Intentional Layout in Sprotty Diagrams: Defining User Interaction

Jette Petzold

Bachelor's Thesis
September 26, 2019

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

Advised by
Sören Domrös

Selbstständigkeitserklärung

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

Kiel,

Abstract

Graphs can be defined visually or textually. Combining these two ways leads to a textual definition of graphs, which is used to automatically generate a diagram based on the textual definition. Unfortunately, automatic layouts do not consider semantic context. This can be solved by allowing the user to define constraints which are then considered by the used layout algorithm.

This thesis and the thesis of Schönberner introduce *Position Constraint* and *Layer Constraint* for the layered layout algorithm in KEITH, which can be set textually or by interacting with the displayed diagram. The layout process is adjusted to consider these constraints. This is done by using the *(semi) interactive* strategies of the layered layout algorithm offered by ELK.

In order to set constraints by interacting with the diagram, the user interaction and a UI is implemented. A constraint can be set by moving a node in the diagram to the desired position. In order to simplify the interaction, the UI visualizes the available layers and positions a node can be moved to and highlights which position the node is assigned to if released. In hierarchical graphs all elements that are not on the same hierarchical level as the moved node are hidden.

Feedback from members of the Real-Time and Embedded Systems Group of the Kiel University is implemented leading to the presented UI. Informal interviews indicate that the basic user interaction combined with the implemented user interface is intuitive. However, more features are desirable such as the creation of a new layer between two consecutive layers. The visualization of hierarchical graphs must be further evaluated since a subset of the participants is irritated while others are consent with the hiding of graph elements.

Acknowledgements

I want to thank Prof. Dr. Reinhard von Hanxleden for the possibility to write this thesis in his working group. Moreover, I want to thank my advisor Sören Domrös for helping us to orientate ourselves in KEITH and his feedback to my thesis. I also want to thank the whole working group, especially Sören Domrös, Niklas Rentz, and Christoph Daniel Schulze for always helping when a question arose. I want to thank Connor for being my colleague and the awesome cooperation.

I want to thank Janina for igniting my creativity and aesthetical skills, Steven Smyth for appreciating my drawings, and my friends for supporting me during my thesis. Finally, I want to thank "the couch" for carrying me through the thesis.

Contents

1 Introduction **1**
1.1 Problem Statement 1
1.2 Outline 2

2 Preliminaries **3**
2.1 Graph Definitions 3
2.2 Layered Layout Algorithm 3
2.3 Eclipse Layout Kernel 4
2.4 Language Server Protocol 4
2.5 Theia 6
2.6 Sprotty 6
2.7 Kiel Environment Integrated in Theia 7

3 Related Work **9**
3.1 Layout 9
3.2 User Interaction 10

4 Intentional Layout **13**
4.1 Constraints 13
4.2 Layout 14
4.3 Server Client Communication 15
4.4 Constraint Calculation 15
4.5 User Interface 17
4.5.1 Layer 17
4.5.2 Position 18
4.5.3 Edges 20
4.6 Set Constraints 20
4.7 Hierarchical Graphs 21

5 Implementation **23**
5.1 melk File 23
5.2 Postprocessor 24
5.3 Layout with Constraints 24
5.4 Language Server Extension 25
5.5 Graph Model 26
5.6 Interactive Module 26
5.7 Interactive View 29
5.8 Diagram Option 30

6 Evaluation **33**
6.1 Expert Feedback 33
6.2 User Study 34
6.2.1 Experiment 1 34
6.2.2 Experiment 2 34

7 Conclusion **37**
7.1 Summary 37
7.2 Future Work 37

Bibliography **39**

Abbreviations **39**

List of Figures

2.1 *Interactive* Layer Assignment 5
2.2 Differences of *semi interactive* and *interactive* strategy in the *Crossing Minimization* phase 5
2.3 *m* IDEs *n* languages problem 6
2.4 KEITH running in a browser 8
2.5 Example of an elkt file and the visualization of the specified graph in KEITH 8

4.1 Procedure for setting a constraint by interacting with the diagram 15
4.2 Calculation of the new layer 16
4.3 Calculation of the new position 17
4.4 Visualization of layers 18
4.5 Visualization of available positions 19
4.6 Visualization when only a layer is selected 19
4.7 Moving a node that has an edge 20
4.8 Icons for the different constraint types 21
4.9 UI in hierarchical graphs 22

5.1 Overview of the process of setting a constraint by interacting with the diagram 23
5.2 Activity diagram for setting IDs by Schönberner [Sch19] 24
5.3 Width calculation 25
5.4 Coordinates of the layers visualized by lines 28
5.5 Overview of the rendering 30

6.1 Zooming into a graph 33
6.2 Experiment 1 35
6.3 Experiment 2 36

Chapter 1

Introduction

Graphs are often used for representing relationships among objects. A graph can be defined in two ways: textual or visual. Using only one of these options has disadvantages. The textual representation of a graph alone becomes confusing really fast with growing graph size and semantic mistakes become harder to detect. Additionally, relationships between objects are not clearly identifiable. Only displaying the graph as a diagram makes it harder to work with. In order to change the graph a user interface must be provided containing the creation of graph elements. Usually only a limited set of these elements is supported. These disadvantages can be avoided by combining the two strategies. The user can write a textual definition of the graph in a specified language and the diagram is automatically generated based on it [SSH13].

When generating the diagram not only the graph elements specified textual must be created, additionally the layout of the graph must be determined, which specifies the positions of the nodes and edges of the graph. Creating the layout manually is time consuming. Not only the initial layout must be done but every time the graph is changed the layout must be adjusted. That is why automatic layout algorithms are used. The goal of automatic layout algorithms is to generate a "nice" drawing of a graph. Since "nice" can not be defined generally, the drawing is based on several aesthetic criteria. These criteria are e.g. number of edge crossings and symmetry. Different algorithms were implemented to generate a layout that fulfils certain criteria [BET+94]. One of them is the force-directed layout algorithm [Han18], which uses simulated forces on the nodes. Adjacent nodes attract each other through an attractive force while non-adjacent nodes repel each other through a repulsive force. Another one is the layered layout algorithm [Han18] which is based on Sugiyama [STT81]. It uses five phases to generate the layout: Cycle Breaking, Layer Assignment, Crossing Minimization, Node Placement and Edge Routing.

The advantage of these algorithms is that the user does not have to take care of the layout of the graph. Besides the aesthetic criteria, the algorithms should fulfill another criteria: preserve the mental map of the user [LLY06]. The mental map is the visual representation the user has in mind when thinking of the graph. If layout algorithms change the layout of a graph drastically when the graph has slightly changed, the mental map of the user differs from the presented layout and the user loses the orientation. Therefore, the algorithms must recalculate the layout based on the current one. But even if the mental map is preserved when the graph is changed, it could be that the initial layout did not match the mental map in the first place. A solution for this problem is allowing the user to change the layout of the graph manually after the automatic layout algorithm generated an initial layout and consider these changes in the following automatic layouts.

1.1 Problem Statement

The goal of this work and the work of Connor Schönberner [Sch19] is to allow the user to change the layout of the graph through interaction with the diagram after an automatic layout algorithm generated an initial layout. These changes should be persistent which means they should not vanish if the graph

is reloaded, in order to preserve the mental map of the user. We focus on the elkt language to specify graphs textual and the layered layout algorithm to layout the graph. The approach is implemented in Kiel Environment Integrated in Theia (KEITH).

In order to change the layout of a graph manually the nodes must be movable. Moving a node must generate a constraint which specifies the change of the graph and is saved such that the automatic layout algorithm can access it. Since this constraint should not vanish when the user reopens the diagram, it must be added to the textual representation of the graph. During the generation of the layout these constraints must be considered.

Additionally to this shared part, the work of Schönberner and me is divided into two parts. The constraints influence each other and therefore must be reevaluated if another one is set. Moreover, the constraints must be reevaluated when the graph is changed. Schönberner focuses on this part in his bachelor thesis [Sch19]. My part is to implement the user interaction. The user must know what can be done to manipulate the layout of the graph and how this is done. Furthermore, the user interaction must be evaluated in order to ensure the usability.

1.2 Outline

In Chapter 2 technologies that are used for the implementation of our approach and main concepts are presented. The related work in Chapter 3 shows different approaches to handle constraints from the user and definitions of user interaction. Chapter 4 illustrates the concept of the intentional layout. It explains several steps that are needed to be done in order to solve the stated problem and the user interaction. The implementation of the concept and the user interaction is explained in Chapter 5. Chapter 6 evaluates the intuitiveness of the user interaction in the diagram. Finally Chapter 7 sums up this thesis and points out future work.

Chapter 2

Preliminaries

In this chapter the main technologies and concepts, which are used in this thesis, are explained. This includes definitions for graphs, the layered layout algorithm, which can be used to create a drawing of a graph, and the Eclipse Layout Kernel (ELK)[1]. Moreover, the Language Server Protocol (LSP)[2], Sprotty[3], and Theia[4] are presented. Finally, KEITH[5] is introduced in which the solution for the stated problem in Section 1.1 is implemented.

2.1 Graph Definitions

A graph is a pair (V, E) where V is a set of nodes and E a set of edges [Han18]. In this thesis only directed graphs are used where edges are ordered pairs of nodes. The first node is the source of the edge and the second node the target. Sources as well as targets are called incident to their edge and the edge is incident to source and target. Nodes are adjacent if they are connected by an edge.

Besides these sets, a graph can be represented by a drawing. A drawing visualize the graph, whereby nodes are represented by points in a plane and edges by curves. In hierarchical graphs nodes can contain further nodes. Hierarchical graphs are defined by triples (V, E, h), where V and E are the same as in the graph definition and h is a function that maps each node to its parent.

A graph can have several connected components [Jan17]. A connected component is a subgraph in which all nodes are connected to each other by paths, that contain one or more edges, and are not connected to any node outside the connected component.

2.2 Layered Layout Algorithm

The layered layout algorithm consists of five phases [Han18]. *Cycle Breaking* is the first one, in which cycles in the graph are detected and eliminated. The elimination is done by reverting an edge of the cycle. Subsequently, the nodes get assigned to layers in the *Layer Assignment* phase. A layer is an ordered set of nodes, whereby adjacent nodes must be in different layers. The layers can be vertical and, therefore, ordered from left to right or horizontal. In the following chapters the layers are always vertical. Long edges are edges that do not connect nodes between consecutive layers. These edges are divided into several segments by introducing dummy nodes in the layers between the connected nodes. In the *Crossing Minimization* phase the ordering of the nodes in each layer is determined whereby the number of crossings of edges should be minimized. The absolute coordinates of the nodes are set in the last two phases. Based on the direction of the layers (horizontal or vertical) the *Node Placement* phase sets the horizontal respectively the vertical coordinates of the nodes. The ordering of the nodes,

[1] `https://github.com/eclipse/elk`
[2] `https://github.com/microsoft/language-server-protocol`
[3] `https://github.com/eclipse/sprotty` and `https://github.com/eclipse/sprotty-theia`
[4] `https://github.com/theia-ide/theia`
[5] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KEITH`

which is determined by the previous phase, is preserved. The *Edge Routing* determines the absolute coordinates of the nodes that are not set yet and the routings for all edges.

Between each pair of consecutive phases intermediate processing slots exists, which can contain several processors. They can also be added before the first and after the last phase. Processors can be used to adjust the layout of the graph in its current state.

In hierarchical graphs each hierarchical level is laid out independently as an individual graph.

2.3 Eclipse Layout Kernel

ELK offers an infrastructure to connect layout algorithms to editors and viewers. Furthermore, some layout algorithms are provided such as the layered layout algorithm[6]. Own layout algorithms can be implemented and used in ELK. Besides other features ELK supports hierarchical nodes. To adjust the layout of a graph layout options can be set (e.g. the layout algorithm that should be used). These options are implemented by annotating graph elements with properties.

In order to know which layout algorithms are available and which layout options they support metadata is used. Therefore, an *ELK Metadata File* must be written for every algorithm to be able to use it. In this file the options an algorithm supports are listed and declared, which usually includes an `ID`, `type`, `label`, `description`, `default` value, `lowerBound`, `upperBound`, and `targets`. `lowerBound` and `upperBound` determine which values can be set for the option while `targets` determine for which graph elements it can be set. If the targets are parents, the option can only be set for nodes that have children.

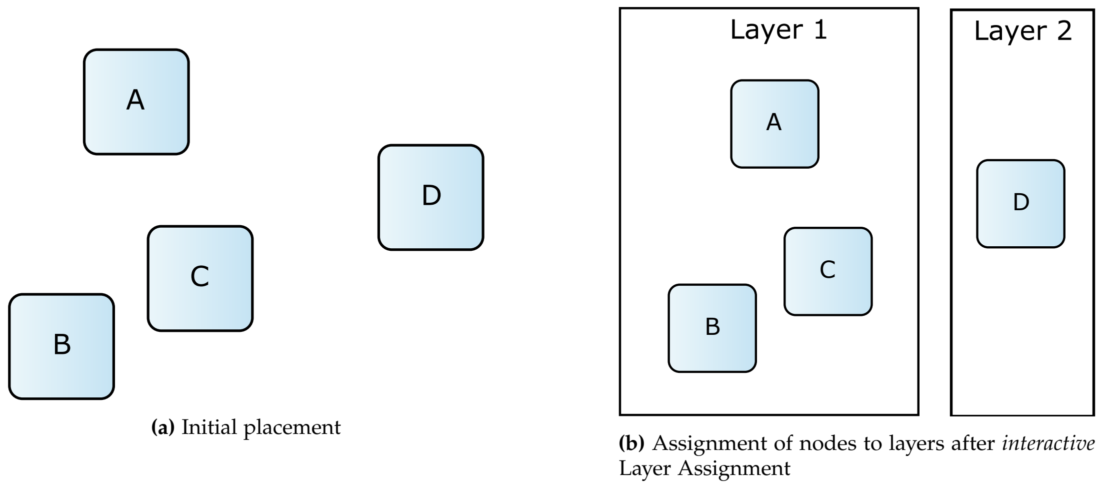
The layered layout algorithm provided by ELK supports several layout options including strategies for all five phases. Each phase has unique strategy values that can be set independently of the strategies of the other phases. The strategy determines how the goal of the phase is achieved. One strategy value, which can be set for each of the phases, is *interactive*. If the strategies of the phases is set to *interactive*, the nodes are placed based on their current coordinates. However, for the *Crossing Minimization* the *semi interactive* strategy is set. The *interactive Layer Assignment* phase assigns the layers based on the horizontal coordinates if the layers are vertical. An example assignment can be seen in Figure 2.1. Nodes whose horizontal coordinates overlap with the shape of another node get assigned to the same layer as this node as long as the nodes are not adjacent. In the *semi interactive Crossing Minimization* phase the ordering of the nodes is not done by minimizing the edge crossings instead the positions are based on the layout option `Position`. This option specifies x and y coordinates of a node. If the y coordinate of a node is lower than another one in the layer, the node is placed above this other node. In contrast to the pure *interactive* strategy, dummy nodes are placed optimal in respect of minimal edge crossings. An example of the differences in the layout can be seen in Figure 2.2.

