hierarchical graphs and layout is currently computed recursively bottom-up. Different types of elements use different layout algorithms. Specifically, SCCharts consist of *regions* and *states*. Regions are arranged using a *rectangle packing* algorithm and states are laid out by the *ELK Layered* algorithm [DLH+21; Sch19]. The layered approach itself was first introduced by Sugiyama, Tagawa and Toda in 1981 [STT81].

In KIELER the graph is then rendered with *Piccolo2D*[1]. In KEITH, the process is a little more involved. KEITH is architecturally split into two parts: a web client, which serves as the user interface, and a *language server*, which provides the client with functionality such as syntax highlighting through the Language Server Protocol (LSP)[2]. This design is used in *Theia*[3], the underlying web IDE that KEITH is based on.

KLighD and ELK are encapsulated in the language server and the process is analogous to KIELER. The laid out graph now has to be sent to the web client and rendered. *Sprotty* [Köh17] is used to render diagrams in KEITH and it internally uses an *SGraph* data structure. Sprotty extends the LSP to support messages regarding diagram generation. KLighD translates the KGraph to an SGraph, which is then sent to the client using the JavaScript Object Notation (JSON) format to be rendered as Scalable Vector Graphics (SVG) in the Document Object Model (DOM) of the browser. Additionally, for correct layout, the text sizes in the diagram are dependent on the browser used and have to be calculated by the client. This means prior to the server-side layout calculation an extra communication step is needed. This *text bounds* calculation is the client-side portion of the layout process.

In all of these steps, the entire graph is always completely processed and passed on to the next stage. The weakness of this system design is that with growing input sizes, i.e., larger models and subsequently larger diagrams, the system hits a limit when the performance is no longer fast enough to meet the requirements of an interactive diagram tool. The goal is to break this process up and explore what advantages alternative approaches can give as well as examining their feasibility in the existing systems. Specifically, this thesis explores the concept of top-down graph visualization in the context of potential performance and usability improvements and also new visualization methods which better utilize the medium of the computer screen.

[1] http://www.cs.umd.edu/hcil/piccolo

[2] https://microsoft.github.io/language-server-protocol/specifications/specification-current/

[3] https://www.theia-ide.org/

1.3 A Top-Down Perspective on Graph Visualization

As we have now established, there are good reasons why graph visualization is often tackled using a bottom-up approach. However, an iterative top-down approach opens the door to many other advantages, and the goal of this thesis is in part to demonstrate feasibility of applying this philosophy to the existing process and also to explore the limitations and benefits of this alternative concept.

For a full conversion to a top-down system, where each part of a diagram goes through the entire process one-by-one, we would need to re-design large parts of KIELER, KEITH, KLighD and ELK from scratch. Because this is a very large undertaking, what we can do instead is make certain modules internally top-down or adapt some interfaces to communicate iteratively.

In KEITH there is a bottleneck in the communication between the language server and the client. The graph structure is sent using JSON and as graphs grow larger this delay becomes very noticeable. A concrete example is the model "Environment_expanded", which is part of the virtual model of a model railway used by the working group. In this thesis this model will also be referred to as "Environment" for brevity. This model is very large, and in KEITH it can take more than 40 seconds for the diagram to be drawn. Therefore, this interface was chosen as the first entry point to apply incremental communication to, since an improvement here would also provide immediate benefits. Going further the idea would be to extend this iterative communication throughout the system.

Incremental communication between KEITH and KLighD was achieved by constructing the SGraph iteratively instead of recursively on the language server and then letting the client request individual pieces of the graph instead of the entire graph all at once. This lets the client already render parts of the diagram, so the user can immediately see something rather than waiting for everything to arrive. Additionally, a simple strategy was implemented to ensure that pieces within the user's viewport were requested before others, so that areas of interest could be shown as soon as possible.

Another idea that is less geared towards how the visualization is created and more oriented to how the result looks. We have two main reasons for approaching this problem from a new perspective.

Firstly, some of the typical constraints for graph visualization stem from the assumption that things need to be on the same zoom level. This means for example that all texts and labels are the same size. However, because we are viewing our graphs on computer screens and can pan freely around in them as well as zoom in and out, we can make the graph more compact and readable at any zoom level.

The second reason is related to the overall system architecture and graph visualization process. If we want a fully iterative process, we need to be able to do that for all parts including layout.

In general, the top-down concept for layout that was for this thesis implemented shrinks nodes using some fixed size constraint and visually scales down its contents to make them fit.

1.4 Related Work

This thesis focuses on the application of graph drawing for dynamically constructing graph drawings of small to large sizes with the particular perspective of exploring top-down techniques. There are several related areas of study with different primary goals. This section will cover some notable examples which share common ideas with this thesis.

1.4.1 Visualization of Very Large Graphs

Kasyanov and Zolotuhin [KZ18] introduce a system for visualizing and navigating very large graphs. The graphs considered were usually too large to efficiently store in memory and computing the layout of them fast enough was another challenge. To address the memory limitation they utilized caching with an embedded SQLite database[4]. This differs from the KEITH system, because we are dealing with a segregated client-server architecture. Furthermore, even the largest graphs we are dealing with are still less than one hundred megabytes large in their serialized JSON form.

What is more interesting for us is their method used to speed up layout time. They use a concept called *multi-aspect layout*. Their use case is interactive exploration of large graphs. To facilitate this with their large input graphs, they construct a multi-aspect layout which is a set of drawings of some subgraphs of the graph. They then use tabs to present these different subgraphs to the user.

Another graph visualization system handling very large graphs with up to 16 million edges is *ASK-GraphView* [AHK06]. The focus here lies again on navigation of large graphs. In this system clustering algorithms are used to construct a hierarchy on arbitrary input graphs. Navigation works in a top-down manner, allowing the user to expand clusters as they descend in the hierarchy.

Although we are also interested in navigating graphs, a major difference in our use case is still there. We are editing the underlying models of the graphs live and want these updated changes to be reflected in the drawn diagram as seamlessly as possible. This is a stark contrast in requirements when compared to only the problem of exploring a fixed, albeit large, model.

1.4.2 Online Dynamic Graph Drawing

Dynamic graph drawing narrows the requirements of the more general problem of graph drawing by requiring that the graph may be modified. This subsequently requires a compromise between fulfilling the expected aesthetics criteria and preserving the viewer's mental map.

Frishman and Tal [FT08] tackled the online version of this problem, meaning the sequence of changes made to the graph were not known beforehand. This means that subsequent layout has to happen very fast.

They build upon the *force-directed layout* in which nodes are modeled as having springs with attractive and repulsive forces between them [FR91]. Their contribution is an algorithm

[4] `http://www.sqlite.org`

that uses the Graphics Processing Unit (GPU) to perform layout. On a GPU parallel computation is the norm, as opposed to a traditional Central Processing Unit (CPU) where single-threaded computation is used. The challenge in leveraging this lies in transforming the calculations needed for the layout into tasks that can be computed in parallel.

An optimization used is the partitioning of nodes to reduce the input size of the algorithm. This is conceptually similar to the top-down concept of applying layout only to outer nodes first, before considering any contained children.

1.4.3 Temporal Granularity in Dynamic Graph Visualization

As already discussed, dynamic graphs are a sequence of graphs representing changes in the graph over time. The work of Burch and Reinhardt [BR17] is an example of offline dynamic graph visualization with an interesting focus. They wanted to show dynamic graphs with variable temporal granularity in one visualization.

The background is that typically a time-to-space mapping is used in which the goal is to preserve the mental map so that subsequent graphs can be easily compared. The downside of time-to-space mappings is that space limitations are reached sooner than a time limit in corresponding time-to-time mappings i.e., animations of the changing graph. Their visualization tool facilitates showing a dense amount of information in the space available as well as enabling a more fine view of the changes across time without breaking the mental model.

1.4.4 Usability Measurement

Part of evaluating software quality is the so called *usability*. Especially of interest for the field of HCI, usability is mainly concerned with how well users can interact with and use a software to fulfill their goals [Bev01].

There are many different standards and models of how to approach the topic of usability during the software development process, and it can be difficult to select an appropriate method fitting the requirements. Models such as Quality in Use Integrated Measurement (QUIM) [SDK+06] aim to consolidate these differences into one unified model which can help to develop a usability measurement theory.

The work done for this thesis only partially deals directly with the interaction between humans and computers and a full usability analysis of the software could be an entirely individual project. It is still important to consider the usability factors though and how additional features may affect them.

Chapter 2

Top-Down Architecture

This chapter gives a detailed overview of the existing architecture and then introduces an alternative top-down concept. We identify where bottlenecks are in the existing system and apply the new ideas to the biggest bottleneck. We will mainly consider the KEITH architecture, because that is where the top-down concepts are applied.

2.1 Current KEITH Architecture

The Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) and the Kiel Environment Integrated in Theia (KEITH) are IDEs with the main purpose of developing SCCharts. KEITH is the newer project, which takes the technologies to the web. This opens the doors to a better accessibility, but comes at the cost of a more complex architecture than KIELER.

First we will take a close look at how the current diagram generation process works in KEITH. This is important to understand where the hurdles are in integrating a different approach. KEITH is split into a web client and a language server. The web client serves as the user interface for the IDE letting users edit code and view diagrams. The language server provides so-called language support to the client. This lets the client use features such as syntax highlighting. In our case we use an extended version of the original LSP that supports diagram related communication.

This general architecture is shown in Figure 2.1. The web client part of KEITH makes use of *Sprotty* to render models. Editing of the textual models also happens on the client. Diagram synthesis and layout are managed by the *KLighD* framework. The client and server communicate changes in the models and laid out diagram using the *LSP*. In the following sections the individual components will be introduced in more detail.

2.1.1 Web Client

At the core of the KEITH client is Theia, a web-based IDE. Theia was designed to be an extensible platform for developing products and it shares similarities with Visual Studio Code (VS Code)[1]. The version KEITH is currently based on relies on a language server and utilizes the LSP. Sprotty is a diagramming framework for the web and drives the diagram rendering in KEITH by extending the functionality of the LSP. It uses a unidirectional cyclic event flow as shown in Figure 2.2 [Köh17].

[1] `https://code.visualstudio.com/`

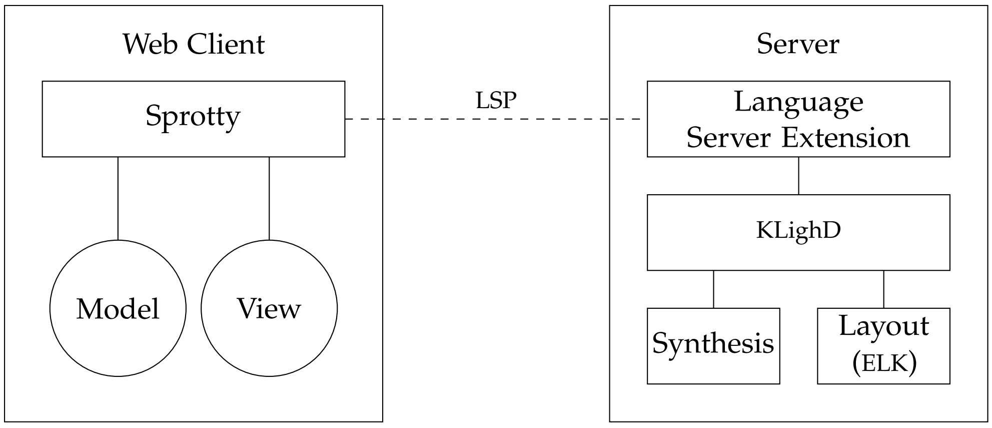


Figure 2.1. Overview of KEITH's architecture.

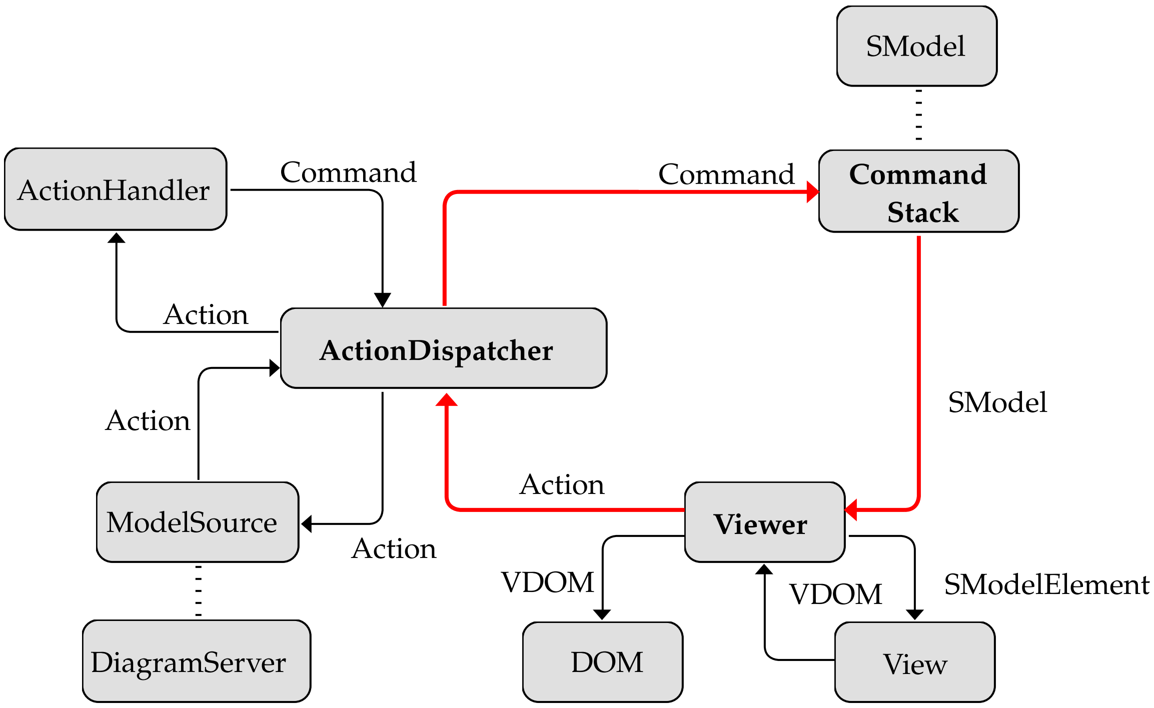


Figure 2.2. Overview of Sprotty's architecture.

The *ActionDispatcher* interacts with the *ModelSource* through *Actions*. The ModelSource can be the locally stored model or a *DiagramServer*. In KEITH for example, a *RequestModelAction* is dispatched to the language server, which responds with a *SetModelAction* that contains an *SModelSchema* (serializable form of an SModel). The ActionDispatcher passes the Action to an *ActionHandler*, which returns a corresponding command. In this case *SetModelCommand* is then pushed onto the *CommandStack*, which is then responsible for executing any commands that have been pushed onto it. In this case the SModel is passed to the *Viewer*, which is then responsible for rendering the individual elements of the model in the browser's DOM. Interactions by the user with the viewer can be sent as new Actions to the ActionDispatcher and the cycle continues.

Work done for this thesis in KEITH builds upon this architecture and expands the existing implementations through new Actions and Commands.

2.1.2 Language Server

The Language Server Protocol (LSP) was conceived as a way to make it easier to support new programming languages in different tools. Traditionally, for each tool the work had to be done individually, because each tool provides its own Application Programming Interface (API). The idea of a *language server* is to provide the language-specific features and communicate with the development tool over a simple protocol. The LSP provides a standardized protocol for the communication between a language server and development tools. This way the language support only has to be implemented once for a language server, and that language server can be re-used many times for different development tools.

For KEITH, Sprotty extends the LSP with actions specifically for generating diagrams. Part of this diagram generation is the automatic layout of the graph, which can be done directly on the client, on a language server, or a mix of both. The final rendering step is performed on the client. The intended usage is that diagrams are generated in one pass.

Sprotty comes with a server-side component that can also handle the Sprotty-specific actions and data structures. These server components can be extended to support new types of diagrams. For KEITH, KLighD contains the extensions of Sprotty and handles the connection to the other components of the system.

2.1.3 Kieler Lightweight Diagrams

KIELER Lightweight Diagrams (KLighD) provides diagramming services and is used by both KIELER and KEITH. In general it provides a framework to connect model sources, diagram synthesis, layout algorithms and viewers. Figure 2.3 shows the architecture and the process used to create model visualizations. Specifically, a textual description is synthesized into a model (Steps 1 and 2). The model is handed back to KLighD (Step 3), which then passes it on to the ELK framework to compute a layout (Step 4). The layout result is then applied to the view model (Steps 5 and 6), which can then be rendered in a viewer. The view model may be changed for example through the use of synthesis options. These changes may require a re-run of the synthesis or the layout and the following process steps (Step 7). It is also possible to modify options that only require a new layout run with the same synthesis. For all these communication steps between the different components KLighD serves as an intermediary framework.

