Intentional Layout in Sprotty Diagrams: Reevaluation gesetzter Constraints

Connor Schönberner

Bachelorarbeit
27. September 2019

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
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,

Zusammenfassung

Die Unterstützung von automatischem Layout für visuelle Programmiersprachen in IDEs für Model-Driven Engineering (MDE) befreit einen von der zeitintensiven Arbeit, ein Layout für die Programme manuell erstellen und bei Änderungen aktualisieren zu müssen.

Zumeist erstellen Layoutalgorithmen ein Layout nur auf Basis des Modells eines Graphen. Semantische Zusammenhänge lassen sie außer Acht. Diese lassen sich durch manuelle Nachbearbeitung einbeziehen. Allerdings sehen Layoutalgorithmen keine Nachbearbeitung des Layouts durch Nutzende vor. Manuelle Änderungen gehen also in Folgelayouts wieder verloren.

Intentional Layout kombiniert automatisches Layout durch einen Layoutalgorithmus mit optionaler interaktiver Nachbearbeitung durch die Nutzenden, deren Anpassungen bei Folgelayouts erhalten bleiben. Diese Arbeit präsentiert Intentional Layout für einen ebenenbasierten Layoutalgorithmus in der IDE KEITH und gelangt dabei zu einer nutzbaren Implementierung. Die Interaktionen auf dem Diagramm führen zur Erzeugung sogenannter *Constraints*, welche den Typ sowie die spezifische Ausprägung der Anpassung im Modell des Graphen persistieren. Der Layoutalgorithmus erfüllt die Constraints in seinem Layout-Prozess, so dass die Anpassung nicht bei Folgelayouts verloren geht. Der Fokus liegt auf sogenannten absoluten Constraints für absolute Positionierung von Knoten.

Darüber hinaus untersucht diese Arbeit, ob und wie die bestehenden Constraints auf dem Modell beim Hinzufügen oder Löschen von Constraints angepasst werden müssen, um die *Mental Map* der Nutzenden zu erhalten. Diese Anpassungen nennt diese Arbeit *Reevaluation*. Es gelingt, Reevaluationen für das Hinzufügen und das Löschen von absoluten Constraints zu erarbeiten. Außerdem gelingt es dieser Arbeit, herauszuarbeiten, welche absoluten Constraints invalide sein können und wie mit ihnen umzugehen ist. Zudem werden Erweiterungen wie das Hinzufügen von Layern ausgearbeitet.

Danksagungen

Zunächst möchte ich Prof. Dr. Reinhard von Hanxleden für das Thema der Arbeit und den Arbeitsplatz an seinem Lehrstuhl danken. Zudem danke ich Sören Domrös für seine Betreuung unseres Projekts und meiner Arbeit. Darüber hinaus danke ich Niklas Rentz und Christoph Daniel Schulze für ihre Unterstützung beim Arbeiten mit KEITH, Eclipse Layout Kernel (ELK) und Schreiben meiner Arbeit und Alexander Schulz-Rosengarten, Lena Grimm und Steven Smyth für das Beantworten von Fragen. Jette möchte ich für unsere tolle Zusammenarbeit danken. Ihr, meinen anderen Freunden und meiner Familie danke ich für ihre moralische Unterstützung. Schließlich möchte ich „der Couch“ dafür danken, dass sie uns durch dieses Semester hindurch ertragen hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Graphen sind vielseitige Strukturen aus Knoten und Kanten: Mit ihnen lassen sich Probleme modellieren, Beziehungen zwischen Objekten ausdrücken und sie kommen bei der visuellen Programmierung zum Einsatz.

Die theoretische Modellierung mit Graphen kann rein textuell erfolgen, wie dies beispielsweise beim Beweisen über Graphen der Fall ist. Für andere Anwendungszwecke wie grafischer Programmierung bietet sich hingegen eine visuelle Repräsentation von Graphen an. Ein *drawing* oder ein *Layout* ist eine Abbildung, die die Positionen der Knoten und den Verlauf, die Routen, von Kanten festlegt, und eine visuelle Repräsentation eines Graphen gestaltet.

Während sich in einem textuellen Format ein Graph mit weniger Aufwand spezifizieren [SSH13] und versionieren lässt, ist ein grafisches Diagramm besser lesbar [SSH14]. Es ist klarer und übersichtlicher und Eigenschaften wie Zusammenhang und Kantenbeziehungen aus ihm klarer ersichtlich als in der textuellen Ausprägung [SSH13].

Um die Vorteile beider Möglichkeiten zu vereinen, bietet sich zur Ergänzung der graphischen View eine Modellrepräsentation in Form einer Domain Specific Language (DSL) an, mit der Modelle in Dateien persistiert werden. Auf dieses Konzept setzt das Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)-Projekt, eine IDE für MDE. KIELER verwendet eine textuelle Repräsentation im `elkt`-Format, das Teil des ELK ist, als Modell für Graphen und Node-Link-Diagramme als View[1].

Domrös [Dom18] und Rentz [Ren18] migrierten KIELER in den Kontext von Web-Technologien: Das Ergebnis ist die Theia-Applikation[2] KEITH. KEITH[3] verwendet das Sprotty-Framework zur Darstellung von Diagrammen.

Zwar sind Diagramme in der Regel lesbarer als textuelle Darstellungen, aber ihre Lesbarkeit steht im direkten Zusammenhang mit dem Layout, auf dem sie aufbauen [SSH14]. Prinzipiell gibt es für die *Layout-Erstellung* zu einem Graphen zwei Möglichkeiten: Einerseits lässt sich ein Solches manuell in einer Software über das Festlegen jeder Knotenposition und jedes Kantenverlaufs erstellen, andererseits kann ein Layoutalgorithmus ein Layout generieren. Der manuelle Ansatz bietet unbeschränkte Freiheit bei der Erstellung eines Diagramms. Allerdings konsumiert der manuelle Erstellungsprozess viel Zeit, erfordert geeignete Tools und das erstellte Layout wird schnell inkorrekt, wenn sich Details im Design mit der Zeit ändern. Solche Änderungen wieder einzupflegen, das Layout also lediglich aktuell zu halten, erfordert viel Aufwand [SSH13]. Schätzungsweise benötigen Nutzende insgesamt, während sie das Modell erstellen und anpassen, sogar 25% ihrer Zeit, um das Layout manuell anzupassen [Kla12].

Dahingegen nimmt ein Layoutalgorithmus, mit deren Entwicklung sich das Feld des Graph Drawing beschäftigt, die Layout-Erstellung und Layout-Anpassung den Nutzenden ab, wodurch die ansonsten dafür aufgewendete Zeit für Verbesserungen am Modell frei wird [SSH14]. Die Designer von Layoutalgorithmen entwerfen diese derart, dass sie bestimmte Ästhetikkriterien optimieren, die die erzeugten Layouts einhalten sollen [Pur97].

[1] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview`

[2] `https://github.com/theia-ide/theia`

[3] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KEITH`

Wie sich zudem gezeigt hat, erhöht sich die Produktivität der Programmierenden beim Verwenden von graphischen Programmiersprachen, wenn Layoutalgorithmen die Layouts der Programme erstellen [FH10].

KIELER erstellt die Layouts für seine View über Layoutalgorithmen des ELK, der Teil des Projekts ist. Diese Arbeit betrachtet ausschließlich den Layered Layoutalgorithmus [SSH14] von ELK, der eine Implementierung des *Sugiyama-Algorithmus* [STT81] ist.

Es ist nicht zwingend gegeben, dass das Layout, das der Layered Layoutalgorithmus standardmäßig erzeugt, der Mental Map der Nutzenden entspricht. Bei ihr handelt es sich um die kognitiv-visuelle Repräsentation des Graphen, die im Verstand der Nutzenden durch die Beobachtung des Layouts entsteht [DG02]. Dies kann der Fall sein, wenn dem Graphen eine Semantik zugrunde liegt, die eine bestimmte Anordnung nahelegt. Solche semantischen Feinheiten bezieht der Layered Layoutalgorithmus nicht mit ein, wodurch die Anordnung nicht so erfolgt, wie die Nutzenden es erwarten.

1.1. Problemstellung

Bestehende Optionen erlauben es, den Stil und gewisse Eigenschaften des Layouts anzupassen. So lassen sich beispielsweise die Größe der Knoten, die Form der Knoten und Layout-Ausrichtung anpassen. Ein Beispiel hierfür ist eine Option, die ausdrückt, ob das Layout von links nach rechts oder von oben nach unten angelegt werden soll.

Allerdings sind die Optionen des Algorithmus nicht darauf ausgelegt, die Eigenschaften von einzelnen Knoten des Layouts anzupassen. Es gibt nur die Möglichkeit, für einen Knoten zu spezifizieren, dass er im ersten oder im letzten Layer des Layouts sein soll. Die Optionen erlauben es also nicht, spezifische Positionen für Knoten festzulegen.

Ein Lösungsansatz ist es, das Layout zuerst mit dem Layered Layoutalgorithmus zu erstellen und es danach manuell via Drag&Drop anzupassen. Hierbei obliegt es den Nachbearbeitenden, ob die Anpassungen sich optisch in das Layout einfügen. Was einschließt, dass die Knoten den festgelegten Abstand zueinander wahren, der Anordnungsstrategie des Layouts folgen und die Kantenverläufe zum restlichen Layout passen. Jeder erneute Durchlauf des Algorithmus erfordert, den manuellen Layout-Anpassungen von Vorne durchzuführen. Schließlich bezieht der Algorithmus die Anpassungen der Nutzenden beim neuen Layout-Prozess nicht mit ein, da sie nicht auf das Modell übertragen werden. Damit sieht sich die manuelle Layout-Anpassung mit ähnlichen Problemen konfrontiert wie die manuelle Layout-Erstellung.

Von Intentional Layout ist die Rede, wenn die Nutzenden beabsichtigte Layout-Anpassungen an Koordinaten von Knoten, den Routen von Kanten oder Eigenschaften eines automatisch erstellten Layouts vornehmen. Diese Anpassungen sollen in folgenden Layout-Prozessen erhalten bleiben. Diese Arbeit befasst sich damit, Intentional Layout für die IDE KEITH und ihre Sprotty Diagrams zu erarbeiten.

Präzisiert ist das Ziel, dass die Nutzenden durch Interaktionen in den Sprotty Diagrams von KEITH ein Layout des Layered Layoutalgorithmus anpassen können. Ist eine Interaktion mit einer Anpassung verknüpft, soll sie zur Spezifizierung von sogenannten Constraints führen. Diese kapseln die Änderungen ein, die die Anpassung vornimmt, und persistieren sie in das Modell. Diese Anpassungen sollen als Optionen des Layered Layoutalgorithmus im Modell des Graphen persistiert werden, damit die Änderungen in Folgelayouts erhalten bleiben. Dafür ist es nötig, das Intentional Layout als Feature in die Kommunikationsstruktur und die Architektur von KEITH zu integrieren. Diese allgemeinen Bestandteile des Intentional Layouts in Sprotty Diagrams ist in Zusammenarbeit im Rahmen einer Projektphase mit Petzold [Pet19] entstanden. Dabei fokussieren wir uns auf die Betrachtung und die Implementierung von Constraints, die einem Knoten einen festen Layer und eine feste Position

zuweisen. Im Weiteren beziehen sich „wir“, „uns“, „unsere“ und so weiter daher auf Petzold und mich.

Das Intentional Layout kapselt interaktiv definierte Layout-Anpassungen in Constraints. Die Auswertung der Constraints im Layout-Prozess führt zur Anwendung der Layout-Anpassungen auf ein bestehendes Layout. Dabei sollte sichergestellt sein, dass dies nur die Layout-Anpassungen sind, die in die Constraints eingekapselt sind, damit der sich verändernde Graph konsistent zur Mental Map der Nutzenden bleibt. Während eine Erhaltung der Mental Map den Nutzenden helfen kann, eine Anwendung zu verstehen, kann das Gegenteil fehlleitend sein [PHG07]. Eine Anpassung von gesetzten Constraints, um die Mental Map der Nutzenden zu erhalten, heiße Reevaluation. Diese Arbeit untersucht, ob und wie diese erfolgen muss, wenn Constraints hinzugefügt, vorhandene Constraints gelöscht oder Knoten hinzugefügt oder gelöscht werden. Zudem erfolgt die konzeptuelle Betrachtung zusätzlicher Constraint-Typen und wie die Reevaluation für diese erfolgen muss.

Abgrenzend widmet sich Petzolds Arbeit [Pet19] der Nutzerinteraktion, um zu ergründen, wie man den Nutzenden das Arbeiten mit dem Intentional Layout möglichst verständlich vermittelt. Dies schließt mit ein, welche Interaktionen benötigt werden, wie diese gestaltet sind und welche Constraints unter welcher Belegung hierbei erzeugt werden.

1.2. Aufbau

Zunächst geht Kapitel 2 auf theoretische Grundlagen und die Technologien ein, die die Projektphase und diese Arbeit verwendet haben. Darauf erfolgt in Kapitel 3 ein Blick darauf, wie verwandte Arbeiten mit Intentional Layout über Constraints umgegangen sind. Daraufhin leitet Kapitel 4 durch die Konzepte zum Intentional Layout. Es umfasst das gesamte Konzept, das wir in der gemeinsamen Projektphase erarbeitet haben. Aufbauend auf dem ersten Konzeptkapitel befasst sich Kapitel 5 mit den Prinzipien der Reevaluationen, den zwei Modi von Intentional Layout, Reevaluationen sowie erweiterten Features. Die Implementierung der Konzepte aus der Projektphase und Implementierung zu den in dieser Arbeit betrachteten Themen finden sich in Kapitel 6. Schließlich fasst Kapitel 7 die Kernthemen, Probleme und Lösungen zusammen und präsentiert offene Probleme und Themen, die Gegenstand zukünftiger Arbeiten sein können.

Kapitel 2

Benutzte Technologien und Grundlagen

Die gemeinsame Projektphase und die Reevaluation gesetzter Constraints stützt sich auf Technologien wie den Layered Layoutalgorithmus, ELK, das LSP und KEITH. Dazu beziehen sich Aspekte dieser Arbeit auf Definitionen, Begriffe und Algorithmen aus dem Gebiet des Graph Drawings.

2.1. Definitionen zu Graphen

Die folgenden Definitionen und Begriffe werden in den nächsten Kapiteln verwendet. Viele von ihnen basieren auf den Definitionen der Automatic Graph Drawing Vorlesung der Universität Kiel im Wintersemester 2017/2018 [Han18], wurden für den Zweck dieser Arbeit jedoch zum Teil erweitert.

Graph Unter einem (gerichteten) Graphen oder auch Digraphen versteht man ein Paar $G = (V, E)$ aus einer Menge Knoten $V = \{v_1, \ldots, v_n\}$ mit $n \in \mathbb{N}$ und einer Menge gerichteter Kanten $E = \{(v_i, v_j) | v_i \in E \wedge v_j \in E\}$ mit $i \in \mathbb{N}$ und $j \in \mathbb{N}$.

Source und Target Bei einer gerichteten Kante einer Kantenmenge $e = (s, t) \in E$ nennt man den Knoten s Source und den Knoten t Target der Kante e. s und t nennt man auch die *Endknoten* der Kante e. §

adjazent und inzident Ein Knoten $u \in V$ ist adjazent zu einem Knoten $v \in V$, wenn $(u, v) \in E \vee (v, u) \in E$ für einen gerichteten Graphen gilt. Adjazente Knoten werden auch *Nachbarn* genannt. Die Nachbarn eines Knoten v sind demnach alle zu ihm adjazenten Knoten. Eine Kante $e \in E$ ist inzident zu einem Knoten $v \in V$, wenn v die Source oder das Target der Kante e eines gerichteten Graphen ist.

Pfad Ein Pfad der Länge n eines Graphen $G = (V, E)$ ist eine Sequenz $(v_1, \ldots, v_n)$ mit $v_1, \ldots, v_n \in V$ und $(v_i, v_{i+1}) \in E$ für $1 \leqslant i < n$.

Zyklus Ein Zyklus ist ein Pfad der Länge $n \in \mathbb{N}$, für den $v_1 = v_n$ gilt.

(a)zyklisch Ein Graph $G = (V, E)$ ist azyklisch, wenn er keine Zyklen enthält, sonst ist er zyklisch.

Ebene Eine Ebene, *plane*, ist eine zweidimensionale Oberfläche, die sich unendlich in beide Dimensionen ausdehnt.

Kurve α heißt Kurve, *curve*, sofern $\alpha \subseteq \mathbb{R}^2$ der Form $\alpha = \{\gamma(x) | x \in [0, 1]\}$, wobei $\gamma : [0, 1] \mapsto \mathbb{R}^2$ eine kontinuierliche Abbildung vom geschlossenen Intervall $[0, 1]$ zur Ebene sei. $\gamma(0)$ und $\gamma(1)$ sind die Endpunkte der Kurve α. Eine Kurve heißt *geschlossen*, wenn sie denselben Start- sowie Endpunkt hat, andernfalls heißt sie *offen*. Eine Kurve heißt *simple*, sofern es in ihr keine wiederholten Punkte abgesehen von dem Start- und Endpunkt gibt.

Drawing/Layout Bei einem Drawing oder Layout handelt es sich um eine Abbildung Γ, die jeden Knoten $v \in V$ eines Graphen auf einen Punkt $\Gamma(v) \in \mathbb{R} \times \mathbb{R}$ und jede Kante $e = (u, v) \in E$ auf eine simple Kurve $\Gamma(u, v)$ mit den Endpunkten $\Gamma(u)$ und $\Gamma(v)$ abbildet. Jeder Knoten $v \in V$ wird

zusätzlich mit einer Breite $w_v \in \mathbb{R}$ und einer Höhe $h_v \in \mathbb{R}$ sowie mit *Ports* assoziiert, die an den *Boundaries*, Grenzen, des Knoten befestigt sind und als Endpunkte der inzidenten Kanten dienen.

Layout-Erstellung und Layout-Anpassung Eine Layout-Erstellung erstellt ein Layout für einen Graphen. Dahingegen verändert eine Layout-Anpassung ein vorhandenes Layout. Das schließt Anpassungen der Knoten-Positionen, der Kanten-Routen und weiterer Eigenschaften des Layouts mit ein.

Layering und Layer Bei einem Layering L handelt es sich um eine Partition $L_1, \ldots, L_n$ von V eines Graphen $G = (V, E)$ mit $n \in \mathbb{N}$, so dass für jede Kante $(u, v) \in R$ gilt $u \in L_i \wedge v \in L_j \implies i < j$. Hierbei nennt man jedes L_i für $i \in \mathbb{N}, i \leqslant n$ Layer mit $L_i = [v_0, \ldots, v_k]$ mit $k \in \mathbb{N}$. Ein Layer ist eine geordnete Menge und kann als Liste, Array oder Sequenz interpretiert werden. Ist ein Knoten v_i an Position i in der Struktur des Layers, sagen wir, dass der Knoten Position i im Layer hat. Diese Position nennen wir im Weiteren auch *Positions-Index* oder *Position-Id*, den die Abbildung $P : V \mapsto \mathbb{N}$ zurückgibt. Es gilt, wenn $v \in V$ an Position i in L_j mit $i, j \in \mathbb{N}$, dann gilt $P(v) = i$. Die Abbildung $L : V \mapsto \mathbb{N}$ liefert den sogenannten *Layer-Index* beziehungsweise die *Layer-Id* eines Knoten zurück. Wenn gilt, dass $v \in L_i$ für $i \in \mathbb{N}$, dann gilt $L(v) = i$.

Proper Layering Wiederum ist ein Proper Layering ein Layering von V eines Graphen $G = (V, E)$, sodass gilt $(u, v) \in E \implies L(u) + 1 = L(v)$.

Layered Graph Ein Layered Graph ist ein Tripel $G = (V, E, L)$, wobei es sich bei L um ein Layering handelt.

unter, über, links, rechts Seien $v, w \in V$ für einen Graphen $G = (V, E, L)$.

- v unter w, wenn $P(v) > P(w)$
- v über w, wenn $P(v) < P(w)$
- v links w, wenn $L(v) < L(w)$
- v rechts w, wenn $L(v) > L(w)$

Hierarchischer Graph Ein hierarchischer Graph ist ein Graph $G = (V, E, h)$ mit einer wie zuvor definierten Menge von Knoten V sowie einer Menge von gerichteten Kanten E und einer Abbildung $h : V \mapsto V \cup \{\perp\}$. h bildet einen Knoten $v \in V$ auf seinen Parent-Knoten ab, sofern ein solcher existiert. Dahingegen entspricht $h^{-1}(v)$ für $v \in V$ den Kinder eines Knoten, die dieser enthält. h induziert einen sogenannten *Containment Graphen* $G^* = (V, E^*)$ mit $E^* = \{(u, v) | u, v \in V \wedge u = h(v)\}$ von G. h ist valide, sofern der Containment Graph kreisfrei ist.

2.2. Der Layered Layoutalgorithmus

Der Layered Layoutalgorithmus [SSH14] von ELK[1] ist eine Implementierung des Sugiyama-Algorithmus [STT81]. Der Sugiyama-Algorithmus begründet die Klasse der ebenenbasierten Layoutalgorithmen, die ebenbasierte Layouts erstellt. Zur Erstellung eines solchen Layouts wendet der Layered Layoutalgorithmus fünf konsekutive Phasen auf einen gegebenen Graphen an:

1. **Cycle Breaking:** Erhält einen möglicherweise zyklischen Graphen als Menge aus Knoten und Kanten als Eingabe und gibt ihn kreisfrei an die nächste Phase weiter.