Based on the layout options ELK generates a layout for the given graph. It only computes the positions and dimensions of the graph elements. The rendering of the drawing must be done elsewhere.

2.4 Language Server Protocol

Integrated Development Environments (IDEs) support language features such as auto-complete or find-all-references. In order to support such features IDE extensions are used which are written for the specific IDE. That is why an extension can not be used for different IDEs. As a consequence an own extension for each IDE that should support a certain language must be implemented. This leads to the m IDEs n languages problem [KPE16] (Figure 2.3). In order to support n languages for m IDEs $n \cdot m$ extensions are needed as seen in Figure 2.3a.

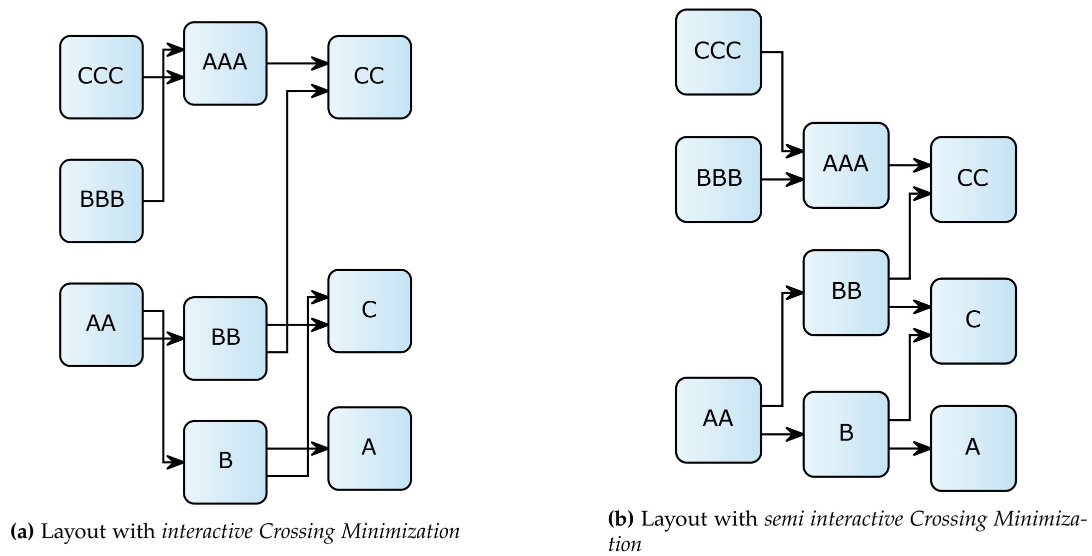
[6] `https://www.eclipse.org/elk/`



(a) Initial placement

(b) Assignment of nodes to layers after *interactive* Layer Assignment

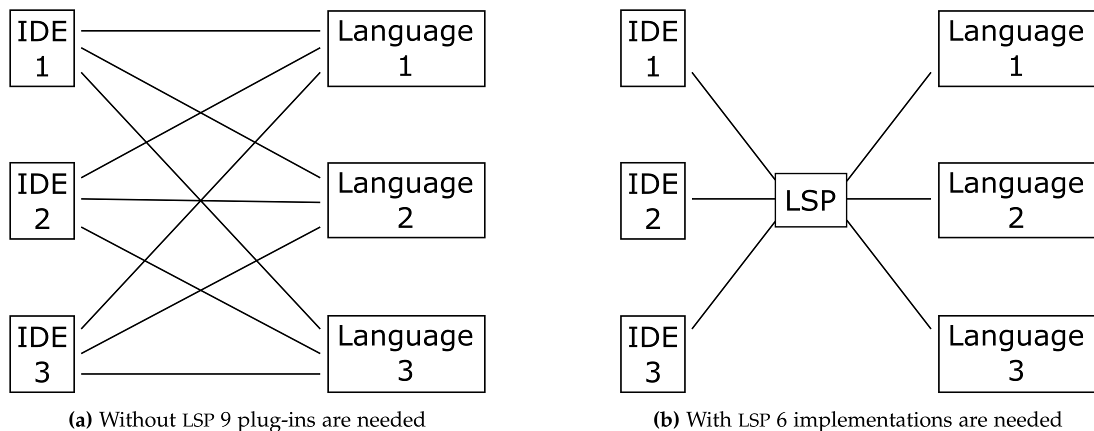
Figure 2.1. *Interactive* Layer Assignment



(a) Layout with *interactive Crossing Minimization*

(b) Layout with *semi interactive Crossing Minimization*

Figure 2.2. Differences of *semi interactive* and *interactive* strategy in the *Crossing Minimization* phase



(a) Without LSP 9 plug-ins are needed

(b) With LSP 6 implementations are needed

Figure 2.3. *m* IDEs *n* languages problem

LSP[7] is an communication protocol based on JavaScript Object Notation (JSON) Remote Procedure Call (JSON-RPC). It is used between a client that can be an editor or an IDE and a server which offers language features. The goal is to implement and offer language support independently of the IDE. With the LSP only one language server per language is needed. The IDE just has to implement the protocol in order to communicate with the server and in that way provide language support. This reduces the number of needed extensions stated in the m IDEs n languages problem to $n + m$ as seen in Figure 2.3b.

2.5 Theia

Theia[8] is an extensible framework to develop multi-language IDEs that is developed by TypeFox[9]. It consists of three components: frontend, backend and Language Server (LS). The backend runs on any server and communicates with the frontend. Additionally, the backend can communicate with different LSs via JSON-RPC, whereby several languages can be supported because of the LSP. The frontend as well as the backend can be extended. The developer can implement own extensions and can use extensions offered by Theia per default such as an text editor.

Theia can be launched as a web-based application with the frontend started in a local browser or as an electron app.

2.6 Sprotty

Sprotty[10] is a web-based graphics framework for diagrams that is developed by Typefox. It uses Scalable Vector Graphics (SVGs) to generate the diagrams and supports own implemented animations as well as morphing by state-changes. Additionally, it can be combined with ELK.

Sprotty can be divided into two parts: client and server. The client holds the model of the diagram, renders it and interacts with the user while the server knows the semantic model and how it is mapped

[7] https://microsoft.github.io/language-server-protocol/overview
[8] https://www.theia-ide.org
[9] https://typefox.io/
[10] https://typefox.io/sprotty-a-web-based-diagramming-framework

to the diagram elements. The two components are communicating with JSON-RPC. However, it is also possible to only use a client without a backend.

An integration in Theia already exists which allows to display the diagram created by Sprotty. Although SVG is supported by several browsers, Chrome is the best according to the developers.

2.7 Kiel Environment Integrated in Theia

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is an IDE for Model-Driven Engineering (MDE), which uses textual representation of graphs in order to create a drawing of the graph. The drawing is generated by performing a synthesis, calculate the size of the graph elements with KIELER Lightweight Diagrams (KLighD), lay out the graph with ELK, and finally render the graph with Piccolo2D[11], which is a graphical framework. Domrös [Dom18] and Rentz [Ren18] migrated KIELER to web technologies resulting in the software KEITH.

KEITH is a web-based implementation in Theia, which is also an IDE for MDE and uses the KIELER backend as a LS. The diagrams are generated by using the Sprotty framework instead of Piccolo2D. The GUI of KEITH running in a browser can be seen in Figure 2.4. KEITH offers an editor in which graphs can be defined textual in certain languages which leads to the generation of a drawing for the specified graph. Supported languages are e.g. Sequential Constructive Statecharts (SCCharts) and elkt, which is a textual representation of an elk graph. The resulting diagram is shown alongside the editor. To the right of the diagram an option menu can be toggled which offers options to adjust the diagram.

The client requests the diagram from the server. The server synthesizes the graph based on the opened file, lays out the graph with the layered layout algorithm of ELK, generates a Sprotty diagram with the calculated values and send this diagram to the client, who renders it. The nodes in the graph that is synthesized are called `KNode` while the nodes of the Sprotty diagram are called `SKNode`.

An example of an elkt file can be seen in Figure 2.5. Nodes are declared by using the keyword "node" and the name of the node. An edge is declared with the keyword "edge", which follows the name of the source, "->", and the name of the target. Nodes can contain further nodes by enclosing node declarations with curly brackets after the node declaration of the parent. Layout options can be set in this way, too.

[11] `http://piccolo2d.org/`

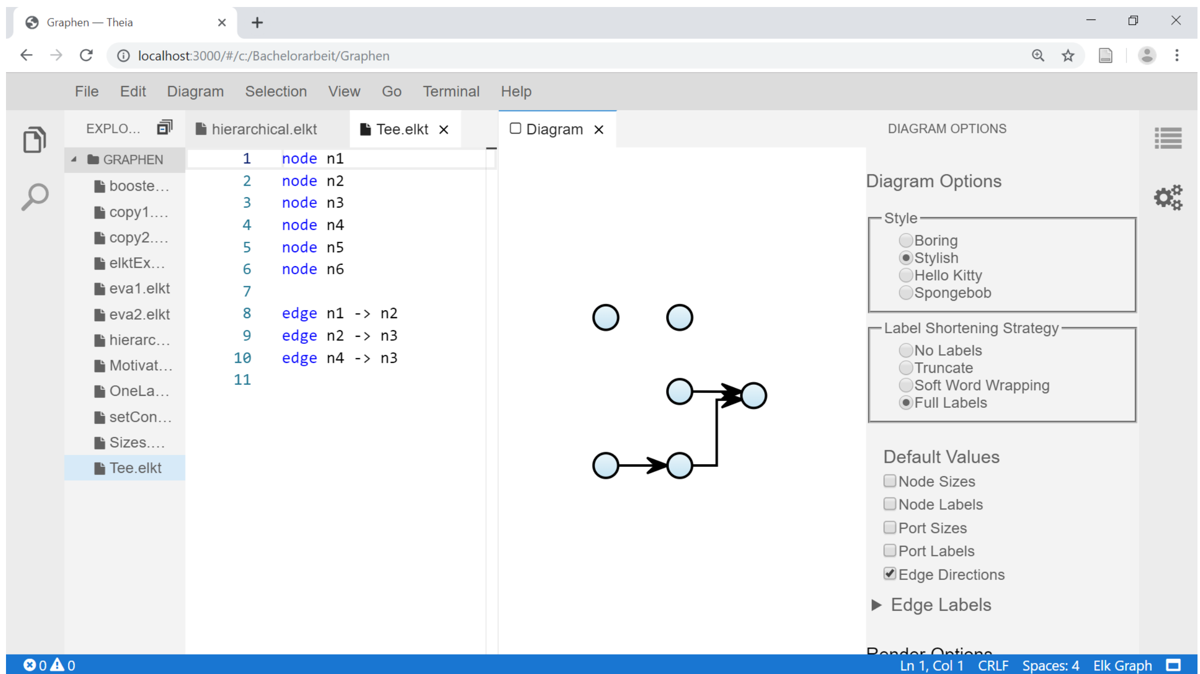


Figure 2.4. KEITH running in a browser

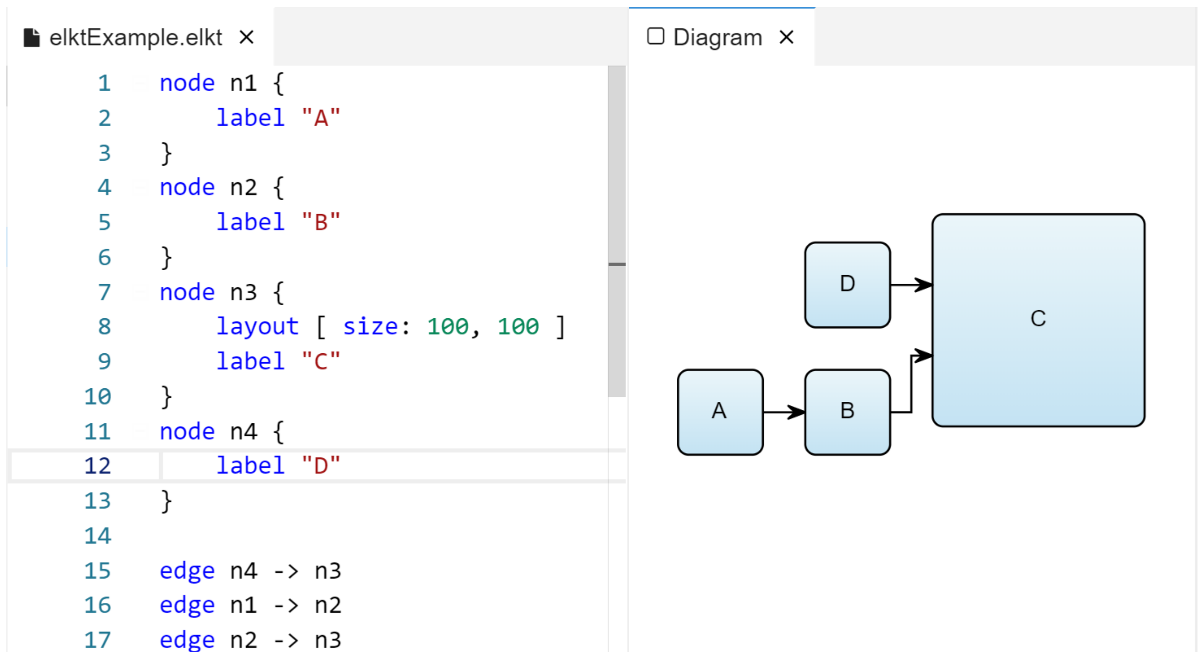


Figure 2.5. Example of an elkt file and the visualization of the specified graph in KEITH

Chapter 3

Related Work

There are several ways to allow the interaction with the diagram. A subset of them depends on the layout algorithm that is used. In this section some of the approaches that already exist are presented. The general layout of the graph considering user-defined changes is observed as well as the user interface that allows the user to interact with the graph.