The principles remain the same for the client-server application of KEITH. There is an additional challenge though. The server cannot accurately determine how much space the

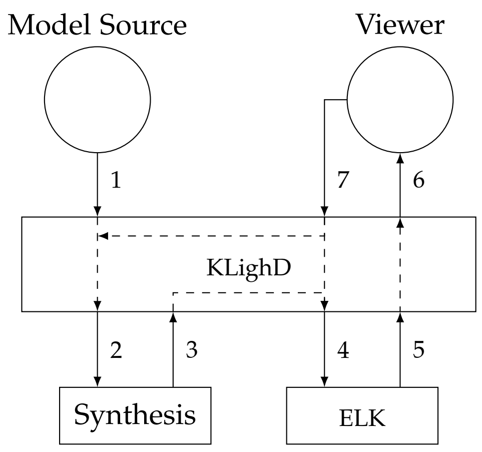


Figure 2.3. Overview of KLighD's architecture based on a diagram made by Christoph Daniel Schulze [Sch19].

rendered texts of the diagram will take up on the client and therefore this part of the layout is done by the client in our case. It is also possible to synchronize fonts between the client and the server, which would let the server compute the required text sizes independently.

2.1.4 Eclipse Layout Kernel

The Eclipse Layout Kernel (ELK) provides tools to connect diagram editors or viewers to automatic layout algorithms. The basic process is as follows. First the graph from the editor must be extracted and converted to an ElkGraph. Then the selected layout algorithm is called and the result applied to the graph. Finally the resulting layout is applied back onto the original graph structure.

ELK can be used by tool developers who want to add layout capabilities to a viewer or editor and algorithm developers who want a graph framework in which they can develop new automatic graph layout algorithms.

ELK itself already comes with a number of standard layout algorithms, such as force-based [FR91], layered [STT81], radial [BET+98], rectangle packing [DLH+21], and more.

Layout of hierarchical graphs is handled by ELK using its *recursive layout engine*. For each node of the graph, a property is set which specifies what layout algorithm should be applied on the node's direct children. The layout engine then recursively computes the layouts bottom-up. Once the children of a node are laid out, the dimensions of that node are known and it can be laid out.

The recursive layout engine can also handle algorithms that can directly lay out hierarchical graphs. For these cases the usual recursive layout calls are not done, and instead the algorithm may manage how the hierarchical layout is calculated.

2.2 Iterative Architecture Concept

The goal of this thesis is the exploration of a top-down approach for creating visualizations, and one aspect of that is the feasibility of applying that abstract idea to the concrete existing solutions implemented in KIELER and KEITH.

In this section a high-level concept of such a fully iterative top-down architecture is described. The system changes that are needed and the solutions that are necessary are discussed.

In principle the idea is to change the pipeline so that instead of performing the synthesis on the entire model, computing the layout on the resulting graph and sending that entire graph back to the client, each step of the process is able to do its job on only a subset of the normal input. This allows to go through the entire process with each individual piece of the diagram.

This concept remains abstract, and in practice we need to identify what parts are able to handle this iterative input. We also need to identify where in the system we would see benefits from this alternative architecture. The quantitative analysis will be covered in Section 2.3.

As has already been touched upon in Chapter 1, we can differentiate between two different types of iterative top-down mechanisms that are necessary to transform the system. Firstly, individual sub-procedures such as layout in ELK, SGraph construction or KGraph synthesis in KLighD could internally be re-designed to work in an iterative manner. The second step is to transform the APIs between the components into ones that also support piece-wise communication and to allow result-merging of partially completed processes.

Because the components are separated, it is possible to only do these transformations in selected parts of the overall system.

2.3 Bottleneck Identification

In order to determine which part of the system could receive the most immediate benefit from an iterative approach, we need to take a closer look at how long each step takes for models of growing sizes.

2.3.1 Benchmark Tests

For this analysis three sample models are used to benchmark these timings. The sizes of the models are given in the number of Lines of Code (LoC). Since the models are synthesized from textual descriptions, LoC are closely correlated to the actual model sizes. Of course, the models themselves can contain different structures that may have smaller impacts in each step, but this is not an extensive analysis. Instead, we are interested in the bigger picture, where the obvious time losses and shortcomings are, so that the right steps can be targeted for improvements. In Table 2.1 the times used for each step of the diagram generation process in KEITH are shown. The steps are grouped by where they take place: client, server or communication between the two.

Table 2.1. Time required in the diagram generation process in KEITH.

	ABRO		Wagon		Environment	
Size (LoC)	25		1541		35 573	
Times	in *ms*	in percent	in *ms*	in percent	in *ms*	in percent
Initial Request	209	78.57	709	46.64	19 287	44.04
Server Steps	**17**	**6.43**	**185**	**12.17**	**4698**	**10.73**
Synthesis	6	2.26	80	5.26	1821	4.16
SGraph	1	0.38	16	1.05	341	0.78
Layout	10	3.76	89	5.86	2536	5.79
Client Steps	**24**	**9.02**	**258**	**16.98**	**4948**	**11.3**
Text Bounds	2	0.75	29	1.91	749	1.71
Rendering	22	8.27	229	15.07	4199	9.59
Communication	**16**	**6.01**	**368**	**24.21**	**14863**	**33.93**
Text Bounds Request	7	2.63	135	8.88	1187	2.71
Text Bounds Response	2	0.75	12	0.79	610	1.39
Model Response	7	2.63	221	14.54	13 066	29.83
Total	**266**	**100**	**1520**	**100**	**43 796**	**100**

Initial Request

The *Initial Request* covers the time from when the textual file of the model is first opened on the KEITH web client until the synthesis on the server begins. This includes sending the textual model to the server and all the SCCharts-specific processing before the diagram generation begins. In the more usual use case that the model has already been opened, this step shrinks down. Instead of sending the entire model upon changes being made, only a change set is sent to the server and applied to the stored model. The SCCharts model, once generated, is also stored on the server, which means that re-triggering the diagram generation process can begin immediately with the synthesis.

Server Process Steps

Synthesis refers to the transformation from domain model to KGraph. Here the domain model is an SCCharts model. *SGraph* refers to the construction of the SGraph on the basis of the KGraph. *Layout* refers to the layout calculation by ELK as well as the mapping of of the layout onto the KGraph and the SGraph. Technically, there are some more steps that happen on the server, but they are not directly part of the diagram generation process, and are therefore included in the *Initial Request* step of the communication category.

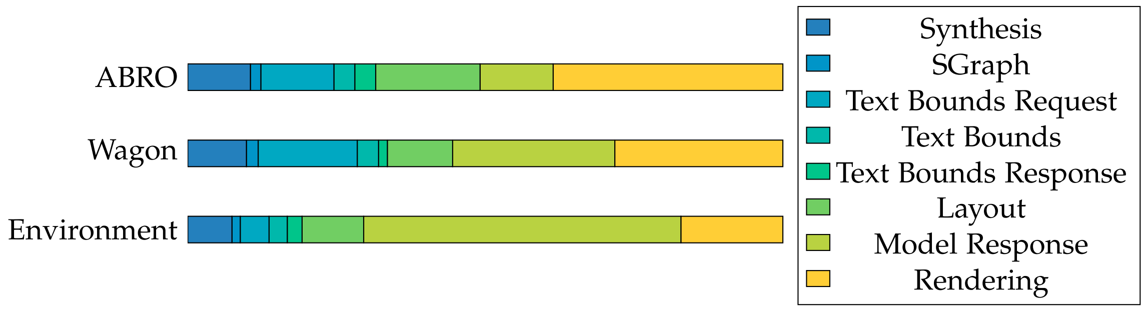


Figure 2.4. Visualization of the length of each step of the diagram generation process beginning with the synthesis on the server.

Client Process Steps

Text Bounds refers to the calculation of the text bounds on the client when the server requests them for its layout computation. *Rendering* refers to the time used to create the SVG renderings used to display the diagram.

Communication Process Steps

The text bounds request and corresponding response steps cover the time used for sending the data between client and server. The *Model* step is the step of the process when the full diagram model is sent to the client to be rendered. In these communication steps the serialization to and from JSON is included in the measurements.

2.3.2 Timing Analysis

As is evident from the data, not all steps of the process are equal. For most of the steps we get roughly linear growth. However, the time required to send the final model to the client grows a lot quicker than the other steps. For the small ABRO model this step only constitutes 2.63% of the entire process, whereas for the largest model this step takes up 29.83% of the process. Another interesting observation is the large overhead of the initial request, which actually drops off significantly as the model grows. This overhead is in practice, however, not particularly relevant, as a model's diagram generation is usually not triggered only once per session. A visualization showing the relative length of the individual steps of the diagram generation process where the initial request is omitted is shown in Figure 2.4.

We are not only going to focus on the time complexity here, but also the absolute numbers to help determine problematic areas of the process. This is because we are most interested in the practical application of reactive live diagram generation, and in this application there are some practical considerations. There is a certain scope of typical diagram sizes that are used, so while scalability is important, we can also just look at the times used and make the observation that 40 seconds wait time until a change to the model is reflected in the diagram is simply not acceptable in this context. To feel responsive, the time until something happens should really be closer to one second. The second model (Wagon) analyzed here already exceeds that boundary and represents a rough model size limit using the traditional bottom-up process.

Even if each step is continuously heavily optimized, there will always remain a limit to the graph size. The iterative concept presented in this thesis opens the door to responsive visualization of large graphs as well.

SGraph generation itself is a fairly fast process, but it provides the entry point to the model transmission step which is the major performance drain. For the largest model investigated here communication takes up 33.93% of the total time used. Therefore, the focus is placed on incremental SGraph generation and piece-wise model transmission.

2.4 Top-Down Approach for Client-Server Communication

This section will give a detailed description of the solutions implemented for this thesis. As already discussed, the main bottleneck where performance is lost is the interface between KLighD and client-side Sprotty, or to put it more precisely, the network communication necessary for sending the model to the client.

The solution presented here consists of several components. On the server the SGraph has so far been constructed recursively. This process is re-designed so it can be performed top-down, which allows a partial on-demand construction of the SGraph. In order to send the model in small pieces rather than all at once, the communication protocol is extended to support requesting and sending specific parts of a graph. Finally, there is the question of how the client should manage its piece requests. For this problem, a naive solution is presented as well as a slightly more sophisticated algorithm which aims to request diagram pieces in an ordering following a prioritization based on the viewport position.

2.4.1 On-Demand SGraph Construction

The SGraph is only required for the final server-side step of sending the model to the client. This means we do not actually need to fully construct it if we are sending the graph in smaller pieces. Only the parts that should be sent need to exist. Layout is calculated on the KGraph and that data is mapped back onto the SGraph afterwards. Because of this, it makes sense to create the SGraph on demand.

Incremental Generator

The SGraph construction process is essentially a traversal of the KGraph, where for each element in the KGraph a corresponding SGraph element is created. The original implementation uses a depth-first traversal. Here, we will use a breadth-first traversal instead. For the most part this is trivial, because the order of visiting the different KNodes does not affect the creation of their SNode counterparts. However, due to the slightly different data structures and specifically how edges are stored in them, special handling of edges is required. In the original implementation this solved by simply generating all the edges at the very end of the construction. This is obviously not a viable approach for the top-down concept. This problem is instead solved by maintaining a list of edges that need to be created. After a piece is created, all these edges are checked and if both nodes incident to an edge already exist it is also created.

Piece Request Manager

The incremental SGraph generator maintains the same interface as the original generator, but it actually has to work differently, because instead of returning the complete SGraph we require the capability to only create one piece per invocation. More specifically, we would like to get specific pieces as requested.

The generator itself is implemented so that it can only generate the next piece of the diagram in a breadth-first traversal. A piece in this case corresponds to a node of the SGraph. In theory pieces could also be larger units containing larger sub-graphs, but in this work a piece is always a single node. With each call to its toSGraph() method the next piece is added to the existing SGraph structure.

In order to handle the requests for specific pieces an additional manager class is added. Every time a request comes in, that request is forwarded to the manager, which returns the desired piece. It maintains a queue of the incoming requests and does several checks. First it checks whether the desired piece has previously been generated. If that is the case, it is simply returned. If it has not been generated before, it lets the generator create pieces until the requested piece is created and returns that.

The piece request manager does not notify its users when all pieces have been requested as maintaining that state information is the responsibility of the using class and in this case specifically the KEITH web client.

2.4.2 Incremental Client-Server Communication

In order to actually realize the incremental communication of pieces between server and client, new Sprotty actions are introduced and the protocol handling the messages on both server and client is expanded. On the client-side the following functionality is necessary.

The client needs to identify pieces to request and to send the those requests to the server. When responses containing pieces arrive, they need to be added to the internal model stored on the server and the diagram has to be re-rendered.

This communication is currently implemented synchronously within the LSP. However, many piece requests are independent of each other and could theoretically also be performed asynchronously. The interfaces on client and server already support this in principle as they both use queues to handle incoming messages and to send responses when they are ready.

Normally, the server does not maintain the state of the diagram generator, but with this incremental approach this becomes necessary. With the initial request of the client the entire process of the synthesis and layout can already be performed as well as the first invocation

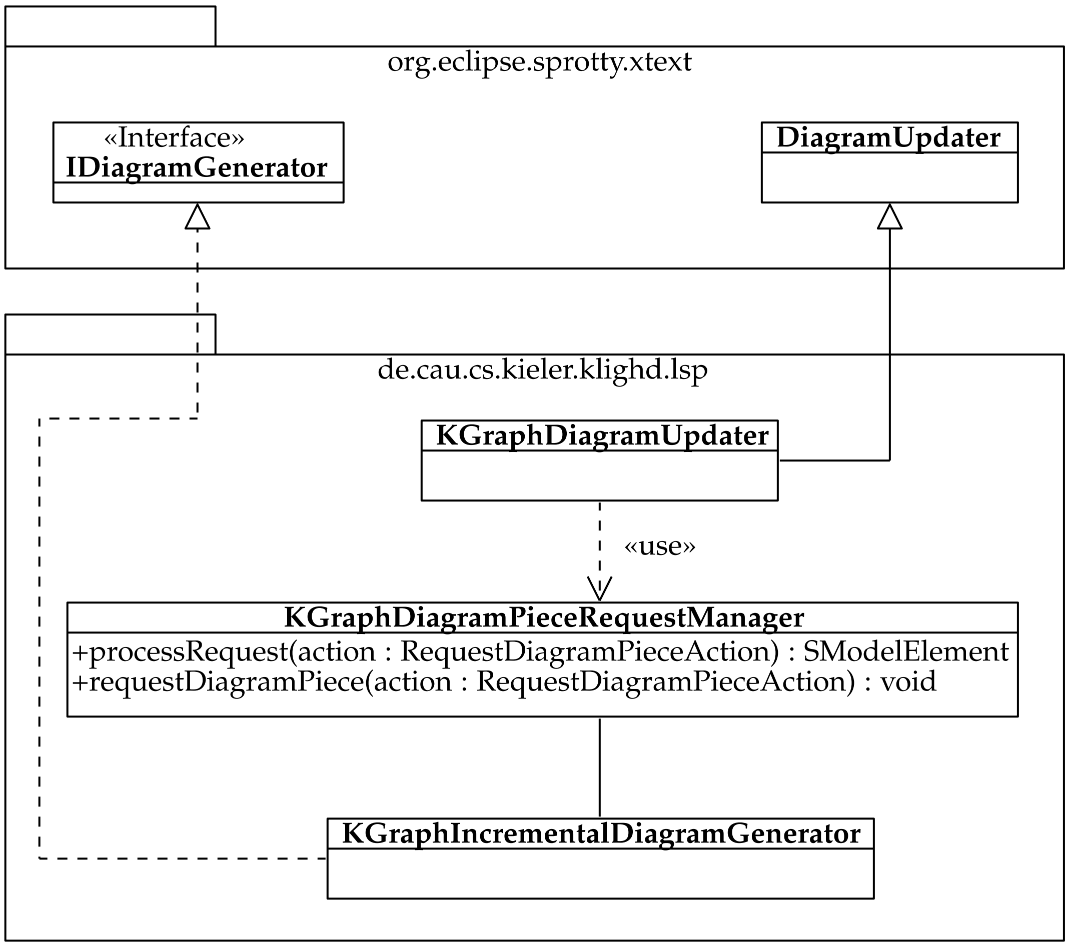

org.eclipse.sprotty.xtext
«Interface»
IDiagramGenerator
DiagramUpdater
de.cau.cs.kieler.klighd.lsp
KGraphDiagramUpdater
«use»
KGraphDiagramPieceRequestManager
+processRequest(action : RequestDiagramPieceAction) : SModelElement
+requestDiagramPiece(action : RequestDiagramPieceAction) : void
KGraphIncrementalDiagramGenerator

Figure 2.5. Class diagram excerpt showing how the piece requests are handed off to the incremental diagram generator.

of the diagram generator. Afterwards, the piece requests are answered until the client sends the final request. If no further child reference stubs are added to the client's queue the client knows that it has received all the pieces it needs.