[1] https://www.eclipse.org/elk/reference/algorithms/org-eclipse-elk-layered.html

2. **Layer Assignment:** Weist jeden Knoten des Graphen einem Layer L_i für $i \in \mathbb{N}$ zu und gibt einen Layered Graphen $G' = (V', E', L)$ an die nächste Phase weiter. Hierbei ist zu beachten, dass der Layered Layoutalgorithmus keine Kanten erlaubt, deren Source und Target im gleichen Layer liegt. Zudem kann der Algorithmus Layer überspannende Kanten anlegen, die Knoten verbinden können, die sich nicht in angrenzenden Layern befinden.
3. **Crossing Minimization:** Minimiert die Kantenüberkreuzungen zwischen Knoten verschiedener Layer durch Umsortierung der Knoten in einem Layer und der Ports der Knoten und gibt den angepassten Layered Graphen an die nächste Phase weiter.
4. **Node Placement:** Weist den Knoten im Layout entweder vertikale Koordinaten zu, wenn die Layer vertikal verlaufen, oder horizontale Koordinaten zu, sofern die Layer horizontal verlaufen.
5. **Edge Routing:** Setzt die jeweils andere Koordinate und bestimmt den geometrischen Verlauf jeder Kante im Layout.

Lediglich die ersten drei Phasen, die *topologischen* Phasen, bestimmen in welchem Layer und welche Position im Layer ein Knoten erhält. Die zwei folgenden *geometrischen* Phasen beeinflussen den Positions-Index und den Layer-Index eines Knoten nicht [Han18].

Darüber hinaus ist der Layered Layoutalgorithmus nicht nur imstande Graphen im Standardformat, also Strukturen aus Knoten und Kanten, zu layouten, sondern kann bei der Layout-Erstellung Knotenports, Label-Größen, Label-Platzierungen sowie Knotengröße einbeziehen. Das Layout von hierarchischen Graphen wird über die Layout-Infrastruktur gelöst: Der Algorithmus wird für jedes Hierarchielevel eines hierarchischen Graphen separat ausgeführt. Weiterhin gibt es vor der ersten Phase, zwischen allen weiteren Phase und nach der fünften Phase einen *Intermediate-Processing-Slot*, in dem *Preprocessors* beziehungsweise *Postprocessors* den aktuellen Stand des Drawings bearbeiten können.

2.3. Interaktive Layoutstrategien

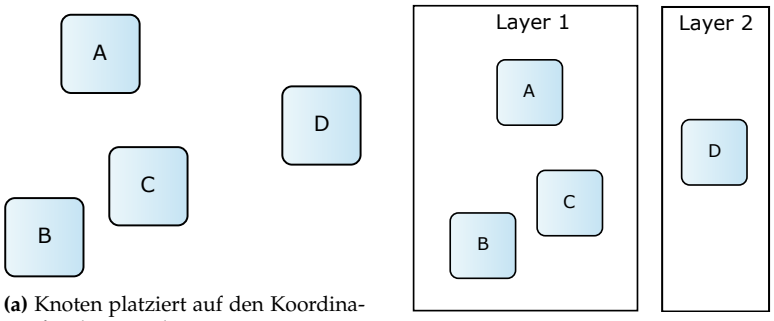
Die Phasen des Algorithmus sind in phasenspezifischen Layoutstrategien implementiert, die das Interface und die Kernaufgabe ihrer Phase erfüllen, wodurch die Implementierungen austauschbar sind. Für das Intentional Layout in Sprotty Diagrams werden die *Interactive Cycle Breaking Strategy*, *Interactive Layer Assignment Strategy* und *Interactive Crossing Minimization Strategy* verwendet. Diese folgen dem gemeinsamen Konzept, dass der Algorithmus nicht vollständig automatisch das Layout erstellt, sondern es auf Basis der Position der Knoten im Input-Graphen vornimmt.

2.3.1. Interactive Cycle Breaking Strategy

Diese Strategie löst Zyklen im Graphen auf, indem sie die horizontale Position der Endpunkte von Kanten betrachtet. Dabei wird entweder die linke Boundary, die zentrierte Boundary oder die rechte Boundary betrachtet.

2.3.2. Interactive Layer Assignment Strategy

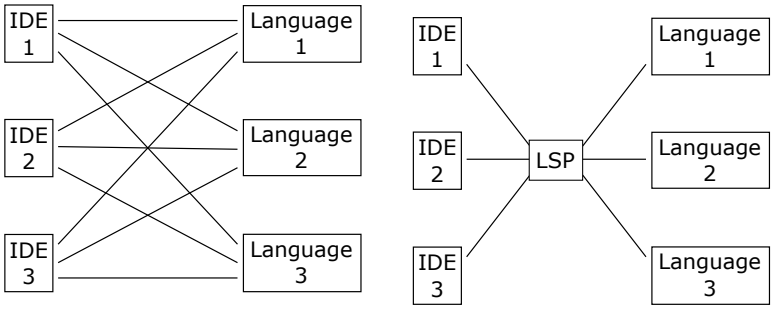
Die Interactive Layer Assignment Strategy ordnet jeden Knoten des Input-Graphen einem Layer zu. Abbildung 2.1 zeigt wie sie dabei vorgeht: Die Zuordnung geschieht von links nach rechts anhand der horizontalen Koordinaten der linken und rechten Boundary eines jeden Knoten. Hierbei platziert die Strategie Knoten, deren Anfang und Endpunkte sich überschneiden, in einem gemeinsamen Layer.



(a) Knoten platziert auf den Koordinaten für die interaktiven Layoutstrategien.

(b) Layer-Bildung durch die Interactive Layer Assignment Strategy.

Abbildung 2.1. Funktionsweise des interaktiven Layouts von Petzold [Pet19].



(a) Knoten platziert auf den Koordinaten für die interaktiven Layoutstrategien.

(b) Layer-Bildung durch die Interactive Layer Assignment Strategy.

Abbildung 2.2. Eine Schema zu LSP mit Bezug auf das *m IDEs n languages-Problem* von Petzold [Pet19].

2.3.3. Semi-Interactive Crossing Minimization Strategy

Schließlich bestimmt die Semi-Interactive Crossing Minimization Strategy die Position jedes Knoten, der kein Dummy-Knoten ist, in seinem Layer, indem sie eine aufsteigende Ordnung auf ihren vertikalen Koordinaten herstellt. Auf den Dummy-Knoten findet bei Verwendung dieser Strategie dennoch eine Minimierung der Kantenüberkreuzungen statt.

2.4. Der Eclipse Layout Kernel (ELK)

ELK, ein *Eclipse Incubation Projekt* [2], bietet eine Infrastruktur, um Layout-Algorithmen mit Diagramm-Editoren oder Viewern zu verbinden. Es ist ebenfalls möglich, eigene Layoutalgorithmen in ELK zu integrieren. Nebst der Infrastruktur stellt ELK eine Reihe von Implementierungen von Layoutalgorithmen bereit. Hierzu zählt auch Layered Layoutalgorithmus, siehe Abschnitt 2.2.

ELK beinhaltet das `elkt`-Format für Graphen. Dieses ermöglicht es, Knoten in der Syntax `node knotenName` und Kanten in der Syntax `edge source target` zu definieren. Zudem lassen sich in dem Format Layoutoptionen für Knoten angeben, die auch Algorithmus-spezifisch sein können.

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

2.5. Language Server Protocol

LSP ist ein *Open Source Communication Protocol*, das JavaScript Object Notation (JSON) Remote Procedure Call (RPC) 2.0 verwendet, um Kommunikation zwischen IDEs oder Source Code Editors und Language Server (LS) herzustellen, die *Rich Editing-Features* für Programmiersprachen bereitstellen. Rich Editing-Features sind beispielsweise Autovervollständigung, Syntax-Highlighting, ein „Go To Definition"-Feature, Einbettung der Dokumentation und Linting. Damit ist es ein Ansatz, das *m IDEs n languages-Problem* zu lösen. Dieses beschreibt, dass man für jede Programmiersprache die Rich Editing-Features für jede IDE implementieren muss, siehe Abbildung 2.2a. Durch das LSP vereinfacht sich die Umsetzung der Rich Editing-Features deutlich, wie Abbildung 2.2b verdeutlicht. Ursprünglich wurde das LSP von Microsoft entwickelt, damit Visual Studio Code mehrere LS unterstützt, die Linter bereitstellen. Daraufhin entwickelte Microsoft das LSP zusammen mit CodeEnvy [3] und Red Hat [4] mit dem Ziel weiter, das Protokoll weiter zu standardisieren. Diese Zusammenarbeit führte zu einer höheren Abstraktion des LSP sowie zur Unterstützung von mehr Sprachen und IDEs.

2.6. KIELER integrated in Theia (KEITH)

KEITH ist das Ergebnis der Migration von KIELER in das Framework für webbasierte IDEs Theia durch Domrös [Dom18] und Rentz [Ren18]. Präzisiert findet in der Theia-Applikation KEITH, das wie KIELER eine IDE für MDE ist, das Backend von KIELER als LS Verwendung, um so die Sprachunterstützungs-Features sowie das Rich Editing von KIELER zu KEITH zu migrieren.

Rentz transferierte das Kieler Light Weight Diagrams (KlighD) Framework aus dem KIELER-Projekt zu KEITH, das für die abstrakte Diagrammgenerierung für die von KIELER unterstützten Sprachen in der Eclipse IDE zuständig ist. Hierfür verwendete Rentz das Sprotty-Framework, das parallel zu Theia entwickelt wird. Sprotty übernimmt, nachdem der LS die erforderlichen Daten an den KEITH-Client gesendet hat, die graphische Darstellung von Diagrammen in KEITH, wofür Scalable Vector Graphics (SVG) zum Einsatz kommen [Ren18]. Damit stellt Sprotty das Äquivalent zum Framework *Piccolo2D* [5] dar, das KIELER für diesen Zweck nutzt.

2.7. Constraints

Unter Layout-Anpassungen versteht man Veränderungen des Layouts, nachdem ein Layout erstellt worden ist. Ein Beispiel hierfür ist die Veränderung von Knoten-Koordinaten. Constraints kapseln die Semantik einer Layout-Anpassung, die interaktiv herbeigeführt werden kann. Sie lassen sich als Eigenschaften den Knoten im Modell hinzufügen und persistieren damit die spezifizierte Layout-Anpassung. Hierbei unterscheiden wir zwischen absoluten Constraints und relativen Constraints.

2.7.1. Absolute Constraints

Absolute Constraints sind Eigenschaften, die Knoten feste Werte zuweisen. Hierbei definieren wir *Position Constraints* und *Layer Constraints*. Sei $G = (V, E)$ ein beliebiger Graph.

Position Constraint Ein Position Constraint mit Wert $v \in \mathbb{Z}$ weist einem Knoten v eine Position $p \in \mathbb{N}$ in seinem Layer L_i mit $i \in \mathbb{N}$ zu, wobei p in Abhängigkeit von v gesetzt wird. Er dient dazu,

[3] https://blog.codenvy.com/press-release-red-hat-codenvy-and-microsoft-collaborate-on-language-server-protocol-8f7c27f2d2ab

[4] https://www.redhat.com/en/about/press-releases/red-hat-codenvy-and-microsoft-collaborate-language-server-protocol

[5] http://piccolo2d.org/

Positionsänderungen in Layern zu persistieren. Dabei wird der Knoten auf die letzte Position eines Layers gesetzt, wenn $v >= |L_i|$.

Layer Constraint Ein Layer Constraint mit Wert $l \in \mathbb{Z}$ weist einen Knoten n einem Layer L_j mit $j \in \mathbb{N}$ zu, wobei L_j in Abhängigkeit von l gesetzt wird. Er dient dazu, eine Verschiebung eines Knoten von einem Layer in einen anderen Layer zu persistieren.

Constraint Layer Constraint Ein Constraint Layer Constraint für einen Knoten p erwartet eine Liste $[z_0, \ldots, z_k]$ für $k \in \mathbb{Z}$. Mit einem derartigen Constraint lassen sich in einem Parent-Knoten neue Layer spezifizieren, die zu den Layern hinzugefügt werden, die das automatische Layout anlegt, wenn es ein Layout für den in ihm enthaltenen Graphen erstellt.

2.7.2. Relative Constraints

Relative Constraints stellen einen Knoten in Relation zu einem anderen Knoten. Folgende Typen von relativen Constraints haben wir konzeptualisiert.

Same Layer Constraint Ein Same Layer Constraint assoziiert einen Knoten n mit einem Knoten m und drückt aus, dass Knoten n im Layer $L_{L(m)}$ platziert werden soll, also in dem Layer, in dem sich Knoten m befindet.

Below Constraint Ein Below Constraint assoziiert einen Knoten n mit einem Knoten m und drückt aus, dass ein Knoten n im Layer von m, also in Layer $L_{L(m)}$, platziert sein sowie eine Position p mit $p > P(m, L_{L(m)})$ haben soll.

Above Constraint Ein Above Constraint assoziiert einen Knoten n mit einem Knoten m und drückt aus, dass ein Knoten n im Layer von m, also in Layer $L_{L(m)}$, platziert sein sowie eine Position p mit $p < P(m, L_{L(m)})$ haben soll.

Left of Constraint Ein Left of Constraint assoziiert einen Knoten t mit einem Knoten a und drückt aus, dass ein Knoten durch das Layout im Layer $L_{L(a)-1}$ platziert werden soll, sofern $L(a) - 1 >= 0$, sonst soll das Layout den Knoten in einem neuen ersten Layer platzieren.

Right of Constraint Ein Right of Constraint assoziiert einen Knoten t mit einem Knoten a und drückt aus, dass der Knoten t durch das Layout in Layer $L_{L(a)+1}$ platziert werden soll.

2.7.3. Abbildungen zu Constraints

Folgende Abbildungen dienen dazu, den Wert eines absoluten Constraints oder den in Relation stehenden Knoten eines relativen Constraints wiederzugeben.

Abbildungen PC , LC, CLC Die Abbildungen $PC, LC : V \mapsto \mathbb{Z}$ liefern den Position Constraint-Wert und Layer Constraint-Wert eines Knoten, sofern diese gesetzt wurden, sonst sind die Bilder für einen Knoten undefiniert. Die Abbildung $CLC : V \mapsto \mathbb{Z}^+$ bildet einen Knoten $v \in V$ auf eine Liste $zs \in \mathbb{Z}$ von Knoten ab, sofern ein Constraint Layer Constraint mit ebendieser Liste auf dem Knoten v gesetzt ist.

Abbildungen SLC, BC, AC, LoC, RoC Die Abbildungen $SLC, BC, AC, LoC, RoC : V \mapsto V$ bilden einen Knoten $t \in V$ auf einen Knoten $r \in V$ ab, sofern ein Same Layer Constraint, Below Constraint, Above Constraint, Left of Constraint oder Right of Constraint für ihn spezifiziert wurde. Andernfalls sind die Bilder der jeweiligen Abbildung für diesen Knoten v undefiniert.

Kapitel 3

Verwandte Arbeiten

Andere Arbeiten haben sich bereits mit der Idee befasst, automatisches Layout um die Einführung von Constraints oder ihnen ähnliche Konzepte zu ergänzen, um Intentional Layout zu ermöglichen.

Böhringer et al. [BP90] verwenden Layout Constraints, um stabilere Folgelayouts zu erhalten, damit die Orientierung der Nutzenden nicht verloren geht. Ähnlich wie wir führen Böhringer et al. absolute Constraints und relative Constraints ein. Sie beschäftigen sich jedoch mehr mit relativen Constraints und zusätzlich mit sogenannten Clusters, Constraints für eine Auswahl von Knoten, während wir uns auf absolute Constraints fokussieren. Böhringer et al. formalisieren die Constraints auf Basis der Koordinaten von Knoten. Dabei restriktieren sie Constraints auf lineare Gleichungen und behandeln unterschiedliche Dimensionen unabhängig voneinander. Ihr Paper führt eine Lösung [Böh89] für ihre Layout Constraints für den Sugiyama-Algorithmus als Hauptbeispiel an. Sie verändern die verschiedenen Phasen des Sugiyama-Algorithmus, so dass er ihre Layout Constraints unterstützt. Anstelle dessen bauen wir unser Intentional Layout auf Basis des Layered Layoutalgorithmus auf und verändern nicht den Algorithmus an sich. Stattdessen nutzen wir die bereits implementierten (Semi) Interactive Layout Strategies von ELK. Böhringer et al. lassen ihren Constraint Manager eine Liste aller eingeführten Constraints verwalten. Der Constraint Manager evaluiert sein Constraint Netzwerk und hält es konsistent. Konsistent bedeutet hierbei, dass keine Layout Constraints widersprüchlich sind. Ist das Netzwerk nicht konsistent, wählt der Manager durch das Deaktivieren von Constraints eine Teilmenge konsistenter Constraints aus. Deaktivierte Constraints werden bei der Evaluation des Netzwerks ignoriert. Wir benötigen für unsere absoluten Constraints keinen Constraint Manager, da es nicht viele Fälle gibt, in denen Widersprüche auftreten können. Eine Erweiterung auf relative Constraints würde es jedoch nötig machen, vermehrt über die Evaluation der relativen Constraints nachzudenken. Hierfür bietet sich eine Interpretation der Constraints als Gleichungen an, ähnlich wie Böhringer et el. es tun. Darüber hinaus generieren Böhringer et al. zusätzliche Constraints nach jedem automatischen Layout, um für sogenannte dynamische Stabilität zu sorgen. Statt extra Constraints zu erzeugen, erarbeitet diese Arbeit Reevaluationen der existierenden Constraints, um die Mental Map zu erhalten. Böhringer et al. haben ihre Lösung in den Graph Editor EDGE [PT90] [93] integriert: Die Modifikationen erlauben den Nutzenden, über ein Interface Constraints hinzuzufügen, zu löschen oder eine Query durchzuführen. Die Oberfläche sendet das entsprechende Kommando an den Constraint Manager. Constraints persistieren die Autoren in das textuelle Graphenformat, Graph Representation Language (GRL) [New88]. Es stellt einen Constraint als eigenes Sprachelement in Form einer Menge von `Attribut:Wert`-Paaren dar. Wir hingegen implementieren Constraints als Layoutoptionen des Layered Layoutalgorithmus im `elkt`-Format von ELK, die sich auf Knoten setzen lassen.

Nascimento und Eades [NE02] erweitern den Sugiyama-Algorithmus um sogenannte User Hints: User Hints sind Layout Constraints, manuelle Anpassungen und sogenannter Focus. Sie sehen Top-Down- und Left-Right-Constraints vor, die damit den von uns definierten relativen Constraints ähneln. Mit den manuellen Anpassungen lassen sich Knoten anders positionieren: Diese Positionen nutzen die Autoren zur Optimierung des Layouts, erstellen für sie aber anders als wir keine Constraints, die das Layout einhalten muss. Also gibt es bei Nasicmento und Eades keine absoluten Constraints, die persistiert werden. Nascimento und Eades haben den sogenannten Focus vorgesehen, durch

den der Layoutalgorithmus nur auf dem fokussierten Bereich eines Graphen angewendet wird. Die beschriebenen Operationen können die Nutzenden interaktiv im System GDHints [NE02] verwenden. Um User Hints einzuführen, passen Nascimento und Eades ähnlich wie Böhringer et al. die Phasen des Sugiyama-Algorithmus an, wohingegen wir die Phasen des Layered Layoutalgorithmus unverändert lassen.

Andere Arbeiten erarbeiten Constraints auf Basis von Force Directed Layout Algorithmen. Solche Algorithmen optimieren die Knotenpositionen eines Layouts, indem sie eine physikalisch inspirierte Funktion minimieren, so dass die Knoten eine nach ihr optimale Position erhalten [Ead84]. Für Force Directed Algorithmen lassen sich andere Constraints definieren als für Layer basierte Layout Algorithmen wie es der Sugiyama-Algorithmus ist. Aufgrund der grundsätzlich anderen Funktionsweise des Algorithmus, unterscheidet sich die Art, wie Constraints für sie umgesetzt werden. Ihre Umsetzung lässt sich nicht direkt auf unseren Ansatz übertragen.

Kamps et al. [KKR96] haben für ihren eigenen Force Directed Layoutalgorithmus, der auf dem Ansatz von Eades [Ead84] aufbaut, Constraints umgesetzt. Sie haben Constraints für fixe Positionen, fixe Distanzen zwischen Knotenpaaren, relative Positionen zwischen zwei Knoten und das Festlegen einer Orientierung einer Teilmenge von Knoten ausgearbeitet. Damit gleichen ihre Constraints sowohl unseren absoluten Constraints als auch relativen Constraints: Unsere absoluten Constraints platzieren einen Knoten jedoch nicht mit fester horizontaler und vertikaler Koordinate, sondern topologisch als *i*-ten Knoten eines Layers. Distanzen lassen sich in unserem Konzept nicht ausdrücken. Kamps et al. verwenden die Modellierungssprache Smalltalk Frame Kit (SFK) [FR92], um die Relationen ihrer Daten auszudrücken. Diese unterscheidet sich wesentlich vom `elkt`-Format, das Knoten und Kanten eines Graphen spezifiziert.