3.1 Layout

Some projects exist that extend automatic layout algorithms in order to support constraints. A subset of the projects can be applied to every layout algorithm while others only work for a specified algorithm.

Böhringer and Paulisch are dealing with the problem that automatic layout algorithms can only generate drawings based on certain aesthetic criteria without considering constraints specified by a user or an application [BP90]. To solve this problem layout constraints are introduced. A general way to extend layout algorithms to support the constraints is shown and Sugiyama's layout algorithm [STT81] is used as an example. Three different constraints are introduced:

Absolute Positioning Determines the node position in a coordinate system.

Relative Positioning Determines the node position in relation to another node. The node can be left/right or above/below another node.

Clusters Collect a set of nodes in a group.

These constraints are defined by linear equations and a Constraint Manager maintains them and keeps them consistent. The phases of the layout algorithm are modified directly to support the constraints. Flat edges, which are edges that connect nodes in the same layer, are allowed by introducing a constraint that specifies that the two nodes must be immediate neighbours in the layer. The approach of Schönberner and me differs at the realization of constraints. The focus is absolute positioning and the constraints are implemented as layout options of nodes instead of equations. This is why a constraint manager is not needed. The validation of constraints, which is taken care of by the manager, is done by Schönberner [Sch19]. Further differences are that the layout algorithm itself is not changed and flat edges are not allowed.

Nascimento and Eades use hints provided by the user to generate good drawings [NE01]. After the graph is laid out the user can give hints which leads to a new drawing. The hints are categorized in three types: Focus, Layout Constraint, and manual changes. Thereby, Layout Constraint is divided into Top-Down and Left-Right constraint. In order to implement this approach the Sugiyama algorithm [STT81] is modified, such that the hints are supported. This modified algorithm is implemented in GDHints. While the user gives hints the system saves the best layout so far, which is determined based on five aesthetic criteria. The user can switch to the best layout or set the current layout as the best at any time. In contrast to the approach of Schönberner and me only relative relationships can be set.

Although manual changes can be done, the changes are not persistent and are only used to optimize the current layout.

Genetic Algorithm And Presentation-Assisted Graphic Object layout System (GALAPAGOS) is an automatic graph layout system [Mas92]. After it laid out the graph with a genetic algorithm, the user can add and modify constraints among graph elements. The constraints the user can specify are: a node is placed at a certain position, two nodes have the same vertical coordinate, or two nodes have the same horizontal coordinate. Since GALAPAGOS uses a different layout algorithm the constraints differ from the ones of Schönberner and me. Instead of specifying a certain position with a constraint, the layer of a node and the position in the layer can be set. Aligning nodes based on their horizontal coordinates corresponds to assigning the nodes to the same layer. However, such relative constraints are not implemented by us.

Several approaches are developed for force-directed layout algorithms. Ryall et al. developed Graph Layout Interactive Diagram Editor (GLIDE) for interactive graph layout [RMS97]. GLIDE is a constraint-based drawing editor, in which the user can interact with the graph even while the system is updating the diagram. The user can set constraints or apply forces directly to nodes by dragging them with the mouse. The constraints that are set by the user are converted to forces that influence the nodes. In addition to two syntactic constraints, which prevent node-node overlap and node-edge overlap, eight semantic constraints are provided.

Another approach is done by Dwyer et al., who present an Incremental Procedure for Separation Constraint Layout of graphs (IPSEP-COLA) [DKM06]. Its key idea is to extend force-directed layout algorithms in order to support so-called separation constraints.

Spritzer and Freitas present "A Physics-based Approach for Interactive Manipulation of Graph Visualizations" [SF08]. In their tool they allow the placement of virtual magnets, which attract nodes that fulfil certain user-defined criteria. This criteria can be based on the topology of the graph, properties of nodes or other magnets. These magnets can be placed by the user after an initial layout is done by a force-directed layout algorithm.

The constraints used in the force-directed layouts can not be applied to the layered algorithm, which is used by Schönberner and me. For example the virtual magnets can not be used, since the layered algorithm does not use forces, and magnets that apply forces on nodes would destroy the layers.

3.2 User Interaction

Besides the implementation of the constraints and the support of them, the user interaction, allowing the user to set constraints, must be specified. There are several ways the user could interact with the diagram. Some of them work for all layout algorithms such as pop-up menus while others are specific for one algorithm. To show the variety of the interaction possibilities some of them are presented here.

The modified Sugiyama algorithm and constraint manager implemented by Böhringer and Paulisch is included in Extendible Directed Graph Editor (EDGE) [PT90]. The constraints they introduced can be set by the user via a set of pop-up menus. In order to change a constraint nodes must be selected by clicking and a menu must be filled out which leads to sending a command to the constraint manager [BP90].

In GLIDE the user can set constraints, which are called Visual Organization Features (VOFs). On the right side of the GLIDE interface are buttons for the different VOFs. Besides moving a node manually in the diagram, a set of nodes an be selected by region-dragging. To these nodes VOFs can be added by pushing the corresponding bottom that is shown on the right [RMS97].

McGuffin and Jurisica improved the user interface in Network Analysis, Visualization, & Graphing TORonto (NAViGaTOR) [MJ09]. The initial layout is done by a force-directed layout. Additionally, nodes can be selected and moved by the user, optionally fix their position while the layout algorithm is used to lay out the other nodes. The selection is done by a combination of lasso and rectangle selection. The available actions, including selection and node-specific actions, are mapped to combinations of mouse button pressed/released and whether that happens on whitespace or a node. Furthermore, the Control key can be pressed which leads to additional actions when combined with dragging nodes/whitespace. Some of the actions invoke a popup radial menu or a pop-up hotbox while others allow to select or move nodes.

In my approach no pop-up menus or buttons for the constraints are used. Constraints are only set by moving a node to the desired position. This can be done since only the absolute positioning of a node is supported so far. When working with relative constraints pop-menus can be useful. Moreover, selecting a group of nodes and assign them constraints is not possible.

Chapter 4

Intentional Layout

This chapter explains the main concept of intentional layout developed by Schönberner and me. The concept consists of the constraints, which are used to save the changes of the graph, the layout process to consider the constraints, and the communication between client and server. For the layout of the graph basically the layered layout algorithm is used, whereby the goal is to not modify the algorithm itself but adjust the layout process in KEITH. The communication is necessary because the calculation of the constraint happens on the client side and only the server can add the constraint to the model of the graph. Constraints should be saved in the model in order to be persistent.

The concept also consists of the calculation of the constraints. When the user interacts with the diagram, the constraint that is set by releasing the node must be calculated based on the position the node is moved to. Thereby, it shall not be allowed to move nodes to another hierarchical level. An User Interface (UI) must be implemented to inform the user which constraints are set by moving a node to certain positions. Besides this information, an overview about the set constraints is given by visualizing the constraints through icons in the diagram. Hierarchical graphs are a special case which must be considered in the concept, too. Overall, the UI should be clear enough to be helpful implying that the visual clutter should be minimized.

While the introduced constraints, the modification of the layout process (also for hierarchical graphs), and the communication between the client and server are dealt with by Schönberner [Sch19] and in this thesis, only in this thesis the calculation of the constraints when a node is moved, the UI, the visualization of set constraints, and the visualization for hierarchical graphs is done. However Schönberner adjusts the calculation of the constraints for his needs [Sch19].

4.1 Constraints

There are several constraints that could be supported as seen in Chapter 3 for example relative positioning and absolute positioning. Since the layered layout algorithm is used, these positionings are split up into several constraints:

Layer Constraint Determines the layer the node should be in, whereby 0 is the first layer.

Position Constraint Sets the position the node should have in its layer. The top position is 0.

left Of/right Of Specifies which layer the node should be in in relation to another node.

below/above Specifies at which position the node should be based on another node.

same Layer as The node should be in the same layer as another node.

Thereby, the first two constraints define the absolute positioning of a node and the other ones the relative positioning. Schönberner and I decided to focus on implementing absolute positioning, because relative relationships can be realized with absolute positioning. The other way around is much

more laborious. The values of these constraints are interpreted as priorities instead of static values. If the *Layer Constraint* is set to 100 but there only exists 4 layers, the node with this constraint is assigned to the 5th layer.

In order to set a constraint for a node the constraint must be saved in the model. As described in Section 2.3 ELK allows to define layout options for nodes, which can be supported by a layout algorithm. By adding options that represent the constraints, these options can be set when a constraint should be set. These options can than be considered in the layout process.

4.2 Layout

Constraints must be considered in the layout by the layout algorithm. There are two possibilities to achieve this. The first one is to modify the layout algorithm itself. In the layered layout algorithm only the first three phases are interesting because the last two set the absolute coordinates and do not change the relative position of nodes as explained in Section 2.2. Each of these three phases (*Cycle Breaking*, *Layer Assignment*, *Crossing Minimization*) can be modified in order to consider the constraints. Another solution is to calculate an initial layout with the unmodified layered algorithm and subsequently adjust the coordinates of the nodes based on their constraints. If another layout is done in which the nodes get assigned to layers and ordering positions based on their coordinates, the resulting layout considers the constraints. We decided to implement the second solution because of the goal to not change the layered layout algorithm itself.

As explained in Section 2.3 ELK offers the *(semi) interactive* strategy for the phases in the layered algorithm. Since the last two phases do not change the relative positions of the nodes the *(semi) interactive* strategy must only be set for the first three phases. This allows us to implement the following procedure to consider the constraints:

1. Generate an initial layout with the layered layout algorithm.
2. Adjust the coordinates of the nodes based on their constraints to set them at the correct position.
3. Set the strategies of the first two phases of the layout algorithm to *interactive* and the strategy of *Crossing Minimization* to *semi interactive*.
4. Generate the final layout of the graph with the layout algorithm.

The advantage of this procedure is that the layout algorithm itself must not be changed. Additionally, strategies that are already implemented can be used, which is why only the coordinates of the nodes must be calculated. Step 1, 3 and 4 are just usage of already implemented functionality. A disadvantage is that it must be taken care of hierarchical graphs particularly.

In order to calculate the correct coordinates of the nodes in step 2 the positions and layers of them must be known. Therefore, internal layout options for nodes are used: one for the *Layer ID* and another one for the *Position ID* of a node. These options are set by a *postprocessor* after the last phase of the layered algorithm. Since an initial layout is done in step 1 these IDs can be used in step 2. When calculating the coordinates the constraints are prioritized over the IDs. All nodes that should be in the same layer get the same horizontal coordinate which depends on the coordinate the nodes in the previous layer get. That is why the calculation begins at the first layer. The coordinate for the first layer can be set to any value while the value for the following layers is set such that the coordinate do not overlap with a node shape of the previous layer. In this way the layout algorithm places the nodes in the desired layers if the *interactive* strategy is set. The vertical coordinates are based on the positions

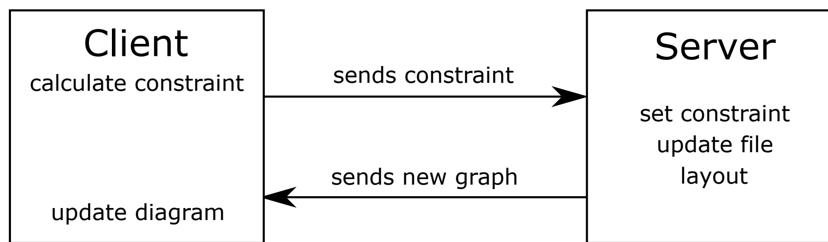


Figure 4.1. Procedure for setting a constraint by interacting with the diagram

the nodes should be in. The top node can be assigned any value and all other nodes get a value that is greater than the node that is directly above them.

Another layout option that is offered by ELK is `separateConnectedComponents`. By default its value is true. This means that the connected components of the graph are laid out separately and then put together. The presented procedure only works properly if this option is set to false. Otherwise nodes with the same *Layer Constraint* value could be placed in different layers if they are in different connected components.

In conclusion: if `separateConnectedComponents` is set to false and the graph goes through the procedure with the presented calculation of the coordinates, the nodes in the resulting graph are set properly according to their set constraints.

4.3 Server Client Communication

The constraints that should be set for a node are determined by user interaction with the diagram which happens on the client side. In order to set the constraints the corresponding layout options of the node must be set. However, the client does not know the semantic model of the graph only the server does, which is why the client must communicate with the server.

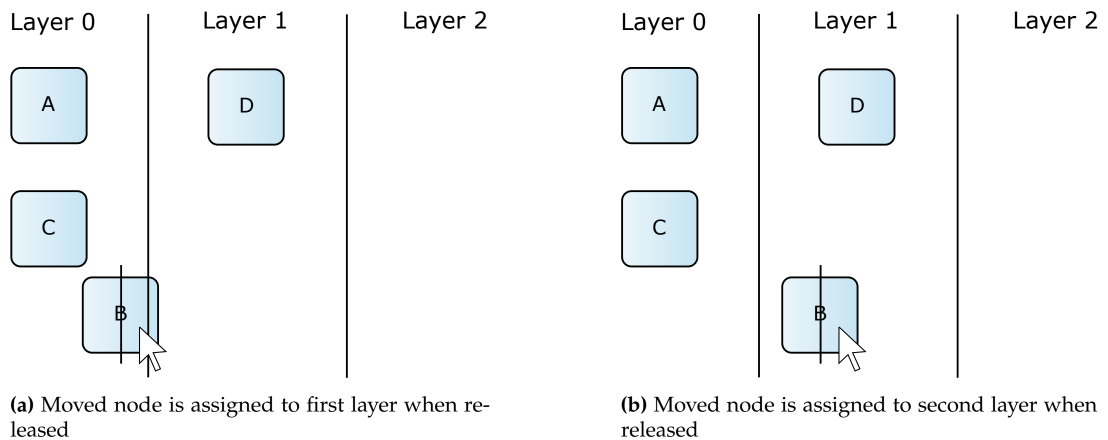
Server and client are communicating through JSON-RPC. The server defines requests and notifications the client can send as well as messages the server can send. Since KEITH did not support constraints before there aren't any definitions of notifications to set layout options of nodes. However, the needed notifications can be added by extending the LS.