Protocol

The diagram request process is initiated by the client using the same action to request the model as in the old method. The response, however, does not contain the full diagram model, but only the root with references to its children. The requests for the individual pieces of the graph are sent using a new piece request action. The responses to those requests contain an SGraph model with the specified piece as the root.

Additionally, the text bounds required for the layout still need to be calculated by the client. For this a new incremental text bounds request is used. This additional communication

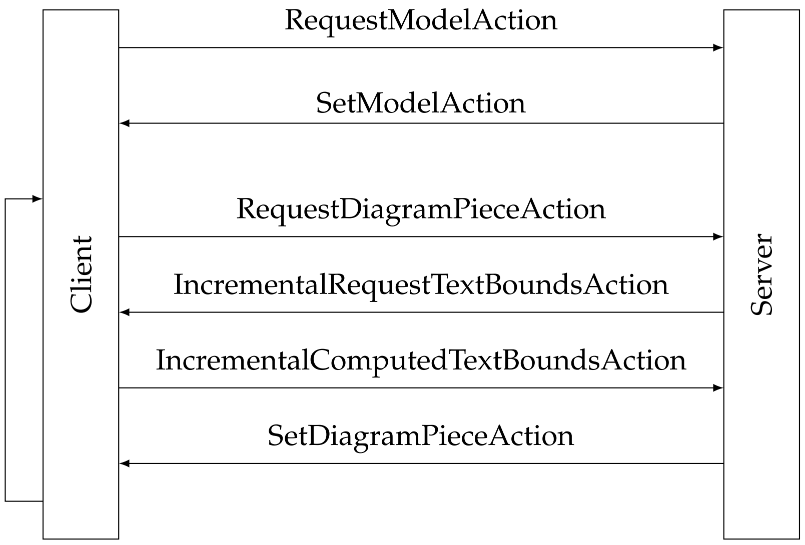

Client
Server
RequestModelAction
SetModelAction
RequestDiagramPieceAction
IncrementalRequestTextBoundsAction
IncrementalComputedTextBoundsAction
SetDiagramPieceAction

Figure 2.6. Incremental message protocol used to transfer the diagram model from the language server to the web client using the LSP.

happens whenever a diagram piece is requested.

Figure 2.6 shows the messages sent in the incremental protocol. *RequestModelAction* and *SetModelAction* are pre-existing actions from Sprotty. Here the semantics of SetModelAction are changed though. It no longer sends the complete model, but only a root stub and it triggers the incremental process. The other actions are all newly introduced, and how they are handled on the client and on the server is explained in the following sections.

Client

After requesting the model from the server, the client receives the first piece of the model, the root. The model is then stored internally by Sprotty. The root contains its child nodes, but without any deep information. Only the IDs of the nodes are stored. The client pushes these references onto an internal queue, which is used to prepare the next piece requests. Then the client gets the next piece reference from the queue and requests the corresponding piece from the server. Arriving pieces again have references to their children, which are added to the queue. Pieces are requested until the queue is empty. Every time a new piece arrives the diagram has to be re-rendered. This has the effect that the initial wait time is short and the user can observe the diagram being built up top-down.

Server

Upon receiving the initial request for the model from the client, the steps up to the SGraph generation are identical to the non-incremental approach. The KGraph is synthesized and layout is performed on it. The incremental concept could still be expanded for these steps though, and this exactly what needs to be done in order to realize a fully incremental architecture. The piece requests are handled as explained in Section 2.4.1.

At this point we are isolating processes and treating them mostly as black boxes. There is an issue with that, however. As part of the layout, the text bounds need to be calculated so that the layout algorithm knows how much space the texts will occupy on the client. This step is dependent on the SGraph construction. Because SGraph construction now happens iteratively, we would need to get the text bounds for each piece and recalculate the layout every time. Here, layout is not performed repeatedly, because that would defeat the purpose of trying to find performance improvements. This currently means that there are some remaining issues with text sizes and positioning. There are two possible solutions for this though. Long term, if the remaining processes can also be performed top-down, this would no longer be an issue. A more short term solution is to move text bounds calculation back to the server by synchronizing the fonts.

2.4.3 Client-Side Request Ordering

With the capability of requesting the diagram pieces node by node from the server comes the question of what order the requests should come in. Since our goal is to use the incremental communication to decrease the wait time until the user can see something, it would be ideal to request whatever should be rendered within the user's viewport.

There are, however, some limitations to what is possible. Whenever a diagram piece is received it comes with references to its children. Because the piece requests work by referencing specific pieces we can only ever request pieces whose parent we have already received. So we need to find a way to determine what piece needs to be requested next to get us closer to the desired viewport location.

Once we have successfully received the pieces that fit in the viewport we are not done. There should be no idling time while building up the diagram. Ideally we request the next pieces in an ordering that maximizes the probability that after a viewport transformation, such as a pan or zoom operation, the pieces in that location are already visible. In the following, two request ordering strategies are introduced, with the grid-based request ordering aiming to fulfill the goal of loading the best pieces first.

Breadth-First Request Ordering

The naive approach is to simply use a breadth-first strategy. As the diagram pieces arrive the child references are simply pushed to a queue of remaining pieces and for every request, the front element of the queue is used until the queue is empty.

This method ensures that all pieces of the diagram will arrive at some point and the diagram will be built up level by level. This solution works well enough when there is no panning or zooming involved, because we can see the entire diagram being constructed uniformly. If we do want to take movement of the viewport into account, a more sophisticated strategy is required.

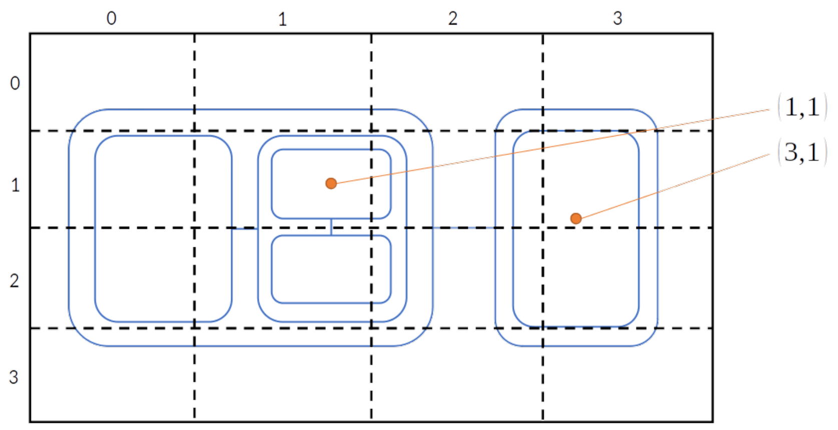


Figure 2.7. Node location to grid cell mapping.

Grid-Based Request Ordering

In this next approach the diagram canvas is divided into a grid with equally sized cells that have a width and height ω. The drawing space is mapped to the grid by a function $f : \mathbb{R}^2 \to \mathbb{Z}^2$. We use $f(x, y) = (\lfloor x/\omega \rfloor, \lfloor y/\omega \rfloor)$. $c_{i,j}$ refers to the cell containing all points $x, y \in \mathbb{R}$ that satisfy $f(x, y) = (i, j)$. As diagram pieces arrive the coordinates of their midpoint are determined as $m = (m_x, m_y)$ and the piece is assigned to the cell $c_{f(m)}$. Figure 2.7 shows how nodes are mapped to grid cells in a simple example.

Because children of a piece are smaller and located within the bounds of the parent piece, we assume that children should also be assigned to the same cells as their parents. In reality this is not necessarily true and depends on how ω is chosen, but it is a good enough estimation in practice. For each cell a queue is created that is initially empty. The child references are stored in the queues of the cells which contain their parent.

When it is time to request the next piece of a diagram, the cell containing the midpoint of the viewport is first computed. Then the first element of the queue corresponding to that cell is requested. If the queue is empty, we first want to get any other pieces that are nearby. The assumption is that the user will be more likely to stay in the region than to move to a very different part of the diagram. In order to determine what pieces are nearby we check the queues of cells in a ring around our center cell. We can check a configurable number of rings, but in any case the first reference found will be requested. If still no child references are found we resort to the breadth-first method until a viewport transformation makes other pieces available again.

It is not trivial to find a good choice for ω and the number of rings to check, because the overall size of the diagram, the sizes of the smallest nodes, and the current zoom level all have an impact on the effect of these settings. With this particular solution the settings have to be set once at the beginning and cannot be adjusted while moving around the viewport. A good rule of thumb for choosing ω is that medium sized nodes should fit inside the cell.

2.5 Performance Analysis

In this section the performance impact of using an incremental client-server communication is analyzed and evaluated. The new process is analyzed and the differences between the classical approach and the new method are highlighted.

First and foremost it should be clear that with the top-down approach we cannot achieve overall faster times than with old method. This is simply due to the fact that we still need to do the same calculations and send the same amount of data in the end. In addition to that, the incremental top-down approach adds a fair amount of overhead, because for each piece of the graph we are now passing data around. With that preliminary consideration we can take a closer look at where we are saving time and where there are still redundancies left.

2.5.1 Initial Request

After the initial model request by the client the KGraph is synthesized and the root of the SGraph is created. Layout is then calculated and applied to the KGraph. This layout data is then mapped onto the SGraph (in this first step just the empty root) and the SGraph is then sent back to the client. The client stores the received model and queues up the child references in the root for future piece requests. The diagram is then rendered and the next piece in the queue is requested.

2.5.2 Piece Requests

For each piece request the following steps happen. Upon receiving a piece request the server generates or retrieves that piece of the SGraph. The text bounds for the current SGraph are requested from the client. The client computes them and sends them back. Then the already existing layout is mapped onto the now larger SGraph structure and the piece that was just generated is sent to the client.

2.5.3 Overview of Performance Impacts

Summarizing, these are the steps and how often they are performed in the top-down approach.

- ▻ *Synthesis* Performed once at the beginning.
- ▻ *Layout* Performed once at the beginning.
- ▻ *SGraph construction* Performed iteratively. The total time used will be a little higher due to additional overhead.
- ▻ *Text bounds calculation* Performed for every piece request. Currently performed for the entire stored SGraph every time. This can be optimized by remembering the texts that have already been sent and only actually transmitting new texts.

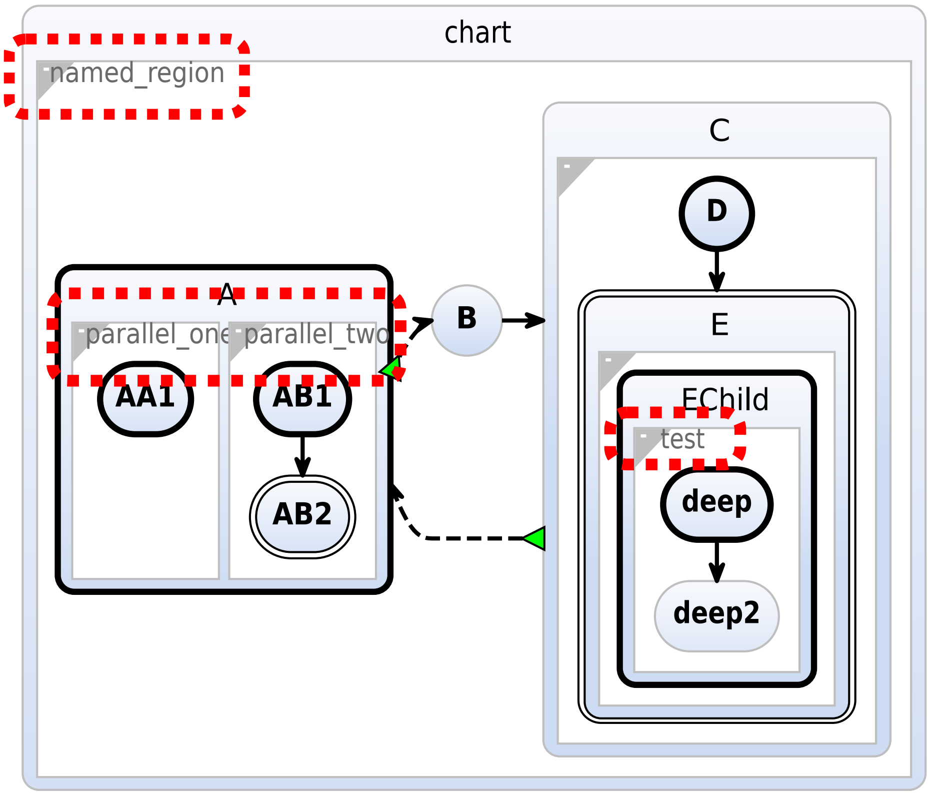


Figure 2.8. The region labels are the most prominent example of the text sizes not being handled correctly. Using a unified font so that the text bounds calculation can happen on the server is solution that would eliminate a lot of the problems related to this process step.

▻ *Rendering* Performed for every piece request, but this is also performed for every viewport transformation. This requires optimization, but this is a different problem.

▻ *Communication* Text bounds related communication is a large additional overhead, since it is performed fully for every piece request. The model sending communication is split apart like the SGraph construction.

As can be seen the major remaining redundancy lies in the text bounds calculation. As already mentioned in Section 2.4.2 the text bounds calculation is dependent on the SGraph construction and the layout step is technically also dependent on the text bounds calculation. This tight coupling of components makes it difficult to individually create top-down solutions for the individual process steps. The compromise made here is to accept small visual bugs, as shown in Figure 2.8, in return for a more performant system.

The solutions already implemented in the scope of this thesis reduce the wait time for the largest diagram, from requesting the diagram until something useful is presented, from over 40 seconds to only a few seconds, which although still not quite the target of one second is a significant step in the right direction. It is important to keep in mind that at this point only the communication is performed incrementally. The transmission of each individual piece now takes up a very small amount of time. This means that the wait time is mostly determined by the other steps in the diagram process that still consume relatively large amounts of time. Further work making more of the time consuming parts of the process also work in a top-down manner, could help realize this performance goal.

Chapter 3

Top-Down Layout

The layout step of automatic graph visualization is an isolated component that is a core contributor to the end result. Since the topic of this thesis is top-down visualization and we have now discussed what that means architecturally, a perhaps more intuitive aspect is the visual impact of calculating layout itself in a top-down manner.

In this chapter the current approach in place in ELK is explained in Section 3.1. In Section 3.2 several layout methods for hierarchical graphs with a top-down idea as their focus are introduced. Several variations and their configuration options are explained in Section 3.3. The results are then evaluated in terms of performance impacts and how they compare visually and aesthetically to the old bottom-up approach in Section 3.4.

3.1 Hierarchical Graph Layout in ELK

Many layout algorithms are designed for graphs that are not hierarchical. Layout algorithms that specifically support hierarchical graphs are a lot more complex to develop and may also only support special cases. Luckily, it is quite straightforward to use normal graph layout algorithms within the process of hierarchical graph layout. This allows building upon the large amount of work that has already been done in the general field of automatic graph layout for the special case of hierarchical graphs.

The simplest approach and the one used in ELK is a recursive graph layout procedure. We define the function $ApplyLayoutAlgorithm(v)$. It computes a layout Γ on the children of v and calculates the size requirements of v. If v has no children, then no further layout is computed at this stage. When the parent of v is laid out, the size of v is determined by externally set size attributes and the micro layout of v. This means for example a node is wide enough so that its label fits in the drawing. Which specific algorithm is applied is not important here and can be set either externally or attached as meta data to the node. With this definition we can now define the layout procedure for hierarchical graphs. This procedure is given in Algorithm 1.