Dwyer et al. präsentieren einen Constraint Graph Layoutalgorithmus [DMW09b]. Darunter verstehen sie eine Generalisierung von Force Directed Algorithmen, die eine physikalisch inspirierte Funktion minimiert, die das Ziel hat, Knoten so zu setzen, dass die Constraints erfüllt sind. Ihr Algorithmus ist zusätzlich in der Lage, die Topologie des initialen Layouts zu erhalten. Auch unser Intentional Layout soll die Topologie des Graphen erhalten, außer Constraints definieren eine spezifische Veränderung der Topologie für bestimmte Knoten. Der Algorithmus von Dwyer et al. ist intendiert für dynamisches Layout verwendet zu werden, lässt sich auch auf durch andere Layout-Techniken erstellte Layouts anwenden. Letzteres ist der intendierte Anwendungsfall für unser Intentional Layout. Dwyer et al. definieren ihre Constraints als sogenannte Separation Constraints: Diese sind Ungleichheits- oder Gleichheits-Constraints, die entweder für die horizontalen oder die vertikalen Positionen von Knoten definiert sind. Damit ähneln sie unserer Idee für relative Constraints. Zudem definieren Dwyer et al. Topology Constraints, die zur Erhaltung der Topologie des Layouts dienen. Das Tool Dunnart [1] verwendet den Algorithmus für kontinuierliche Layout-Anpassung durch User-Interaktion [DMW09a]. In Dunnart lassen sich die Separation Constraints interaktiv einführen und anpassen. Zudem nutzt ein Netzwerk-Diagramm-Browser von Dwyer et al. ihn [DMS+08]. Er updatet mit dem Algorithmus das Layout einer detaillierten View eines Netzwerkteils, während der Nutzer den Fokus-Knoten ändert, Knoten-Cluster kollabiert oder expandiert. Anders als in KEITH gibt es in Dunnart keine textuelle Repräsentation des Graphen. Dunnart ist keine IDE für MDE wie KEITH, sondern ein grafischer Diagrammeditor.

[1] `https://github.com/mjwybrow/dunnart`

Kapitel 4

Intentional Layout

Intentional Layout ist eine konzeptuelle Erweiterung für KEITH, die auf dem Layered Layoutalgorithmus aufbaut. Dabei passt das Konzept den Algorithmus nicht an, sondern verwendet die existierenden interaktiven Layoutstrategien, siehe Abschnitt 2.3. Dadurch trägt unsere Lösung zur Modularität bei. Interaktionen im Client von KEITH sind logisch mit ihnen zugeordneten Layout-Anpassungen verknüpft. Ebendiese sind in Constraints gekapselt, die an den Server gesendet und im Modell persistiert werden. Der für sie angepasste Layout-Prozess wertet die in die Constraints gekapselten Layout-Anpassungen wiederum aus und wendet sie auf das Layout an, welches der Layered Layoutalgorithmus initial erstellt hat.

Dabei fokussiert sich das Intentional Layout von Petzold und dieser Arbeit auf Position Constraints und Layer Constraints, also zwei absolute Constraints, siehe Abschnitt 2.7. Wir haben diesen Fokus gewählt, weil absolute Constraints den Effekt von relativen Constraints ausdrücken können und der Layered Layoutalgorithmus ebenfalls mit absoluten Layer- und Positions-Indices arbeitet.

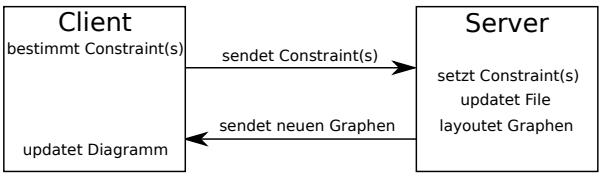


Abbildung 4.1. Kommunikationsablauf und Workflow des Intentional Layouts von Petzold [Pet19]

4.1. Workflow des Intentional Layouts

Damit sich das Intentional Layout in die Architektur von KEITH einfügt, gilt es, die Kommunikation zwischen dem LS und dem Client durch eine `LanguageServerExtension` zu erweitern. Abbildung 4.1 zeigt schematisch wie die Kommunikation funktioniert. Die LSP stellt ein Interface mit Nachrichten für alle Interaktionen bereit. Interagieren die Nutzenden mit dem Diagramm von KEITH, bestimmt der Client, ob und welche definierte Interaktion ausgelöst wurde. Daraufhin bestimmt der Client die zu setzenden Werte für die Constraints, die die Interaktion kapseln. Setzen Interaktionen neue Constraints auf Knoten, kann es zu unbeabsichtigten Layout-Änderungen wie Reihenfolgenveränderungen der nicht bewegten Knoten kommen. Dies lässt sich durch Anpassung der Constraint-Werte auf dem Modell verhindern. Schließlich sendet der Client die für die Interaktion vordefinierte Nachricht, die ein Kapselobjekt für die Werte der berechneten Constraints beinhaltet, an den LS. Daraufhin betrachtet der LS wiederum die neuen Constraints im Kontext der bereits gesetzten Constraints auf dem Modell und passt diese so an, dass nur die durch die neuen Constraints bestimmten Änderungen stattfinden. Diese Anpassung wird im Folgenden *Reevaluation* oder *Reevaluierung* genannt. Darauf folgend werden die Anpassungen der Constraints und die neuen Constraints auf das Modell übertragen. Diese

Veränderung des Modells löst einen neuen interaktiven Layout-Prozess aus, in diesem werden die Koordinaten der Knoten derart gesetzt, dass die interaktiven Layoutstrategien, siehe Abschnitt 2.3, die Constraints erfüllen.

4.2. Interaktives Erstellen von Constraints im Client

Interaktives Spezifizieren von Constraints findet im Diagramm im Client von KEITH statt. Hierbei gibt es drei grundlegende Interaktionen. Das generelle Prinzip ist unabhängig von der genauen Umsetzung der Interaktion: Wenn eine Interaktion durchgeführt wurde, für die ein Constraint definiert ist, sollen für diese Interaktion passende Constraints erzeugt und zu dem Modell hinzugefügt werden. Wie genau die Interaktionen definiert, den Nutzenden kenntlich gemacht und welche Constraints unter welcher Belegung welchen Interaktionen zugeordnet werden, findet sich in „Intentional Layout in Sprotty Diagrams: Defining User Interaction“ von Petzold [Pet19]. Es folgt eine Betrachtung der drei grundlegenden Interaktionen:

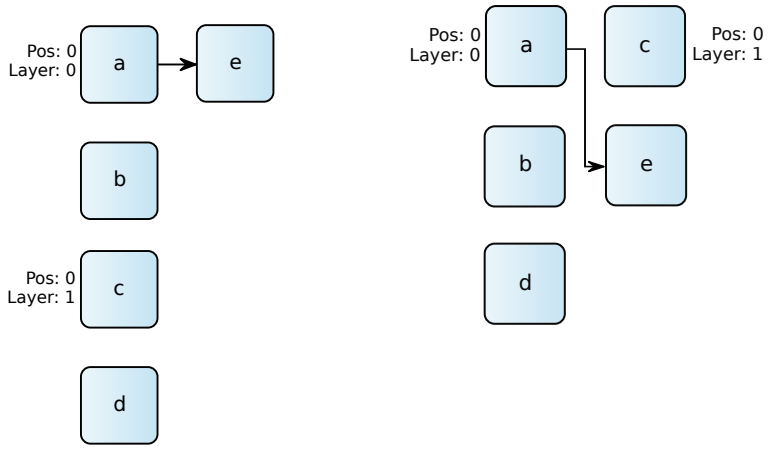
1. **Layer-Änderung:** Bewegen eines Knoten v von seinem Layer $L_{L(v)}$ in einen anderen Layer L_t mit $L(v) \neq t, t \in \mathbb{N}$
2. **Positionsänderung:** Bewegen eines Knoten u von seiner Position $P(u)$ zu einer anderen Position $j \in \mathbb{N}$ in seinem Layer $L_{L(u)}$
3. **Kombinierte Layer- und Positionsänderung:** Kombination beider voriger Fälle. Die Kombination von Positions- und Layer-Änderung lässt sich theoretisch durch das Senden von einer Positions- und einer Layer-Änderungsnachricht lösen. Allerdings geht dabei der Kontext verloren, dass beide Constraints gleichzeitig auf dem gleichen Stand des Modells gesetzt werden, und nicht nacheinander. Zudem reduziert die Kombination in einer Nachricht den Kommunikationsaufwand.

Für jede der drei grundlegenden Interaktionen gibt es eine Nachricht, die an den Server gesendet wird. Die Werte dieser Constraint-Typen bestimmt der Client anhand der Layer-Indices und Positions-Indices, der im Layout vorhandenen Knoten.

4.3. Layoutprozess für Intentional Layout

Intentional Layout soll zusätzlich zum automatischen Layout mit dem Layered Layoutalgorithmus erfolgen. Hierfür passen wir den Layout-Prozess an, der ausgeführt wird, wenn ein Modell, auf dem Constraints definiert sind, gelayoutet werden soll. Dieser funktioniert wie folgt:

1. **Initiales Layout:** Erstelle ein initiales Layout mit dem Layered Layoutalgorithmus, so dass alle Knoten eine horizontale und vertikale Koordinate erhalten. Ein Beispiel für das initiale Layout eines Graphen zeigt Abbildung 4.2a.
2. **Auswertung der Constraints und Koordinatenanpassung:** Werte die Constraints aus, die auf dem Modell spezifiziert sind und passe das Layout beziehungsweise die Koordinaten der Knoten derart an, dass das folgende Layout die Semantik der gesetzten Constraints erfüllt.
3. **Intentional Layout:** Dieser zweite Layout-Durchlauf passt das Layout so an, dass das resultierende Layout die Semantik der gesetzten Constraints erfüllt. Er verwendet die drei in Abschnitt 2.3 erläuterten interaktiven Layoutstrategien des Layered Layoutalgorithmus und deaktiviert die Option `SeparateConnectedComponents`. Abbildung 4.2b zeigt das Intentional Layout, das der Layout-Prozess aufbauend auf dem initialen Layout in Abbildung 4.2a generiert.



(a) Dies ist das initiale Layout des Beispielgraphen.

(b) Mithilfe der Koordinaten aus Abschnitt 4.4 erzeugt der zweite Layout-Aufruf dieses Layout.

Abbildung 4.2. Initiales Layout und erzeugtes Zweitlayout eines Beispielgraphen mit Constraints.

Die Option `SeparateConnectedComponents` muss deaktiviert werden, damit der Layered Layoutalgorithmus den gesamten Graphen in einem Durchgang layoutet. Ansonsten lässt ELK den Layered Layoutalgorithmus die Zusammenhangskomponenten des Graphen einzeln layouten. Sie nicht zu deaktivieren, hat mitunter zur Folge, dass Knoten, die durch Layer Constraints für verschiedene Layer spezifiziert wurden, im gleichen Layer platziert werden. Denn es kann vorkommen, dass der Layered Layoutalgorithmus ihre Zusammenhangskomponenten zusammenschiebt.

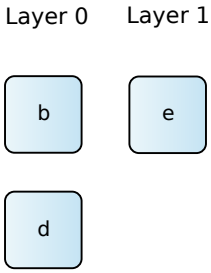
4.4. Constraint-Auswertung und Koordinatenanpassung

Der Fokus des Projekts von Petzold und dieser Arbeit liegt auf Position Constraints und Layer Constraints mit Werten $v \geqslant 0$. Für weitere Constraint-Typen und Anpassungen muss dieses Konzept erweitert werden. Eine solche Erweiterung stellt die Erlaubnis von Layer Constraints mit negativen Werten dar, siehe Abschnitt 5.5.2, und das Einführen von Constraint Layer Constraints, siehe Abschnitt 5.8.

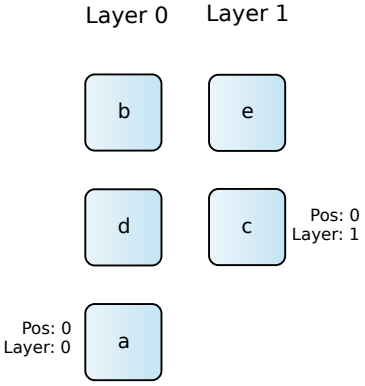
Damit der zweite auf das initiale Layout folgende Layout-Prozess mit aktivierten interaktiven Layoutstrategien tatsächlich die auf dem Modell gesetzten Constraints erfüllt, gilt es die Koordinaten der Knoten des Inputgraphen $G = (V, E)$ so anzupassen, dass die interaktiven Layoutstrategien ein Layout anlegen, das die Constraints erfüllt.

Das Layering des Graphen wird während der Koordinatenanpassung intern als Liste von Listen dargestellt. Die Anpassung durchläuft folgende Phasen. Die Beispielgrafiken in Abbildung 4.3 vollziehen die Koordinatensetzung exemplarisch aufbauend auf dem Layout aus Abbildung 4.2a nach.

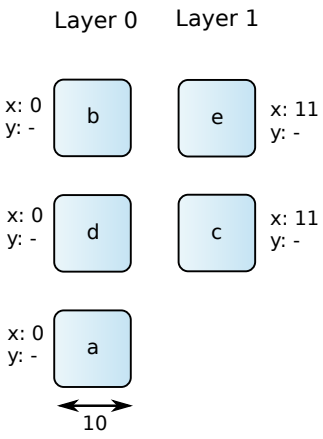
1. **Knoten ohne Constraint betrachten:** Füge jeden Knoten $v \in V$, auf dem kein Layer Constraint gesetzt ist, in die $L(v)$-te Layer-Liste. Für das Beispiel führt das zu der Ausprägung der Layer-Listen in Abbildung 4.3a.
2. **Constraint-Knoten hinzufügen:** Sortiere die Knoten, die einen *Layer Constraint* haben, aufsteigend nach dem Layer Constraint-Wert. Füge jeden Knoten $vc \in V$, auf dem ein Layer Constraint gesetzt



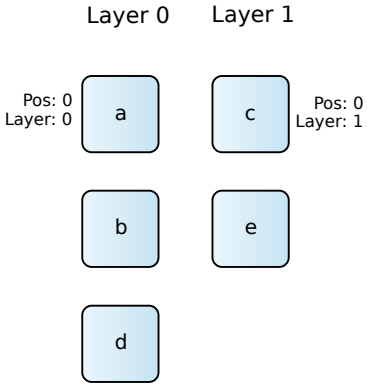
(a) Zunächst fügt die Koordinatenanpassung alle Knoten ohne Layer Constraint zu ihrer jeweiligen Layer-Liste hinzu.



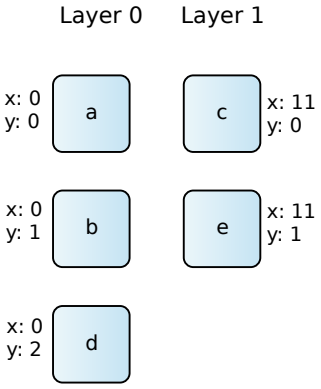
(b) In Schritt 2 fügt die Koordinatenanpassung die Knoten mit Layer Constraint zu ihrer Layer-Liste hinzu.



(c) In Schritt 3 werden die horizontalen Koordinaten der Knoten gesetzt.

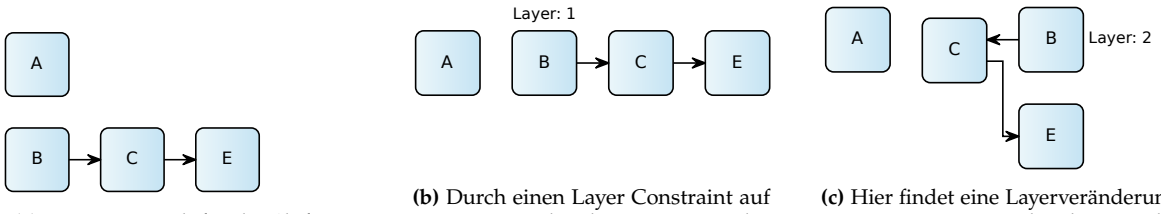


(d) In Schritt 4 fügt die Koordinatenanpassung die Knoten mit Position Constraint in die Position ein, die dem Wert des Position Constraint entspricht.



(e) Schließlich setzt Schritt 5 die vertikalen Koordinaten der Knoten.

Abbildung 4.3. Die Constraint-Auswertung und Koordinatenauswertung anhand des Beispielgraphs aus Abbildung 4.2.



(a) Ausgangsgraph für das Shifting.

(b) Durch einen Layer Constraint auf Knoten *B* werden die Knoten *C* und *B* geshiftet.

(c) Hier findet eine Layerveränderung von Knoten *B* statt, der die Ursache von Shifting-Operationen war.

Abbildung 4.4. Dieses Beispiel zeigt die Funktionsweise von Shifting.

ist, in die $LC(vc)$-te Layer-Liste. Gibt es in der letztgenannten Liste Knoten, die zu *vc adjazent* sind, muss dies durch Shifting behandelt werden, siehe Abschnitt 4.4.1. Abbildung 4.3b zeigt die Layer-Listen aus Abbildung 4.3b, die um die Knoten mit Layer Constraint erweitert wurden.

3. **Horizontale Koordinaten setzen:** Betrachte nun jede Layer-Liste Ll. Alle Knoten in der gleichen Layer-Liste erhalten die gleiche horizontale Koordinate. Die Breite des breitesten Knoten in der Layer-Liste bestimmt die Breite des Layers, da die interaktiven Layoutstrategien Knoten, deren horizontale Positionen ihrer linken und rechten Boundary überlappen, in den gleichen Layer einfügt. Das Ergebnis dieses Schritts für die Layer-Listen aus Abbildung 4.3b zeigt Abbildung 4.3c.

4. **Ordnung in Layer-Listen herstellen:** Betrachte jede Layer-Liste Ll. Teile Ll in eine Liste *nodes*, die nur aus Knoten ohne Position Constraint besteht, und in eine Liste *constraintNodes* aus Knoten mit Position Constraint auf. Sortiere die Liste *nodes* auf Basis des Layer-Index und die Liste *constraintNodes* nach dem Position Constraint-Wert jedes Knoten. Füge daraufhin jeden Knoten *vc* aus *constraintNodes* an der $PC(vc)$-te Position in *nodes* ein, sofern $PC(vc) < |nodes|$ gilt. Sollte $PC(vc) >= |nodes|$ gelten, füge *vc* am Ende der Liste *nodes* ein. In Abbildung 4.3d wird die Ordnung der Knoten für den Stand des Layout-Prozess in Abbildung 4.3c hergestellt.

5. **Vertikale Koordinaten setzen:** Betrachte jede Layer-Liste Ll. Der Knoten an Position 0 verbleibt an seiner vertikalen Position. Jeder Knoten v_i für $i > 0$ erhält die vertikale Position $y_i = y_{i-1} + h_{v_{i-1}} + 1$. Das Ergebnis dieses Schritts für den Beispielgraph zeigt Abbildung 4.3e.

4.4.1. Shifting

Das Intentional Layout ermöglicht es, Layer Constraints zu spezifizieren, die einen Knoten t in einen Layer verschieben, in dem sich zu ihm adjazente Knoten befinden. Da der Layered Layoutalgorithmus Kanten zwischen Knoten im gleichen Layer verbietet, muss ein solcher Fall behandelt werden. Im Folgenden werden diese Kanten auch *flache Kanten* genannt.

Einerseits bietet sich ein Verbot von Aktionen an, die zu flachen Kanten führt. Diese Option reduziert jedoch die Ausdrucksstärke von Constraints und ist damit nicht zu präferieren. Andererseits lässt sich das Vorkommnis adjazenter Knoten in einem Layer lösen, indem man die zu dem Knoten t adjazenten Knoten in einen anderen Layer bewegt. Hierbei kann es wieder zu adjazenten Knoten in einem Layer kommen, wodurch weitere Knoten verschoben werden müssen. Dieses Verschieben, das über mehrere Layer hinweg kaskadieren kann, heiße Shifting. Wir haben uns für dieses Vorgehen für das Intentional Layout entschieden.

Abbildung 4.4 zeigt, wie das Setzen eines Layer Constraint mit Wert 1 auf Knoten B in Abbildung 4.4a zu Shifting der Knoten B und C in Abbildung 4.4b führt. Shifting findet in Schritt 2 der

Koordinaten-Anpassung statt, vergleiche Abschnitt 4.4. Das bedeutet, dass Shifting erfolgt, wenn die Auswertung Knoten mit Layer Constraints zu ihren Layer-Listen zuordnet.

In der Constraint-Auswertung und Koordinatenanpassung funktioniert Shifting auf folgende Weise: Wird ein Knoten t in die Layer-Liste des Layers $L_{LC(t)}$ geschoben, werden alle zu ihm adjazenten Knoten, die sich bereits in der Layer-Liste befinden, aus ihr und damit aus dem Layer geshiftet. Dabei wird ausschließlich nach rechts geshiftet, also zur benachbarten Layer-Liste mit nächsthöherem Index.

Dieses Vorgehen ist konzeptuell einfach, da nicht entschieden werden muss, wann man nach links und wann nach rechts shiftet, und somit leichter zu implementieren. Weitere Betrachtung der Shifting-Strategie erfolgt in Abschnitt 5.4.1.

Beim interaktiven Anpassen von Layouts, die unter anderem durch Shifting entstanden sind, kommt es zu einem auf den ersten Blick nicht erwarteten Effekt, der eine direkte Folge des interaktiven Layout-Prozesses ist, siehe Abschnitt 4.3. Diesen Effekt zeigt Abbildung 4.4. In Abbildung 4.4b kommt es durch den Layer Constraint mit Wert 1 auf Knoten B zu Shifting von Knoten C in Layer L_2. Daraufhin wird Knoten E in L_3 geshiftet. Wird nun ein neuer Layer Constraint mit Wert 2 auf B gesetzt, muss der Layout-Prozess den Knoten C für das Folgelayout nicht shiften, weshalb es auch Knoten E nicht shiften muss. Also platziert der Layout-Prozess den Knoten C in L_1 und Knoten E in L_2. Wird in einem Layout ein Knoten s aufgrund eines Layer Constraints von Knoten v geshiftet und wird dem Knoten v ein neues Layer Constraint mit anderem Wert zugewiesen, dann „bewegt“ sich s im Folgelayout auf die Position zurück, die s im initialen Layout innehat. Das passiert, weil jedes interaktive Layout auf Basis eines initialen Layouts gestaltet wird. Es erfolgt somit keine Berücksichtigung der Vorzustände der Constraints. Stattdessen zählt nur der aktuelle Zustand der Constraints und in diesem gibt es keine Ursache, die zum Shiften von s führt, wodurch der Eindruck entsteht, dass sich s ohne Zutun der Nutzenden seine Position im Layout ändert. Dies kann eine Beeinträchtigung der Mental Map zu Folge haben, wenn die Nutzenden sich dieses Verhaltens von Knoten bei Shifting nicht bewusst sind. Ein Weg diesen Effekt zu verhindern, findet sich in Abschnitt 5.4.3.