The general procedure for setting a constraint for a node by interacting with the diagram is shown in Figure 4.1. After the client calculated the constraint that should be set, this constraint is send to the server. On the server side the corresponding layout option is set for the specified node, the file is updated based on the new model, and a new layout is generated. Subsequently, the new graph is send to the client who updates the diagram.

4.4 Constraint Calculation

Besides allowing the user to write the desired constraints in the editor the constraints shall be set through interaction with the diagram. Therefore, it must be specified which interactions with the diagram lead to setting constraints and which constraints these are.

As seen in Section 3.2 there are different ways of allowing interaction with the diagram. One possibility for setting a *Layer Constraint* or *Position Constraint* is to allow the moving of nodes. The constraint that is set through the moving can than be calculated based on the position a node is moved



(a) Moved node is assigned to first layer when released

(b) Moved node is assigned to second layer when released

Figure 4.2. Calculation of the new layer

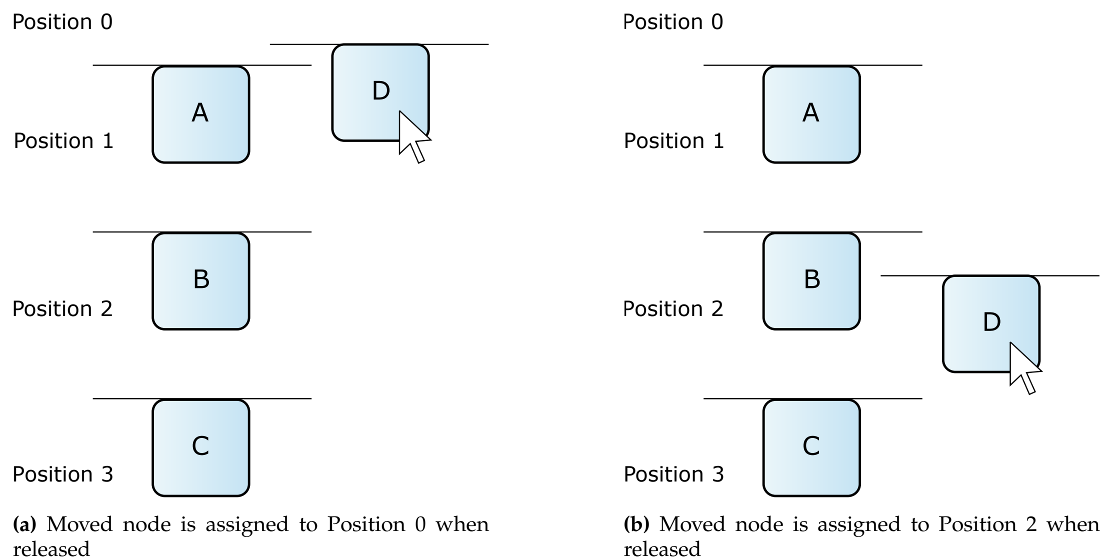
to. The relevant information about the position the node is moved to is the layer and ordering position the node would have when it is placed at the new position.

In order to determine the new layer the coordinates of all existing layers are calculated. The layer the moved node is assigned to depends on the horizontal coordinates, which can be seen in Figure 4.2. For each pair of consecutive layers the middle of them is calculated. If the horizontal middle of the moved node is lower than the middle of the layers, it is assigned to the first layer otherwise to the second layer of the pair. Additionally, it is possible to create a new layer and assign it to the node wherefore the node must be moved to the right of the last layer. Create new layers could also be allowed by moving the node to the left of the first layer or between two consecutive layers. However, this is not implemented yet. That is why the node is assigned to the first layer if it is moved to the left of it.

Since the new position in the layer only depends on the nodes in the same layer, first the new layer and then the new position is calculated. With the information about the layer all nodes that are in the same layer can be determined by the *Layer ID* and *Layer Constraint* of the nodes. The new position of the moved node is then calculated by comparing its vertical coordinate with the coordinates of the other nodes in the layer as shown in Figure 4.3. If the value is lower than all of the others, the new position is 0 otherwise it is the position of the node with the greatest value lower than the moved one increased by 1.

The node can be moved to any existing layer at any position. As long as the layer as well as the position of the moved node do not change, no constraint is set when the node is released only the diagram is reloaded. If the layer stays the same but the position of the node changes, only the *Position Constraint* is set. Only the *Layer Constraint* is set when the layer of the moved node changed and it is released above or below the current layer. In order to determine whether a node is below or above a layer, the vertical coordinates of all layers are determined. These coordinates are based on the coordinates of the uppermost and the lowermost node. Both constraints are set if the layer changed and the node is within the vertical coordinates of the layer.

It is possible that some positions are not allowed for a node to be moved to if other nodes already have constraints. One case is that adjacent nodes are not allowed to be in the same layer. The solution of this problem is explained by Schönberner [Sch19].



(a) Moved node is assigned to Position 0 when released

(b) Moved node is assigned to Position 2 when released

Figure 4.3. Calculation of the new position

4.5 User Interface

The user must get an idea of the result of moving a node. Therefore, visual feedback is necessary. Since the constraints are calculated based on the position a node is moved to, the feedback should contain information about what constraints are set if the moved node is released on a certain position. For each of the two constraints an own visualization is defined.

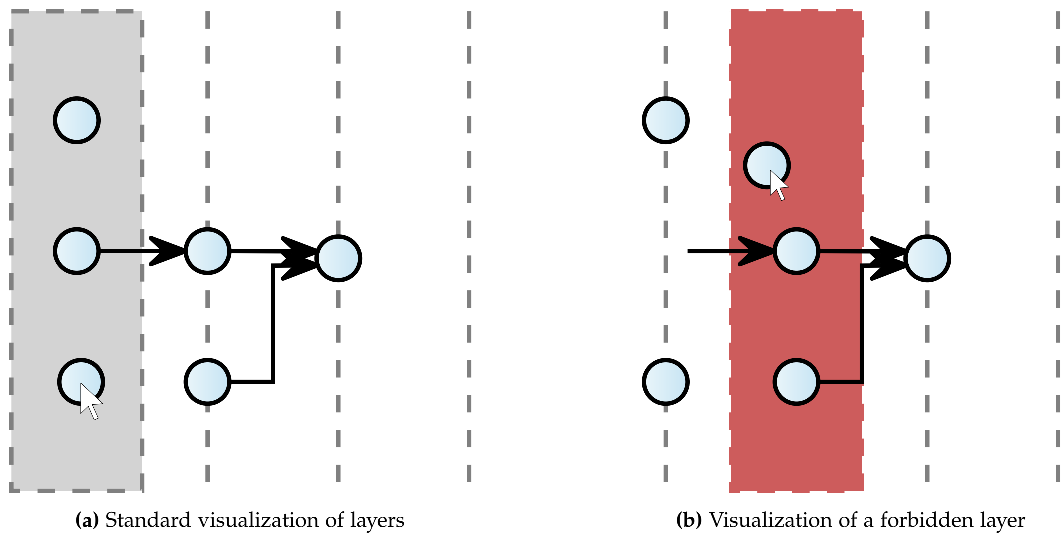
If only the node is moved and the incident edges are ignored, these edges point to empty space. That is why it must be taken care of the incident edges.

4.5.1 Layer

In the shown diagram the graph consists of several layers, which contain the nodes of the graph. The visualization of the layers can be seen in Figure 4.4a. With the *Layer Constraint* a node can be assigned to another layer than the one it is currently in. Therefore, the node must be moved into the desired layer. Since the user should know to which layer a node is assigned when releasing it at a certain position, the layer the node would currently be assigned to is visualized by a dashed rectangle. The rectangle surrounds all the nodes that are in the current layer. All other layers the node could be assigned to by moving it to another position are visualized through vertical dashed lines, which are placed in the middle of the layers. In this way the user only sees dashed lines when zooming in, which is why the rectangle is also highlighted in lightgrey in order to prevent confusion. The colour changes to grey if the node is moved above or below the layer and hence only a *Layer Constraint* should be set as explained in Section 4.4 and shown in Figure 4.6.

The possible creation of a new layer is visualized by an additional vertical line to the right of the last layer. If a node is moved to that layer, the vertical line becomes a rectangle and releasing the node leads to the creation of the new layer. This can only be done if the moved node is not already the only one in the last layer of the graph.

The positions of the visualizations are based on the coordinates of the layers which calculation is explained later in Section 5.6. As a consequence the diagram always contains a rectangle indicating the



(a) Standard visualization of layers

(b) Visualization of a forbidden layer

Figure 4.4. Visualization of layers

current layer that would be assigned to the moved node and several lines indicating all other layers the node could be assigned to when a node is moved.

As explained in Section 4.4 adjacent nodes are not allowed to be in the same layer. If a node has a *Layer Constraint* and an adjacent node is moved to the same layer, this layer is highlighted red instead of grey to indicate that this is not allowed which can be seen in Figure 4.4b.

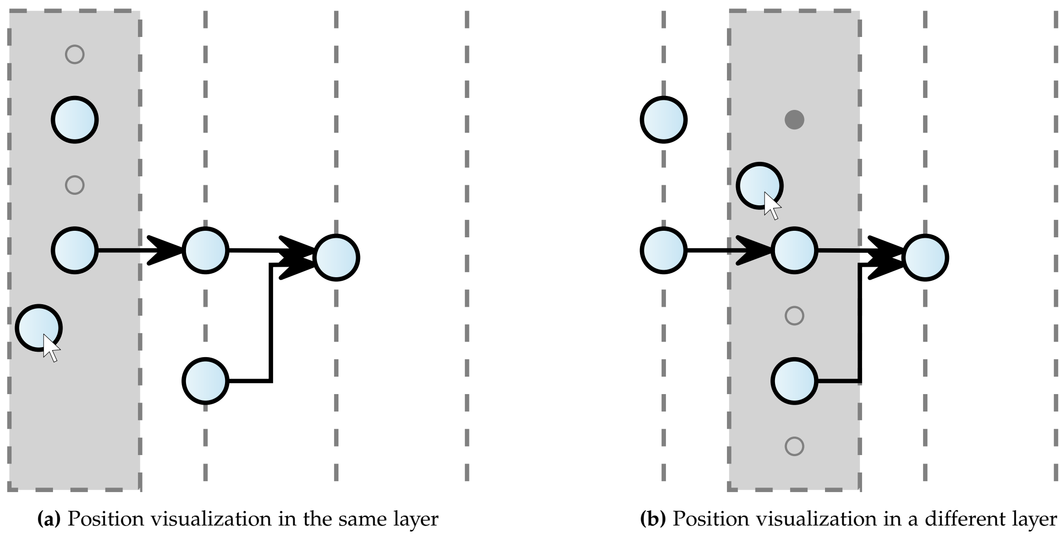
4.5.2 Position

Besides the *Layer Constraint* also the *Position Constraint* can be set through interaction with the diagram. In each layer of the graph the nodes are ordered in a sequence from top to bottom. The *Position Constraint* determines the position of the node in this sequence. Although, a node can be assigned to any position in any layer, only the available positions in the current layer are relevant. To assign a node to a position in another layer the node must be moved to this layer first. However, this makes the other layer the current layer, which is why it is sufficient to only focus on the positions in the current layer.

The positions in the current layer the node can be assigned to depends on the original position of the moved node. The visualization of these available positions can be seen in Figure 4.5. Generally, the positions a node can be assigned to are above the first node, between two consecutive nodes, and below the last node. The position above the first node respectively below the last node is excluded if the moved node was originally the first respectively the last node in this layer. A position between two consecutive nodes is excluded if the moved node was already originally between these two nodes.

The available positions in the current layer are visualized by circles, in the following referred by *Node Slots*. Circles are chosen because of their similarity to radio buttons. The *Node Slots* are placed above the first node, in the middle of two consecutive nodes, and below the last node. A filled circle indicates that the node would be assigned to the position the *Node Slot* represents when the node is released.

However, the *Node Slots* are not shown at all if only the *Layer Constraint* should be set, which can be seen in Figure 4.6.