3.1.1 Layout of SCCharts

In the case of SCCharts, which is the graph drawing application this thesis focuses on, diagrams are constructed out of *states* and *regions*. States are inside regions and regions are inside states. There are some differences in how layout of states versus how layout of regions is handled. Applying a layout algorithm to a node means computing the layout of the children of that

Algorithm 1: RecursiveGraphLayout

```
Input  : Graph $G = (V, E)$ with hierarchy function $\tau$
Output: Layout $\Gamma$ of graph $G$
1 begin
2     Choose $r \in V$ such that $\tau(r) = \perp$
3     forall $n \in \{v \in V | \tau(v) = r\}$ do
4         $\tau(n) \leftarrow \perp$
5         $G' \leftarrow G$
6         $\tau' \leftarrow \tau$
7         remove all vertices from $G'$ and $\tau'$ which are not $n$ and not descended from $n$
8         $RecursiveGraphLayout(G', \tau')$
9     $ApplyLayoutAlgorithm(r)$        // layout determines dimensions of $r$
```

node. The layout algorithm applied within regions is a layered algorithm. This means the states within a region are laid out in layers. The layout algorithm applied within states, on the other hand, is a rectangle packing algorithm. This means that the regions within the state are arranged to fit inside the state while preserving their order in the reading direction. Whitespace is removed by enlargening the regions after they have been positioned.

For the layered algorithm there is a further option available which controls the direction of the layout. This can in general be any one of four directions (*up*, *down*, *left*, *right*). For SCCharts this is restricted to several configurations. The simple approach is to set the algorithm to *horizontal* or *vertical* layout. In these cases the layered algorithm is set to always use the right or down directions respectively. Alternatively, the direction can be set to alternate between right and down. This gives us two more options: *HV-layout* and *VH-layout*. HV-layout is the default SCCharts look.

Figure 3.1 shows the result of this layout.

3.2 Top-Down Layout Concept

There are two separate concepts that *top-down layout* could refer to and therefore we need to make clear distinctions about what is talked about where, especially as the concepts are not mutually exclusive.

First of all, coming from the topics discussed so far, especially in Chapter 2, top-down layout can be understood as simply a different implementation than the existing bottom-up method. The goal might be that the resulting diagram looks the same using both solutions. The top-down method would then primarily provide the architectural advantages and flexibility as highlighted in the previous chapter. However, a top-down algorithm which provides the same output as the bottom-up approach is simply not possible, due to layout steps higher up in the hierarchical graph requiring computed layout information of lower nodes.

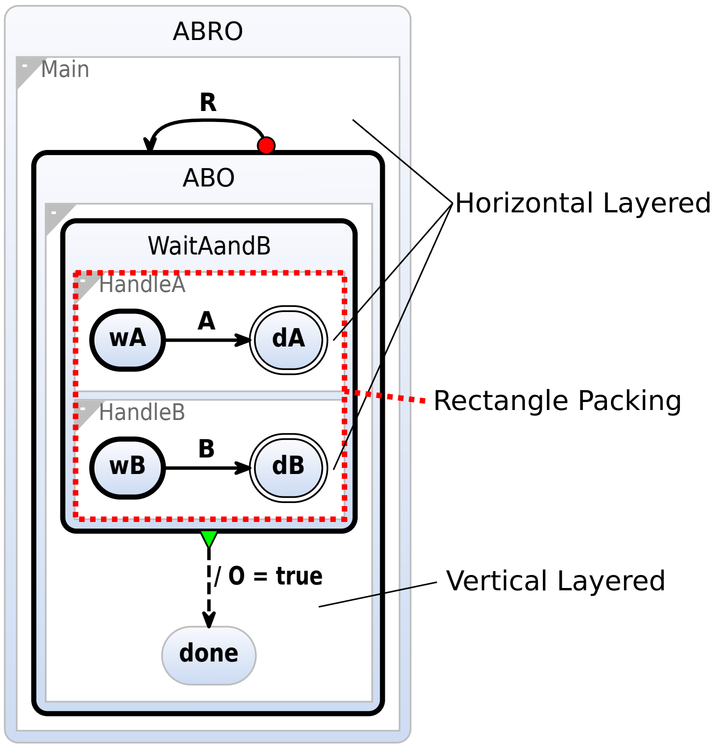


Figure 3.1. Different layout algorithms combined to create the overall SCCharts layout.

There is another aspect of top-down layout that is heavily focused on in this chapter. One of the goals of the bottom-up layout strategy is to use common size for elements such as texts and small nodes so that it can be easily read at a constant zoom level. For the applications of editing SCCharts this is generally not actually a necessary constraint, because the work is happening on computer screens, where we can zoom to arbitrary detail. So with this in mind we can explore what kind of layouts can be created when ignoring nodes lower down in the hierarchy while laying out top-level elements.

On paper these two ideas go hand in hand, but due to the nature of the existing system where the concepts are applied, there are some limitations especially regarding actually performing full top-down layout, which will be covered later. The focus is therefore rather on what the resulting diagrams can actually look like, partially using top-down implementations and also utilizing the existing frameworks.

Figure 3.2 shows a visualization of the top-down concept. Nodes that are located lower down in the hierarchy are scaled down to fit within their parent nodes. The ratio of node dimensions to their label sizes remains similar which lets the user view the entirety of a node while at the same timing being able to read its label or other text attached to that node.

An important idea behind this top-down layout is the notion of *compactness*. Instead of nodes that are on the same hierarchical level being spread far apart, they appear close together and can be viewed together at a glance without extra viewport transformations being

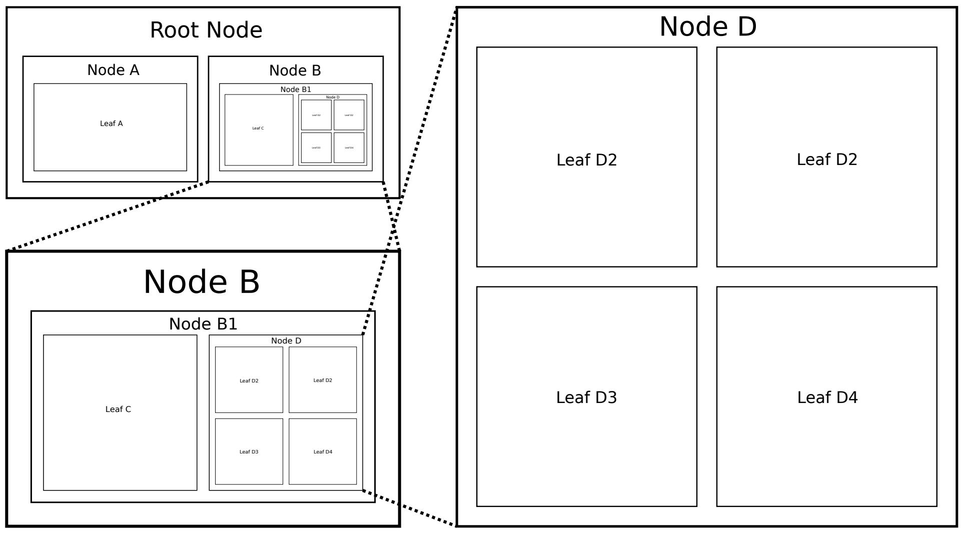


Figure 3.2. Top-down layout concept. Inner nodes are scaled down to fit into their parents. Zooming in closer makes their labels legible.

necessary. The motivation is that this helps make navigating a diagram much more efficient than in the bottom-up approach, because panning back and forth between distant nodes is no longer necessary and, while zooming in on a node, the next hierarchy level already becomes visible and legible. This allows the user to more directly move from node to node.

3.2.1 Top-Down Layout Engine

In order to actually perform layout using a top-down ordering we need to replace the recursive layout engine with something else. The solution is a top-down layout engine which externally provides the same interface, but internally uses a different approach to compute layout for hierarchical graphs. This new engine should complement rather than replace the recursive engine so that both engines may be used depending on the settings chosen by the user.

Although the existing interface for layout engines is abstract, the system was not designed to actually support switching layout engines during run time. The concrete engine that is used is bound once upon start-up. In order to enable dynamic selection of the layout engine, a third generic layout engine is introduced, which serves only the purpose of calling the correct layout engine for a layout run. Which engine should be used is set via synthesis options in the IDE and then as a property on the graph structure itself during the synthesis. The described layout engine switching mechanism is illustrated in Figure 3.3.

From an abstract point of view it is now quite straight-forward to implement the top-down layout engine, however, in practice it turns out it is not quite so simple. The existing algorithms called during graph layout and the structures used for storing the graph and its metadata generally assume that the graph is laid out bottom-up. This makes the implementation of an actual working prototype using this engine quite complex. The challenge here is that the

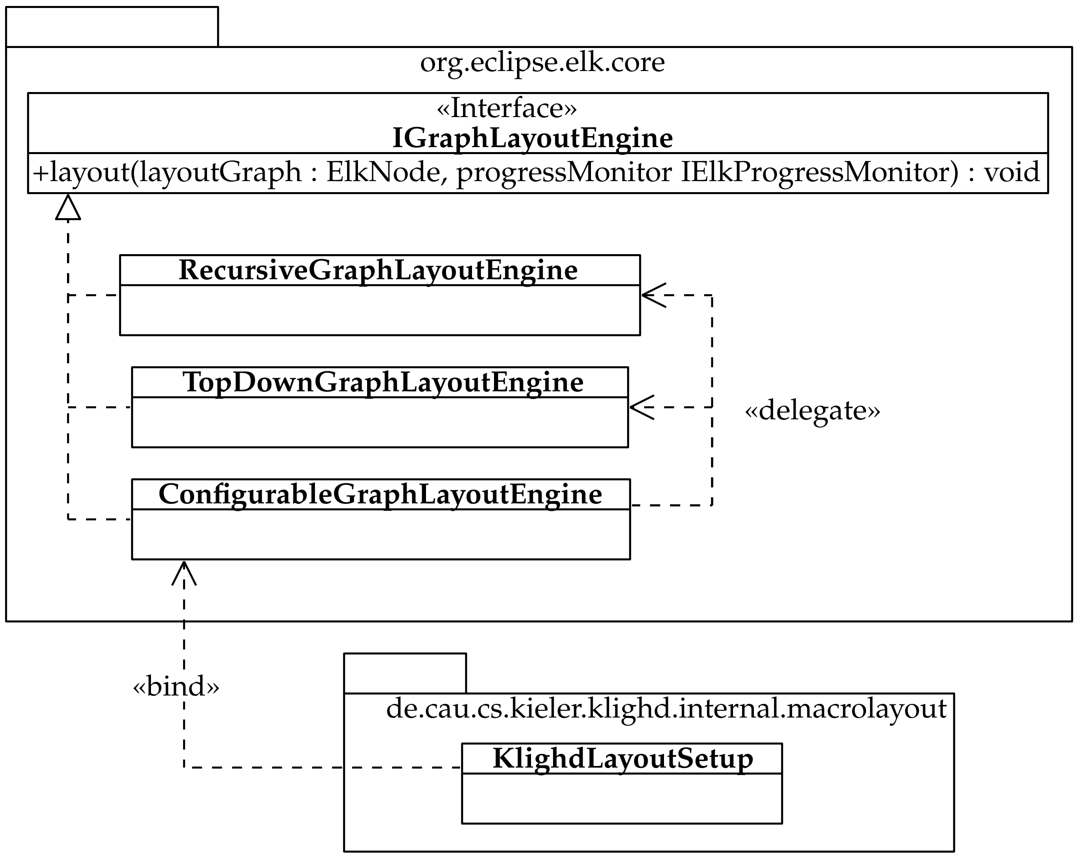

org.eclipse.elk.core
«Interface»
IGraphLayoutEngine
+layout(layoutGraph : ElkNode, progressMonitor IElkProgressMonitor) : void
RecursiveGraphLayoutEngine
TopDownGraphLayoutEngine
ConfigurableGraphLayoutEngine
«delegate»
«bind»
de.cau.cs.kieler.klighd.internal.macrolayout
KlighdLayoutSetup

Figure 3.3. Class diagram excerpt illustrating how the optional switching of layout engines is realized.

layout of nodes' positions is only part of the layout process. The label of a node also affects the necessary size of a node, which in turn affects the overall layout. Because of this it is not possible in ELK to simply set the child node sizes and then compute a layout, because the result will be affected in the wrong order by these extra calculations. This is an example, where creating a new system to perform this specific task top-down, would probably end up being easier than trying to integrate the new concept into the existing system.

But since we are not only interested in the computation order, but also the potential visual results, there is another method we can use which makes it possible to combine top-down ideas with the working system.

3.2.2 Hybrid Layout

The method used to achieve a visualization with a top-down aesthetic within the the existing system is a hybrid approach. The hierarchical layout is still performed recursively bottom-up using the recursive layout engine, but on the way down the hierarchy, the node sizes are constrained without knowing the size of the content. This mimics the process in a true top-down ordering, where the layout of the high-level nodes has to be performed before

Algorithm 2: HybridGraphLayout

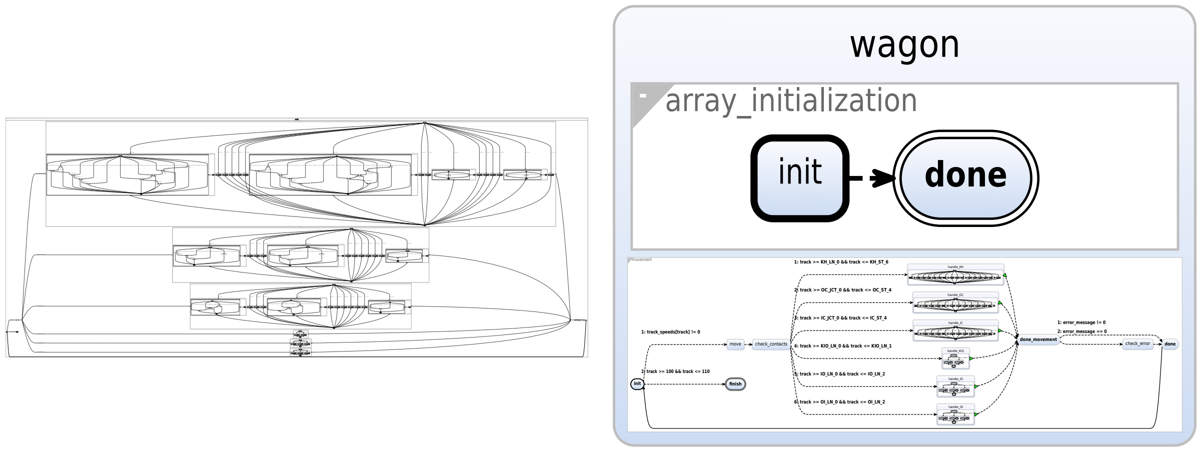
```
Input  : Graph $G = (V, E)$ with hierarchy function $\tau$
Output : Layout $\Gamma$ of graph $G$
1  begin
2      Choose $r \in V$ such that $\tau(r) = \bot$
3      forall $n \in \{v \in V | \tau(v) = r\}$ do
4          $\tau(n) \leftarrow \bot$
5          $G' \leftarrow G$
6          $\tau' \leftarrow \tau$
7          remove all vertices from $G'$ and $\tau'$ which are not $n$ and not descended from $n$
8          $RecursiveGraphLayout(G', \tau')$
9          scale $n$ according to its scale factor
10     $ApplyLayoutAlgorithm(r)$
       /* The size constraint is set for every node (usually all the same), the
          desired size can be the width or height computed by the layout or can be
          a more complicated metric describing the space required to draw the
          layout.                                                              */
11     $c \leftarrow$ size constraint of $r$
12     $d \leftarrow$ desired size of $r$
13     $s \leftarrow c/d$                          // scale factor
14     attach $s$ as property to $r$
```

their contents have been laid out. In order to then fit the laid out children into their parents with restricted sizes, the entire child content area is scaled down to fit the given space. This modified recursive layout is described in Algorithm 2.

The node size constraints are chosen in such a way that each node aims to be the same size when viewed at its correct zoom level. This has the effect that text sizes always remain legible while viewing the entirety of a node as opposed to a bottom-up layout, whereas in large diagrams, top-level labels tend to get lost in the large node drawings, see Figure 3.4. While this problem can also be approached using dynamic rendering techniques, which only modify the resulting drawing depending on the current zoom level, dealing with this issue directly during layout provides an elegant alternative to this issue.

In Figure 3.4b the top and and bottom regions don't align and the region labels are also different sizes. This is a consequence of scaling down the nodes to fit within a constrained size. It is not possible to change this on the layout side, but this is something that could be changed relatively easily during the rendering process.