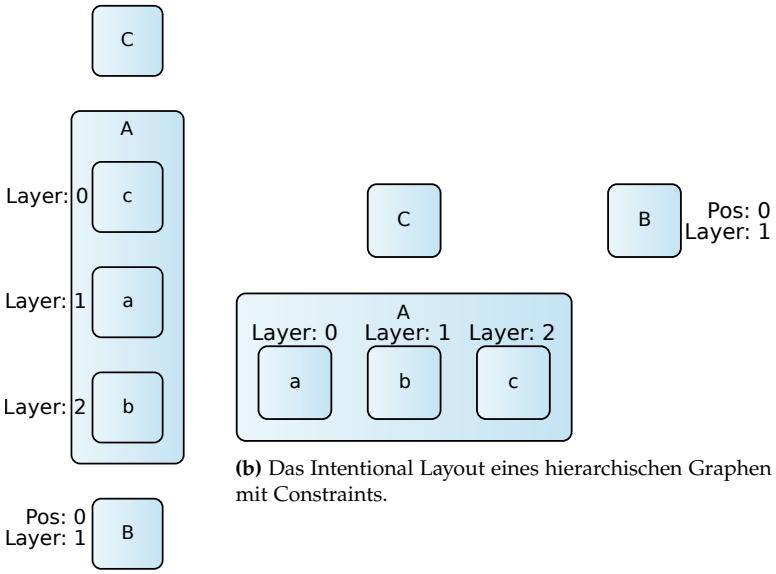
4.5. Intentional Layout in hierarchischen Graphen

Der interaktive Layout-Prozess für hierarchische Graphen erfolgt isoliert für jedes Hierachielevel des Graphen, also analog zu der Weise, wie der Layered Layoutalgorithmus mit hierarchischen Graphen umgeht. Ein Beispiel für ein Intentional Layout eines hierarchischen Graphen mit Constraints präsentiert Abbildung 4.5b. Diese Abbildung zeigt, dass Intentional Layout die Größe eines Layouts, in diesem Fall die Breite des Knoten A, im Vergleich zum initialen Layout in Abbildung 4.5a ändern kann. Daher sollte das Intentional Layout für hierarchische Graphen in einem Depth-First-Vorgehen erstellt werden. Auf diese Weise kann die geänderte Größe von Kindknoten beim Festlegen der Koordinaten in Parent-Knoten einfließen. Ansonsten kommt es unweigerlich dazu, dass die Layer-Platzierung von Knoten fehlerhaft ist, da die interaktiven Layoutstrategien Knoten anhand der Überlappung der Positionen ihrer linken und rechten Boundary in Layern platziert, siehe Abschnitt 2.3.

Als Konsequenz aus diesem Vorgehen beziehen sich Constraints für hierarchische Graphen auf das Hierarchielevel des Knotens, für den sie definiert sind, beziehungsweise auf die Knoten, die in einem Knoten enthalten sind.

4.6. Vor- und Nachteile des Layout Prozess

Unser Layout-Prozess wird bei jeder Aktualisierung des Modells erneut durchgeführt. Dadurch baut jede weitere Anpassung des Layouts durch eine interaktive Aktion nicht auf dem aktuellen Stand des



(a) Das initiale Layout eines hierarchischen Graphen mit Constraints.

(b) Das Intentional Layout eines hierarchischen Graphen mit Constraints.

Abbildung 4.5. Ein hierarchischer Graph mit Constraints.

Layouts und seiner aktuellen Ausprägung von Constraints auf. Anstelle erfolgt ein initiales Layout, auf das der interaktive Layout-Prozess die bisherigen Constraints, von denen einige möglicherweise angepasst wurden, und die neuen Constraints anwendet. Alternativ könnte ein Intentional Layout sich nicht auf das initiale Layout stützen, sondern inkrementell auf dem aktuellen Zustand der Constraints aufbauen.

Unsere statische Lösung hat den Vorteil, dass jedes angezeigte Intentional Layout sich allein durch die Synthese des Modells reproduzieren lässt. Also selbst, wenn die IDE abstürzt und neugestartet oder die Datei auf einen anderen Rechner übertragen wird, wird allein auf Basis des Modells wieder dasselbe Layout erzeugt. Das Layout ist unabhängig von Zwischenzuständen, die möglicherweise bei der Erstellung des Layouts existiert haben. Andererseits lässt dieses Konzept keinen Rückschluss auf Vorzustände des Layouts zu einer anderen Ausprägung der Constraints zu. Dadurch lassen sich ältere Zustände nicht ohne Weiteres wiederherstellen. Dies spielt beispielsweise beim Löschen von Constraints eine Rolle, siehe Abschnitt 5.7. Dadurch entspricht das Folgelayout beim Löschen zum Teil nicht dem zu erwartenden Layout, läuft also der Mental Map der Nutzenden zu wider.

Die interaktiven Layoutstrategien zu verwenden, hat den Vorteil, dass unsere Lösung somit theoretisch unabhängig von Veränderungen des Algorithmus ist. Dazu entfällt der Aufwand, eigene Strategien für die Phasen auszuarbeiten. Dennoch ist unser Ansatz aufgrund von hierarchischen Graphen nicht optimal: Intentional Layout kann die Breite eines Layouts verändern, die auf den Parent-Knoten angewendet werden muss, weil die interaktiven Layoutstrategien ein Layout auf Basis der Knotenkoordinaten erstellen. Allerdings ist die Berechnung der resultierenden Breite nicht trivial, da mehrere Features Einfluss auf die Breite nehmen, siehe Abschnitt 4.5. Lösen lässt sich dies über eigene interaktive Layout-Strategien, die es erlauben, direkt den Layer und die Position eines Knoten festzulegen. Diese Arbeit betrachtet keine solche eigenen interaktiven Layoutstrategien. Dies wäre ein Gegenstand zukünftiger Arbeiten, siehe Abschnitt 7.2.

Kapitel 5

Reevaluation gesetzter Constraints und Erweiterungen

Aufbauend auf dem vorigen Kapitel untersucht dieses, ob und wie eingeführte Constraints reevaluiert werden müssen, um die Mental Map der Nutzenden möglichst zu erhalten. Darüber hinaus geht es auf erweiterte Konzepte für das Intentional Layout ein. Zunächst geht Abschnitt 5.1 auf Prinzipien ein, die bei dem Erweitern des Konzepts für weitere Constraints und bei der Reevaluation zu beachten sind. Darauf untersucht Abschnitt 5.2 den Unterschied zwischen der Spezifikation von Constraints im Text-Editor und der interaktiven Spezifikation im Diagramm von KEITH. Abschnitt 5.3 betrachtet, warum Reevaluation für Position Constraints und Layer Constraints nötig ist und wie sie funktioniert. Weitere Betrachtungen zu Shifting und seine Reevaluation findet sich in Abschnitt 5.4. Aufbauend auf den vorigen Abschnitten untersucht Abschnitt 5.5 zwei Erweiterungen für Layer Constraints und Position Constraints. Abschnitt 5.6 betrachtet die Validierung von Position Constraints und Layer Constraints. Dahingegen beschäftigt sich Abschnitt 5.7 mit dem Löschen von Position Constraints und Layer Constraints und Abschnitt 5.8 sich mit Constraint Layer Constraints. Schließlich betrachtet Abschnitt 5.9 das Hinzufügen von Knoten und Kanten und Abschnitt 5.10 präsentiert Überlegungen zu relativen Constraints.

5.1. Prinzipien bei der Reevaluation und erweiterten Intentional Layout Features

Die weitgehende Erhaltung der Mental Map der Nutzenden ist ein wichtiges Kriterium für das Intentional Layout und die Reevaluation von Constraints. Hierfür sollen die Techniken zur Reevaluation sowie Erweiterungen des bisherigen Konzepts für Intentional Layout sich an gewisse Prinzipien halten:

1. **Keine nicht beabsichtigte Änderung des Layouts und der Mental Map:** Nur semantisch definierte Layout-Anpassungen sollten durchgeführt werden. Wird ein Constraint interaktiv spezifiziert, soll diese Spezifikation lediglich die Änderung an dem Layout vornehmen, die seine Semantik spezifiziert. Dies schließt folgende Punkte mit ein:
   (a) Es soll möglichst zu keinen nicht intendierten Änderungen in der Knotenordnung kommen.
   (b) Es soll möglichst zu keinen vermeidbaren nicht intendierten Änderungen der Kantenrouten kommen.
2. **Keine Constraints für unbetroffene Knoten:** Knoten ohne Constraints, die nicht das Ziel von Layout-Anpassungen sind, sollen durch die in Constraints gekapselten Layout-Anpassungen keine Constraints erhalten, außer hierfür müssten zu viele Sonderfälle betrachtet werden.
3. **Keine vermeidbaren Constraint-Änderungen:** Interaktionen verändern Constraints auf ihren Zielknoten. Gleichzeitig passen sie indirekt betroffene Constraints an, um Prinzip 1 zu erfüllen. Aller-

dings sollten sich nur Constraints verändern, wenn sich Prinzip 1 sonst nicht einhalten lässt. Die Constraints von den Interaktionen unbetroffenen Knoten sollten unverändert bleiben.

Prinzip 1 zielt darauf ab, dass sich die Layout-Anpassung möglichst an das hält, was die Mental Map der Nutzenden wahrscheinlich erwartet.

Prinzip 2 ist einerseits sinnvoll, weil das Vergrößern des Modells mit vermeidbaren Constraints das Modell weniger verständlich werden lässt, da mehr Text für das gleiche Layout kognitiv verarbeitet werden muss. Andererseits führen absolute Constraints dazu, dass die Position oder der Layer von Knoten fixiert wird. Dadurch hat der Layered Layoutalgorithmus keinen Einfluss mehr auf die jeweilige Größe des Knotens. Desto mehr Knoten also Constraints haben, je weniger Einfluss hat der Layered Layoutalgorithmus auf Position und Layer. Dadurch profitieren die Nutzenden also weniger vom automatischen Layout. Nicht intendierte Constraints sind demnach zu vermeiden.

Prinzip 3 soll undefinierte Seiteneffekte auf Constraints von der Interaktion unbetroffene Knoten vermeiden. Da solche nicht intendiert und daher für die Nutzenden schwer nachvollziehbar sind. Zudem können solche nicht intendierten Seiteneffekte auf Constraints zu einer Verletzung von Prinzip 1 führen, wenn sie das Layout in zukünftigen Folgezuständen verändern.

Die Reihenfolge der Prinzipien ist anhand ihrer Bedeutsamkeit gewählt. Bei Konflikten sollte die Wahrung von Prinzip 1 priorisiert werden, da die Erhaltung der Mental Map das vorrangige Ziel des Intentional Layouts darstellt. Stehen Prinzip 2 und Prinzip 3 in Konflikt ist Prinzip 2 zu priorisieren. Eine weitere Limitierung der Eingriffsmöglichkeiten des Layered Layoutalgorithmus sieht diese Arbeit als schlimmer an als einen Seiteneffekt auf bestehende Constraints.

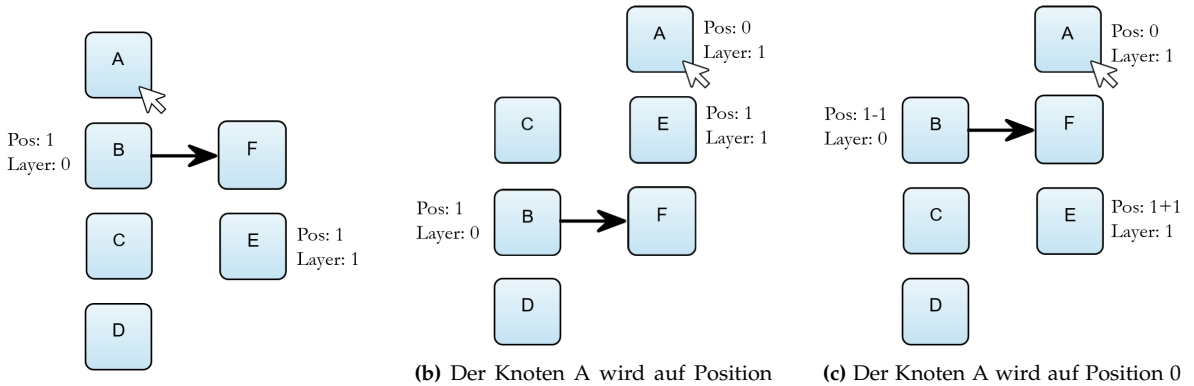
5.2. Vergleich zwischen Interaktivität und Spezifizieren im Texteditor

Es sind zwei Modi vorgesehen, um Constraints zu spezifizieren: Einerseits lassen sie sich per Interaktionen in den Diagrammen von KEITH erzeugen, andererseits direkt im Texteditor textuell im Modell spezifizieren.

Die interaktiv ausgelösten Layout-Anpassungen kapselt das Intentional Layout in Constraints im Modell. Als Folge ist es möglich, alle Anpassungen, die interaktiv durchführbar sind, ebenfalls direkt im Texteditor zu spezifizieren. Umgekehrt ist es nicht möglich, alle Constraints, die im Texteditor setzbar sind, auch interaktiv zu erzeugen: So lassen sich im Texteditor Layer Constraints setzen, die über die Anzahl an bestehenden Layern hinausgehen.

Weiterhin unterscheidet sich die Art und Weise, wie Intentional Layout bei einer Interaktion im Vergleich zum Spezifizieren im Texteditor erfolgt.

Führen die Nutzenden eine Interaktion im Diagramm von KEITH durch, entscheidet das System, welchen Constraints diese entspricht und belegt diese mit entsprechenden Werten. Darüber hinaus werden diese neuen Constraints im Kontext der bereits existierenden Constraints betrachtet. Das wird getan, weil das Hinzufügen von Constraints unbeabsichtigte Änderungen im Layout nach sich ziehen kann, wenn die bereits gesetzten Constraints nicht angepasst werden. Ein einfaches Beispiel hierfür ist, wenn zwei Knoten auf Position 0 im Layer L_1 gesetzt werden. In diesem Fall ist nicht spezifiziert, welcher Knoten zuerst platziert wird. Bei anderen Constraints kann es zu unbeabsichtigten Veränderungen in der Knotenreihenfolge kommen. Beide Fälle verletzen Prinzip 1, siehe Abschnitt 5.1. Eine zur Interaktion passende Reevaluation passt gesetzte Constraints derart an, dass sie die unbeabsichtigten Änderungen beim Setzen der neuen Constraints verhindert. Erst hiernach erhält das Modell die neuen Constraints der Interaktion und die durch Reevaluationen angepassten Constraints.



(a) Der unveränderte Beispielgraph.

(b) Der Knoten A wird auf Position 0 in Layer 1 bewegt. Es findet keine Reevaluation statt.

(c) Der Knoten A wird auf Position 0 in Layer 1 bewegt. Es findet eine Reevaluation statt.

Abbildung 5.1. Eine Positions- und Layer-Änderung wird mit und ohne Reevaluation angewendet.

Im Texteditor von KEITH spezifizieren die Nutzenden ohne Unterstützung neue Constraints. Sie müssen somit wissen, welche Constraint-Komposition das gewünschte Layout erzeugt. Sie müssen außerdem wissen, dass die Constraints weitere Veränderungen im Layout hervorrufen können, die nicht Teil ihrer definierten Semantik sind. Wenn sie beispielsweise für einen Knoten ein Layer Constraint und ein Position Constraint setzen, kann es passieren, dass dies die Knotenreihenfolge der unbetroffenen Knoten in dem Ziel-Layer verändert, vergleiche Abschnitt 5.3.1.

Eine Reevaluation beim direkten Spezifizieren im Texteditor ist nicht wünschenswert, weil sich von den Nutzenden eingetragene, vielleicht unfertige Werte während des Tippens ändern können. Dies führt zu einer Einschränkung der Nutzbarkeit des Editors. Außerdem verletzt dies Prinzip 3, siehe Abschnitt 5.1.

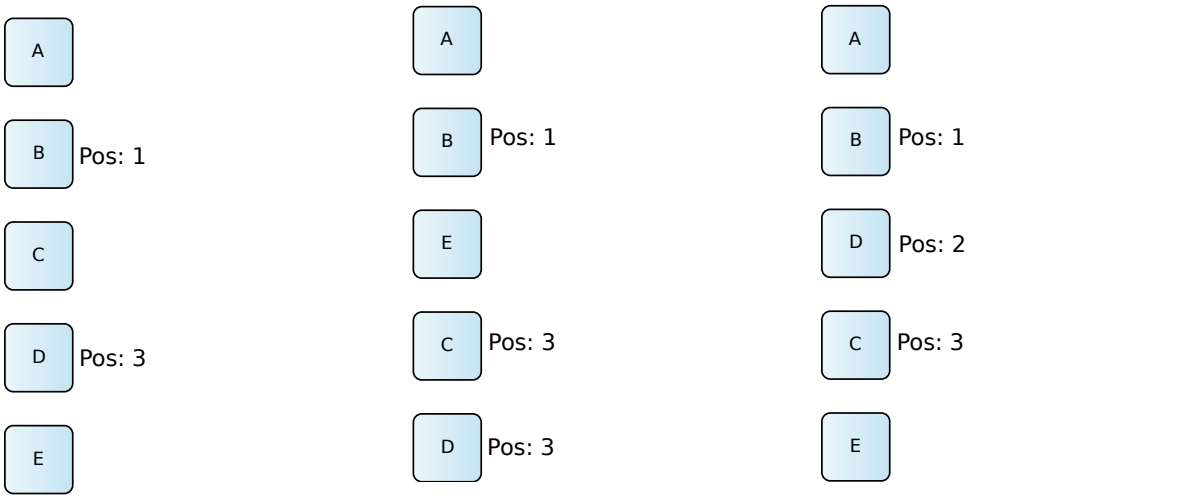
Davon abgesehen ist nicht ohne Weiteres feststellbar, was der alte Stand und was der neue Stand des Modells ist, da in KEITH bei jedem Layout-Prozess eine vollständige Synthetisierung des Modells durchgeführt wird. Dies ist der Fall, weil es in KEITH in der derzeitigen Version keine Aktualisierung auf Basis von Datei-Unterschieden gibt. Im interaktiven Modus schickt der Client die neuen Constraints dem Server in Nachrichten zu, weswegen der alte Stand des Modells eindeutig ersichtlich ist.

5.3. Reevaluation von Position und Layer Constraints

Führt eine Interaktion für einen Knoten $v \in V$ eines gelayouteten Graphen $G = (V, E, L)$ eine Positions-, Layer- oder beide Änderungen auf einmal durch, werden diese durch Position Constraints, Layer Constraints oder beide ausgedrückt. Dabei soll sich die Layout-Anpassung auf die spezifizierten Änderungen beschränken. Die genaue Bedeutung hängt von der Änderung ab.

5.3.1. Kombinierte Positions- und Layeränderung

Soll der Knoten v_i für $i \in \mathbb{N}$ eine Position in einem anderen Layer erhalten, wird sowohl ein Layer Constraint als auch ein Position Constraint gesetzt. Der Knoten v_i wird von der Position $i = P(v_i)$ aus dem Ursprungs-Layer $L_{L(v_i)} = [v_0, \ldots, v_{i-1}, v_i, v_{i+1} \ldots, v_k]$ entfernt und in den Layer $L_{LC(v_i)} = [n_0, \ldots, n_{j-1}, n_j, n_{j+1} \ldots, n_l]$ an der Position $j = PC(v_i)$ eingefügt mit $k, l \in \mathbb{N}$. Nach dem Durchführen der Änderung soll, unabhängig von den bisher gesetzten Constraints, gelten:



(a) Dies ist der unveränderte Beispielgraph.

(b) Der Knoten *C* wird auf Position 3 geschoben. Es findet keine Reevaluation statt.

(c) Der Knoten *C* wird auf Position 3 geschoben. Es findet eine Reevaluation statt.

Abbildung 5.2. Eine Interaktion die Position des Knotens *C* in seinem Layer.

$L_{L(v_i)} = [v_0, \ldots, v_{i-1}, v_{i+1} \ldots, v_k]$ und $L_{LC(v_i)} = [n_0, \ldots, n_{j-1}, v_i, n_j, n_{j+1} \ldots, n_l]$. Die Ordnung von Knoten im Ursprungs-Layer und die Ordnung der Knoten im Ziel-Layer soll also abgesehen von der Entfernung im Ersteren und dem Einfügen im Letzteren gleich bleiben. Allerdings werden diese Forderungen mitunter verletzt, wenn bereits Constraints auf Knoten in dem Ursprungs-Layer und dem Ziel-Layer gesetzt sind. Ein Beispiel für einen solchen Fall zeigt Abbildung 5.1b. Die durch das Bewegen von Knoten A auf Position 0 im Layer L_1 ausgehend von Abbildung 5.1a entsteht. Dabei kommt es in Abbildung 5.1b zu Veränderungen in der Knotenreihenfolge in Layer L_0 und Layer L_1 im Vergleich zum ursprünglichen Layout in Abbildung 5.1a.