(a) Position visualization in the same layer

(b) Position visualization in a different layer

Figure 4.5. Visualization of available positions

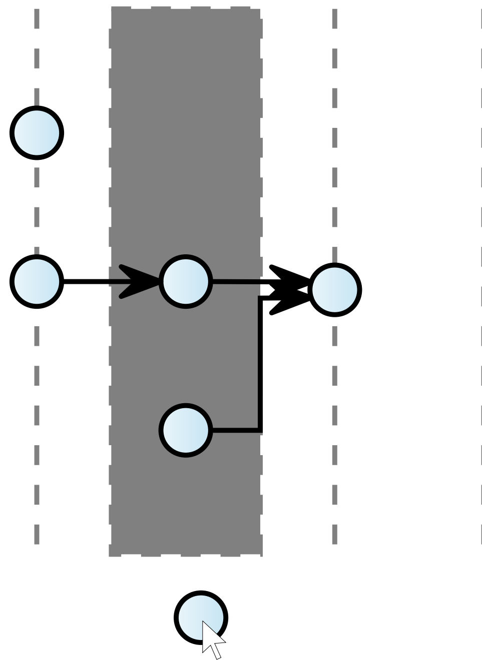
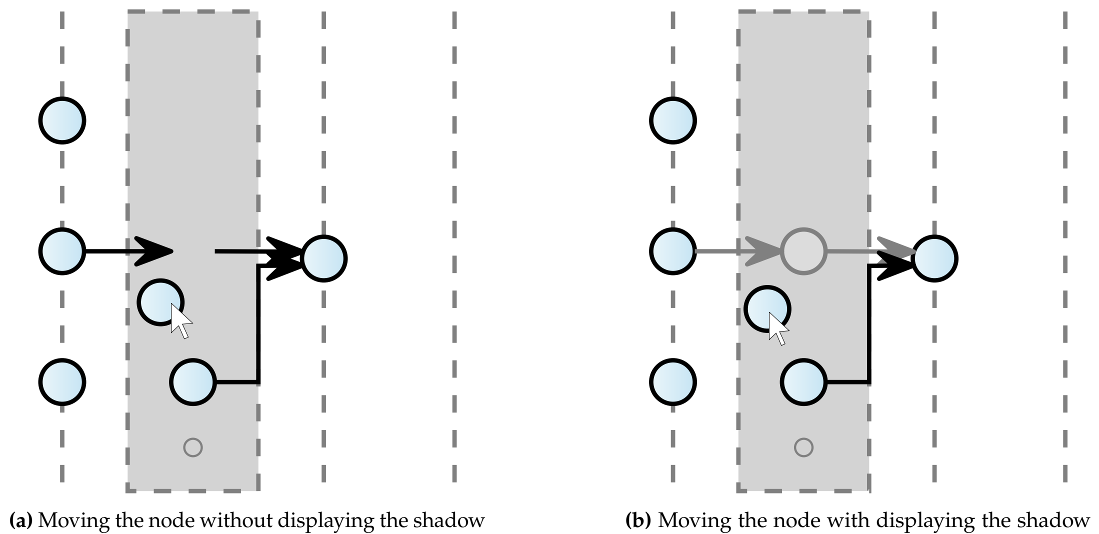


Figure 4.6. Visualization when only a layer is selected



(a) Moving the node without displaying the shadow

(b) Moving the node with displaying the shadow

Figure 4.7. Moving a node that has an edge

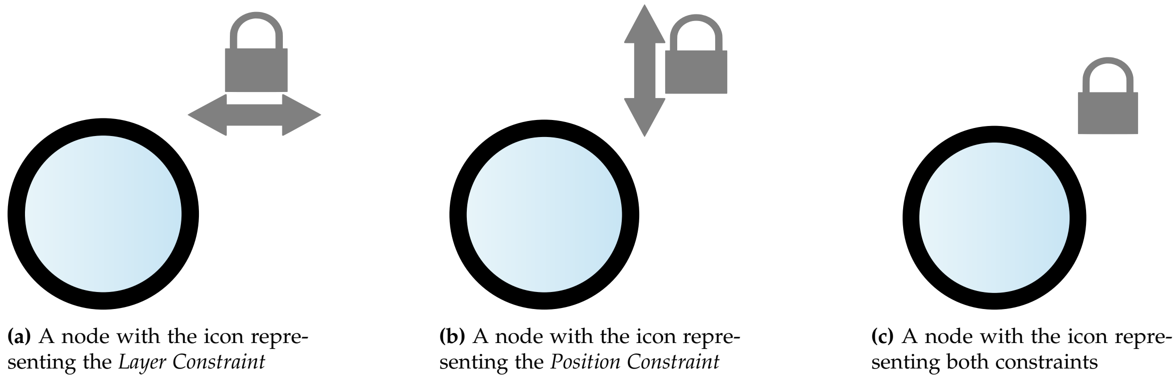
4.5.3 Edges

When a node is moved, the incident edges are not moved at all which leads to edges that point to nothing as seen in Figure 4.7a. One option to avoid this, is to move the edges together with the node but that increases the visual clutter without added value. Even if the edges are moved too as soon as the node is placed the layout of the graph changes which leads to a new routing of the edges. That is why the user does not know the routing of the edge although it is moved. However, when the edges are not moved they point to nothing. This is avoided by placing a shadow of the moved node at the originally position of it as seen in Figure 4.7b. This shadow has the same shape as the original node but is greyed out. In this way not only the edge problem is solved but also the user knows the original position of the moved node even if it has no edges. Additionally, the incident edges are greyed out too to indicate that they belong to the moved node.

4.6 Set Constraints

In order to give the user an overview about the constraints which are currently set, the different constraint types are visualized by an icon attached to the nodes as seen in Figure 4.8. If a node has a *Position Constraint* as well as a *Layer Constraint*, the icon attached to the node is a lock. In case that only one of the constraints is set the icon is a lock with an arrow. The arrow is horizontal if the *Layer Constraint* is set, to indicate that the horizontal position of the node is determined by the constraint. Analogously, the arrow is vertical if the *Position Constraint* is set, to indicate that the vertical position is determined.

Since the graph and therefore the number of icons can become large, which results in visual clutter, the user should decide whether the icons are shown or not. In order to allow the user this decision a new diagram option is introduced: *Show Constraint*. As default this option is false in order to prevent visual clutter but in the diagram option menu it can be set to true which leads to the showing of the icons.



(a) A node with the icon representing the *Layer Constraint*

(b) A node with the icon representing the *Position Constraint*

(c) A node with the icon representing both constraints

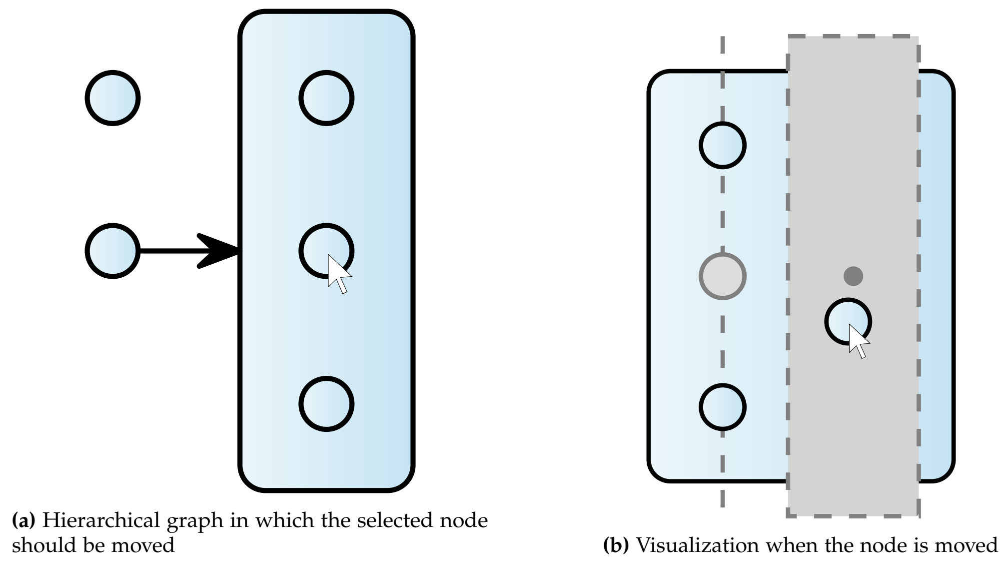
Figure 4.8. Icons for the different constraint types

4.7 Hierarchical Graphs

Calculating and setting of constraints should not only work on graphs with one hierarchical level. In order to support the constraints also for hierarchical graphs, the constraint of a node must be calculated based on the nodes in the same level and not all nodes of the graph. Therefore, if the constraint is calculated because of the movement of a node, only the layers in the current level are observed while determining the *Layer Constraint* of the moved node. Since the node shall not be movable to another level, the current level is always the level the node is originally in.

In order to support the constraints properly it must be considered that the shape of a parent node can change when the children of it are laid out. If this changed size is not taken into account in step 2 of the layout process explained in Section 4.2, the coordinate of a layer could overlap with the shape of the parent node which leads to the combining of the supposed two layers into one. As a consequence the coordinates of the nodes in the deepest hierarchical level must be calculated first and the size of the parent must be adjusted accordingly to the resulted width. Thereby, the coordinates are set based on the other nodes in the same hierarchical level ignoring the nodes on other levels.

The UI in the diagram of hierarchical graphs can be seen in Figure 4.9. It is only shown for the hierarchical level the moved node is in since nodes can not be moved to another level. All elements of the graph that are neither in the same level nor the parent of the moved node are hidden in order to highlight that the node can only be moved in its level. Additionally, the parent of the moved node is resized in order to contain the UI as long as it is not the root of the graph.



(a) Hierarchical graph in which the selected node should be moved

(b) Visualization when the node is moved

Figure 4.9. UI in hierarchical graphs

Chapter 5

Implementation

This chapter explains the technical details regarding the implementation of the concept presented in the previous chapter. The general structure of KEITH can be seen in the master's theses of Domrös [Dom18] and Rentz [Ren18]. Schönberner and I extend the implementation of them in order to realize the interaction with the diagram. In the following sections only these modifications and extensions are explained.

In order to implement the concept, a melk file and a *postprocessor* must be created, the new procedure for the layout must be implemented and the LS extended. An abstract overview of the process can be seen in Figure 5.1. All of this is done in collaboration with Schönberner [Sch19]. The user interaction presented in the previous chapter is implemented through an interactive module, an interactive view, and a new diagram option, which are presented in this thesis.

5.1 melk File

As mentioned in Section 2.3, for each algorithm in ELK a melk file exists in which the options the algorithm supports are listed and defined. The constraints should be supported by the layered layout algorithm and therefore the melk file for this algorithm, which is `Layered.melk`, must be modified. In this file the two layout options `layerChoiceConstraint` and `positionChoiceConstraint`, representing the two introduced constraints in Section 4.1, as well as the `layerID` and `positionID` are added. The default value and lower bound of these options is -1 and the target of each is node which means that the options can be set for every node. Additionally, the option `interactiveLayout` is introduced which default value is false and target is parents which allows to set the option only for nodes that have children. This option determines whether the diagram should be changeable through interaction and

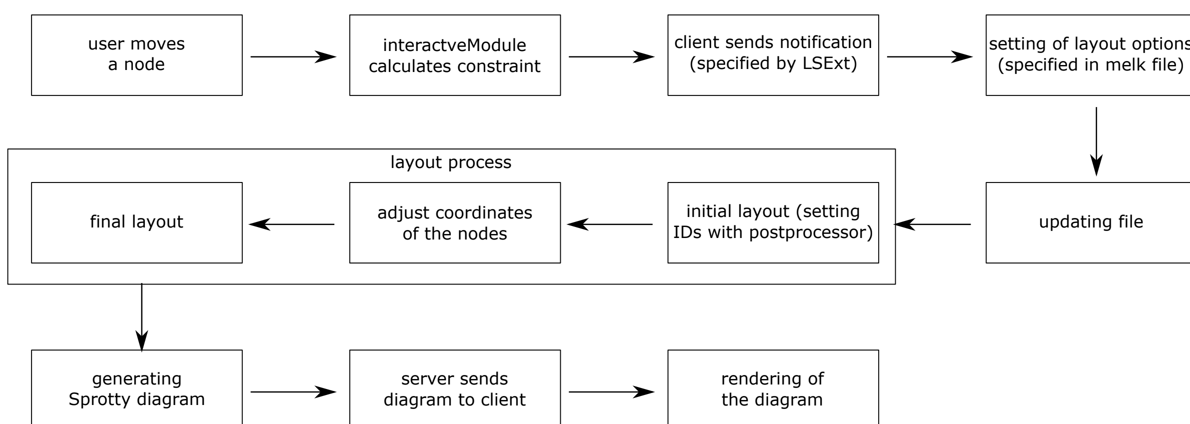


Figure 5.1. Overview of the process of setting a constraint by interacting with the diagram

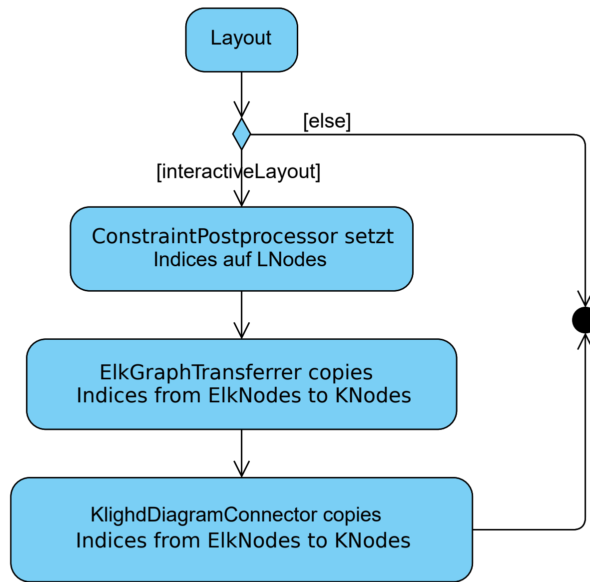


Figure 5.2. Activity diagram for setting IDs by Schönberner [Sch19]

whether the constraints should be considered in the layout.

5.2 Postprocessor

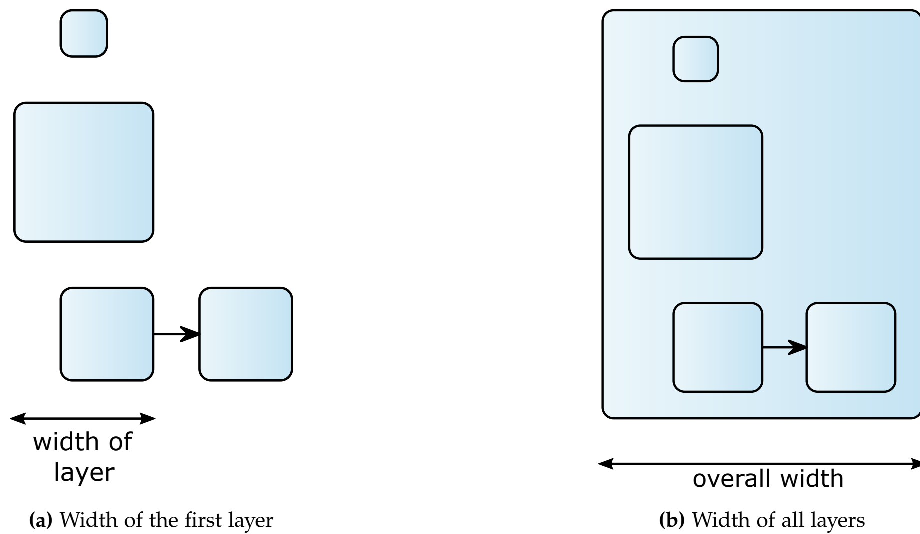
In order to set the `layerID` and `positionID` of the nodes a new *postprocessor* is implemented: the `ConstraintsPostprocessor`. The process to set the IDs on the `KNodes` is shown in Figure 5.2. Since the layout algorithm works with `LNodes`, the processor sets the layout options for the IDs of the `LNodes` to the values the algorithm calculated. In `ElkGraphLayoutTransferrer` the layout of the `LNodes` is applied to the `ElkNodes`. Applying the two new layout options to them is added by getting the option of the `LNode` and setting it for the corresponding `ElkNode`. The values of the IDs of the `ElkNodes` are applied to the `KNodes` in the `KlighdDiagramLayoutConnector`.