The other benefit of this top-down layout is that it becomes much easier to parse a node in its entirety and to get an overview of what it contains. This is due to two factors. Firstly, because the contents are scaled down to fit into the parent it is possible to see the entire node and its content at a legible zoom level without the need to pan or zoom the screen.



(a) Wagon model laid out using bottom-up approach. Labels of large states are impossible to read while viewing the entirety of the state.

(b) Wagon model laid out using top-down approach. Labels of all states are always legible while viewing the entirety of the state.

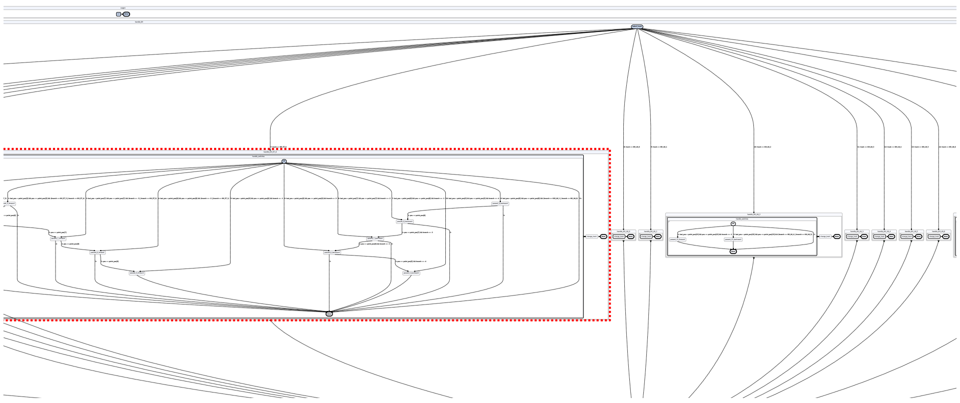
Figure 3.4. Comparison of legibility of labels in large states.

There are extreme cases where this is still necessary of course, but these cases are very rare. These cases are when there are a large number of nodes on a single hierarchy level. At that point the result is dependent on the layout algorithm responsible for arranging the nodes and the influence of the hierarchical layout diminishes. For the use case of SCCharts that is being considered here, this is not very common as they generally make a lot of use of hierarchy.

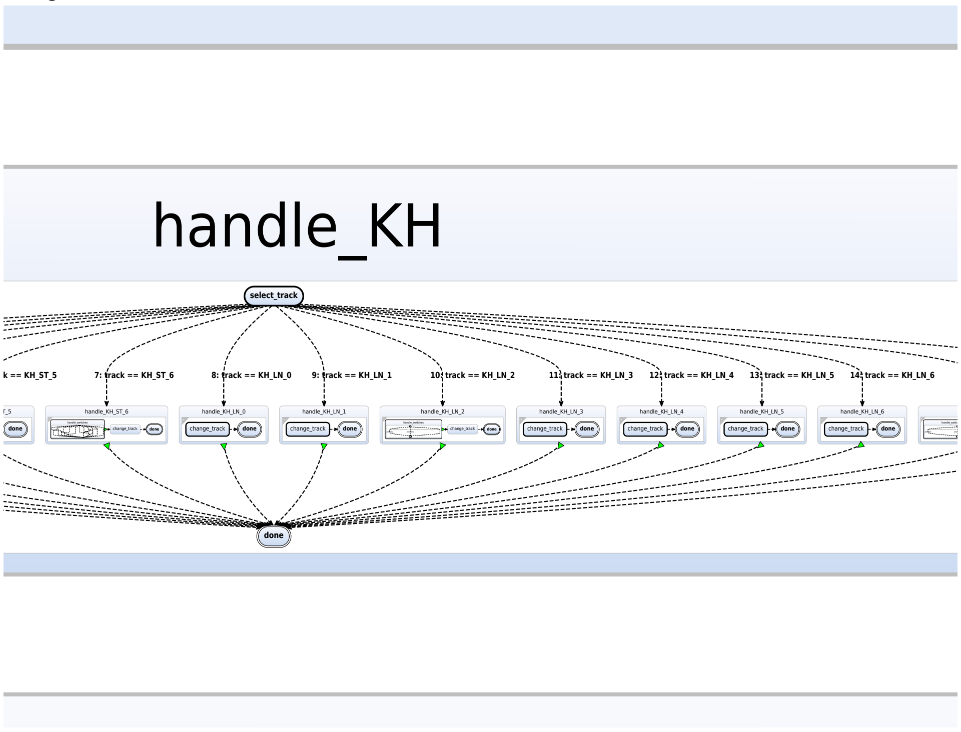
In the bottom-up approach, on the other hand, it is often the norm that the viewport needs to be adjusted while parsing the diagram. The second factor is that the child nodes themselves all have a similar size relative to each other regardless of the number of children they themselves have. So nodes on the same hierarchy level can no longer be dwarfed by neighbouring very large nodes as would be the case in a bottom-up layout, see Figure 3.5.

This is the solution chosen here, but there is another idea that we can look ahead into only the next layer of a node to adjust the amount of space given to that node. For example a node with five direct children might be given more space than a node with only two direct children. In this hybrid approach this can be quite easily achieved by adjusting the node size constraints dynamically. In the case of SCCharts this would not immediately have the desired effect though. This is because of the way regions and states switch for every layer. Often there is only one region in a state and what actually determines the space required by the state is then not its child region, but rather that region's children. This makes this heuristic not generally applicable which is why the more general solution was chosen.

This covers the general concept, but there are a number of variation possibilities and configuration options, which will be covered in detail in Section 3.3.



(a) In the bottom-up case, larger nodes such as *handle_KH_ST_6*, highlighted in red, dwarf their neighbours making it difficult to view several nodes next to each other. Besides that the labels are all hardly legible requiring further zooming.



(b) In the top-down case, all the nodes on the same level are sized in a way that makes large and small nodes legible at the same time.

Figure 3.5. Comparison of resulting layouts when very large states are next to very small states. Both figures show a very similar portion of the wagon model.

3.3 Configuration Options

There are a variety of different ways to set the constraints for the top-down layout, and evaluating the resulting diagram variations effectively requires flexible switching of the specific parameters and their combinations used. To this end several synthesis options are introduced that control how top-down layout is computed. These configuration settings and their effects are explained in this section.

3.3.1 Layout Engine

There is an option to toggle between the top-down layout engine and the recursive layout engine. The remaining options are currently only supported by the recursive layout engine, because as previously already stated, the top-down engine did not prove to be a suitable solution and the hybrid approach was implemented as an extension of recursive layout. When the recursive layout engine is selected, top-down layout can be turned on and off using a separate setting.

3.3.2 Constraint Type

The constraint type defines what aspect of the nodes is used to calculate the scale factor to be applied. In general the scale factor s is based on a size constraint c which defines the amount of space available for the layout of a node and a desired size d. What exactly the desired size represents depends on the chosen constraint type. The scale factor is computed by $s = c/d$. After the layout of a node is computed, all laid out elements are scaled with the factor s.

To better understand the effects of the available constraint options, example figures are included. The same diagram is laid out each time using a different setting. As a reference, Figure 3.6 shows the diagram laid out using the bottom-up approach. In the following example figures for the different size constraints red bars are added to the diagrams to illustrate, which dimension is constrained during the layout. These red bars are scaled along with the part of the diagram they correspond to, but it is important to understand that they represent a fixed unit length. How this unit length is chosen and what happens when it is adjusted will be covered in Section 3.3.3.

Fixed Width

When *fixed width* is selected, all nodes on the same hierarchical level are scaled in such a way that their final widths are the same. The main visual effect is that node labels become very large. This has the downside that, when the labels are relatively longer than the constrained width, extra whitespace is added to the child area to accommodate the larger label. Dynamic constraints, which are introduced later, help alleviate this issue. In Figure 3.7 we can see how the fixed width setting affects the layout. As can be seen quite well at the top-level node, due to the constrained width, the label *N* becomes relatively large. The nodes *A*,*B* and *C* are all the same width, but *B* is slightly taller, because its children require a little more space.

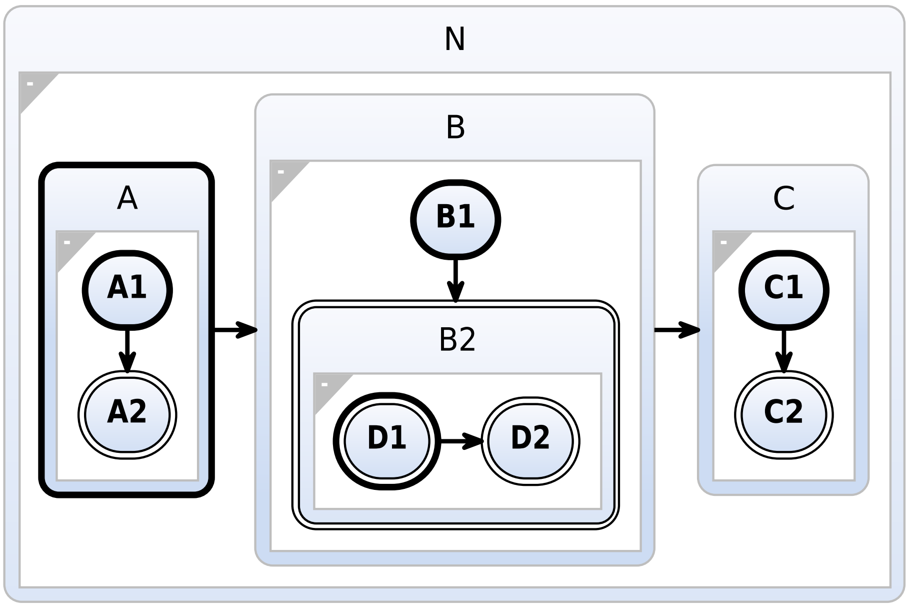


Figure 3.6. Reference for comparison. Diagram laid out using bottom-up approach.

Fixed Height

Fixed height is analogous to fixed width only that heights are constant. For both *fixed width* and *fixed height* the size of the non-fixed dimension depends on the actual contents of the node. With this setting the label width issue is not a problem anymore, but a similar issue happens when the child layout is tall and needs to be scaled down further to fit within the limited height. An example diagram with fixed height can be seen in Figure 3.8. Since the width can now stretch to accommodate the children, the *N* label is now smaller relative to the overall size of the node. The child nodes *A*,*B* and *C* are all now constrained to the same height, which means that *B*'s children need to be scaled down slightly more than before.

Optimized

When always fixing the width or the height, the result can often be that some nodes are very tall or very wide compared to their neighbours. The *optimized* setting alleviates this by dynamically constraining the width or the height depending on the resulting scale factor. The constraint which results in a scale factor closer to 1 is chosen as this helps keep scales between different hierarchy levels closer together and also reduces the amount of very tall or wide nodes. This setting also reduces the issues with fixed width and height, as it usually selects the setting that causes less problems. Besides considering the scale factor when deciding which dimension to fix, it might also be useful to take the layout direction into account. Figure 3.9 shows which constraint is chosen for each node. The important nodes to note here

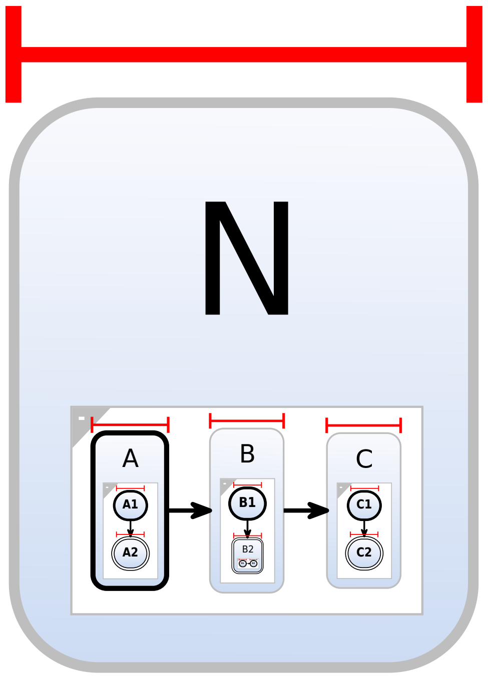


Figure 3.7. Diagram laid out using top-down approach with size constraint type set to *fixed width*.

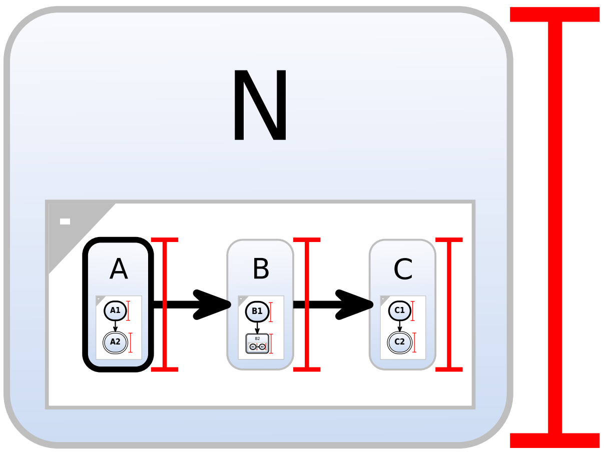


Figure 3.8. Diagram laid out using top-down approach with size constraint type set to *fixed height*.

are *N*,*A*,*B*,*C* and *B*2. The other nodes are atomic nodes, which effectively means fixed width and fixed height have the same effect. *N* and *B*2 are scaled using a fixed height, because this results in less down-scaling. *A*,*B* and *C* are scaled using fixed width instead, because this results in less down-scaling for them. This is also quite visible when comparing these nodes directly between Figure 3.7 and Figure 3.8.

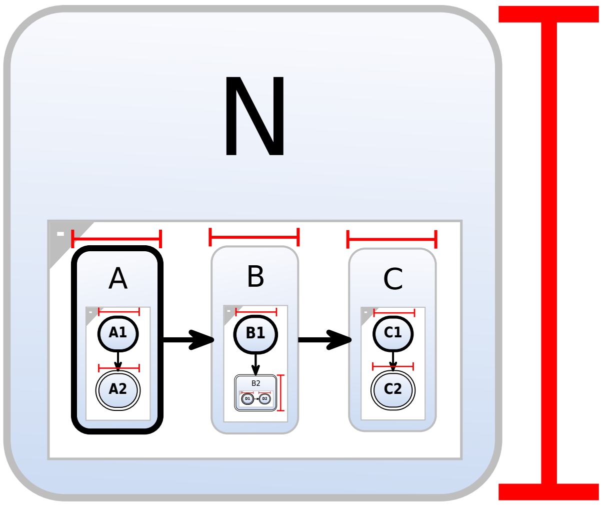


Figure 3.9. Diagram laid out using top-down approach with size constraint type set to *optimized*.

Square

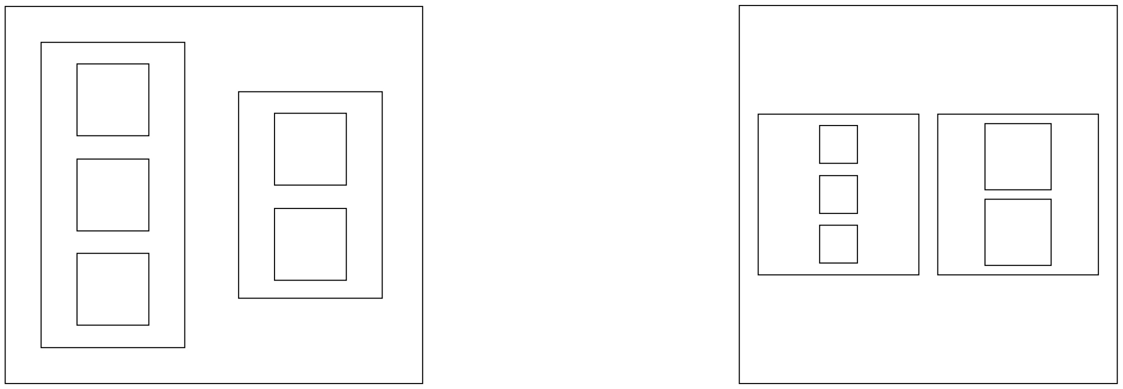
This option so far exists only as a concept and a solution to fully implement the idea still needs to be developed. The aim of the *square* option is to more closely emulate a true top-down layout process within the hybrid method used here. The idea is to fix both the width and the height of a node at the same time, completely independent of the content of the node. When the content of the node is laid out it has to then be scaled down according to the long edge of its drawing space. The result would be more down-scaling and more whitespace than with the other constraint types. Figure 3.10 illustrates what the layout result using this option would look like.