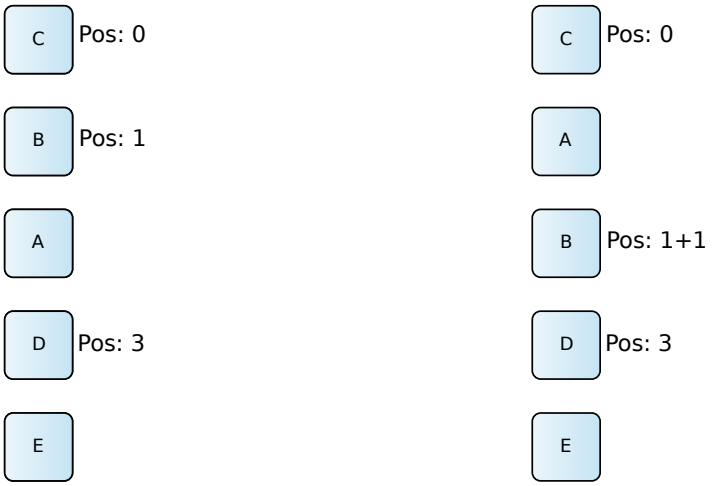
Für den Ursprungs-Layer kann die am Anfang dieses Abschnitts aufgestellte Forderung verletzt werden, wenn es einen Knoten u mit Position $P(u) > i$ gibt, der diese Position wegen eines Position Constraint innehat, da nach dem Entfernen aus dem Layer ein Knoten ohne Position Constraint die frei gewordene Position über dem Knoten u einnimmt. Dies geschieht aufgrund des interaktiven Layout-Prozesses und der Auswertung des Constraints, die immer auf das erste Layout zurückgeführt werden, siehe Abschnitt 4.3 und Abschnitt 4.4.

Im Ziel-Layer kann wiederum ein Knoten ohne Position Constraint verdrängt werden, wenn es einen Knoten w mit $PC(w) = h$ mit $h \geqslant j$ gibt. Wenn $h = j$ liegt sogar ein semantischer Konflikt vor, da es zwei Knoten mit demselben Position Constraint im Ziel-Layer gibt. Die Reihenfolge im Layer hängt daraufhin von der Sortierung in der Constraint-Auswertung ab.

Diese Verletzungen werden verhindert, wenn die Position Constraint-Werte für jeden Knoten u in $L_{L(v_i)}$ mit $P(u) > i$ und $PC(u) > i$ dekrementiert werden und Postion Constraint-Werte jedes Knoten x mit $P(x) > j$ und $PC(x) > j$ inkrementiert werden.

5.3.2. Positionsänderung im eigenen Layer

Soll ein Knoten v_i auf Position $i = P(v_i)$ im Layer $L_{L(v_i)} = [v_0, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n]$ mit $n \in \mathbb{N}$ auf eine andere Position $t <= n$ in demselben Layer geschoben werden, wird dies durch einen Position Constraint mit $PC(v_i) = t$ realisiert.



(a) Der Knoten *C* wird auf Position 0 geschoben. Es findet keine Reevaluation statt.

(b) Der Knoten *C* wird auf Position 0 geschoben. Es findet eine Reevaluation statt.

Abbildung 5.3. Dies ist eine weitere Interaktion, die die Position des Knotens *C* in seinem Layer verändert.

Nach dem Verschieben soll gelten: $L_{L(v_i)} = [v_0, \ldots, v_{k-1}, v_i, v_k, \ldots, v_n]$ mit $k \in \mathbb{N}$. Es soll also die Knotenreihenfolge im Layer abgesehen von der Positionsänderung von v_i bestehen bleiben. Ähnlich wie in Abschnitt 5.3.1 kann es beim einfachen Setzen des neuen Constraints zu einem Nachrücken oder Verdrängen von Knoten sowie zu einer Position Constraint-Wert-Kollision kommen. Abbildung 5.2b entsteht durch eine Interaktion, die ausgehend von Abbildung 5.2a Knoten *C* auf Position 0 bewegt, wodurch der Knoten *A* verdrängt wird. In Abbildung 5.2b bewegt eine Interaktion den Knoten *C* ausgehend von Abbildung 5.2a auf Position 3. Dadurch befindet sich der Knoten *E* nach der Änderung nicht mehr unter *D*. Es entsteht außerdem eine Kollision von Position Constraints, da Knoten *C* und Knoten *D* beide denselben Position Constraint innehaben.

Abstrahiert ist die Positionsänderung im eigenen Layer ein Spezialfall des Abschnitt 5.3.1. Der Knoten v_i wird zuerst aus dem Layer $L_{L(v_i)}$ entfernt und an der neuen Position j in Layer $L_{L(v_i)}$ wieder eingefügt. Die gleiche Reevaluation wie in Abschnitt 5.3.1 ist anwendbar. Diese dekrementiert die Position Constraints aller Knoten u mit $P(u) > i$ und inkrementiert die Position Constraints aller Knoten w mit $P(w) >= t$. Dies verhindert sowohl eine Veränderung in der Knotenreihenfolge als auch eine Kollision von Position Constraint-Werten. Allerdings führt dieser Ansatz zu viele Operationen durch: Denn es werden Position Constraints von gewissen Knoten sowohl dekrementiert als auch inkrementiert. Dies liegt daran, dass ein Teil der Knoten sich vor und nach der Verschiebung des Knoten immer noch unter dem Knoten beziehungsweise immer noch über dem Knoten befinden. Mit diesem Wissen und dem Bekanntsein der alten und neuen Position, lässt sich die Reevaluation anpassen: Hierbei gibt es zwei Fälle zu unterscheiden – zum einen $t > i$ zum anderen $t < i$. $t = i$ tritt nicht auf, da dieser Fall keiner Positionsänderung entspricht.

- $(t < i)$ – Die Zielposition t liegt also „über“ der ursprünglichen Position i. Daher werden lediglich die Position Constraint-Werte der Knoten inkrementiert, die nach der Verschiebung unter v_i liegen, aber vor der Verschiebung dies noch nicht taten. Das heißt, es werden die Position Constraint-Werte von jedem Knoten $o \in V$ mit $P(o) > t \wedge P(o) < i$ inkrementiert. Diese Reevaluation wird in Abbildung 5.2c durchgeführt. Den Vorzustand des Layouts zeigt Abbildung 5.2a.
- $(t > i)$ – Die Zielposition t liegt also „unter“ der ursprünglichen Position i. Es werden lediglich die

Position Constraint-Werte der Knoten dekrementiert, die nach der Verschiebung nicht mehr unter v_i liegen, aber vor der Verschiebung dies noch taten. Das heißt, es werden die Position Constraint-Werte von jedem Knoten $o \in V$ mit $P(o) <= t \wedge P(o) > i$ dekrementiert. Diese Reevaluation wird in Abbildung 5.3b durchgeführt. Den Vorzustand des Layouts zeigt Abbildung 5.2a.

5.3.3. Layer-Änderung

Es ist ebenfalls möglich, lediglich eine Layer-Änderung interaktiv zu spezifizieren. Soll ein Knoten v_i auf Position $i = P(v_i)$ im Layer $u = L(v_i)$ $L_u = [v_0, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n]$ mit $n \in \mathbb{N}$ einem anderen Layer mit Layer-Index $t = LC(v_i)$ zugewiesen werden. Für diesen Ursprungs-Layer soll nach der Verschiebung gelten: $L_u = [v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n]$. Dies ist äquivalent zum Entfernen eines Knoten aus seinem Layer wie in Abschnitt 5.3.1. Daher müssen die Position Constraint-Werte von Knoten u mit $P(u) > i$ dekrementiert werden.

Es ist nach der Definition von Layer Constraints irrelevant auf welcher Position der Knoten v_i sich nach dem Anwenden der interaktiven Layoutstrategien befindet. Da die interaktiven Layoutstrategien die Position von Knoten nicht selbst bestimmen, muss dennoch eine Position gesetzt werden.

Sollte auf v_i ein Position Constraint definiert sein, gibt es zwei Möglichkeiten: Entweder der Position Constraint wird beibehalten oder er wird gelöscht, weil er für einen anderen Layer gesetzt wurde. Wird der Position Constraint beibehalten, führt das Intentional Layout eine kombinierte Layer- und Positions-Änderung durch, wie sie Abschnitt 5.3.1 behandelt. Entscheidet man sich, ihn zu löschen, ist die Situation äquivalent zu dem Fall, dass kein Position Constraint gesetzt ist.

In diesem Fall gilt es zu entscheiden, welche Position für den Knoten v_i zu wählen ist. In der Constraint-Auswertung und Koordinatenanpassung bestimmt die Positions-Index auf welcher Position ein Knoten platziert wird, sofern kein Position Constraint für ihn existiert. Die Constraint-Auswertung platziert v_i demnach auf Position i, wenn die Position nicht durch einen Position Constraint belegt ist, oder auf die nächstmögliche Position j mit $j > i$, für die es keinen Knoten $o \in L_t$ mit $PC(o) = j$ gibt. Allerdings kann diese Entscheidung dazu führen, dass die Knotenreihenfolge im Ziel-Layer verändert wird, da der v_i somit einen anderen Knoten von seiner Position verdrängt. Dies lässt sich verhindern, indem Position Constraint-Wert jedes Knoten $u \in L_t$ mit $P(u) > j$ inkrementiert wird. Zudem kann sich die Knotenreihenfolge sich im Ausgangs-Layer verändern. Um das zu verhindern, dekrementiere alle Positions Constraints von Knoten unter der Ausgangsposition i also die Positions Constraints von Knoten $u \in L_t$ mit $P(u) > i$. Damit entspricht die Reevaluation derjenigen, die bei kombinierter Positions- und Layer-Änderung erfolgt, wie sie in Abbildung 5.1a exemplarisch gezeigt wird.

Alternativ gibt es die Möglichkeit, den Knoten v_i im Ziel-Layer auf der nächsten freien Position f zu platzieren. Damit ist $f \in \mathbb{N}$ die kleinste Position im Ziel-Layer, für die es keinen Knoten $k \in V$ mit $P(k) = f$ und $PC(k) = f$ gibt. Es gilt also für $f : \forall v \in V : P(v) \neq f \wedge PC(v) \neq f$. In diesem Fall ist keine Reevaluation notwendig, da keine Constraint-freien Knoten unter Position f vorhanden sind und v_i keinen Position Constraint erhält.

5.3.4. Layerausleerung

Sowohl bei der kombinierten Position- und Layer-Änderung als auch bei der Layer-Änderung kann es sein, dass die Nutzenden einen Knoten verschieben, der der einzige Knoten seines Layers ist. Dies stellt kein Problem für die beschriebenen Konzepte dar und führt zu keinen nicht spezifizierten Layout-Veränderungen. Visuell rücken, wenn ein Layer L_i mit $i \in \mathbb{N}$ interaktiv ausgeleert wird, die Layer rechts von diesem Layer, also mit Layer-Index $j > i$ eine Position nach links. Dies lässt sich auf

dem Modell nachempfinden, indem man für alle Knoten $v \in V$ mit definiertem Layer Constraint und $LC(v) > i$ den Layer Constraint-Wert dekrementiert.

Das bis dato betrachtete Konzept für Intentional Layout hat eine Schwäche, die diese Reevaluation notwendig macht: Die Bestimmung von Constraint-Werten im Client wird anhand der Positions-Indices und Layer-Indices vorgenommen, siehe Kapitel 2. Leert eine Interaktion einen Layer aus, führt dies zu fehlerhafter Setzung von Layer-Constraints, wenn Knoten rechts des ausgeleerten Layers platziert werden.

Mit der Erweiterung für Layer Constraints mit Wert $v > |Layers|$, die in Abschnitt 5.5.1 betrachtet wird, ist diese Reevaluation nicht mehr notwendig.

Sie durchzuführen hat den Vorteil, dass die beobachtbare Entwicklung auch auf das Modell übertragen wird. Allerdings verletzt dies Prinzip 3. Es besagt, dass keine Constraint-Änderungen vorgenommen werden sollen, die sich vermeiden lassen. Nichtsdestotrotz führt diese Anpassung zu keiner Änderung im Layout und damit zu keiner der Mental Map. Zudem vergrößert sich das Modell nicht, da keine Constraints hinzukommen. Prinzip 1 und Prinzip 2 werden also erhalten. Demnach handelt es sich um eine annehmbare Verletzung der Prinzipien, siehe Abschnitt 5.1.

Konsequenterweise sollte diese Reevaluation durch eine Option ausschaltbar sein, damit das Intentional Layout möglichst prinzipientreu bleibt.

5.4. Genauere Betrachtung von Shifting

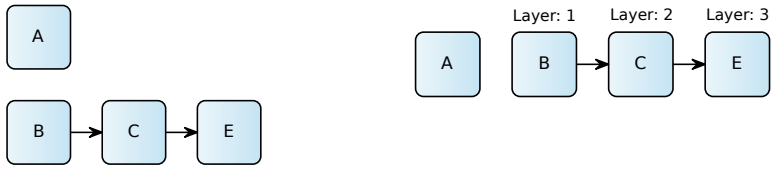
Den Layer eines Knoten in einem gelayouteten Graphen anpassen zu können, führt, wie Abschnitt 4.4.1 erläutert, zu der Notwendigkeit von Shifting. Im Zusammenhang mit gesetzten Constraints muss betrachtet werden, wie Shifting mit Knoten mit Constraints umgehen soll. Außerdem wird der Grund für die gewählte Shifting-Strategie beleuchtet.

5.4.1. Hintergrund der Shifting-Strategie

Abschnitt 4.4.1, der das Shifting im interaktiven Layout-Prozess einführt, legt fest, dass in unserem Konzept für Intentional Layout Knoten stets nach rechts geshiftet werden. Darüber hinaus hat die Wahl der Shifting-Strategie für den Unterschied zwischen Interaktivität und Texteditor, siehe Abschnitt 5.2 sowie für die Reevaluation Relevanz.

Beginnen Pfade im ersten Layer oder enden im letzten Layer kann es je nach Shifting Strategie notwendig sein, einen Knoten in einen neuen Layer am linken Rand oder am rechten Rand zu shiften. Das Layering eines Layouts muss also am linken Rand oder am rechten Rand durch die Einführung eines neuen Layers erweiterbar sein. Shiftet man stets nach rechts, muss dies nur am rechten Rand geschehen. Shiftet man sets nach links, muss dies nur am linken Rand möglich sein. Bei bedingtem Shiften ist wahrscheinlich beides nötig, dieser Fall wurde im Zuge dieser Arbeit jedoch nicht betrachtet.

Im aktuellen Konzept für Constraint-Auswertung könnte ein neuer Layer am linken Rand als neue Layer-Liste mit Layer-Index 0 eingeführt werden. Angenommen dies sei möglich, ohne den Knoten ohne Layer Constraint ein Layer Constraint geben zu müssen. Damit bleibt Prinzip 2 aus Abschnitt 5.1 gewahrt. Dann müssen die bereits eingeführten Layer Constraints inkrementiert werden, um die Mental Map beizubehalten. Ohne diese Reevaluation platziert der Layout-Prozess alle Knoten, die nach dem Hinzufügen der neuen Layer-Liste in ihre Layer-Listen hinzugefügt werden sollen, eine Layer-Liste zu weit links. Was daran liegt, dass die Layer Constraints auf Basis der alten Layer-Nummerierung erstellt wurden. Zudem entsteht eine Inkonsistenz, da die Constraintauswertung einen Teil der Layer Constraints nach der alten Nummerierung auswertet und einen anderen Teil nach der Neuen. Die Reevaluation verhindert das erste Problem und die Inkonsistenz. Beide verletzen Prinzip



(a) Dies ist der Ausgangsgraph, auf dem Shifting stattfindet.

(b) Shifting wird durch Layer Constraints fixiert.

Abbildung 5.4. Shifting mit Setzen von Layer Constraints.

1, da kein einfach vorhersehbares Layout entsteht, es also nicht mit der Mental Map konform sein kann, siehe Abschnitt 5.1. Allerdings wird das Shifting während des interaktiven Layout-Prozesses durchgeführt, also nach dem Reevaluieren und Setzen der Constraints. Demnach ändert sich das Modell unabhängig vom Eingriff der Nutzenden während des Layouts. Als Standardoption stellt dies keine gute Wahl dar, wie Abschnitt 5.2 herausstellt. Demnach ist die Entscheidung immerzu nach rechts zu Shiften, auf dem aktuellen Stand des Konzepts sinnvoll: Auf diese Weise ist sichergestellt, dass sich das Modell nicht während des interaktiven Layout-Prozesses ändern muss.

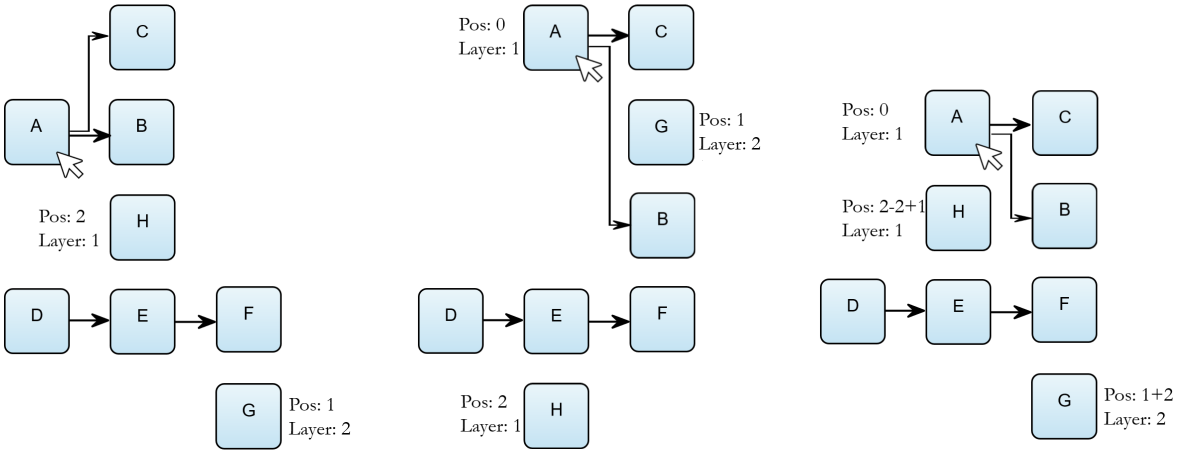
5.4.2. Adjazente Knoten mit Layer Constraint beim Shifting

Shifting, wie es Abschnitt 4.4.1 einführt, ist notwendig, wenn eine Layer-Änderung einen Knoten v in einen Layer L_t bewegt, in dem es zu v adjazente Knoten gibt. Hierbei gilt es zu erarbeiten, ob eine solche Aktion erlaubt sein soll, wenn sich im Layer L_t zu v adjazente Knoten mit Layer Constraint befinden. Angenommen das ist erlaubt. Das heißt, dass jene Layer Constraints geupdatet werden müssen, da ihre Knoten aufgrund des Verbots von flachen Kanten geshiftet werden müssen. Ein Layer Constraint wird auf einem Knoten gesetzt, damit das Layout seine Positionierung in diesem Layer erzwingt. Den hierbei gesetzten Layer zu ändern, widerspricht Prinzip 3, vergleiche Abschnitt 5.1. Dieses sieht vor, dass Interaktionen möglichst keine Constraints von nicht direkt betroffenen Knoten ändern sollen. In diesem Fall wird die Intention hinter den Layer Constraints sogar zerstört. Als Folge kann in Folgelayouts eine Veränderung der Mental Map stattfinden, was Prinzip 1 verletzt, vergleiche Abschnitt 5.1.

Demnach ist es nicht sinnvoll, eine solche Aktion vorzunehmen, außer sie geschieht unter Einbeziehung der Nutzenden beispielsweise als erweiterte Option. Das Spezifizieren von Layer Constraints für einen Knoten v für einen Layer mit Layer-Index l, ist standardmäßig also nur dann gestattet, wenn es in Layer L_l keine Knoten mit Layer Constraint gibt.

Dazu kommt eine weitere Beobachtung: Sei s ein Knoten, der aufgrund einer Layer-Änderung geshiftet werden muss. Dann soll dieser in den nächsten rechten Layer geshiftet werden, in dem sich kein zu Knoten s adjazenter Knoten a mit einem Layer Constraint befindet. Ansonsten müsste nach der zugrundeliegenden Strategie dieser Knoten a geshiftet werden. Obige Betrachtung hat jedoch ergeben, dass Layer Constraints sich nicht ohne Intention der Nutzenden ändern sollten.

Unser Konzept vermeidet das Shiften von Knoten mit Layer Constraints in der Koordinaten-Auswertung aus Abschnitt 4.4 in Schritt 2, indem die Knoten mit Layer Constraints sortiert werden. Die Constraint-Auswertung fügt Knoten mit Layer Constraints demnach nicht nur nach den Knoten ohne Layer Constraints, sondern auch nach Knoten mit kleineren Layer Constraint-Werten zu ihrer Layer-Liste hinzu. Dies führt dazu, dass Knoten des gleichen Pfades, die einen Layer Constraint haben, nicht geshiftet werden.



(a) Dies ist der Ausgangsgraph, der zur Erklärung der Reevaluation zum Shifting dient.

(b) Das Setzen eines Layer Constraints auf Knoten A führt zu Shifting ohne Reevaluation.

(c) Das Setzen eines Layer Constraints auf Knoten A führt zu Shifting mit Reevaluation.

Abbildung 5.5. Diese Grafik zeigt auf, welche unbeabsichtigten Layout-Änderungen bei Shifting auftreten können und wie Reevaluationen sie verhindern können.

5.4.3. Fixieren von Shifting

Abschnitt 4.4.1 erläutert, dass das Setzen eines neuen Layer Constraints auf einem Knoten, dessen voriger Layer Constraint zu Shifting geführt hat, dazu führt, dass die geshifteten Knoten ihre Position verändern. Zur Vermeidung kann man Shifting durch Layer Constraints fixieren, wie es Abbildung 5.4 zeigt. In Abbildung 5.4a führt das Setzen eines Layer Constraint mit Wert 1 auf Knoten B zu Shifting von Knoten C nach L_2 und von Knoten E nach L_3. Sollen B und C in Folgelayouts nicht ihre Position ändern, lässt sich dies durch Setzen entsprechender Layer Constraints für diese Layer gewährleisten, siehe Abbildung 5.4b. Formalisiert heißt dies: Wenn das Setzen eines Layer Constraints auf einem Knoten $v \in V$ zu Shifting von einem Knoten $s \in V$ nach $L_l \in L$ mit $l \in \mathbb{N}$ führt, wird ein Layer Constraint mit $LC(s) = l$ gesetzt.