Processors are added to the `LayoutProcessorConfiguration` in `GraphConfigurator`, which is why `ConstraintsPostprocessor` is also added to the configuration. It is inserted in the intermediate processing slot after the edge routing phase, because then the layer and position of the nodes do not change anymore.

5.3 Layout with Constraints

After the graph is synthesized, the graph needs to go through the process described in Section 4.2. This process is implemented in `InteractiveLayout` in the `calcLayout` method. If the `interactiveLayout` option is true, initially `separateConnectedComponents` is set to false, the strategies of the layout phases are set to the standard strategies, and the `interactiveLayout` option is set to true for every parent in the graph. Subsequently, the initial layout is done on the graph with the layered layout algorithm.

Afterwards, the coordinates of the nodes are calculated independently for every hierarchical level whereby the ones for the deepest hierarchical level must be calculated first as explained in Section 4.7.



(a) Width of the first layer

(b) Width of all layers

Figure 5.3. Width calculation

Before the coordinates for the nodes in a level can be set, nodes that are supposed to be in the same layer must be grouped in order to set their coordinates identical. First the nodes without a *Layer Constraint* are assigned to layers based on their *Layer ID* they got in the initial layout. Afterwards, the nodes with a *Layer Constraint* are added to the layers based on the value of the constraint, beginning with the node with the lowest value. When a node is assigned to a layer because of a *Layer Constraint*, adjacent nodes that are already in the same layer must be shifted. These nodes get assigned to the layer right of the one they were in before. This lead to the shifting of the adjacent nodes that are in this layer. This continues until no adjacent nodes are in the same layer. Schönberner explains the shifting of nodes in more detail in his thesis [Sch19]. After all nodes are assigned to layers the coordinates for the nodes can be set. For each layer the horizontal coordinate of the nodes is set to the same value which is the sum of the coordinate of the previous layer and the width of it increased by 1. The width of the layer is calculated such that it contains all nodes that should be in it as seen in Figure 5.3a. The coordinate of the first layer is 0. The combined width of all layers, as seen in Figure 5.3b, is calculated wherefore the padding and spacing are needed which are also layout options. The size of the parent is adjusted based on this width. This is required in order to calculate the coordinates in the hierarchical level above properly. The calculation of the graph's width is not accurate for arbitrary graphs. Only adding the padding and spacing to the width of the layers is not sufficient when e.g. port labels are used. However, at the moment only the padding and spacing are considered which can be improved in further work. At last, the vertical coordinates are set by sorting the nodes without a *Position Constraint* based on their *Position ID* and then adding the nodes with a *Position Constraint* at the correct positions in each layer.

Since now the coordinates of each node are set, the strategies of the phases in the layered layout algorithm are set to *(semi) interactive* and a new layout is generated.

5.4 Language Server Extension

In order to define messages that can be send between the client and the server the LS has to be extended. Since both constraints should be settable the extension needs two notifications: One for setting the

Position Constraint and another one for setting the *Layer Constraint* of a node. These notifications must contain the value to which the constraint should be set and the node for which it should be set. This allows the client to send a notification based on the constraint it calculated. Furthermore, a notification for setting both constraints and a notification to refresh the layout are needed. These messages are defined in `ConstraintsCommandExtension` and implemented in `ConstraintsLanguageServerExtension`. The messages are:

setPositionConstraint/setLayerConstraint Expects a `PositionConstraint` respectively a `LayerConstraint` and sets the constraint for the node. Both contain the id of the node, the uri of the current file and the value to which the constraint should be set. Based on the id the corresponding `KNode` can be found and the layout option of it set to the given value.

setStaticConstraint Expects a `StaticConstraint` which contains the id of the node, the uri of the file, and a value for the *Position Constraint* as well as for the *Layer Constraint*. The method sets both constraints for the given node.

refreshLayout Refreshes the layout of the graph for the given uri.

As a reaction to these notifications the model is updated. In the first two methods the corresponding `ElkNode` of the given `KNode` must be found and the layout option of it must be set.

Since the set constraints must be visible in the editor, only updating the model if the server gets a notification, is not sufficient. After the model is updated, the file in the editor that defines the graph must be updated as well. This is done be deleting the old model and saving the new model in the file. This causes a change in the editor, which leads to the generation of a new diagram, because every time the editor changes a new synthesis is started. In this way the constraint is persistent and as a result not vanish when the diagram is reloaded.

5.5 Graph Model

The `KGraphDiagramGenerator` class provides a method to generate the Sprotty diagram based on the laid out `KGraph`. Among other elements `SKNodes` are generated based on the `KNodes`. Since the client works with these `SKNodes` and the layout options of the `KNodes` are needed, these options must be added to the generated `SKNodes`. Therefore, the `SKNode` has an attribute for each introduced layout option which is set in the `generateNode` method according to the option of the `KNode` that is used for the generation. These attributes are: `layerId`, `posId`, `layerCons`, `posCons` and `interactiveLayout`. The `interactiveLayout` option can only be set for parents at the server side which is why the attribute of a `SKNode` is set according to the root of the `KGraph`.

Since an edge should be greyed out if an incident node is moved, the source as well as the target of the edge must be known. That is why in the `generateEdge` method the source and target of the generated `SKEdge` must be set according to the source and target of the `KEdge` that is used for the generation.

5.6 Interactive Module

The moving of nodes is enabled by adding the `moveFeature` provided by Sprotty to the `SKNodes`. The `SKNode` class has a method called `hasFeature` that determines which features are supported. Since the nodes should only be movable if `interactiveLayout` is true, this method only returns true for the `moveFeature` if this is the case.

For the calculation of the constraints that should be set the original position of the node that is moved and the current position the node is moved to must be known. Therefore, a `MouseListener` is needed. Sprotty already provides a `moveModule` with a `MoveMouseListener`. However, just loading the `moveModule` is not sufficient because it does not calculate and send constraints. That is why a new implemented module is used: `interactiveModule`.

In contrast to the `moveModule`, the `interactiveModule` uses the `InteractiveMouseListener` instead of the `MoveMouseListener`. The `InteractiveMouseListener` extends the `MoveMouseListener` and overrides the `mouseDown` as well as the `mouseUp` method. If the target of the `mouseDown` method is a node, the original position of the node is saved in two new attributes of the `SKNode`: `shadowX` and `shadowY`. Additionally, the boolean `isShadow` is set to true. In the `mouseUp` method the constraint that should be set for the target is calculated if the target is a node and has been moved, which can be checked with the `hasDragged` variable of `MoveMouseListener`. Besides these modifications the methods do the same as the ones in the superclass. Since a node should also be movable when the user clicks on the label of it, the method in the superclass is called with the corresponding node as target if the original target is a label.

In order to calculate the constraints several functions are introduced in the `ConstraintUtils` file.

getLayers Expects all nodes that are in a certain hierarchical level and calculates the coordinates of the different layers in this level. To save these coordinates the class `Layer` is introduced with the attributes `topY`, `botY`, `leftX`, `rightX` and `mid`. Accordingly, an array of layers can be returned.

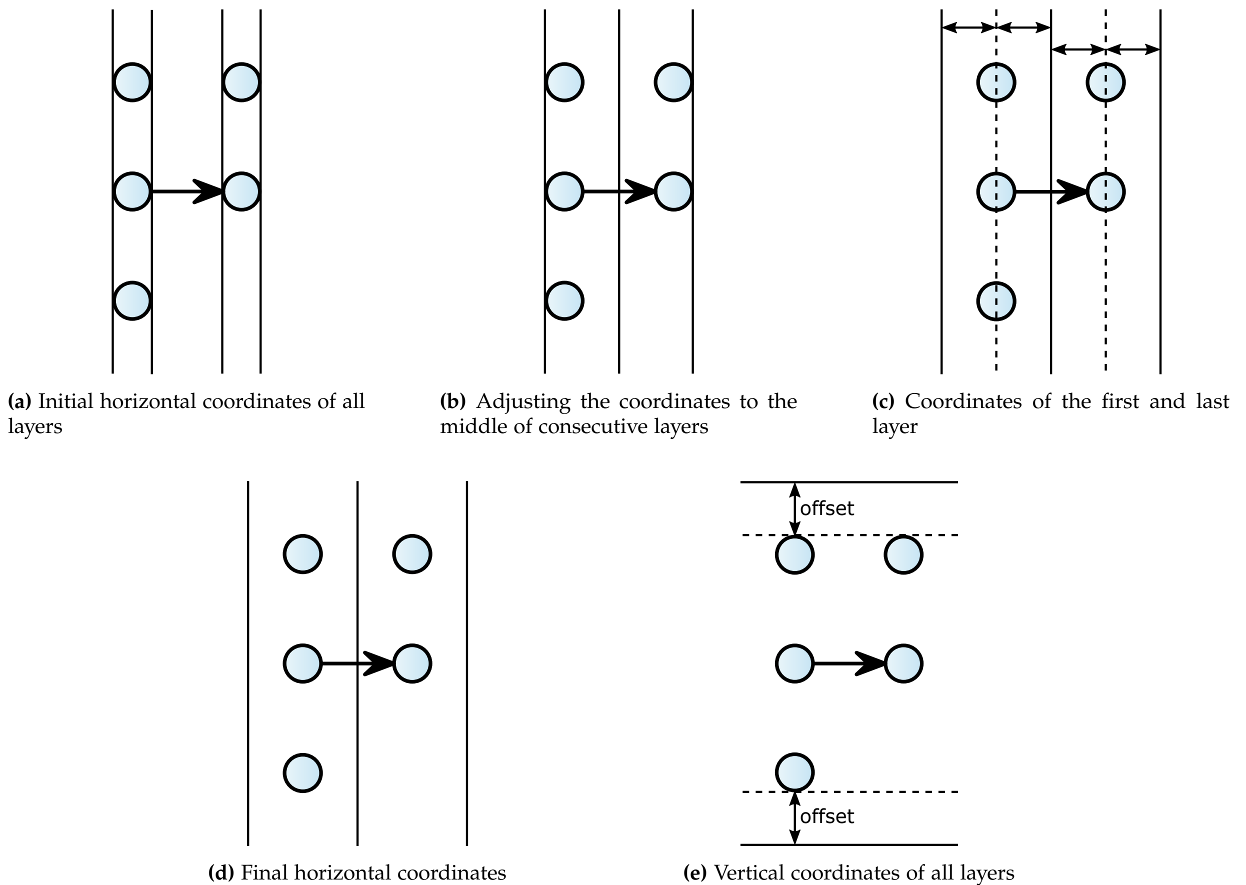
The calculation of these coordinates is done in the following way (visualized in Figure 5.4):

1. `leftX` is set to the vertical coordinate of the leftmost node in the layer and `rightX` to the rightmost vertical coordinate the shape of a node in the layer is at. Thereby, the nodes that are in one layer are filtered based on their `layerID`.
2. For every layer `leftX` and `rightX` are updated such that for two consecutive layers the `rightX` of the first and the `leftX` of the second layer is set to the middle of the layers based on the previous values of the attributes.
3. `leftX` of the first layer is set such that `mid` is still the middle of the layer.
4. `rightX` of the last layer is set, such that `mid` is still the middle of the layer.
5. The vertical coordinates are set identically for every layer. `topY` is the coordinate of the uppermost node in the graph reduced by an offset while `botY` is the greatest coordinate the shape of a node is at in the graph increased by an offset.

getNodesOfLayer Given a layer and the nodes of the graph the nodes that are in this layer are returned. To determine whether a node is in the given layer the `layerID` of the node is checked.

getLayerOfNode Calculates the layer of the given node by using the given layers and the vertical coordinate of the middle of the node. If it is within the horizontal coordinates of a layer, the id of this layer is returned with the exception of the first layer, which is also the layer the node is in if the node is left of its `leftX`. Another case is that the node can be right of `rightX` of the last layer. In this case the id of the last layer increased by one is returned provided that the node is not originally the last one in the last layer. Otherwise the id of the last layer is returned.

getPosInLayer Determines the position the given node has in the layer by comparing the vertical coordinate of the node with the vertical coordinates of the other nodes that are in the same layer.



(a) Initial horizontal coordinates of all layers

(b) Adjusting the coordinates to the middle of consecutive layers

(c) Coordinates of the first and last layer

(d) Final horizontal coordinates

(e) Vertical coordinates of all layers

Figure 5.4. Coordinates of the layers visualized by lines

shouldOnlyLCBeSet States whether only a *Layer Constraint* should be set by comparing the horizontal coordinate of the given node to the coordinates of the layers. If the node is above or below the layer, only the *Layer Constraint* should be set.

With these functions the constraint of the moved node can than be determined as described in the following procedure:

1. Calculating the layers of the graph with `getLayers`.
2. Determine the layer of the moved node by calling `getLayerOfNode` with the calculated layers and the target of the `mouseUp` method.
3. Filter the nodes that are in the same layer by calling `getNodesOfLayer` with the layer of the moved node.
4. Calling `getPosInLayer` with the previous gotten values.

In order to call the first three methods additionally all nodes in the same hierarchical level must be known. These nodes are determined by filtering the children of the parent of the target resulting in a

list of KNodes that are in the same level. After the layer and the position of the target are calculated a constraint is send to the server.

As mentioned in Section 4.4 the send constraints depend on the difference between the original and the new position of the node. The original position is determined by the positionID and layerID while the new position is calculated by calling getPosInLayer as described above. Whether only a *Layer Constraint* should be send is determined by calling the shouldOnlyLCBeSet method. In order to be able to send different constraints classes are introduced for each constraint type: LayerConstraint, PositionConstraint, and StaticConstraint, which is used if both constraints should be set. Based on the constraint that should be set the corresponding notification introduced in Section 5.4 is send to the server. If no constraint should be set, the refreshLayout notification is send.