Even though the option is called square here, a more general descriptor could be *fixed aspect ratio*. The problem does not change when width and height are constrained to different sizes so long as they are both constrained. For the use case of computer screens, a fixed aspect ratio corresponding to the screen would probably be the most sensible restriction to create an optimal viewing experience.

3.3.3 Dynamic Constraints

To fully understand what this option does we need to first go into a little more detail about what exactly the size constraint does. The size constraint is a number that is set as a property on a node and it tells the layout engine how many pixels the constrained dimensions should occupy in the drawing. The ratio between this constraint and the space required by the node content determines the scale factor applied to the nodes children.

By default the node size is set to the size of a minimal SCCharts state as shown in Figure 3.11. This value of this size can be modified using a slider. The result is an interesting visual effect. When the value chosen is smaller than the default, the scale of the internal nodes starts



(a) Hierarchical graph laid out bottom-up, where node size is dependent its children. All leaf nodes are the same size.

(b) Hierarchical graph laid out top-down constraining all nodes on a level to the same square size.

Figure 3.10. Square constraint option concept.

becoming smaller faster and nodes with some content will have smaller labels than nodes with no content. To an extent this is exactly expected behaviour, because of the constraints. When the size constraint is set higher than default, we can observe the opposite. Nodes with child areas will become relatively larger than empty nodes. In Figure 3.12 a comparison between the default size constraint, smaller and larger values is shown.

There may be benefits to using the non-default size constraints, but some further restrictions are probably necessary. For example, to restrict the magnifying effects described above, the final scale factor that is applied can be capped. A simple method to achieve a good visual result is explained in the following.

At the default size setting, nodes with no further children look as intended, that is, they look the way would also look in a bottom-up layout. As originally explained, the idea of the top-down layout is to make all nodes on the same hierarchy level the same size. In practice, however, this can actually make it hard to read the contents of nodes containing children next to atomic nodes. To alleviate this issue, an option called *dynamic node size constraint* can be toggled and when enabled this simply doubles the size constraint of all nodes which have

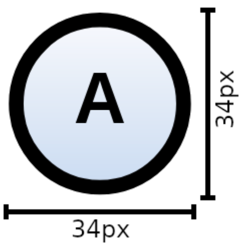
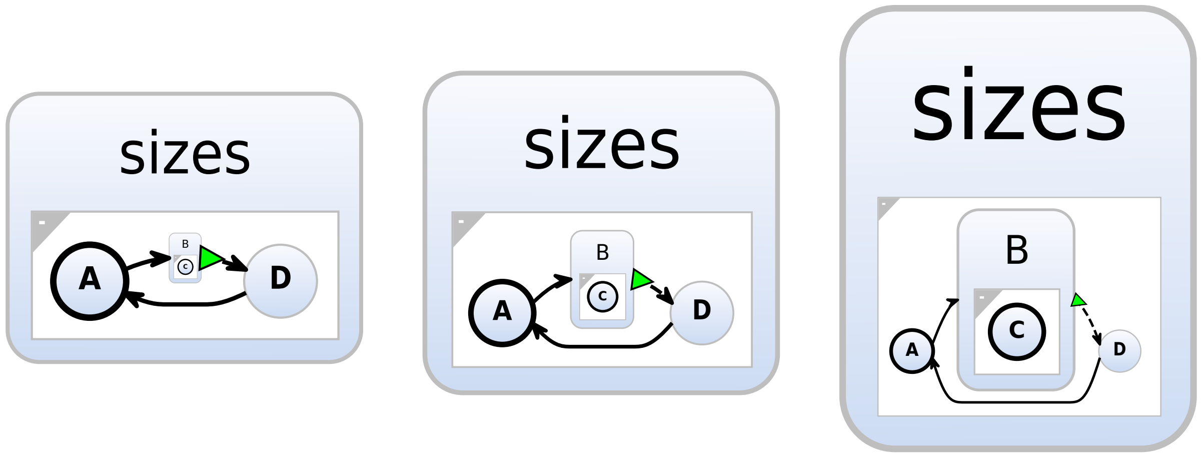


Figure 3.11. Dimensions of a single, atomic SCCharts state.



(a) Size constraint set to 15.

(b) Size constraint set to default value of 34.

(c) Size constraint set to 94.

Figure 3.12. Comparison of effect of changing the node size constraint values. Dynamic node size constraints are disabled here. As can be seen, when we deviate away from the default size, an undesirable distortion of the nodes relative sizes starts appearing.

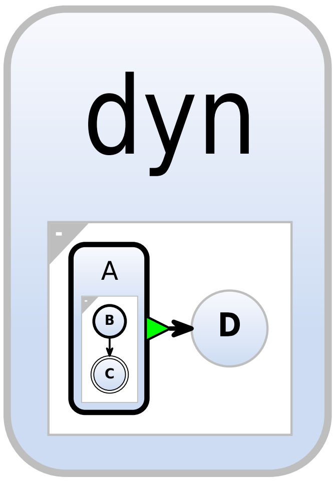
children. The difference in the two appearances can be seen in Figure 3.13. The figure shows the effect for both fixed width and fixed height. In both cases the result is a more balanced overall look.

3.3.4 Automatic Layout Direction

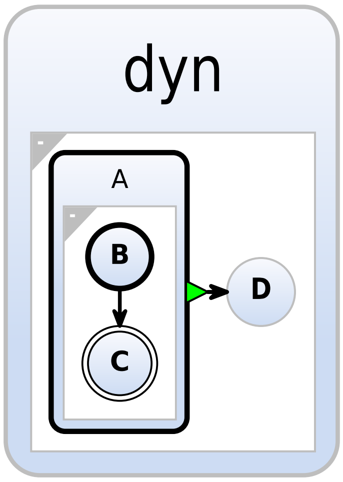
Part of the visual aesthetic of SCCharts comes from the layout direction. As explained in Section 3.1.1, SCCharts use a layered algorithm to lay out states within a region. A layered layout can be oriented either horizontally or vertically. To balance the aspect ratio of SCCharts diagrams, the default approach is to use a so-called *HV-layout* which just means that the direction used by the layered algorithm switches back and forth and starts with a horizontal direction. The direction can also be set to alternate, but start with vertical and also use only horizontal or only vertical.

This does not force a balanced aspect ratio though, and therefore in the spirit of top-down layout a new setting is introduced. *Automatic direction* can be used with or without top-down layout. When it is enabled the layout of a hierarchy level is first performed with the default direction. The dimensions used by the resulting layout are saved and then layout is computed again, but using the other direction. The aspect ratios of the resulting layouts are compared and the one that is closer to a pre-defined ratio is used for the final layout.

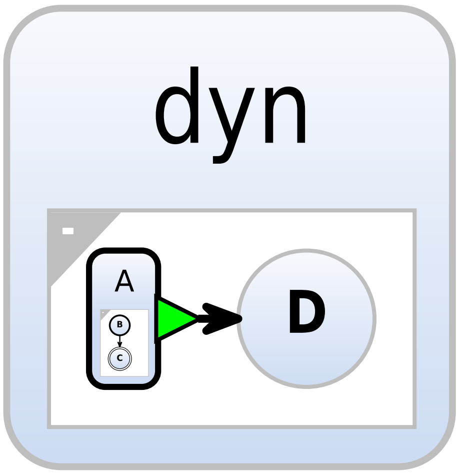
Using the aspect ratio to decide which direction to use results in less long and thin nodes and an overall more balanced look. The downside is that the uniform look achieved by alternating the direction in every step is no longer there. However, part of the usefulness for top-down layout is actually achieved by optimally using the available screen space, which this



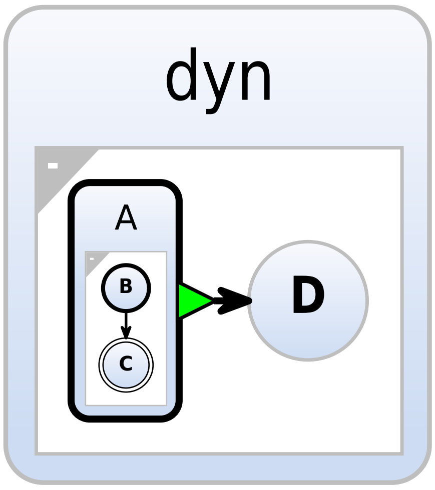
(a) Dynamic node size constraint is disabled. Fixed width.



(b) Dynamic node size constraint is enabled. Fixed width.



(c) Dynamic node size constraint is disabled. Fixed height.



(d) Dynamic node size constraint is enabled. Fixed height.

Figure 3.13. Comparison of the effect of having dynamic node size constraint disabled and enabled.

option helps with. Aspect ratio is a relatively simple method of determining the best direction and there are other metrics that may be considered as well. One such more advanced measure could be the *max scale measure*, which aims to make labels as legible possible by determining the largest possible scale they can be displayed at [RES+17].

Enabling this option increases layout time by a small factor, because every time the layered algorithm is normally executed it is now executed two or three times. This could in theory be reduced to only a factor of two, but that would require more memory to cache the results and the necessary copying of the data would introduce other additional performance costs. Since layout is not such an expensive process and the time complexity is not significantly impacted, this is a reasonable solution for this experimental feature.

3.4 Top-Down Layout Evaluation

This section covers the performance implications top-down layout has and compares the diagrams resulting from top-down layout with diagrams laid out using the traditional bottom-up approach. We also take a look at how a use case of SCCharts is affected by using both approaches. The use case that is examined is the navigation within a diagram.

3.4.1 Performance Impacts

While performance is not the main focus in this chapter, it is nonetheless important to understand what a new process costs in terms of performance. This helps keep the system fast and can also prevent unexpected problems showing up later.

In general the procedures used to create the top-down layouts use the existing system and build upon that. The complexity of that system will serve as our baseline reference. While calculating a top-down layout with any of the constraint types, the time complexity does not actually increase as all that happens is a little extra calculation at each step to determine the scale factor.

There are, however, two things which do have a considerable impact, the first is the option to automatically choose the layout direction and the other is related to rendering with our large scale differences.

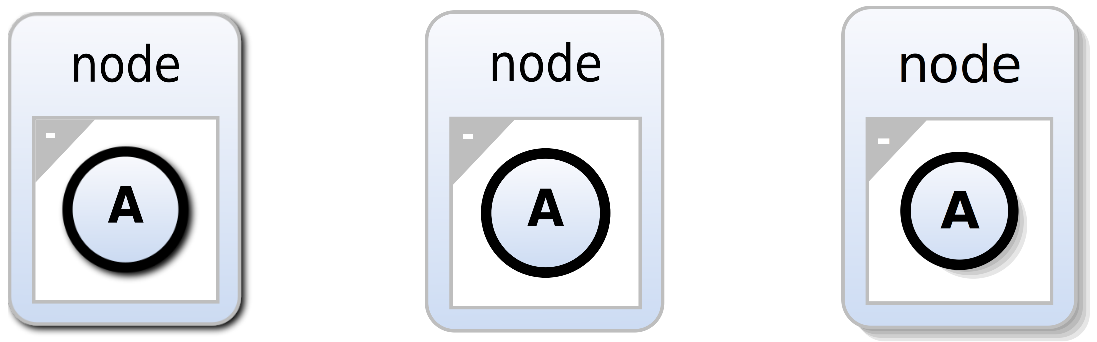
Automatic Direction

Enabling the setting to automatically choose the layout direction for the layered algorithm during the layout process comes with a performance cost. Both possibilities are computed in every instance where the layered algorithm is called and if the first possibility is chosen, that layout has to be recomputed. The result is that all the layered algorithm calls end up using approximately twice as long in the best case and three times as long in the worst case. The factor is not quite as high for the entire layout process, because SCCharts layout also includes rectangle packing which is not affected here.

While the time used does increase, the time complexity actually does not. This is good, because it makes this a viable experimental feature to view alternative diagram layouts, which scales similarly to the normal layout process.

Shadow Rendering in KEITH

Part of the visual design of SCCharts includes a shadow drawn behind states to give them a more three-dimensional look. See Figure 3.14 for a comparison of a diagram drawn with shadows enabled and disabled. The way they are rendered in KEITH causes severe rendering slowdowns when zooming in on them due to the current implementations of shadows using



(a) Shadows enabled in KEITH. **(b)** Shadows disabled in KEITH. **(c)** Shadows enabled in KIELER.

Figure 3.14. Comparison of a diagram drawn with and without shadows.

gradients requiring much more rendering time in current SVG renderers. Zooming in very far is, however, a core navigational requirement for a top-down layout and therefore shadows have to be disabled, when using top-down layout. This is also the reason the diagrams shown in this thesis are drawn without shadows. In the future, a different implementation to render shadows in KEITH could be used which does not slow down rendering time when zoomed in closely. An implementation closer to the one currently used in KIELER could be more suitable.

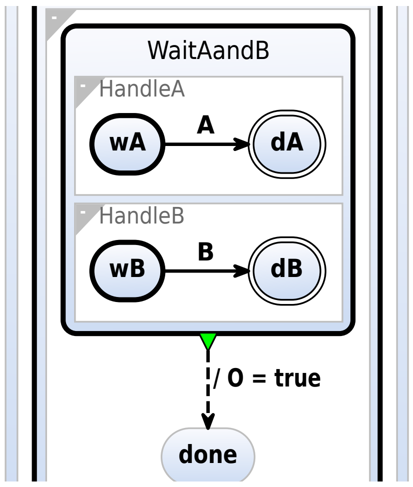
3.4.2 Aesthetics Comparison

The concepts developed for this thesis were put into practice for SCCharts, therefore it makes sense to compare the *aesthetics* and the *look-and-feel* of the bottom-up layout with the new top-down ideas. This section will go over what remains similar and where the differences in the layout results lie. Furthermore, the advantages of each approach are highlighted.

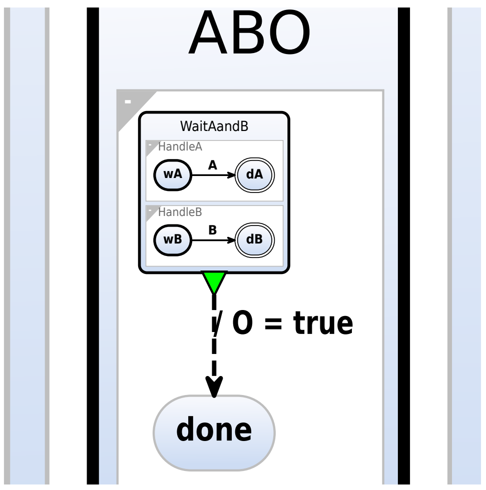
When viewing a top-down layout of an SCCharts diagram on a single hierarchy level, the general look is analogous to a diagram laid out bottom-up, this is demonstrated in Figure 3.15. This means that many aesthetics criteria are simply maintained when switching from the bottom-up layout to the top-down layout. This includes measures such as the distances between nodes, edges and combinations thereof. Edge labels in particular are still a little different in their relative appearance. This is due to their hierarchical location within the diagram model, which affects how they are scaled. This could be further adjusted in the future, depending on what look is deemed best.

There are some differences though. The first most obvious difference is that similar elements no longer have a similar size. The most prominent example of this are texts. This look is of course intended, as hierarchy levels can now be visually differentiated by their scale and the name of a state is always large enough to be readable in relation to the size of the state, see Figure 3.4.

Figure 3.15b showcases a problem in the top-down layout implementation that still remains to be fixed. Although edge labels are scaled down together with other elements on the same hierarchy level they do not always end up in quite the right position.



(a) Zoomed in excerpt of ABRO laid out bottom-up.



(b) Zoomed in excerpt of ABRO laid out top-down.

Figure 3.15. Comparison of aesthetics within a hierarchy level.

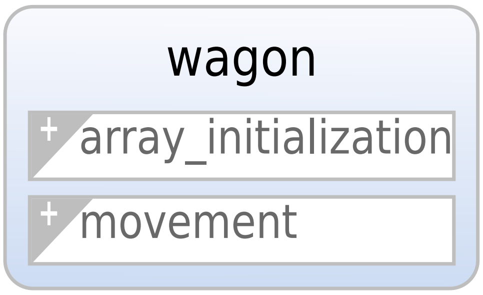


Figure 3.16. Wagon model laid out using bottom-up approach with all regions collapsed.

The only way bottom-up layout can provide similar legibility as achieved by top-down layout is by collapsing regions as shown in Figure 3.16. Collapsing and expanding regions always shifts the layout slightly, which can conflict with the mental map of a user. While using top-down layout it may not be necessary to collapse and expand regions to switch between high-level and low-level viewing, therefore maintaining the mental map at the highest possible degree by not changing the layout at all.