Diese Idee verletzt Prinzip 2, siehe Abschnitt 5.1. Deswegen sollte sie nicht standardmäßig beim Shifting erfolgen.

5.4.4. Reevaluation beim Shiften

Die Bewegung des Knoten v in den Layer L_t entspricht der kombinierten Layer- und Positionsänderung aus Abschnitt 5.3.1. Allerdings bleibt es nicht bei dieser Änderung. Knoten in L_t, die zu v *adjazent* sind, müssen geshiftet werden. Sie werden also von L_t in L_r mit $r \geqslant t + 1$ bewegt. r ist hierbei der kleinste Layer rechts von t, in dem es keinen zu v adjazenten Knoten mit Layer Constraint gibt.

Hat der Knoten s einen Position Constraint, sollte dieser beibehalten werden, da die Verschiebung des Knoten v keine Entfernung von Position Constraints einbezieht und diese daher nicht nachvollziehbar ist. Die Position im Ziel-Layer entspricht dann dem Position Constraint. Hat der Knoten v keinen Position Constraint, sollte der Knoten mit demselben Positions-Index in den Layer L_r angeordnet werden, die er zuvor hatte, damit der Knoten, sofern es keinen anderen Knoten $a \in L_r$ mit $PC(a) = P(v)$ gibt, auf derselben Position verbleibt, damit die Kantenrouten möglichst gleich bleiben. Dies gilt insbesondere für gerade Kantenrouten, die ohne einen solchen Knoten a gerade

bleiben. So kommt es zu weniger Veränderungen im Layout und somit auch zu weniger Änderungen in der Mental Map.

Im Ziel-Layer L_r gilt es wiederum zu prüfen, ob es zu dem geshifteten Knoten adjazente Knoten gibt. Für diese muss der Prozess wiederholt werden. Wie Abbildung 5.5b zeigt, kann Shifting zur Veränderung der Knotenreihenfolge im Ausgangs- und Ziel-Layer führen. In diesem Fall befindet sich Knoten E in Abbildung 5.5b anders als in Abbildung 5.5a nicht mehr unter Knoten H und Knoten G nicht mehr unter Knoten F.

Naive Reevaluation

Die naive Lösung ist analog zum Abschnitt zur Layer-Änderung und lässt sich an Abbildung 5.5c nachvollziehen: Sei $s \in V$ ein zu shiftender Knoten in einem Layer mit Layer-Index u auf Position $p \in \mathbb{N}$. Dann werden die Position Constraint-Werte aller Knoten $u \in L_u$ dekrementiert, für die gilt $P(u) \leqslant p$. Im Ziel-Layer wiederum werden die Position Constraint-Werte aller Knoten u_2 inkrementiert, für die gilt $P(u_2) \geqslant p$. Dabei sind die Position Constraints der geshifteten Knoten beizubehalten. In Abbildung 5.5c wird diese naive Reevaluation für ein Layout vorgeführt, in dem eine Interaktion einen Knoten A von Poisition 0 aus L_0 auf Position 0 in L_1 bewegt. Dadurch muss der Layout-Prozess die Knoten C und B shiften. Daher wird der Positions Constraint von Knoten H wegen der Knoten C und B zwei Mal dekrementiert und wegen des Knotens A einmal inkrementiert. Dies ist der Fall, weil B und C aus dem Layer von H geshiftet werden und A in den Layer von H bewegt wird. Der Position Constraint von G wird hingegen zwei Mal inkrementiert, weil C und B über seiner Position in den Layer von G geshiftet werden.

Effizientere Reevaluation

Ähnlich wie bei der Positionsänderung ohne Layer-Änderung kommt es mitunter zu überflüssigen Rechenoperationen. Beim Shiften rückt häufig ein Knoten auf die Ausgangsposition eines geshifteten Knotens nach. Ist dies der Fall, muss es für diese Position keine Reevaluation geben. Die Reevaluation bei Layer-Änderungen, die Shiften nach sich ziehen, funktioniert wie folgt:

1. Seien LP_s und LP_t Listen von Paaren. Diese Paare haben die Form (l, p), wobei p ein Positions-Index und l ein Layer-Index ist. Im Weiteren heißen Paare dieser Form *Layer-Positionen-Paare*. LP_s sammelt die Layer-Positionen-Paare, von denen Knoten wegbewegt werden. LP_t sammelt solche Layer-Positionen-Paare, auf die Knoten bewegt werden.

2. Müssen nun Knoten in L_l geshiftet werden, lege ihre Layer-Position Paare in LP_s ab, bestimme das Folge-Paare, wo sie hin geshiftet werden, und lege diese in LP_t ab. Prüfe daraufhin, ob das Shiften zu erneutem Shiften führt und wiederhole das Aufsammeln der Paare.

   Für das Beispiel in Abbildung 5.4, in dem der Knoten A auf Position 0 in Layer L_1 geschoben wird, bedeutet dies: $LP_s = [(0,0),(1,0),(1,1)]$ und $LP_t = [(1,0),(2,0),(2,1)]$.

3. Prüfe zusätzlich, ob Knoten auf ihre ursprüngliche Position zurückwandern können: Ist dies der Fall, lege für diese Knoten ein Paar in LP_s mit ihrer Ausgangslage und ein Paar in LP_t für ihre Ziel-Position im Ziel-Layer an. Im Beispiel gibt es keine Knoten, die auf ihre initiale Position gesetzt werden.

4. Nun entferne alle Paare aus LP_s und LP_t, die sich in $LP_s \cap LP_t$ befinden. Denn gibt es ein Paar sowohl in LP_s als auch LP_t bedeutet dies, dass eine Position in einem Layer, von der ein Knoten wegbewegt wird, wieder aufgefüllt wird. Somit ist für diese Position keine Reevaluation nötig.

Für das Beispiel bedeutet dies: $LP_s \cap LP_t = [(1,0)]$ also $LP_s = [(0,0),(1,1)]$ und $LP_t = [(2,0),(2,1)]$.

5. Hiernach wird jedes Paar LP_s so wie eine Position in einem Layer reevaluiert, von der ein Knoten aus seinem Layer bewegt wird, und jedes Paar in LP_t so wie eine Position in einem Layer, auf die ein Knoten bewegt wird. Dies wird in Abschnitt 5.3.3 für Knoten beschrieben, deren Layer geändert wird. Werden Position und Layer geändert, findet sich die Reevaluation in Abschnitt 5.3.1.

   Im betrachteten Beispiel gilt $LP_s = [(0,0),(1,1)]$ und $LP_t = [(2,0),(2,1)]$. Die Position Constraints werden unter den beschriebenen Layer-Positionen-Paaren reevaluiert. Für das Beispiel heißt das: Der Position Constraint von Knoten H wird dekrementiert. Der Position Constraint von G wird zwei Mal inkrementiert. Ohne Schritt 3 wird der Position Constraint von Knoten H zusätzlich einmal inkrementiert und dekrementiert.

Diese effizientere Reevaluation bezieht ebenfalls die Reevaluation für den durch die Interaktion bewegten Knoten ein. Will man hingegen diese Reevaluationen getrennt abarbeiten, dürfen für den Zielknoten der Interaktion keine Layer-Positionen-Paare erstellt werden.

5.5. Erweiterung von Layer Constraints und Position Constraints

Bis dato betrachtet das Konzept lediglich Position und Layer Constraints, die einen Knoten maximal die letzte Position in einem Layer und maximal in einen Layer hinter dem letzten Layer einordnen. Zudem ist es bis dato nur vorgesehen, Layer Constraints mit Wert $v \geqslant 0$ zu spezifizieren. Diese beiden Limitierungen werden in diesem Abschnitt aufgehoben, damit Constraints zusätzliche Bedeutungen transportieren können.

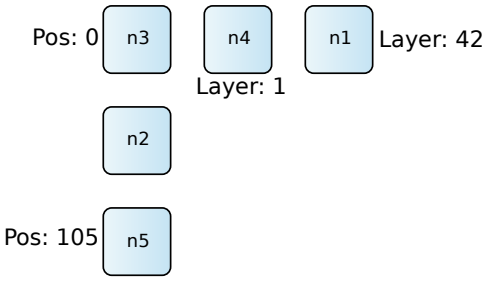


Abbildung 5.6. Ein Graph mit Positionen überspringende Position Constraints und Layer überspringende Layer Constraints

5.5.1. Layer und Positionen durch absolute Constraints überspringen

Das bis dato betrachtete Konzept für absolute Constraints behandelt Position Constraints und Layer Constraints so, dass sie einem Knoten v nur Positionen p und Layer mit Layer-Index l mit $p <= |L_l|$ und $l <= |L|$. Es lassen sich ergo keine leeren Positionen und leeren Layer durch Constraints „überspringen“. Das ist problematisch, da sie sich im Texteditor definieren lassen, unabhängig davon, ob das interaktiv im Diagramm der Fall ist, vergleiche Abschnitt 5.2. Genauer betrachtet handelt es sich also um absolute Constraints mit folgenden Eigenschaften:

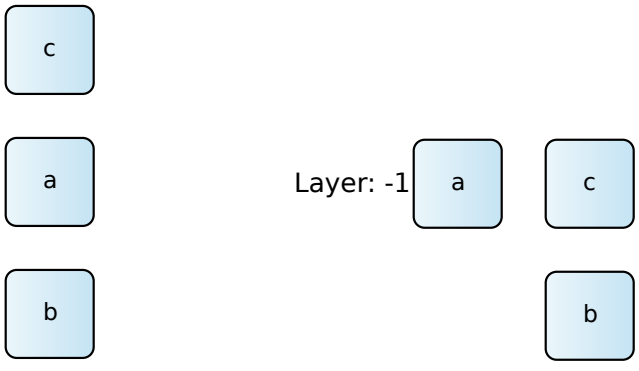
- Position Constraints, die einem Knoten v eine Position p in ihrem Layer $L_{L(v)}$ mit $p > |L_{L(v)}|$ zuweisen.

▻ Layer Constraints, die einem Knoten v mit einem Layer Constraint-Wert lc versehen, für den $lc > |L|$ gilt.

Solche Constraints könnte man theoretisch verbieten, da sie streng genommen dem Knoten v keine valide Position beziehungsweise keinen validen Layer zuordnen. Sie zu erlauben, hat jedoch den Vorteil, dass sich so Constraints mit einer zusätzlichen semantischen Bedeutung definieren lassen. Position Constraints für einen Knoten v mit einem Wert p mit $p > |L_{L(v)}|$ drücken aus, dass $p - |L_{L(v)}|$ zusätzliche Knoten zwischen dem mit dem Constraint versehenen Knoten und dem Knoten mit der nächst kleineren Position platziert werden könnten. Layer Constraints für einen Knoten v mit einem Wert l mit $l > |L|$ drückt aus, dass es $l - |L|$ weitere Layer zwischen L_l und dem linken Nachbar-Layer geben kann. Weiterhin lässt sich durch einen solchen Constraint ein Knoten in einen neuen Layer am rechten Rand des Layouts platzieren.

Solche Constraints zuzulassen, hat Konsequenzen für das Setzen von Constraints im Client: Wollen die Nutzenden einen Position Constraint für einen Knoten v für eine Position $p > 0$ im Layer L_l interaktiv festlegen, für den gilt, dass der Knoten n auf Position $p - 1$ einen Position Constraint mit Wert $PC(n) > |L_l|$ hat, muss der Wert des neuen Position Constraint anhand dieses Werts bestimmt werden. Gilt $L(v) = L(n)$ und $P(v) < P(n)$ muss also gelten $PC(v) = PC(n)$ und in allen anderen Fällen $PC(v) = PC(n) + 1$. Es muss nur der nächste Knoten über der Zielposition betrachtet werden, da Positions Constraints mit einem Wert $val > |L_l|$ keine Knoten unter sich haben können, die keinen Position Constraint haben.

Soll dahingegen ein Knoten $w \in V$ interaktiv in Layer l verschoben werden, muss hierfür betrachtet werden, ob es einen Knoten mit einem wie oben beschriebenen Layer Constraint gibt. In diesem Fall legt nicht der Wert l allein fest, welchen Wert der Layer Constraint erhalten muss, sondern der Abstand zwischen dem linksnächsten Layer, der durch einen solchen Layer Constraint betroffen ist, und der Wert dieses Layer Constraints. Formal heißt dies: Bestimme den Knoten $b \in V$ mit $L(b) < l$, für den gilt, dass $l - L(b)$ minimal ist. Dann muss gelten, dass $LC(w) = LC(b) + l - L(b)$.



(a) Dieser Beispielgraph dient als Ausgangslage für negative Layer Constraints.

(b) Ein negativer Layer Constraint platziert den Knoten a in einem neuen Layer am linken Rand des Layouts.

Abbildung 5.7. Beispiel für das Einfügen eines Layers am linken Rand des Layouts durch einen negativen Layer Constraint.

5.5.2. Negative Layer Constraints

Im derzeitigen Stand des Konzept lassen sich lediglich Knoten in neuen Layern am rechten Rand des Layouts platzieren. In Abbildung 5.7 zeigt sich, dass eine Erweiterung von Layer Constraints um negative Werte dies am linken Rand ermöglicht. Wird ein Layer Constraint für einen Knoten n auf einen Wert $v < 0$ gesetzt, dann soll n in den $|v|$-ten *Layer* links des Layers L_0 platziert werden. Beispielsweise lässt sich der Knoten a in Abbildung 5.7a durch einen Layer Constraint mit $LC(a) = -1$ in den ersten Layer links von L_0 platzieren. Diese Anpassung muss in der Constraint-Auswertung und Koordinaten-Anpassung einbezogen werden. Für jeden negativen Constraint-Wert muss am linken Rand der Liste von Layer-Listen eine neue Liste erstellt werden. Dies funktioniert auf folgende Weise: Zunächst ist dafür der minimale Layer Constraint-Wert ml zu finden, um dann eine Liste von leeren Layer-Listen mls mit $|mls| = |ml|$ zu erstellen. Hiernach konkateniere diese Liste von Layer-Listen mit der restlichen Liste von Layer-Listen. Damit die ursprüngliche 0-te Layer-Liste bekannt bleibt, sollte ein Feld $firstLayer = |mls|$ angelegt werden. Dieser Wert lässt sich dazu nutzen, die Layer-Liste zu bestimmen, in die ein Knoten mit einem Layer Constraint-Wert $v > 0$ gehört. Werden die Layer-Indices im Layout-Prozess nicht für negative Layer erweitert, ist der Wert $firstLayer$ auch für die Constraintwert-Berechnung im Client und Reevaluation relevant. Die gewählte Vorgehensweise die Layer-Listen-Verwaltung für negative Layer Constraint-Werte auszudehnen, hat den Vorteil, dass die Constraint-Auswertung und Koordinatensetzung aus Abschnitt 4.4 nicht angepasst werden muss.

Allerdings muss das Setzen des richtigen Constraints auf der Client-Seite erweitert werden: Wird ein Knoten n in einem Layer links des links vom L_0 platziert, sollte der Client einen Layer Constraint senden, der sich am nächsten rechten Layer orientiert oder an dem Layer, in dem der Knoten platziert wird.

5.6. Validierung des Modells mit Position Constraints und Layer Constraints

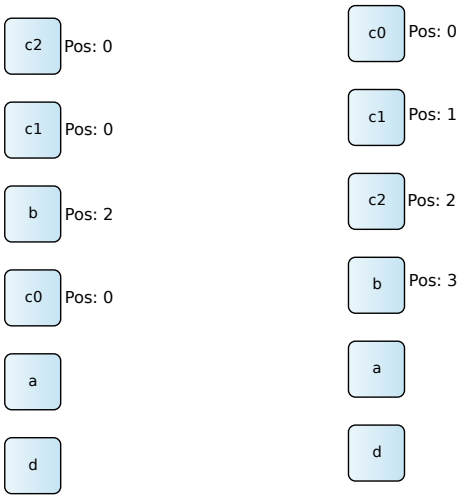
Während davon ausgegangen wird, dass interaktives Spezifizieren von Constraints das Modell immer in einem validen Zustand hinterlässt, lassen sich im Texteditor Constraints definieren, die aufgrund anderer Constraints nicht valide sind. Bei den absoluten Constraint-Typen Position Constraint und Layer Constraint gibt es zwei Varianten von invaliden Constraints:

1. **Kollision von Position Constraints:** Es darf immer nur einen Knoten geben, der auf eine Position p in einem Layer L_l mit $l \in \mathbb{N}$ durch einen Layer Constraint gesetzt wird. Denn die Semantik von Position Constraints sieht mehrere Knoten mit demselben Wert nicht vor, wodurch es keine eindeutige Ordnung für Knoten mit demselben Position Constraint-Wert in einem gemeinsamen Layer gibt. Ein Beispiel hierfür zeigt Abbildung 5.8.

2. **Adjazente Knoten mit demselben Layer Constraint-Wert:** Wie in Abschnitt 5.4 erläutert, sollten Layer Constraint nicht geshiftet werden. Demnach ist es verboten, adjazente Knoten mit Layer Constraints in einen gemeinsamen Layer zu setzen.

5.6.1. Invalide absolute Constraints deaktivieren

Kommen solche invaliden Constraints im Modell vor, gibt es kein korrektes Layout. Um ein solches zu gewährleisten, empfiehlt es sich, die Invalidität verursachenden Constraints zu deaktivieren. Sie sind also nicht im Layout-Prozess zu erfüllen, aber auch nicht aus dem Modell zu löschen. Solche



(a) Die Knoten $c0$, $c1$ und $c2$ verfügen über den gleichen Position Constraint. Dies ist eine Invalidität.

(b) Hier wird die Position Constraint Kollision aus Abbildung 5.8a mit der Technik aus Abschnitt 5.6.2 gelöst.

Abbildung 5.8. Eine exemplarische Kollision von Position Constraints und ihre Lösung.

Constraints zu löschen, verstößt gegen das Prinzip 3, also dagegen Constraints auf dem Modell nicht zu verändern, wenn dies vermeidbar ist, siehe Abschnitt 5.1. Außerdem verstößt es gegen die Überlegung, dass eine standardmäßige Reevaluation im Modell nicht ratsam ist, vergleiche Abschnitt 5.2.

Bei kollidierenden Position Constraints sollte nur das erste Auftreten eines Knotens v mit einer Position p für einen Layer L gültig sein. Alle Weiteren sind zu ignorieren.

Damit es keine adjazenten Knoten mit demselben Layer Constraint im Modell gibt, muss entschieden werden, welche Constraints bei einer Kollision von Position Constraints deaktiviert werden sollen. Eine Möglichkeit ist, alle ausgehenden Kanten jedes Knoten zu betrachten: Falls auf dem Target ein Layer Constraint für den Layer der Source gesetzt ist, kann man den Layer Constraint des Targets deaktivieren. Analog ist dies mit den eingehenden Kanten jedes Knoten möglich. Beide Mengen, also die eingehenden und ausgehenden Kanten, zu betrachten, stellt zu viel Rechenaufwand dar, da die Vereinigung aller eingehenden Kanten aller Knoten und die Vereinigung aller ausgehenden Kanten aller Knoten jeweils die Menge alle Kanten eines Graphen darstellt.

5.6.2. Korrigieren von invaliden absoluten Constraints

Automatisch die invaliden Constraints durch Veränderung zu korrigieren, ist ohne Zutun der Nutzenden nicht sinnvoll, siehe Abschnitt 5.2. Zudem verstößt dies gegen Prinzip 3 aus Abschnitt 5.1. Als Zusatzoption ist das Korrigieren jedoch denkbar. Kommt es zu kollidierenden Position Constraints, gibt es keine festgelegte Ordnung auf den Knoten mit dem gleichen Position Constraint-Wert. Es ist aber klar, dass sie ohne Unterbrechung untereinander zu platzieren sind. Demnach lässt sich die Kollision wie folgt lösen:

1. **Knoten einer Kollision sammeln:** Gibt es einen kollidierenden Position Constraint-Wert col in einem Layer L_l mit $l \in \mathbb{N}$, dann sammelt man alle Knoten $v \in L_l$ mit $PC(v) = col$ in einer Liste $k_l = [v_0, \ldots, v_t]$ für $t \in \mathbb{N}$.

2. **Kollision auflösen:** Man löst die Kollision durch Iteration über diese Liste und Addition des

Listenindex i des jeweiligen Knotens v_i auf seinen Position Constraint.

3. **Reevaluation:** Es kann im Modell Knoten geben, die einen Layer Constraint haben, der kleiner als t und größer als col ist. Durch das Auflösen können also weitere Kollisionen entstehen. Zudem kann es zu einer Veränderung in der Knotenordnung kommen, wenn Knoten Position Constraints haben, deren Werte größer-gleich t sind. Hat ein Knoten v einen Position Constraint mit $col < PC(v)$, dann erhöht man seinen Position Constraint um $t - PC(v)$, falls $PC(v) < t$ gilt, und sonst um t.

4. **Korrektur des gesamten Modells** durch wiederholtest Vorgehen für jeden Position Constraint-Wert, für den es zu Kollisionen kommt.