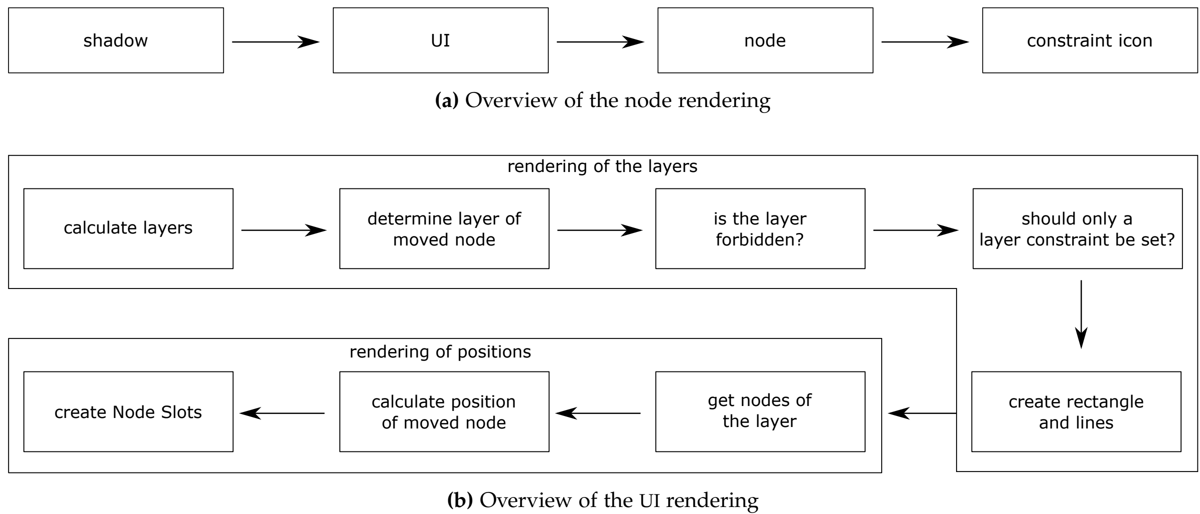
5.7 Interactive View

The shown diagram in KEITH is generated by render methods that are called on the elements of the graph. In order to adjust the diagram these methods must be modified. SKNodes are rendered by the render method defined in KNodeView which is called for each node in the graph. An overview of the rendering of a node is shown in Figure 5.5a. In addition to the rendering of the graph elements, the UI presented in Section 4.5 must be rendered which is done in the render method for the nodes. If the render method is called for the parent of the moved node, the UI is rendered too as long as interactiveLayout is true and a node is moved. An overview of rendering the UI is shown in Figure 5.5b.

In order to calculate the coordinates for the visualization of the layers, which can be seen in Figure 4.4a, the methods described in Section 5.6 are used. With getLayerOfNode the current layer of the moved node is determined. For each layer getLayers calculates, an SVG object is created: if the layer is the current one, a rectangle is created otherwise a vertical line. The coordinates of these objects are the same as the ones of the layers. The new empty layer on the right-hand side of the last layer starts on the coordinate the last layer ends and has the same width as the last layer. Whether the shown rectangle should be red and hence indicating a forbidden layer is determined by calling isLayerForbidden. So far a layer is only forbidden if an adjacent node of the moved node is already in the layer and has a *Layer Constraint*. As explained in Section 4.5.1 the colour of the rectangle also depends on the constraints that will be set. The method shouldOnlyLCBeSet is used to determine whether only a *Layer Constraint* will be set.

For the current layer the coordinates for the *Node Slots* indicating the available positions as seen in Figure 4.5 must be calculated. The current position of the moved node and the other nodes in the same layer must be known wherefore getPosInLayer and getNodesOfLayer mentioned in Section 5.6 are used. The vertical coordinates of the *Node Slots* are determined by the coordinates of the enclosing nodes and the horizontal coordinate of the *Node Slots* is the middle of the layer. If a *Node Slot* is drawn above the first node respectively below the last one, the *Node Slot* is placed in the middle of the first node and the top of the layer respectively the last node and the bottom of the layer. Whether the *Node Slot* is filled is determined by comparing the calculated position of the moved node with the position the *Node Slot* is representing. The colour of the *Node Slots* is red if the layer is forbidden, which is determined by calling isLayerForbidden. In order to determine whether the *Node Slots* are drawn at all shouldOnlyLCBeSet is used.

The shadow of the moved node, which can be seen in Figure 4.7b, is rendered if the isShadow attribute of the node, which is currently rendered, is true. In order to do this the node is rendered twice in its render method: Once with isShadow set to false, which results in the standard rendering and once with isShadow set to true, which results in the shadow. If the attribute is true, the bounds of



(a) Overview of the node rendering

(b) Overview of the UI rendering

Figure 5.5. Overview of the rendering

the node calculated by `findBoundsAndTransformationData` are equal to the ones the node had before it was moved by using `shadowX` and `shadowY`. The colour of the node is determined by `getSvgColorStyles`, which is why in this method the colour is calculated based on the `isShadow` attribute.

In order to grey out incident edges of the moved node, `KEdge` gets the attribute `moved` which is set to true in the rendering method of the edge before the edge is rendered, if an incident node is moved. In the rendering of the edge, the `getSvgColorStyles` method is used to determine the colour. That is why in this method the `moved` attribute is checked and the colour set to grey if it is true.

Whether a graph element is hidden because it is not in the same hierarchical level as the moved node, as seen in Figure 4.9, is determined by calling `isChildSelected`. It determines whether a child of the given `KNode` is currently selected. For the edges this function is called with the parent of the edge and they are only rendered if one of the children is selected. A node is only rendered if one of its children or another child of its parent is selected. Since the parent of the moved node should be large enough to contain the UI, the `KNode` has the attributes `hierWidth` and `hierHeight` which contain the size the node must have. The default value of these attributes is 0 and is reset at the beginning of the node rendering. After the rendering of the layer visualization `hierWidth` is set such that all layers are inside the node and `hierHeight` is the same as the original height. In `findBoundsAndTransformationData` this new size is used to determine the bounds if `hierHeight` is not the default value.

5.8 Diagram Option

The diagram option to show set constraints introduced in Section 4.6 is only needed on the client side, which is why it is not added to the synthesis options already existing on the server side. An option that is only needed on the client side is a `RenderOption` which includes among others a `name`, an `initialValue`, and a `currentValue`. The `RenderOptions` keeps track of all theses options and is injected in the `KNodeView` in order to adjust the rendering based on the options.

If the user changes a `RenderOption` in the diagram options, the `currentValue` of the option is updated in `DiagramOptionsViewContribution` just like the synthesis options except that not all option types are supported yet. The option type can be `check`, `choice`, `range`, or `separator` but only the first

one is currently implemented. Unlike the synthesis option the `RenderOption` is not sent to the server in order to update the value. Instead it is updated in the injected `RenderOptions` with the update method it provides. Subsequently, a notification is send to the server in order to update the diagram.

An option is defined in the `RenderOptions` class. At the moment only *Show Constraint* is declared with an initial and current value of false. If this option and `interactiveLayout` is true, icons are shown indicating the constraints that are set, which is done in `KNodeView`. When a node is rendered, SVG objects are created based on the values of the constraints of it as explained in Section 4.6.

Chapter 6

Evaluation

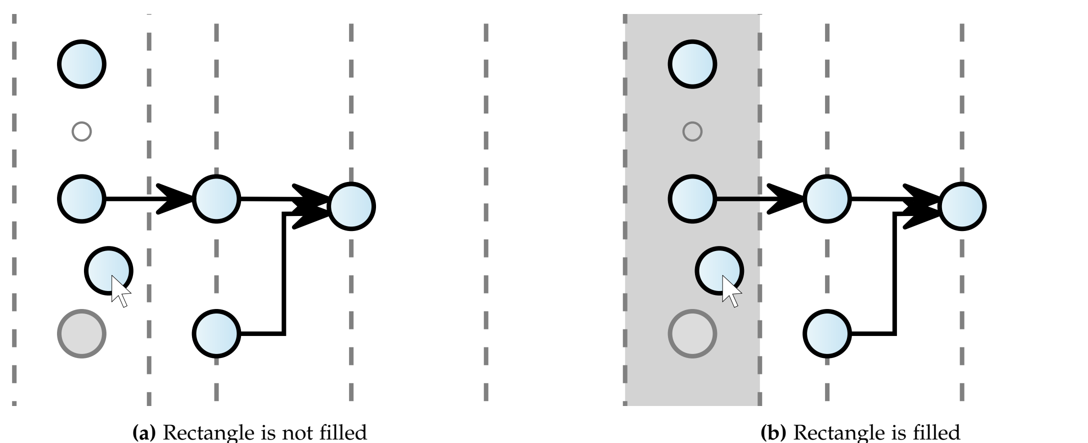
The user interaction with the graph should be as intuitive as possible. In order to determine the usability of the presented UI, members of the Real-Time and Embedded Systems Group of the Kiel university offered feedback. Additionally, informal interviews were conducted in order to obtain insights in the usability for people with little to no previous knowledge.

6.1 Expert Feedback

During the development of the user interaction intermediate results were shown to members of the Real-Time and Embedded Systems Group. One aspect that was criticized is that the visualization of the current layer was only a dashed rectangle. When zooming into the graph only dashed lines were visible, as seen in Figure 6.1a, confusing the user which lines visualize the current layer. This is why the rectangle is now highlighted in grey as explained in Section 4.5.1.

At first, it was only allowed to set a *Position Constraint* or both constraints together. Only setting a *Layer Constraint* was not possible. However, the user should not be forced to write the constraint textual when only a *Layer Constraint* is wanted. As a consequence, setting a *Layer Constraint* alone is realized as presented in Section 4.4.

The visualization of constraints through icons was always enabled. It could not be turned off, which was criticized since the visual clutter increases with more set constraints that are visualized. In order to prevent this the option described in Section 4.6 is implemented.



(a) Rectangle is not filled **(b)** Rectangle is filled

Figure 6.1. Zooming into a graph

6.2 User Study

The informal interviews consists of two experiments. These experiments are not representative since only a few people participated. Nevertheless, they give a small insight in the usability and which features must be further improved and evaluated. In both experiments the same ten persons participated. Each one of them knows graphs in general and the layout of the graph is explained to them shortly.

6.2.1 Experiment 1

The aim of the first experiment is to determine whether the basic user interaction (moving nodes to certain positions) with the presented UI is intuitive. This is done by showing the participants the graph seen in Figure 6.2a. Their task was to group the A's in one layer, the B's in one layer, and the C's in one layer. An example of the fulfilled task can be seen in Figure 6.2b. Subsequently, the nodes of each layer should be sorted such that the single letters are at the top and the triple letters are at the bottom as shown in Figure 6.2c. After each task the participants are asked for feedback to the experienced user interaction.

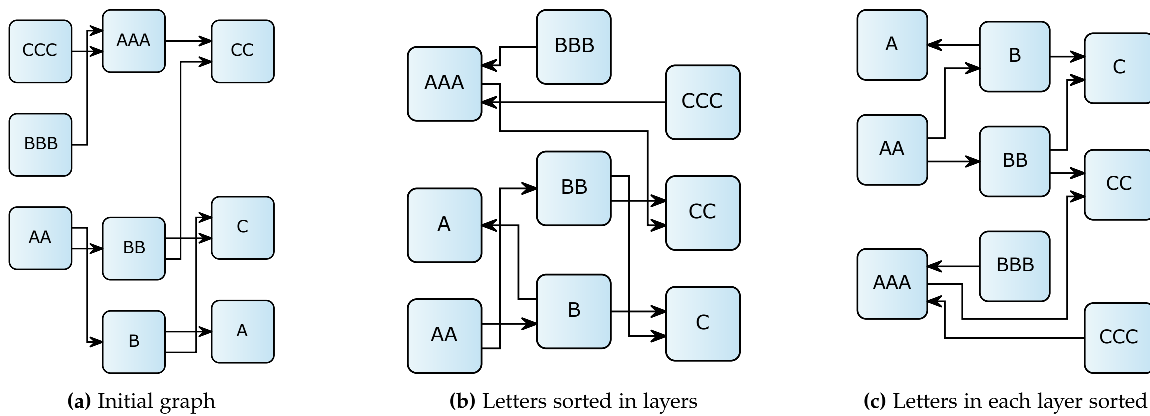
All participants agreed that it is unintuitive to not be able to move nodes by clicking on the label of them. The general movement of nodes and where they are placed by releasing at a certain position is intuitive and understandable for all participants. Three participants mentioned the *Node Slots*, representing the available node positions, as helpful but two participants said that they could be a little bigger or flashier. The layer visualization and the shadow is mentioned by one person as helpful. A red layer, indicating that the layer is forbidden, is seen by two participants who appreciate it, but one of them desires more explanation why the layer is forbidden. Three participants get confused by the large changes of the graph when only one node is moved. Overall the participants agreed that the general movement of nodes is user-friendly.

Although only few people participate, the feedback of them can be used to improve the user interaction. Intuitively it is clicked in the middle of a node when wanting to move it which results in clicking on the label. Since it is irritating when a node can not be moved by clicking there, this problem is solved as explained in Section 5.6. The size of the *Node Slots* is limited by the spacing between two nodes but can be improved by adjusting it based on the node sizes in order to be better seen. Visualizing the forbidden layer can be optimized by showing a little text box, which explains why this layer is forbidden. Another option is to highlight the reason for the conflict, in this case the edge of the node, instead of the whole layer. The reason of the large graph changes was that the layout of the graph changed when the diagram is refreshed by clicking on a node. That happened because the initial layout calculated in the layout process was not always the same. This problem has been solved after the evaluation.

6.2.2 Experiment 2

In the second experiment the interaction with hierarchical graphs is observed by using the graph seen in Figure 6.3a. The task is to sort the letters in the parent nodes alphabetically, whereby the user decide whether this is done vertically or horizontally. An example of the fulfilled task can be seen in Figure 6.3b. Subsequently, the node n3 should be in the middle of the graph as seen in Figure 6.3c. After the task, the participants are asked for their opinion regarding hiding the graph elements that are not in the same hierarchical level as the moved node and whether it is confusing.