Another important difference also related to sizes on a hierarchy level is the size variation of nodes drawn on the same hierarchy level. In a bottom-up diagram there are often cases where very small nodes are drawn next to very large nodes. This makes it difficult to see them together and to understand their relation at a glance. In a top-down diagram, on the other hand, all nodes that share a parent are drawn with very similar sizes, depending on which particular size constraint is chosen. This makes it easy for a viewer to quickly get an overview of all the components inside a node and how they are connected. An example diagram demonstrating this difference is shown in Figure 3.5. Looking at the internal workings of any individual node is then achieved by zooming down into it.

Another measure which is influenced by this change is the overall compactness of the drawing. In a top-down layout there is much less whitespace than in a bottom-up layout of the same diagram. This means that elements are far less spread apart and can be viewed easily on a screen with minimal panning of the viewport.

The differences between the two layout approaches help distinguish the different use cases of the layout types. In bottom-up layout, the primary navigational tool is panning, which makes it a great layout method for diagrams that are printed on physical media. A large sheet of paper can only be panned. Zooming can be emulated by stepping away from or closer to the paper, but this is very limited compared to the possibilities on a computer screen. The resolution of a sheet of paper can be increased by increasing the physical size of the paper. Top-down layout, in contrast, uses zooming as the more important navigational operation. This makes it very suitable for screen-based applications.

Apart from differentiating the layouts by the medium on which they can be suitably viewed we can also consider how a user interacts with a diagram. This can mean navigation from one node to another, searching for specific nodes or understanding the connections presented in the diagram. The layout and presentation of the diagram is crucial to make these tasks easier or harder for a user to perform. Taking this into account when creating a visualization to provide the user with a diagram that makes these tasks easy and do not require additional cognitive load, should always be an important factor when evaluating the quality of a diagram drawing.

3.4.3 Comparison of a Navigational Task

Navigation within a diagram is important, especially for diagrams which are large enough that the details cannot all be taken in at a glance. This section will go over the differences between bottom-up and top-down layout when faced with a relatively simple navigation task. The focus here will be on the viewport transformations necessary during navigation. For a full quantitative analysis and comparison average navigation times for different examples would also be interesting, we will specifically be looking into the amount of panning and zooming operations that are necessary.

Before constructing an artificial comparison we need to first consider how navigation is usually done in bottom-up layouts, how it can be done using top-down layout and how comparable these methods really are. In a large diagram laid out using bottom-up layout, texts are usually too small to read when viewing the entire diagram. Users are forced to *hunt-and-peck* for the nodes they want to navigate to. What this means is that the user will make a guess about where they think the node they are looking for might be and zoom in to check the labels at that location. If it is not the right place, they will zoom back out and try somewhere else. In a diagram laid out using top-down layout, on the other hand, this is easier. When viewing the entire diagram, the top-level nodes are legible and the user can directly zoom in on the correct location. These different navigation styles can hardly be compared, but what we can do is remove the random guessing part and assume a direct and known path to a goal. In this scenario we can get an estimate for distances in a diagram, which helps illustrate the relative compactness of a top-down layout.

The task is a simple task of going from the top-level zoomed out view to a leaf node. Since we are not so much interested in search times, but rather how directly the goal can be reached, the list of parent nodes are known beforehand. This way, searching is limited to searching for the next node on a hierarchy level. To keep the process comparable, the operations are restricted in the way they may be used so that the process will alternate between a zoom operation to make node labels legible and a pan operation to locate the correct next node. In the case of the bottom-up layout, all node labels are legible on the same zoom level, so there is essentially only one initial zoom required and all subsequent operations are pans. The task is performed on the wagon model, because it is a relatively large model which forces a certain amount of viewport transformations. Top-down layout here was computed with the size constraint set to *optimized* and dynamic size constraints enabled.

We are interested in finding out how much zooming is necessary to make labels legible and how much panning is required to center the viewport on the next node in the sequence before the next zooming step. We can measure the distance between each step and compute the sum to determine the total panning distance. The distance is always measured from the midpoint of a node to the midpoint of the next node.

In the bottom-up case, since there is no zooming, determining the distance is simple. In the case of top-down layout, there is a zoom operation between each pan operation and therefore we need to ensure that distances use a standardized distance measure. The way this is solved is by measuring distances relative to the zoom level with text sizes being the defining standard measure. This also gives comparability between bottom-up and top-down layout. The task compared here consists of the following path through the wagon model.

wagon → movement (region) → handle_IC → handle_IC_ST_4 → handle_switches → init

Here, *movement* is the only region listed, because it is the only named region. All other states only have a single region which therefore do not have a large impact on the navigation so the path can be simplified by combining them with their parent node.

The results of this navigational task are shown in Table 3.1. The units for measuring the distance are the values as calculated by the layout of the diagram before any scaling is applied to the renderings on the client. This ensures the comparison between bottom-up and top-down is made at the same relative scale. The total pan distance is computed as the sum of all individual steps and the total zoom distance as the product of all steps. The distances always refer to the distance to the previous step, so in the first step the pan distance is 0 and the zoom distance is 1. Internally, zooming is implemented by scaling down the inner nodes. Here, we will use the inverse of the scale factor, the *zoom factor*, which represents the factor by which nodes need to be enlarged in the viewport until they reach the size at which they become legible.

To get an idea of what it feels like to navigate through this relatively large diagram, Figure 3.17 shows the first few steps of the navigation task in both the bottom-up layout and

Table 3.1. Navigation distance measured in pan distance and zoom factor. Measurements are relative to the previous step.

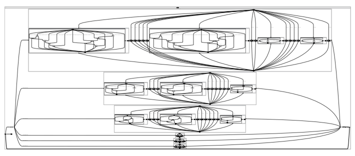
Node	Bottom-Up Pan	Top-Down Pan	Top-Down Zoom
wagon	0	0	1
movement (region)	49.31	51.74	8.13
handle_IC	1779.82	165.6	1.5
handle_IC_ST_4	649.98	305.87	8.9
handle_switches	110.74	115.23	2.03
init	295.58	278.74	8.4
Total	**2885.42**	**928.88**	**1854.51**

the top-down layout. As can be seen quite clearly, at the zoom levels presented, the texts in the bottom-up layout are not legible. This means when actually navigating the graph the user is forced to zoom in close to read labels and then out again to reposition the viewport appropriately. In the top-down layout, on the other hand, the labels become gradually visible, which lets the user smoothly transition from one node to the next. The top-down layout also helps to improve the readability of the diagram by minimizing the amount of visible extra information. The focus is always more on the direct children of a node.

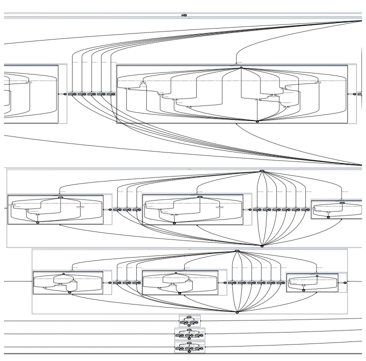
The first observation that can be made is that the total panning distance when navigating the top-down layout is significantly shorter. Of course this distance does not just disappear, but rather is converted into a zoom distance. While of course zooming is also possible on the diagram laid out bottom-up, the top-down layout actually leverages this as a sort of third dimension and through that shortens distances between points of interest.

The task chosen here leads to a node which is positioned fairly central on the diagram. The bottom-up layout benefits a lot from this, because if we were moving to the edge of the diagram in the case of the wagon model, we would see a total panning distance of around 10 000. This is of course assuming there is no searching necessary, because the user already knows exactly where to navigate to. In practice this is usually not the case.

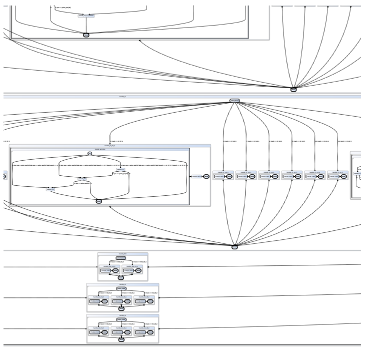
What we can see is that the biggest benefits happen in the initial steps of the task. In the top-down case a small pan is enough to locate the next node which can then be zoomed into, whereas in the bottom-up case a lot of distance has to be covered. The benefits only really start becoming significant after about three levels of nesting. Diagrams with less depth than that do not benefit very much from top-down layout, because in those cases everything still remains readable without having to zoom in.



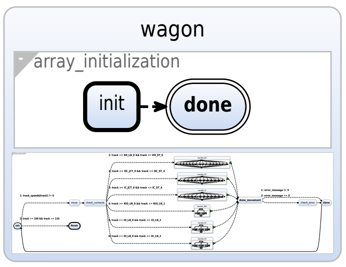
(a) Bottom-up layout. Entire model in view.



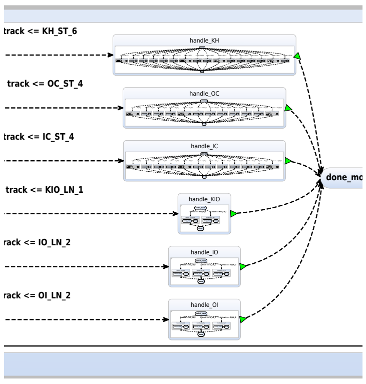
(b) Bottom-up layout. *handle_IC* node is visible along with the other nodes above and below it.



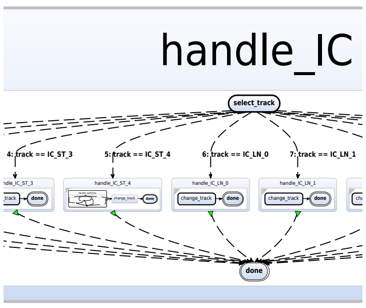
(c) Bottom-up layout. The children of *handle_IC* are in the center of the image. *handle_IC_ST_4* is the large node.



(d) Top-down layout. Entire model in view.



(e) Top-down layout. *handle_IC* node is visible along with the other nodes above and below it. Here all nodes occupy the same amount of vertical space.



(f) Top-down layout. Zooming in on the children of *handle_IC*. Surrounding nodes are no longer in view.

Figure 3.17. Comparison of the navigation through a large diagram laid out with bottom-up layout and top-down layout. The figures aim to capture a very similar cutout of the diagram for each step.

Chapter 4

Discussion

In this chapter the results of chapters 2 and 3 are summarized and discussed. Going further they are examined within the overall context of top-down graph visualization and how the solutions developed help advance this overarching topic. Finally, we take a look how further improvements could be made and what the strategy for extending the top-down concept could be.

4.1 Summary of Results

This section will go over the individual areas of interest of the thesis and discuss the concepts, solutions and results, as well as what can be learned from the work done. We will go over some of the remaining issues, how they could be addressed, and what the next development steps could be.

4.1.1 Top-Down Architecture

A major goal of this thesis is to provide alternate solutions to the scaling problems faced by the current architectural solutions. The path chosen here is an incremental top-down architecture design, which flips the existing paradigm on its head. The questions posed in this research were concerned with the feasibility of implementing parts of the architecture in a top-down manner and also what benefits this provides in practice.

The work of this thesis focused on two specific parts of the KEITH system, the SGraph generation step and the communication between client and server when sending the diagram model. For the SGraph generation, an incremental on-demand construction mechanism was created which serves as the entry point for the incremental communication by only delivering the parts of the SGraph that should be sent in any individual message. Instead of sending the entire model in one request-response message exchange, the client now requests individual pieces of the model and the server sends only that piece. This results in many small messages rather than one large message. The client can decide in which order pieces of the diagram should be requested and can render all pieces it has already received.

What we can see with these solutions is that the concept works in practice. It is possible to transform parts of the overall architecture into an incremental pattern while treating the rest of the system as a black box. This in principle is promising for taking the same approach with other parts of the system, such as synthesis and layout.

Is a top-down architecture useful? To answer this question we need take a look again at the use cases. KEITH is an IDE primarily focused on the development of visual diagrams using textual descriptions of models. When modifying a model, the corresponding diagram should update quickly enough that the developer does not need to wait for change to show up. Based on previous research from the field of HCI, one second is a fairly reasonable upper limit for such an interaction. Models smaller than the wagon model examined in this research can generally be updated in about one second or less using the old architecture. For these small models the incremental solution does not provide any advantage over the traditional system. Once models start getting larger though, the user experience just continually degrades. Here the incremental approach helps by letting the web client begin rendering portions of the diagram relatively quickly. So, to answer the question whether the incremental approach is useful, we have to differentiate between small and large models. These are different use cases and the top-down architecture definitely opens the door to making interactive exploration and editing of larger models viable.

In order to extend the top-down architecture to the rest of the system, certain questions need to answered. For steps like synthesis and layout it needs to be determined whether working incrementally from the top is possible and if not what needs to change to make it possible. At least for the layout portion we have already begun exploring this in this thesis and will get back to it in the next section. If the entire process can eventually be performed incrementally, diagrams could potentially become much larger and partial layout of a diagram also becomes an option. It is important to note though, that implementing these solutions and integrating them in the existing system which was not at all designed to support this kind of process, may not be the easiest approach to creating a fully incremental system.

4.1.2 Top-Down Layout

Aside from the general architecture and the communication between client and server, this thesis also explored the top-down concept in the context of automatic graph layout. This was motivated by the architectural considerations, but also serves as an exploration into visually different layout results when calculating layouts from the top downwards and not dependent on the lower levels of hierarchical graphs.

Automatic graph layout of hierarchical graphs poses a unique set of challenges when considering a top-down approach. The naive approach of treating layout as a black box and redefining the interface does not actually work when you start getting into the implementation details, at least not if the result should stay the same. This is due to the simple fact that layout of hierarchical graphs as it is performed in existing solutions depends on lower level elements being laid out before their parents.

This necessitates any solution for a top-down layout implementation to also create a new different layout result. How to technically solve this and what these solutions might look like was the focus in this thesis. For the architectural side, a top-down layout engine was experimented with as a means to truly compute layout top-down. Because this did not provide satisfactory results the focus shifted a little more to the visual side. The recursive layout engine was used, but certain constraints set top-down on the downwards pass of the recursion. The resulting layout variations provide interactive usable concepts for what a top-down layout can look like in practice.

Besides scalability for larger input graphs being a motivator of top-down visualization, there is another aspect that makes top-down layout specifically interesting. Stemming from the use case of live editing of models in software, the medium on which these diagrams are visualized is a computer screen. An important feature is that we can very flexibly zoom in and out on diagrams and that is something that bottom-up laid out diagrams do not leverage very much. They rely much more heavily on panning, because a diagram is spread out over a large space. Zooming here is used more to increase panning speed by zooming out, panning and then zooming back in to the same zoom level. In a top-down layout as presented in this thesis panning is kept to a minimum. Zooming is used not only to make small things larger and texts readable, but in a navigational sense to reach points of interest in a diagram. Panning is still useful when the layout on one level contains many elements that may be difficult to fit onto one screen, but mostly the layout corresponding to the current zoom level is concentrated within the boundaries defined by the viewport. A caveat of this is of course that elements on different levels are difficult to see together. It can, however, be argued that this is an acceptable trade-off compared to bottom-up layout, where neighbouring nodes can often not be viewed together at a single zoom level.

The solutions presented in this thesis still rely on an all-in-one process, whereas for the future development of fully incremental solutions, a truly top-down solution would be desirable. Solving the problems of the top-down layout engine provides one path to reach this goal. Since the layout algorithms used here operated on single non-hierarchical layers this is not a problem in theory.

4.2 Top-Down Graph Visualization

After reviewing the individual ideas explored in detail in this thesis, this section will now concentrate more on the overall topic of top-down graph visualization. This is a complex task requiring coordination of many smaller components. Integrating these small components into such a new overarching paradigm comes with challenges, some of which were already encountered within this work. How to prepare for these problems and deal with them when they come up is covered in this section.

4.2.1 Integration of Top-Down Solutions

So far we have considered the incremental communication and the top-down layout in isolation. This makes sense during development, because a simplified component is easier to understand, analyze and debug than a larger system. Eventually these individual components, and also potential future top-down components, should be integrated and function smoothly together. In this section we will take a closer look at what this entails for the case combining incremental communication with top-down layout and what new challenges arise. First we will analyze what happens when we just naively use both new technologies together and what works well and what does not. Then ideas to address the issues are presented.