In Abbildung 5.8b löst diese Technik die Kollision aus Abbildung 5.8a. Soll eine automatische Nachbearbeitung adjazente Knoten mit einem Layer Constraint durch Wert-Änderung korrigieren, muss entschieden werden, welche Knoten im Layer verbleiben sollen und welche nicht. Hierbei stellt sich die Frage, wohin letztere Knoten bewegt werden sollen. Beim Verschieben eines der Knoten in einen anderen Layer und der folgenden Anpassung des Layer Constraint kann wiederum eine invalide Situation des zu behebenden Typs auftreten. Er tritt sogar zwangsläufig auf, wenn die Nachbearbeitung mehrere betroffene Knoten in den gleichen Ersatz-Layer schiebt. Dies lässt sich prinzipiell durch Shifting auflösen, allerdings hat Abschnitt 5.4.2 festgehalten, dass das Shiften von Knoten mit Layer Constraints nicht sinnvoll ist, da es die intendierte Semantik zerstört. Das gleiche Argument gilt generell für das Verändern von Layer Constraints. Position Constraint legen implizit fest, dass Knoten zwischen anderen Knoten zu platzieren sind. Daher lassen sich kollidierende Position Constraints korrigieren. Für Layer Constraints gibt es eine solche vom absoluten Wert losgelöste Semantik nicht.

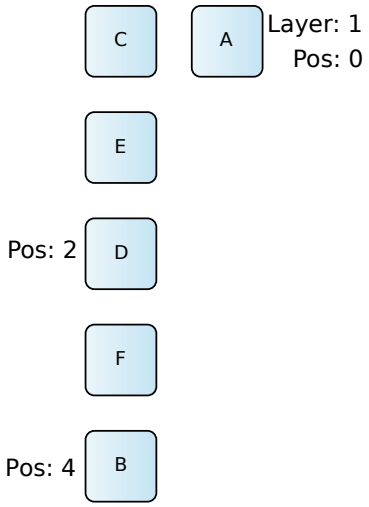
5.7. Position und Layer Constraints löschen

Da sich Position und Layer Constraints interaktiv setzen lassen, bietet sich an, diese interaktiv löschen zu können. Angenommen dies sei möglich.

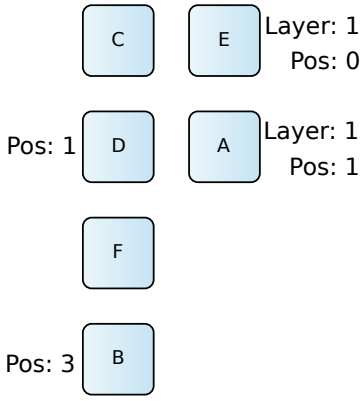
Seien auf einem Knoten b ein Position Constraint und ein Layer Constraint mit $LC(b) = lc$ und $PC(b) = pc$ gesetzt. Löscht man nun den Layer Constraint von b, wird der Layout-Prozess b im Folgelayout in dem Layer platzieren, in dem b sich im initialen Layout befindet. Dies ist nicht möglich, falls es in diesem Layer einen zu b adjazenten Knoten a gibt, der entweder in den Layer geshiftet wurde oder sich wegen eines Layer Constraints mit $LC(a) = lc$ in ihm befindet. In beiden Fällen wird b geshiftet, siehe Abschnitt 4.4.1. Löscht eine Interaktion den Position Constraint von b, wird der Layout Prozess b auf der Position platzieren, die b im initialen Layout hat, sofern diese frei ist. Ist sie durch einen Knoten a mit Position Constraint $PC(a) = pc$ belegt, platziert der Layout Prozess b auf der nächsten Position im Layer, auf der es keinen Knoten mit Position Constraint gibt.

Je nachdem, ob ein Layer Constraint, ein Position Constraint oder beides gelöscht wird, findet also eine Layer-Änderung, eine Position-Änderung oder beide statt. Wie zuvor betrachtet, können solche Änderungen unbeabsichtigte Veränderungen in der Knotenreihenfolge eines Layers zur Folge haben. Diese lassen sich durch die bereits betrachtete Reevaluationen verhindern. Die verschiedenen Reevaluationen finden sich in Abschnitt 5.3 und für Reevaluation bei Shifting in Abschnitt 5.4.4.

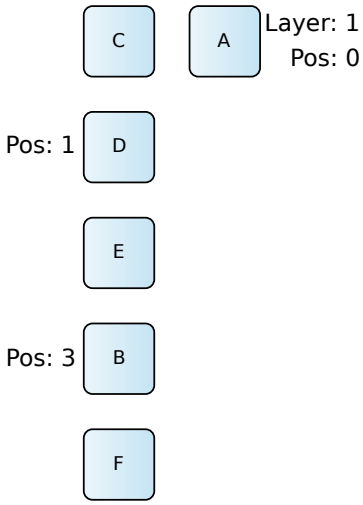
Ein Beispiel hierfür zeigt Abbildung 5.9: Ausgehend von Abbildung 5.9a erhält Knoten E ein Layer Constraint und ein Position Constraint, so dass $LC(E) = 1$ und $PC(E) = 0$ gilt. Das führt zum Folgelayout in Abbildung 5.9b. Daraufhin löscht eine Interaktion beide Constraints. Ohne Reevaluation führt dies zu Abbildung 5.9c, in dem E den Knoten F auf eine Position unter B verdrängt. Wie Abbildung 5.9d zeigt, lässt sich durch eine Reevaluation nach Abschnitt 5.3.1 lösen. Die Position



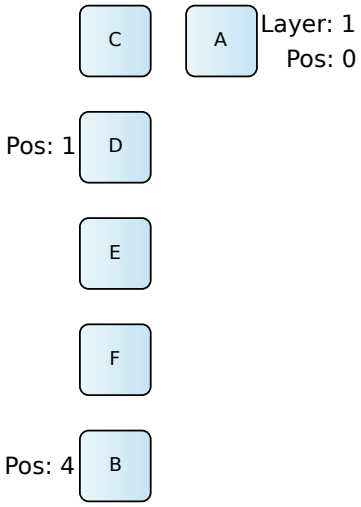
(a) Dies ist der Ausgangsgraph, auf dem ein Constraint gesetzt wird, das gelöscht werden soll.



(b) Dieses Folgelayout kommt durch das Setzen von Constraints auf Knoten *E* zustande.



(c) Dieses Folgelayout entsteht durch das Löschen der Constraints von Knoten *E* ohne Reevaluation.

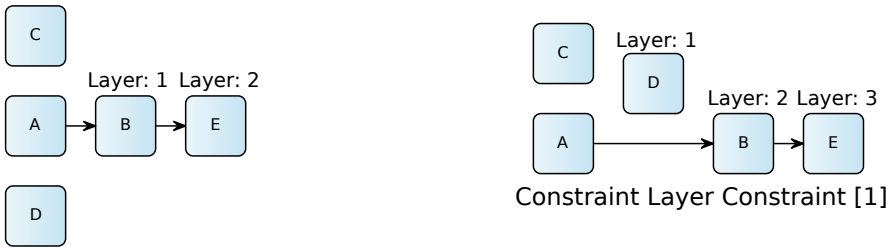


(d) Dieses Folgelayout entsteht durch das Löschen der Constraints von Knoten *E* mit Reevaluation.

Abbildung 5.9. Eine Übersicht zum Löschen von absoluten Constraints.

Constraints unter der Position, die E im Folgelayout nach der Löschung hat, werden also inkrementiert. In diesem Fall wird also der Position Constraint von B inkrementiert.

Haben Interaktionen Constraints gesetzt, die daraufhin gelöscht werden, lässt sich das beim Setzen ursprünglich vorliegende Layout nicht immer wiederherstellen. Dies liegt daran, dass vergangene Ausprägungen von Constraints verloren gehen, vergleiche Abschnitt 4.6. In Abbildung 5.9 passiert dies aufgrund der Reevaluation für die Layer Änderung von E, die auf Basis des Layouts in Abbildung 5.9a erfolgt und zu Abbildung 5.9b führt. Nach dem Löschen befindet sich E, unabhängig davon, ob beim Löschen reevaluiert wird oder nicht, nicht mehr auf seiner ursprünglichen Position 0 in L_0.



(a) Dies ist das Layout eines Graphs, auf dem ein Constraint Layer Constraint eingeführt werden soll.

(b) Knoten D wird in einen neuen Layer zwischen die ursprünglichen Layer L_0 und L_1 eingefügt.

Abbildung 5.10. Beispiel für das Einfügen eines Layers zwischen existierenden Layern durch ein Constraint Layer Constraint.

5.8. Constraint Layer Constraints

Constraint Layer Constraints kapseln das Hinzufügen neuer Layer zum Layout zwischen existierenden Layern. Dies ist notwendig, weil Layer Constraints für das Bewegen von Knoten in neue Layer zwischen existierenden Layern nicht ausreichen. Dies liegt daran, dass die Knoten ohne Layer Constraint in ihren Layern verbleiben sollen. Constraint Layer Constraints sind auf dem Parent-Knoten eines Layouts setzbar und erwarten eine Liste $zs = [z_0, \ldots, z_w]$ für $w \in \mathbb{N}$, $i \leqslant w$, $z_i \in \mathbb{Z}$. Dahinter verbirgt sich, dass für jedes z_i ein neues Layer an Layer-Index z_i eingefügt werden soll. Platziert eine Interaktion einen Knoten am aktuellen linken oder rechten Rand in einen neuen Layer, sollte dies nicht mit einem Eintrag in einen Constraint Layer Constraint einhergehen, sondern mit einer der Erweiterungen aus Abschnitt 5.5 vonstatten gehen. Weiterhin sollte nur ein Constraint Layer-Eintrag existieren, wenn es auch einen Knoten gibt, der sich in diesem Constraint Layer befindet. Abbildung 5.10 zeigt exemplarisch, wie Constraint Layer Constraints funktionieren sollen: In Abbildung 5.10b spezifiziert ein Constraint Layer Constraint mit Eintrag 1 einen neuen Layer L_1, der sich zwischen den Layern L_0 und L_1 aus Abbildung 5.10a befindet. Zudem platziert ein Layer Constraint mit $LC(D) = 1$ den Knoten D aus Abbildung 5.10a in dem neuen Layer.

5.8.1. Anpassung des interaktiven Layoutprozesses

Für diesen neuen Constraint-Typ gilt es, den interaktiven Layout-Prozess zu überarbeiten, aufgezeigt in Abschnitt 4.4 und erweitert in Abschnitt 5.5. Direkt nachdem die Constraint-Auswertung alle Knoten ohne Layer Constraint seine jeweilige Layer-Liste zugeordnet hat, ist für jedes $z_i \in zs$ eine neue Layer-Liste an Position z_i einzufügen. Dieses Vorgehen gewährleistet, dass alle Knoten ohne Layer Constraint, in dem Layer verbleiben, das sie beim interaktiven Einfügen neuer Layer innehaben.

5.8.2. Reevaluierung beim Erstellen von Constraint Layer Constraints

Führt eine Interaktion einen neuen Layer zwischen zwei vorhandenen Layern ein, verschiebt dies das Layout beginnend am Index des Layer-Einfügens einen Layer nach rechts. Diese Verschiebung muss bei den Layer Constraints rechts des Index nachvollzogen werden, da ansonsten der Layout-Prozess die betroffenen Knoten einen Layer weiter links positioniert als zuvor. Das heißt, sei $z_{neu} \in \mathbb{Z}$, dann inkrementiert die Reevaluation alle Layer Constraint-Werte mit $LC(v) \geqslant z_{neu}$ für $v \in V$ und fügt z_{neu} in zs ein. In Abbildung 5.10 wird diese Reevaluation für das Einführen eines neuen Layers mit Layer-Index 1 angewendet, um Layer Constraints auf den Knoten B und E zu inkrementieren.

5.8.3. Ausleerung von Layern anpassen

Die Ausleerung von Layern kann dazu führen, dass Constraint Layer-Einträge keinen Effekt haben und überflüssig sind. So kann ein Constraint Layer durch Layer-Änderungen eines Knotens ausgeleert werden: Gibt es keinen Knoten $v \in V$ mit $LC(v) = z_i$, dann lösche z_i aus zs. Die Ausleerung von Layern kann dazu führen, dass ein z_i lediglich einen neuen Layer am linken oder rechten Rand einfügt. In diesem Fall decken die Erweiterungen aus Abschnitt 5.5 das Hinzufügen der Layer bereits ab. Der Eintrag z_i ist also überflüssig und daher zu löschen. Man löscht z_i also, wenn es keinen Knoten $v \in V$ gibt, für den gilt, dass $LC(v) \leqslant z_i$ oder $LC(v) \geqslant z_i$. Diese Prüfungen verhindert, dass das Modell durch zusätzliche Einträge vergrößert wird, die effektlos sind.

5.8.4. Löschen eines Eintrags aus einem Constraint Layer Constraint

Angenommen ein Eintrag eines Constraint Layer Constraints wird durch eine Interaktion gelöscht. Da beim interaktiven Einführen eines Eintrags alle Layer Constraints hinter dem damit eingeführten Layer inkrementiert werden, liegt nahe, beim Löschen eines Eintrags die Layer Constraints hinter ihm zu dekrementieren. Dann nimmt die Reevaluation billigend in Kauf, dass sie dies auch für Constraint Layer Constraints tut, die nicht interaktiv erzeugt wurden. Wie in Abschnitt 5.1 in Prinzip 3 festgehalten, ist es zu vermeiden, Constraints zu verändern, wenn dies nicht notwendig ist. Ergo sollte hinter einem gelöschten durch einen Constraint Layer Constraint erzeugten Layer nicht dekrementiert werden.

Es stellt sich die Frage, was mit einem Knoten $a \in V$ mit $LC(a) = z_i$ geschehen soll. So oder so verändert a durch das Löschen von z_i aus dem Constraint Layer Constraint seinen Layer. Daher wird auch eine Reevaluation nötig, siehe Abschnitt 5.3.3. Der Layer Constraint auf a hängt semantisch mit dem Constraint Layer Constraint zusammen. Daher lässt sich wie folgt argumentieren: Der Layer Constraint von a verliert ohne den Eintrag im Constraint Layer Constraint seine Bedeutung, da der Layer Constraint gerade aussagte, dass der Knoten a in jenem neu eingefügten Layer sein soll. Demnach ist der Layer Constraint von a zu löschen, vergleiche Abschnitt 5.7.

Angenommen man will den Layer Constraint auf a dennoch beibehalten, weil sich keine Constraints auf dem Modell ändern sollen, die sich nicht unbedingt ändern müssen, siehe Prinzip 3 in Abschnitt 5.1. Dann wird a durch den Layer Constraint mit $LC(a) = z_i$ in den Layer vor dem Layer eingefügt, den der Eintrag z_i spezifiziert hat.

5.9. Knoten und Kanten interaktiv einführen

Im Allgemeinen ist es nicht vorgesehen, dass interaktive Mittel in KEITH neue Elemente einführen, anstelle dessen soll die Spezifizierung neuer Elemente im Texteditor erfolgen. Für manche Nutzenden

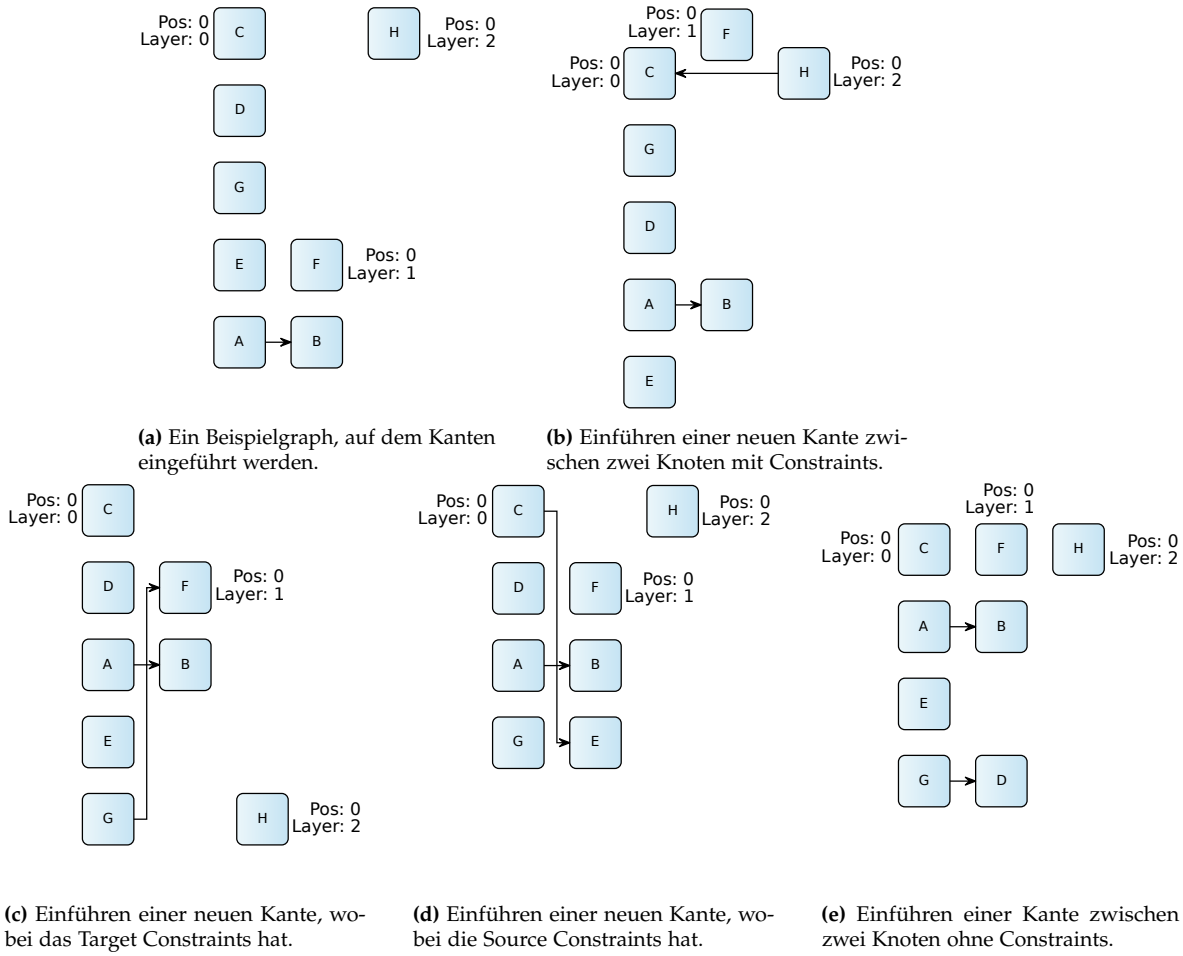
und für andere IDEs kann dies jedoch ein wünschenswertes Feature sein. Zudem hat Abschnitt 5.2 vorgestellt, dass eine Reevaluation auf Basis von Änderungen im Texteditor nicht gut nachvollziehbar und nicht möglich ist. Bei interaktiver Einführung neuer Elemente ist eine Reevaluation durchführbar und (wahrscheinlich) besser nachvollziehbar.

5.9.1. Knoten einführen

Angenommen es gibt eine Interaktion, die einen neuen Knoten v_{neu} zum Graphen hinzufügt. Dieser Knoten v_{neu} könnte zum einen ohne Constraints und zum anderen mit Constraints zum Graphen hinzugefügt werden. Letzteres ist zum Beispiel denkbar, wenn die Interaktion gleichzeitig erlaubt festzulegen, wo der Knoten v_{neu} sich befinden soll. Soll der Knoten v_{neu} keine Constraints erhalten, platziert der Layered Layoutalgorithmus den Knoten v_{neu} im linkesten Layer, also in L_0, da `SeparateConnectedComponents` deaktiviert ist, vergleiche Abschnitt 4.3. Der Knoten sollte so hinzugefügt werden, dass er allen platzierten Knoten ohne Position Constraint nachfolgt. So wird keine Reevaluation nötig, da sich die Knotenreihenfolge nicht ändern kann, weil Knoten mit Position Constraint ihre Position garantiert erhalten, siehe Abschnitt 4.4. Wie schon betrachtet, ist es möglich, einen Position Constraint, einen Layer Constraint oder beide auf einem Knoten zu setzen. Jede dieser Optionen entspricht der Operation im Ziel-Layer der schon betrachteten Positionsänderung, siehe Abschnitt 5.3.2, Layer-Änderung, siehe Abschnitt 5.3.3, oder kombinierten Positions- und Layer-Änderung, betrachte Abschnitt 5.3.1. Damit zieht das Knotenhinzufügen die gleiche Reevaluation im Ziel-Layer nach sich, die zur passenden Änderungs-Aktion gehört. Ansonsten kommt es zu Veränderung in der Knotenreihenfolge im Ziel-Layer.

5.9.2. Kanten hinzufügen

Angenommen es gibt eine Interaktion, die es erlaubt, eine neue Kante $e_{neu} = (s, t)$ zwischen zwei im Graphen existierenden Knoten s und t zu erstellen. Das Hinzufügen neuer Kanten stellt ein Stein des Anstoßes für den Grundsatz dar, die Mental Map der Nutzenden möglichst unverändert zu lassen. Es liegt also eine Verletzung von Prinzip 1 vor, siehe Abschnitt 5.1. Wenn eine Aktion eine neue Kante einführt, bleiben s und t nur dann mit Sicherheit auf ihren bestehenden Positionen, wenn für beide sowohl ein Layer Constraint als auch Position Constraint gesetzt ist. Fehlt ein Layer Constraint auf dem Target aber nicht auf der Source wird das Layer Assignment im initialen Layout das Target in den nächsten rechten Layer der Source platzieren. Dies zeigt Abbildung 5.11d, das durch das Hinzufügen einer Kante zwischen den Knoten C und $E4$ im Layout in Abbildung 5.11a zustande kommt. Fehlt ein Position Constraint auf der Source, ändert sich ihre Position, wenn eine Kante eingeführt wird. Dies zeigt Abbildung 5.11c, die eine neue Kante zwischen den Knoten G und F im Vergleich zum Layout in Abbildung 5.11a hat. Fehlen Position Constraints ändern mitunter beide Knoten ihre Position in der Crossing Minimization-Phase: Im Layout in Abbildung 5.11a wird eine Kante zwischen den Knoten G und D eingeführt. Dies führt zum Folgelayout in Abbildung 5.11e, in dem beide Knoten ihre Position geändert haben und D in den Layer L_1 gewechselt ist. Dies lässt sich durch Fixierung der Knoten durch Constraints vermeiden. In Abbildung 5.11b gibt es eine neue Kante zwischen den Knoten H und C im Vergleich zum vorigen Layout in Abbildung 5.11a. Da beide Knoten sowohl Position Constraint als auch Layer Constraint haben, ändern sich Layer und Position beider Knoten nicht. Constraints auf Knoten einzuführen, die noch kein Constraint haben, verletzt Prinzip 2, siehe Abschnitt 5.1. Denn das Modell sollte nicht leichtfertig vergrößert werden, wenn sich dies vermeiden lässt. Ungeachtet, wie das System die Kante einführen soll, wird es also eine Verletzung der Prinzipien geben. Dies legt nahe, vom interaktiven Einführen von Kanten abzusehen.