Four participants did not notice that other graph elements are hidden. After they knew it, one of them concluded that nodes can not be moved to another hierarchical level. Two participants were

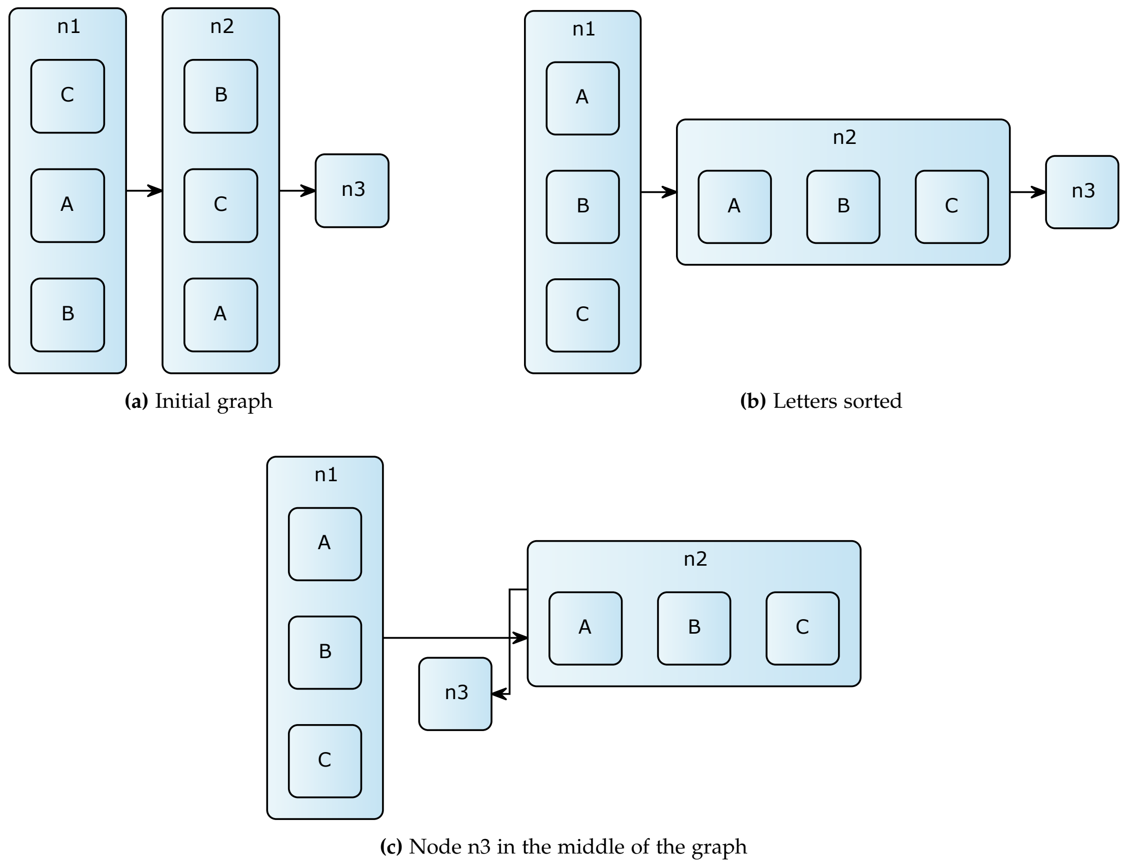


(a) Initial graph **(b)** Letters sorted in layers **(c)** Letters in each layer sorted

Figure 6.2. Experiment 1

confused and suggest that the graph elements should be greyed out instead of vanishing. On the contrary four participants are content with the presented solution. One of them states that the context focus is intuitive and prevent the attempt to change the hierarchical level of nodes. After moving the n3 node to the middle of the graph six participants express their desire to create a layer between two existing layers by moving the node between them. Two participants state that the possibility to create a layer to the left of the first one would be helpful.

These results of the experiment show that the UI in hierarchical graphs must be further evaluated. While some people are irritated by hiding graph elements, other people are consent with it or not notice it at all. When deciding to not hide other graph elements they could be greyed out to indicate that they are currently not relevant. However, then it could happen that users attempt to change the hierarchical level of the moved node. Regarding the creation of new layers, the concept of create them left of the first layer or between two consecutive layers is already explained in Schönberners bachelor's thesis [Sch19]. However, it is not implemented yet.



(a) Initial graph

(b) Letters sorted

(c) Node n3 in the middle of the graph

Figure 6.3. Experiment 2

Chapter 7

Conclusion

This thesis, together with the thesis of Schönberner [Sch19], shows a way to realize intentional layout in Sprotty diagrams. The focus of this thesis is the user interaction with the diagram generated by Sprotty. In this chapter the introduced constraints and their consideration in the layout process as well as the defined user interaction are summarized. Additional features that can be implemented and occurring problems by transferring the presented concept to SCCharts are described as future work.

7.1 Summary

The user can change the displayed diagram by adding constraints to nodes in the textual definition of the graph. With these constraints absolute positions of the nodes can be set. A *Layer Constraint* determines the layer of the node and a *Position Constraint* the position in the layer. The layout process is adjusted such that the constraints are considered. That is achieved by using the layered layout algorithm for an initial layout, setting the coordinates of the nodes according to their constraints, and generating the final layout by using the layout algorithm and *(semi) interactive* strategies. This layout process is done for every hierarchical level individually, whereby the deepest level is laid out first in order to ensure that the nodes in higher levels get the correct coordinates.

Setting constraints is also possible by interacting with the diagram. Based on the position a node is moved to, suitable constraints are calculated and set. In order to set a constraint through interaction the LS is extended. The interaction with the diagram happens on the client side, which is why the client must send the desired constraint to the server, who can set the constraint in the model. As a result the changes are persistent. An UI is implemented to visualize which constraints are set when a node is released on a certain position. The UI consists of the layers in the graph, available positions in the current layer, and a shadow indicating the original position of the moved node. In hierarchical graphs all graph elements that are not in the same hierarchical level as the moved node are hidden. Set constraints are visualized through icons, which can be enabled by a diagram option.

Evaluating the implemented user interaction in informal interviews shows that the basic concept is user-friendly and delivers ideas for additional features such as layer creation between two consecutive layers. However, further evaluation is needed for the UI in hierarchical graphs.

7.2 Future Work

In order to validate the results of the informal interviews an academic survey can be conducted with more participants resulting in representative outcomes. Especially the UI in hierarchical graphs must be further evaluated. The results of the evaluation described in Section 6.2 consist of additional features such as the creation of layers between consecutive layers, which can be implemented in further work.

Other aspects that need to be evaluated are the movement of taller nodes and the usability of the user interaction in big graphs. The problem of big graphs is that not all nodes are displayed in the

visible area, provided it is not zoomed out which would lead to tiny nodes. Especially setting only the *Layer Constraint* can be inconvenient when a node must be moved to the top or bottom of the graph.

The coordinate calculation in hierarchical graphs must be improved. At the moment the width calculation of parents is not always correct. Only the padding and spacing is respected but e.g. not the width of port labels. A way to avoid the handling of hierarchical graphs is to implement new strategies for the layered layout algorithm, which lay out the graph similar to the *(semi) interactive* strategies but uses layer and position IDs instead of the coordinates of the nodes.

An additional feature that can be implemented is the possibility to delete constraints by interacting with the diagram. At the moment the user can delete constraints only in the editor. However, a reevaluation of constraints is not possible when a constraint is deleted in the editor. Since the constraints can be set interactively, it would be convenient to allow interactive deletion. Moreover, switching between editing modes interferes with the workflow. An option to allow interactive deletion is to open a menu when the user right-clicks on a node or on an icon visualizing the set constraints. In the menu the different constraints can be shown with the option to delete them individually or altogether.

The introduced constraints can be extended as well. We focused on implementing absolute constraints but as mentioned in Section 4.1 relative constraints are possible, too. Setting these constraints can be realized by moving a node to another one to which the moved node should have a relation to. This can lead to opening a menu in which the user can choose which relation should be set for the two nodes. In order to support these further constraints properly the layout process must be adjusted, too. Additionally, further cases in reevaluating constraints covered by Schönberner [Sch19] must be observed e.g. circles can arise by setting several relative constraints.

Elkt is so far the only language that supports our constraints. In further work the presented concept can be applied to more languages such as SCCharts. When applying the approach to SCCharts some aspects must be considered. The direction of the layers can be different at each hierarchical level and is not static. Based on the direction the UI and the constraint calculation must be adjusted. Additionally, it must be specified, which graph elements should be movable. Allowing the `KNodes` to be movable can be too generic, leading to the movement of objects that should not be movable at all. In order to place labels of edges further layers are used. These layers must be filtered in order to prevent the movement of nodes to such layers.

Bibliography

[BET+94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry* 4.5 (Oct. 1994), pp. 235–282. ISSN: 09257721. DOI: 10.1016/0925-7721(94)00014-X. URL: https://linkinghub.elsevier.com/retrieve/pii/092577219400014X (visited on 08/25/2019).

[BP90] Karl-Friedrich Böhringer and Frances Newbery Paulisch. "Using constraints to achieve stability in automatic graph layout algorithms". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '90. event-place: Seattle, Washington, USA. New York, NY, USA: ACM, 1990, pp. 43–51. ISBN: 978-0-201-50932-8. DOI: 10.1145/97243.97250. URL: http://doi.acm.org/10.1145/97243.97250 (visited on 08/26/2019).

[DKM06] T. Dwyer, Y. Koren, and K. Marriott. "IPSep-CoLa: an incremental procedure for separation constraint layout of graphs". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 821–828. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.156.

[Dom18] Sören Domrös. "Moving model driven engineering from eclipse to web technologies". 2018.

[Han18] Reinhard von Hanxleden. "Automatic graph drawing". Published: University Lecture. 2018.

[Jan17] Klaus Jansen. "Algorithmen und datenstrukturen". Published: University Lecture. 2017.

[KPE16] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. "The IDE portability problem and its solution in monto". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. event-place: Amsterdam, Netherlands. New York, NY, USA: ACM, 2016, pp. 152–162. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997368. URL: http://doi.acm.org/10.1145/2997364.2997368 (visited on 08/22/2019).

[LLY06] Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. "Mental map preserving graph drawing using simulated annealing". In: Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation - Volume 60. Australian Computer Society, Inc., Jan. 1, 2006, pp. 179–188. ISBN: 978-1-920682-41-5. URL: http://dl.acm.org/citation.cfm?id=1151903.1151930 (visited on 08/25/2019).

[Mas92] T. Masui. "Graphic object layout with interactive genetic algorithms". In: *Proceedings IEEE Workshop on Visual Languages*. IEEE Workshop on Visual Languages. Seattle, WA, USA: IEEE Comput. Soc. Press, 1992, pp. 74–80. ISBN: 978-0-8186-3090-3. DOI: 10.1109/WVL.1992.275781. URL: http://ieeexplore.ieee.org/document/275781/ (visited on 08/27/2019).

[MJ09] M.J. McGuffin and I. Jurisica. "Interaction techniques for selecting and manipulating subgraphs in network visualizations". In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 937–944. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.151. URL: http://ieeexplore.ieee.org/document/5290697/ (visited on 08/28/2019).

[NE01] Hugo A. D. Nascimento and Peter Eades. "User hints for directed graph drawing". In: *Graph Drawing*. International Symposium on Graph Drawing. Springer, Berlin, Heidelberg, Sept. 23, 2001, pp. 205–219. DOI: 10.1007/3-540-45848-4_17. URL: https://link.springer.com/chapter/10.1007/3-540-45848-4_17 (visited on 08/27/2019).

[PT90] Frances Newbery Paulisch and Walter F. Tichy. “Edge: an extendible graph editor”. In: *Software: Practice and Experience* 20 (S1 June 1, 1990), S63–S88. ISSN: 0038-0644. DOI: 10.1002/spe.4380201307. URL: https://vpn.uni-kiel.de/proxy/407e3244/https/onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380201307 (visited on 08/27/2019).

[Ren18] Niklas Rentz. “Moving transient views from eclipse to web technologies”. 2018.

[RMS97] Kathy Ryall, Joe Marks, and Stuart Shieber. “An interactive constraint-based system for drawing graphs”. In: *Proceedings of the 10th Annual Symposium on User Interface Software and Technology (UIST)* (1997). DOI: 10.1145/263407.263521. URL: https://dash.harvard.edu/handle/1/2258867 (visited on 08/27/2019).

[Sch19] Connor Schönberner. “Intentional layout in sprotty diagrams: reevaluation gesetzter constraints”. unpublished bachelor thesis. 2019.

[SF08] Andre Suslik Spritzer and Carla M. D. S. Freitas. “A physics-based approach for interactive manipulation of graph visualizations”. In: Proceedings of the working conference on Advanced visual interfaces. ACM, May 28, 2008, pp. 271–278. ISBN: 978-1-60558-141-5. DOI: 10.1145/1385569.1385613. URL: http://dl.acm.org/citation.cfm?id=1385569.1385613 (visited on 08/27/2019).

[SSH13] Christian Schneider, Miro Sponemann, and Reinhard von Hanxleden. “Just model! — putting automatic synthesis of node-link-diagrams into practice”. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). San Jose, CA, USA: IEEE, Sept. 2013, pp. 75–82. ISBN: 978-1-4799-0369-6. DOI: 10.1109/VLHCC.2013.6645246. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6645246 (visited on 08/25/2019).

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. “Methods for visual understanding of hierarchical system structures”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (Feb. 1981), pp. 109–125. ISSN: 0018-9472. DOI: 10.1109/TSMC.1981.4308636.

Abbreviations

EDGE Extendible Directed Graph Editor

ELK Eclipse Layout Kernel

GALAPAGOS Genetic Algorithm And Presentation-Assisted Graphic Object layout System

GLIDE Graph Layout Interactive Diagram Editor

GUI Graphical User Interface

IDE Integrated Development Environment

IPSEP-COLA Incremental Procedure for Separation Constraint Layout of graphs

JSON JavaScript Object Notation

JSON-RPC JSON Remote Procedure Call

KEITH Kiel Environment Integrated in Theia

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KLighD KIELER Lightweight Diagrams

LS Language Server

LSP Language Server Protocol

MDE Model-Driven Engineering

NAViGaTOR Network Analysis, Visualization, & Graphing TORonto

SCCharts Sequential Constructive Statecharts

SVG Scalable Vector Graphic

UI User Interface

VOF Visual Organization Feature