On the surface, turning both top-down layout and incremental communication on works. The layout will look as expected and the pieces arrive one after the other to complete the diagram. Upon closer inspection though, somewhat unexpected behaviour occurs. The order in which pieces appear does not seem to be related to the viewport position. This is not the case, the viewport position is still relevant for determining which piece to request next. However, due to the nature of top-down layout, pieces tend to all be in one or a few grid cells. As we zoom further in the grid cells become larger relative to the node sizes. This is the core problem, the grid based piece requesting strategy was developed under the assumption that nodes are spread out over an area and that panning was the main way to reach them. Because of this, the grid based approach works for bottom-up layout, but not for top-down layout where nodes are more spread out over the zoom depth.

In the following a new piece requesting strategy concept is outlined, which aims to be more layout agnostic and also does not require setting vague parameters such as grid size and ring count.

First, the absolute positions of nodes are computed and stored upon piece arrival. This is similar to what the grid based idea does, but requires a different data structure and is more information. When a new piece needs to be requested, the viewport computes its intersection with all pending pieces and uses this as the basis for selecting which piece to request. The rough ordering would be: pieces completely enclosed by the viewport, pieces partially covered by the viewport, any other pieces. When multiple pieces are completely within the viewport, the distance between their midpoint and the midpoint of the viewport can be used as a further differentiation.

The challenge of this concept is the performance. In the grid based method, there are also a lot of checks, but data is stored in hash maps whose data access has a time complexity of $O(1)$. To achieve similarly quick results using the intersection method a solution could be to define some kind of hash function which maps a bounding box to some value which can also be directly compared to another hashed bounding box. The grid based approach is already a very rough version of this idea, but it is a solution that does not adapt to arbitrary zoom levels.

There is a good lesson to be taken away from this specific integration of components. While developing components isolated from each other, this only works when their interfaces remain stable. In this case the piece requesting strategy was developed with the original layout. The software interfaces did not change with the top-down layout later, but the algorithm output itself did. The diagram layout was now different enough to invalidate some of the assumptions that the piece requesting mechanism was built on. This example just shows how important it can be to define a clear specification of parts ahead of time, especially when multiple components depend on these details. When this is not possible to do precisely, as can arguably be said in the case of top-down layout, it should then be the focus to create general purpose solutions such as the intersection method rather than the grid method, even if that comes at a potential performance cost.

4.2.2 Hybrid Solutions

There are plenty of system components where there are good reasons why keeping a bottom-up process can make sense. For example, a process is already very fast and requires a lot of effort to change, or a process just plain does not work when using a different order of operations. This is, to some extent, the case with layout, which is why we are looking at alternative visualization options in the first place.

For components that fall into these categories we can apply hybrid solutions similar to what was done for top-down layout in this thesis. These solutions provide a stepping stone in transforming a process. They help in prototyping implementations that can provide working results and can be integrated into the existing infrastructure.

There are of course some caveats. While such hybrid solutions can provide decent intermediate results, they may also hinder overall progress by introducing new tech debt and reinforcing dependence on old systems. At some point when reinventing a system it makes more sense to rework parts from the ground up, rather than continuously modifying an existing system. Particularly when the foundation is being reimagined, it can mean a lot more work to make everything now also work well together with both the old and the new core concept.

Furthermore, there are some specific future concepts that might not be viable to even achieve when using hybrid solutions. One such example is visualization of unbounded graphs. This by definition requires a completely incremental approach and is therefore incompatible with a bottom-up method.

4.3 The Future

The implementation of incremental communication for sending a diagram model from the language server not only improves the interface usability for very large graphs in the current KEITH system, but it also serves as a working example demonstrating how top-down incremental ideas can be integrated in the overall bottom-up architecture.

Going forward it is important to identify the correct development priorities. Following the lessons learned while working on this thesis, there are some key takeaways that can help in the continued development and improvement of the ideas of the thesis. Firstly, it is important to note that a major architectural flip from a completely bottom-up system to an iterative top-down system is a tremendous challenge that should not be underestimated. Even when just focusing on specific parts, there are many dependencies that need to be well understood in order to integrate a working solution. Working through this component by component is definitely recommended while at the same time improved decoupling should be pursued as well.

Top-down layout showcases the challenges of reversing the layout process for hierarchical graphs as well as presenting possible solutions. Furthermore, there are now working prototypes that can be further refined to hone in on the desired final look of top-down layouts. With these foundations the door is open to further improvements, expansions of the ideas within the system or application of top-down visualization concepts in new contexts.

There are still quite a few open topics in the top-down layout part of this thesis, which should be addressed and solved. One of the most useful aspects of the implementation provided here is that it serves as a working example, which can be used for experimental evaluation on its usability and also provides a clear picture of what is desirable and possible as well as what should still be changed or revised.

Chapter 5

Conclusion

This thesis introduces the concept of top-down graph visualization and demonstrates a practical application of the idea within the KIELER and KEITH projects. With the development of an incremental communication protocol between client and server we accomplish two goals. The short-term goal of improving the user interface responsiveness for large graphs is improved by drastically reducing the wait times for changes within the diagram to become visible to the user. For an interactive system this is crucial to provide a good user experience. The new protocol also lays the foundation for further work expanding the top-down idea to other parts of the diagram visualization process.

The top-down layout concepts and solutions presented in this thesis are showcase examples for what top-down layouts can look like and where their strengths lie. Top-down layout demonstrates a clear advantage in readability of deep hierarchical graph visualizations over the classical bottom-up layout when viewing diagrams on a computer screen. Especially the task of navigating a large diagram and finding specific nodes is made a lot easier. While bottom-up layout requires users to hunt-and-peck for the labels by guessing where certain nodes could be and then zooming in and out until they guess successfully, top-down layout lets users gradually zoom in and is capable of always showing relevant labels at a readable size.

The work done for this thesis opens the door to further exploration, development and evaluation of top-down concepts with the focus of usability, performance and scalability. The results presented in this thesis provide opportunities for future research and development, and the features created so far demonstrate that further work in this topic is feasible, practical and useful. While the focus here has been on the practical integration of top-down ideas into the KIELER and KEITH projects, the concepts are very applicable to the more general context of graph visualization.

5.1 Future Work

With Chapter 4 covering the overall evaluation of the work done for this thesis and an abstract view of the road ahead, we can now identify some of the concrete steps necessary for achieving those goals. This section briefly summarizes some concrete goals and desirable next steps for continuing the work on top-down graph visualization.

5.1.1 Asynchronous Piece Requests

Currently, the incremental client-server communication uses a synchronous messaging protocol. This already provides a benefit in decreasing the time until the user can start interacting with the diagram, but there is a relatively simple further improvement that can be made. An asynchronous protocol would further increase speeds by further reducing idle time that is spent waiting for messages to arrive. In order for asynchronous requests to work, we need to consider the dependencies between messages. A new piece can only be requested if it is parent has already been received and inserted into the model. Other than that, pieces can be requested in an arbitrary ordering. This ordering is already completely handled on the server and the client and therefore, for asynchronous messages, the only change that needs to be made is to create and dispatch the correct Sprotty actions and handle synchronized modifying of the queue data structures safely.

5.1.2 Architectural Improvements

Some of the challenges with creating a new process within an existing system come from the architecture of the system, in particular from the coupling and dependencies between components. The main example of difficulties caused by this in this thesis is the handling of the text bounds calculation. It is a process that belongs to the layout step, but is dependent on the client and is therefore intertwined with the other communication steps. In the bottom-up architecture this is not a major problem, but when transforming the process to an iterative approach we are suddenly faced with many co-dependent components, which all need to be carefully taken care of to ensure the final system still functions as intended. Minimizing or removing these couplings would be a major contributor to easing the development and subsequent integration of individual top-down components.

5.1.3 Top-down Layout Tweaks

The top-down layout implementation still has some bugs that were already mentioned. The edge label problem, where edge labels are not positioned where they should be, still needs to be addressed and the presentation of regions that are scaled differently can be improved upon at the rendering level. On the face these issues seem fairly simple, but the challenge comes from the interplay of layout and rendering. The constraints used to created the top-down layout can be further tweaked, particularly a more advanced approach to dynamic constraints could create even better results. Lastly, a working top-down layout engine as a true top-down layout solution is a desirable next step as the general concept, and the desired visual results have now been covered quite thoroughly.

5.1.4 Rendering Optimization

Rendering as a process step is not covered very much in this thesis, but it plays an important role in both performance optimization and the presentation of the laid out diagrams. From the results in Section 2.3 we can see that rendering is not the biggest bottleneck in terms of scalability, but it still consumes a significant amount of time, which makes it interesting to look into further optimization opportunities.

5.1.5 Navigation Tasks

Navigation is a key use case of large diagrams, and while it was not a core focus of this thesis, top-down layout should be studied more in-depth specifically regarding its usefulness in navigating graphs. Studies covering many different graphs and configurations could help determine where precisely the advantages of top-down and bottom-up layout are with respect to navigation. We have so far stipulated that top-down layout has advantages specifically on the interactive computer screen medium, but further research is necessary to validate this claim. Especially, the task of comparing the navigation within a bottom-up layout versus a top-down layout in a quantitative manner is not trivial. Section 3.4.3 touches upon this topic, but there is a lot left to investigate.

5.1.6 Layout of Unbounded Graphs

In a fully top-down architecture, graph size would no longer be a limiting factor, as we would only need to handle the parts of the graph that are currently in view and could omit the rest. Of course, there remain some layout specific challenges, which are not so easy to solve in this manner and some dependency to other parts of the diagram will generally still be there. In practice this does limit how much can be incrementally built up.

Incremental layout of unbounded graphs is, however, an interesting concept that deserves some consideration when discussing top-down layout. While the case of SCCharts is not a suitable use case, there are clear applications of it, for example the visualization of very large or dynamically changing data. Any work done in the area of top-down visualization may also provide benefits to unbounded automatic graph layout.

One specific example of a feature that could also be useful for top-down layout in KEITH is incremental removal of pieces in addition to the piece insertion. This is useful when the space on the client is limited. While this could be added to the current implementation relatively easily, there remains a challenge, because the SGraph structure is built up incrementally on the server while the pieces are requested. If pieces were now removed and then later requested again, the server should only send that piece without any dangling children. This requires some clever data structures to handle this without creating a lot of new overhead with copying parts of the SGraph around.

5.1.7 Integration with the Google Maps for Models Project

The Google Maps for Models project is a master project offered to students by the Real-Time and Embedded Systems Group. The aim of the project is to incorporate concepts used in online maps such as Google Maps[1] into the visualization of models, in particular of SCCharts.

The most recent work by Bleßmann, Jöhnk and Pöhls [BJP21] on this project has focused client-side rendering improvements that make the user interface more responsive by reducing the amount of rendered SVG elements. Instead of rendering elements that are too small to be seen, they render a large readable title of higher level node. This achieves a similar effect as top-down layout, helping users locating nodes quickly without having to zoom in and out.

Their solutions improve the performance once the full diagram is loaded in the client, but the wait time is still an issue. Integrating the incremental diagram loading of this thesis with their work would provide KEITH with an overall performance improvement. Furthermore, it would be very interesting to not only compare their solution to top-down layout, but also to see the effect of having both active at the same time.

[1] `https://www.google.com/maps/`

Bibliography

[AHK06] James Abello, Frank van Ham, and Neeraj Krishnan. "Ask-graphview: A large scale graph visualization system". In: *IEEE Trans. Vis. Comput. Graph.* 12.5 (2006), pp. 669–676. DOI: 10.1109/TVCG.2006.120.

[BET+98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

[Bev01] Nigel Bevan. "International standards for HCI and usability". In: *International Journal of Human-Computer Studies* 55.4 (2001), pp. 533–552. ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.2001.0483. URL: https://www.sciencedirect.com/science/article/pii/S1071581901904835.

[BJP21] Bennet Bleßmann, Felix Jöhnk, and Mika Pöhls. *Project report for Google Maps for models*. Project Report. Kiel University, Department of Computer Science. Sept. 2021.

[BR17] Michael Burch and Thomas Reinhardt. "Dynamic graph visualization on different temporal granularities". In: *2017 21st International Conference Information Visualisation (IV)*. IEEE. 2017, pp. 230–235.

[BRS+07] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. "The aesthetics of graph visualization". In: *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAe'07)*. Banff, Alberta, Canada: Eurographics Association, 2007, pp. 57–64.

[DET+94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry: Theory and Applications* 4 (June 1994), pp. 235–282.

[DLH+21] Sören Domrös, Daniel Lucas, Reinhard von Hanxleden, and Klaus Jansen. "On order-preserving, gap-avoiding rectangle packing". In: *Proceedings of the 13th International Conference on Information Visualization Theory and Applications (IVAPP'21), part of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP'21)*. INSTICC. SciTePress, 2021, pp. 38–49. ISBN: 978-989-758-488-6. DOI: 10.5220/0010186400380049.

[Dom18] Sören Domrös. "Moving model-driven engineering from Eclipse to web technologies". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Nov. 2018.

[ELM+91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. "Preserving the mental map of a diagram". In: *Proceedings of the First International Conference on Computational Graphics and Visualization Techniques*. 1991, pp. 34–43.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. “Graph drawing by force-directed placement”. In: *Software—Practice & Experience* 21.11 (1991), pp. 1129–1164. ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.4380211102.

[FT08] Y. Frishman and A. Tal. “Online dynamic graph drawing”. In: *IEEE Transactions on Visualization and Computer Graphics* 14.4 (July 2008), pp. 727–740. ISSN: 1941-0506. DOI: 10.1109/TVCG.2008.11.

[GKW14] Michael Glueck, Azam Khan, and Daniel J Wigdor. “Dive in! enabling progressive loading for real-time navigation of data visualizations”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2014, pp. 561–570.

[Gra15] Martin Grandjean. “Introduction à la visualisation de données : l’analyse de réseau en histoire”. In: *Geschichte und Informatik* 18/19 (Jan. 2015), pp. 109–128.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[Kar11] Mårten Karlberg. “Mobile map client API: design and implementation for Android”. In: *LUMA-GIS Thesis* (2011).

[Köh17] Jan Köhnlein. *Sprotty – a web-based diagramming framework*. https://www.typefox.io/blog/sprotty-a-web-based-diagramming-framework. June 2017.

[KZ18] V. Kasyanov and Timur Zolotuhin. “A system for visualization of big attributed hierarchical graphs”. In: *International Journal of Computer Networks & Communications* 10 (2018), pp. 55–67.

[LS10] Zhicheng Liu and John Stasko. “Mental models, visual reasoning and interaction in information visualization: a top-down perspective”. In: *IEEE transactions on visualization and computer graphics* 16.6 (2010), pp. 999–1008.

[MEL+95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. “Layout adjustment and the mental map”. In: *Journal of Visual Languages & Computing* 6.2 (June 1995), pp. 183–210. DOI: 10.1006/jvlc.1995.1010.

[Mil68] Robert B Miller. “Response time in man-computer conversational transactions”. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. 1968, pp. 267–277.

[Nor96] Stephen C. North. “Incremental layout in DynaDAG”. In: *Proceedings of the Symposium on Graph Drawing*. Vol. 1027. LNCS. Springer, 1996, pp. 409–418. ISBN: 978-3-540-60723-6. DOI: 10.1007/BFb0021824.

[Oys14] Fred the Oyster. *Radial tree - graphic statistics in management*. Oct. 2014. URL: https://commons.wikimedia.org/wiki/File:Radial_tree_-_Graphic_Statistics_in_Management.svg.

[Ren18] Niklas Rentz. “Moving transient views from Eclipse to web technologies”. `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf`. Master’s thesis. Kiel University, Department of Computer Science, Nov. 2018.

[RES+17] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. “Generalized layerings for arbitrary and fixed drawing areas”. In: *Journal of Graph Algorithms and Applications* 21.5 (2017), pp. 823–856. DOI: `10.7155/jgaa.00441`.

[Sch19] Christoph Daniel Schulze. *Text in diagrams: challenges to and opportunities of automatic layout*. Kiel Computer Science Series 2019/4. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, CAU Kiel, 2019. DOI: `10.21941/kcss/2019/4`.

[SDK+06] Ahmed Seffah, Mohammad Donyaee, Rex B Kline, and Harkirat K Padda. “Usability measurement and metrics: a consolidated model”. In: *Software quality journal* 14.2 (2006), pp. 159–178.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. “Just model! – Putting automatic synthesis of node-link-diagrams into practice”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC ’13)*. San Jose, CA, USA: IEEE, Sept. 2013, pp. 75–82. DOI: `10.1109/VLHCC.2013.6645246`.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. “Methods for visual understanding of hierarchical system structures”. In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (Feb. 1981), pp. 109–125.