(a) Ein Beispielgraph, auf dem Kanten eingeführt werden.

(b) Einführen einer neuen Kante zwischen zwei Knoten mit Constraints.

(c) Einführen einer neuen Kante, wobei das Target Constraints hat.

(d) Einführen einer neuen Kante, wobei die Source Constraints hat.

(e) Einführen einer Kante zwischen zwei Knoten ohne Constraints.

Abbildung 5.11. Verschiedene Situationen beim Einführen von Kanten.

Will man es dennoch einführen, ist das Prinzip, die Mental Map der Nutzenden möglichst aufrechtzuerhalten, zu priorisieren, siehe Abschnitt 5.1. Demnach sollte die Standardaktion zum interaktiven Kanteneinführen die aktuelle Position und den aktuellen Layer von s und t durch einen Position Constraint und einen Layer Constraint fixieren, sofern dies nicht der Fall ist. Da somit beide Knoten auf ihrer Position verbleiben, folgt aus dem Vorgehen kein Bedarf an Reevaluation.

5.10. Überlegungen zu relativen Constraints

Relative Constraints stellen eine Erweiterung des Intentional Layouts dar, wie wir es bis dato betrachtet haben. Das bisherige Konzept ist auf Position Constraints und Layer Constraints ausgelegt. Beide Constraint-Typen setzen Knoten mit absoluten Werten auf Positionen beziehungsweise Layer, die vom initialen Layout und anderen absoluten Constraints abhängen. Dabei werden Layer Constraints vor Position Constraints ausgewertet.

Ein relativer Constraint weist einem Knoten t einen Knoten r zu. Je nach Constraint-Typ bestimmen der Layer-Index oder der Positions-Index von r den Layer-Index oder den Positions-Index von t,

vergleiche Abschnitt 2.7 und Abschnitt 4.4. Position oder Layer von r müssen also bestimmt werden, bevor die durch relative Constraints bestimmten Größen auf t gesetzt werden sollen. Der Layer von Knoten t beziehungsweise seine Position ist also abhängig vom Layer beziehungsweise von der Position von Knoten r. Relative Constraints sollten wie absolute Constraints interaktiv hinzufügbar sein. Wie dies umsetzbar ist, ist also auszuarbeiten. Weiterhin ist zu klären, wie genau die Layoutoptionen für relative Constraints anzulegen sind. Wird ein relativer Constraint auf einem Knoten gesetzt, stellt sich die Frage, ob sich dadurch lediglich die Position und der Layer des Zielknotens im Vergleich zum Vorlayout ändern sollte. Die Betrachtungen für absolute Constraints legen dies nah, allerdings sollte diese Überlegung überprüft werden. Im Folgenden wird diese Überlegung als sinnvoll angenommen.

Böhringer et al. betrachten relative Constraints in ihrem Ansatz für Constraints für den Sugiyama-Algorithmus [BP90], siehe Kapitel 3. Sie übersetzen ihre relativen Constraints in Formeln in Form von Gleichungen über die horizontalen und vertikalen Koordinaten ihrer Knoten. Sie betrachten die beiden Dimensionen dabei getrennt. Auf eine ähnliche Weise lassen sich unsere relativen Constraints in Formeln in Form von Gleichungen und Ungleichungen über die Position und den Layer der Knoten t und r ausdrücken.

Die Übersetzung aller relativen Constraint-Typen in Formeln und eine Auflistung dieser, ist dabei eine offene Aufgabe. Übersetzt man alle gesetzten relativen Constraints eines Graphen, ergibt sich jeweils eine Menge an Gleichungen über alle Layer und je eine für Positionen in einem Layer. Da das Setzen von Knoten in seinen Layer festlegt, in welchem Kontext die Position eines Knoten zu betrachten ist, sollte die Betrachtung der Layer bei relativen Constraints zuerst erfolgen. Die Auswertung der relativen Constraints sollte zeitlich nach der für absolute Constraints erfolgen, weil die Positionen oder Layer von Knoten mit absoluten Constraints dann feststehen. Aufgrund der zuvor durchlaufenen Auswertung von Layer Constraints hat das durch sie ausgelöste Shifting ebenfalls stattgefunden. Das Feststehen von Layer und Positionen, die durch absolute Constraints gesetzt sind, lässt sich durch entsprechende Gleichungen ausdrücken. Allerdings können Same Layer, Left of und Right of Constraints ebenfalls nach dem Auswerten der Layer Constraints dazu führen, dass ein Zielknoten durch einen relative Constraint mit zu ihnen adjazenten Knoten in einem Layer platziert werden. Im Layered Layoutalgorithmus ist es verboten, adjazente Knoten in dem gleichen Layer zu platzieren. Verletzt es nicht die Constraints der Knoten in dem Layer, lässt sich dies durch Shifting lösen. Hierbei stellt sich jedoch die Frage, ob Shifting bei relativen Constraints zugelassen werden sollte. Es zuzulassen, erhöht die Zahl an benötigten Gleichungen, da die Layer von Knoten nicht feststehen, die keinen den Layer beeinflussenden Constraint haben. Zudem verändert Shifting des Layout, was zu einer Beeinträchtigung der Mental Map führen kann. Wenn die Constraints Shifting nicht gestatten, beispielsweise weil einer von ihnen ein Layer Constraint ist, dann darf der Zielknoten nicht in dem Layer platziert werden. Dies ist zum Beispiel der Fall, wenn ein Knoten B mit einem Knoten mit Layer Constraint im gleichen Layer sein soll, in dem es einen anderen Knoten mit Layer Constraint gibt, der zu B adjazent ist. Um das Verbot von adjazenten Knoten in einem Layer in den Kontext der Ungleichungen für relativen Constraints zu transportieren, lassen sich Ungleichungen hinzufügen, die besagen, dass die Layer aller adjazenten Knoten ungleich sein müssen. Die Erhaltung der Mental Map sollte bei relativen Constraints wie bei absoluten Constraints ein wichtiges Kriterium sein. Also sollten Layer Constraint freie Knoten nur dann ihr Layer ändern, wenn dies durch Shifting geschieht. Dies lässt sich durch eine bedingte Gleichung ausdrücken: Solange ein Knoten A ohne den Layer betreffenden Constraint nicht mit einem adjazenten Knoten in einem Layer ist, soll A sein Layer beibehalten. Solange kein Knoten aufgrund eines relativen Constraints die Position eines Knoten ohne Position Constraint einnimmt, sollte er seine Position beibehalten. Im Prinzip entspricht die Überprüfung und die Lösung der hierbei entstehenden Constraint-Gleichungen ein durch Constraints ausgedrücktes Problem in der Constraint-

Programmierung. Hier überschneidet sich unsere Definition für Constraints im Intentional Layout und die Definition von Constraints in der *Constraint-Programmierung*. In Letzterer sind Constraints bestimmte prädikatenlogische Formeln. Nichtsdestotrotz sollten beide Definitionen miteinander vereinbar sein, da sich alle unsere Constraints in Formeln beziehungsweise Ungleichungen umwandeln lassen. Ist es also möglich, alle relativen Constraints und die oben genannten Zusatzbedingungen in Constraints der Constraint-Programmierung umzuwandeln, dann ist ein Constraint-Löser imstande sowohl die Überprüfung als auch das Finden einer Lösung, also ein die Constraints erfüllendes Layout, zu übernehmen.

Kapitel 6

Implementierung des Intentional Layout und der Reevaluation

Der Fokus dieser Arbeit liegt auf dem Konzept, das sie in Kapitel 4 und in Kapitel 5 erläutert. Dem entsprechend ist weniger Arbeit und Zeit in die Implementierung geflossen. Die implementierten Konzepte und Details zur Strukturierung des Intentional Layouts folgen in diesem Kapitel. Abbildung 6.1 zeigt, wie die verschiedenen Implementierungsbestandteile miteinander zusammenhängen. Die Implementierung, die in der Projektphase entstanden ist, findet sich in Abschnitt 6.1 bis Abschnitt 6.7. Darauf folgen die Implementierungsdetails zur Reevaluation in Abschnitt 6.8.

6.1. Eingeführte Properties und ihre Bedeutung

Die in Abschnitt 2.7 eingeführten Constraints sind als Layoutoptionen, also als `Properties`, des Layered Layoutalgorithmus implementiert. Dazu kommen weitere Layoutoptionen, die für die Implementierung des Intentional Layouts verwendet werden. Es folgt eine Auflistung aller zur `Layered.melk` hinzugefügten Layoutoptionen.

- **InteractiveLayout:** Ein `Boolean`, der auf der Wurzel einer Ressource gesetzt sein muss, wenn das Intentional Layout verwendet werden soll.
- **PositionId:** Speichert den Positions-Index eines Knoten als `int`. Unsere Implementierung arbeitet bisher nur mit natürlichen Zahlen als Positions-Indices.
- **LayerId:** Speichert den Layer-Index eines Knoten als `int`. Unsere Implementierung arbeitet bisher nur mit natürlichen Zahlen als Layer-Indices.

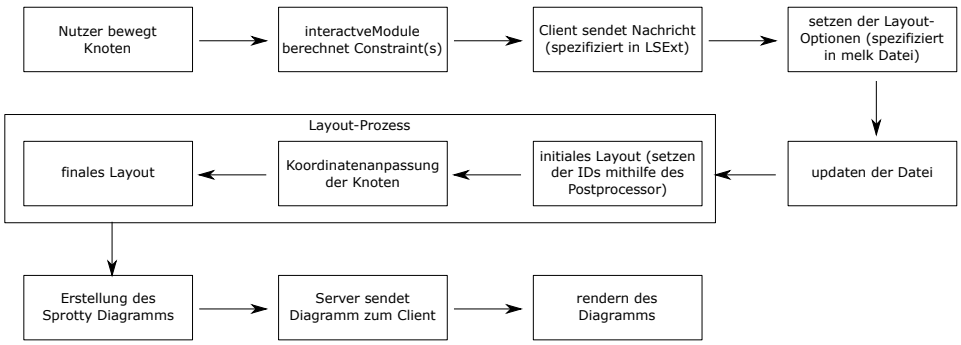


Abbildung 6.1. Ein Schema von Petzold [Pet19], das den Zusammenhang der einzelnen Teile der Implementierung darstellt.

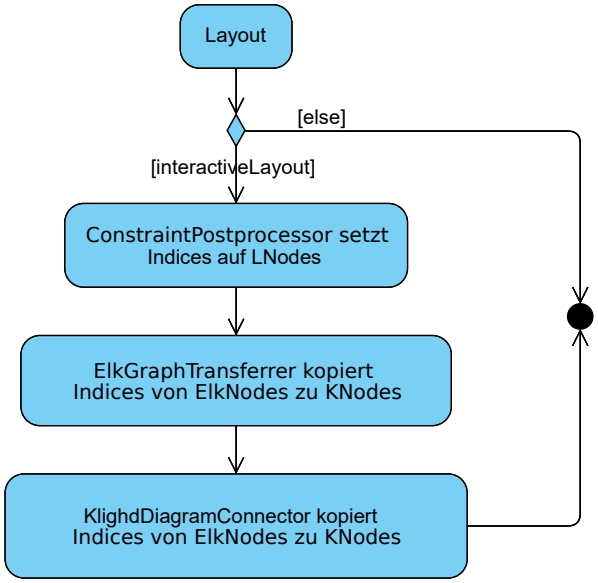


Abbildung 6.2. Dieses Aktivitätsdiagramm beschreibt, wie die Layer und Position Indices von Knoten gesetzt werden.

- **PositionChoiceConstraint:** Speichert den Position Constraint-Wert eines Knoten als `int`. Derzeit ist er als `Default` auf -1 gesetzt und seine untere Grenze ist ebenfalls -1.
- **LayerChoiceConstraint:** Speichert den Layer Constraint-Wert eines Knoten als `int`. Derzeit ist er als `Default` auf -1 gesetzt und seine untere Grenze ist ebenfalls -1. Diese Wahl ist im Anbetracht von Abschnitt 5.5.2 nicht sinnvoll und anzupassen, wenn negative Layer Constraints implementiert werden.
- **InitialLayerId:** Speichert den Layer-Index aus dem initialen Layout. Diese `Property` ist notwendig für die Reevaluation bei Shifting sowie bei der Reevaluation beim Löschen von Constraints.
- **InitialPositionId:** Speichert den Positions-Index aus dem initialen Layout. Diese `Property` ist notwendig für die Reevaluation bei Shifting sowie bei der Reevaluation beim Löschen von Constraints.

6.2. ConstraintsPostprocessor

Grundsätzlich arbeiten wir auf der Serverseite mit der `KNode`- und der `ElkNode`-Repräsentation von Graphen. Dabei handelt es sich bei `ElkNodes` um die Knoten, die aus der Ressource synthetisiert werden. `KNodes` sind hingegen die interne Modell-Repräsentation des Graphen. Der Layered Layoutalgorithmus arbeitet wiederum mit einem `LGraph` deren Knoten `LNodes` sind. Allerdings verfügen sowohl `KNode` als auch `ElkNode` über keine `Property` für den Layer-Index und den Positions-Index, die beide vom Layered Layoutalgorithmus auf dem zugehörigen `LGraph` gesetzt werden. Für das Intentional Layout ist essentiell den Layer-Index sowie den Positions-Index der Knoten im bestehenden *Layout* zu kennen, ansonsten lassen sich Constraints nicht sinnvoll setzen. Beide lassen sich für jeden Knoten mithilfe des *Layouts* berechnen, da die Knotenposition Rückschluss auf den Layer eines Knoten und seine Position in diesem Layer zulässt. Dies führt jedoch zu Overhead. Stattdessen erweitern wir die Properties

des Layered Layoutalgorithmus durch eine `LayerId` und eine `PositionId`. Diese werden durch den `ConstraintsPostprocessor` nach der Edge Routing-Phase gesetzt. Dafür iteriert der Postprocessor über alle Layer des `LGraph` und speichert den Layer-Index und den Positions-Index jedes Knoten in der jeweiligen Layoutoption des Knoten. Es ist wichtig an dieser Stelle, die Indices nicht auf Dummy-Nodes zu setzen, da diese im angezeigten *Layout* nicht auftauchen und die Indices somit fehlerhaft werden. Nachdem die `Properties` auf den `LNodes` gesetzt sind, gilt es die `Properties`, die den Indices entsprechen, auf den anderen Repräsentationen des Graphen zu setzen. Zunächst geschieht das für die `ElkNode`-Repräsentation des Graphen in der Klasse `ElkGraphLayoutTransferrer` in der Methode `applyNodeLayout()`. Darauf werden die `Properties` in der Klasse `KlighdDiagramConnector` in der Methode `applyLayout()` auf die `KNode`-Repräsentation der Knoten kopiert. Diesen Ablauf zeigt Abbildung 6.2.

Da sich der `KlighdDiagramConnector` in einem anderen Projektteil von KIELER befindet als ELK, hat man keinen Zugriff auf die `Properties` für den Positions-Index und den Layer-Index. Abhilfe verschafft, diese lokal mit der gleichen `Id` zu erstellen, wodurch sie für den Compiler dieselben Klassen sind.

6.3. Language Server Extension

Jeder Constraint-Typ sowie das gleichzeitige Setzen eines Position- als auch Layer Constraint verfügt über eine eigene Nachricht zum Hinzufügen und zum Löschen. Jeder Constraint-Typ verfügt über eine eigene Datenkapsel-Klasse, deren Instanzen den Nachrichten übergeben werden.

6.4. Client

Unsere Erweiterung ist auf der Client-Seite von KEITH in ein neues Moduls, das `Interactive-Module`, implementiert. Wir tauschen den `MouseMoveListener`, der Mausaktionen im KEITH-Client verarbeitet, gegen den `InteractiveMouseListener`. In diesem werden Methoden gerufen, die prüfen, ob eine Aktion durchgeführt wurde, die dem Verschieben eines Knoten in einen anderen Layer oder auf eine andere Position entspricht. Ist dies der Fall, wird eine passende Nachricht gesendet.

6.5. Server

Die Server-Seite von KEITH übernimmt das Setzen der Constraints auf dem Modell, die meiste Reevaluation sowie den interaktiven Layout-Prozess als auch das damit verbundene Bestimmen passender Koordinaten für die interaktiven Layout-Strategien.

6.5.1. Constraints-Setzen

Sendet der Client dem LS eine Nachricht, die ein Constraint-Kapselobjekt enthält, soll der Server die Constraints, die dieses Objekt beschreibt, auf dem Modell setzen. Dafür wird zunächst über den `DiagramState` der `KGraphLanguageServerExtension` mithilfe der Ressource `URI` aus dem Kapselobjekt der `ViewContext` des beschriebenen Graphen bestimmt. Das geschieht in der `getRoot()`-Methode. Durch diesen kann die `getKNode()`-Methode den `KNode`, auf dem die Constraints gesetzt werden sollen, mit dem `KGraphElementIDGenerator` nachschlagen. Dieser `KNode` wird in dem Kapselobjekt, das in der Nachricht vom Client enthalten ist, durch eine String `Id` referenziert. An diesem Punkt setzen die in Abschnitt 6.8 betrachteten *Reevaluationen* an. Danach werden die neuen Constraints auf dem `KNode`

gesetzt. Die KNode-Repräsentation eines Knoten dient an dieser Stelle als Zwischenspeicher, bevor die Constraints auf das Modell der Ressource übertragen werden. Dann wird mit der updateSourceCode() das Modell in der Ressource des Graphen geupdatet. Zunächst werden hierbei für jeden Knoten mit der Methode copyAllConstraints() die Werte der Constraint-Properties von dem betrachteten KNode auf seinen ElkNode kopiert, da die Knoten des Modells im ElkNodes sind. Darauf folgend gibt die Methode getResourceFromUri() mithilfe der URI der Ressource und einem Injector die Instanz der Ressource zurück. Dann wird das Modell der Ressource gelöscht, das neue Modell eingefügt und gespeichert. Die Speicherung löst einen neuen Layout-Prozess aus, der in Abschnitt 6.5.2 behandelt wird.

6.5.2. Intentional Layout

Für die Realisierung des Intentional Layouts greifen wir in die Logik ein, die auf der Server-Seite von KEITH für die Generierung neuer Layouts für die Diagramme der Client-Seite zuständig ist - dies geschieht im KGraphDiagramUpdater. In der dort vorhandenen createModel-Methode fügen wir einen Aufruf von calcLayout(), unserer äußersten *Layout*-Methode ein. Dieser Aufruf wendet unseren angepassten Layout-Prozess auf das zuvor aus der Ressource synthetsierte Graphen-Modell an. Sie implementiert den interaktiven Layout-Prozess, wie er in Abschnitt 4.3 beschrieben wird. Dafür haben wir mit der Methode onlyLayoutOnKGraph(String rootID) den Layout-Code der LayoutEngine ausgegliedert. Dieser führt das Layout ausschließlich auf einer Modell-Repräsentation durch und wandelt diese nicht in eine View-Repräsentation des Graphen um, wie es sonst geschehen würde. Sie entspricht dem, was Abschnitt 4.3 das initiale Layout nennt. Darauf erfolgt das Setzen der Koordinaten mit einem Depth First-Vorgehen mit setCoordinatesDepthFirst(), das Setzen der Layout-Strategien auf der Wurzel des Modells und schließlich ein zweiter Aufruf des Layouts mit onlyLayoutOnKGraph(String rootID).

6.6. Koordinaten-Setzung

Die Methode setCoordinates() implementiert die Constraint-Auswertung und die Koordinaten-Setzung, wie sie Abschnitt 4.4 ausführt. Dabei fasst calcLayerNodes() die Schritte 1 und 2 zusammen: Sie erstellt die den Layern entsprechenden Layer-Listen und sammelt in ihnen zunächst die Knoten ohne Layer Constraint und fügt daraufhin die Knoten mit Layer Constraint zu den Layer-Listen hinzu. Darauf folgend führen setXCoordinates() und setYCoordinates() Schritt 3, 4 und 5 durch. Sie setzen also die horizontalen Koordinaten, stellen eine Ordnung auf den Layer-Listen her und setzen mithilfe dieser die vertikalen Koordinaten auf die Weise, wie sie im zugehörigen Abschnitt 4.4 erläutert wird.

6.7. Support für hierarchische Graphen

Die Methode setCoordinatesDepthFirst() stellt die Implementierung für den Support von hierarchischen Graphen bereit. Um Intentional Layout für hierarchische Graphen zu unterstützen, rufen wir setCoordinates() für jede Hierarchie-Ebene des Graphen einzeln auf. Dabei gehen wir mit einem Depth-First-Vorgehen vor, da das Intentional Layout die Breite und Höhe des *Layouts* verändern kann, siehe Abschnitt 4.5. Sind die Koordinaten eines Graphen angepasst, wird die Breite seines Parents abhängig von der horizontalen Koordinate des letzten Layers, des Paddings und Spacings der Knoten in ihm gesetzt. Allerdings handelt es sich herbei um eine zu starke Simplifizierung: Es wird außer

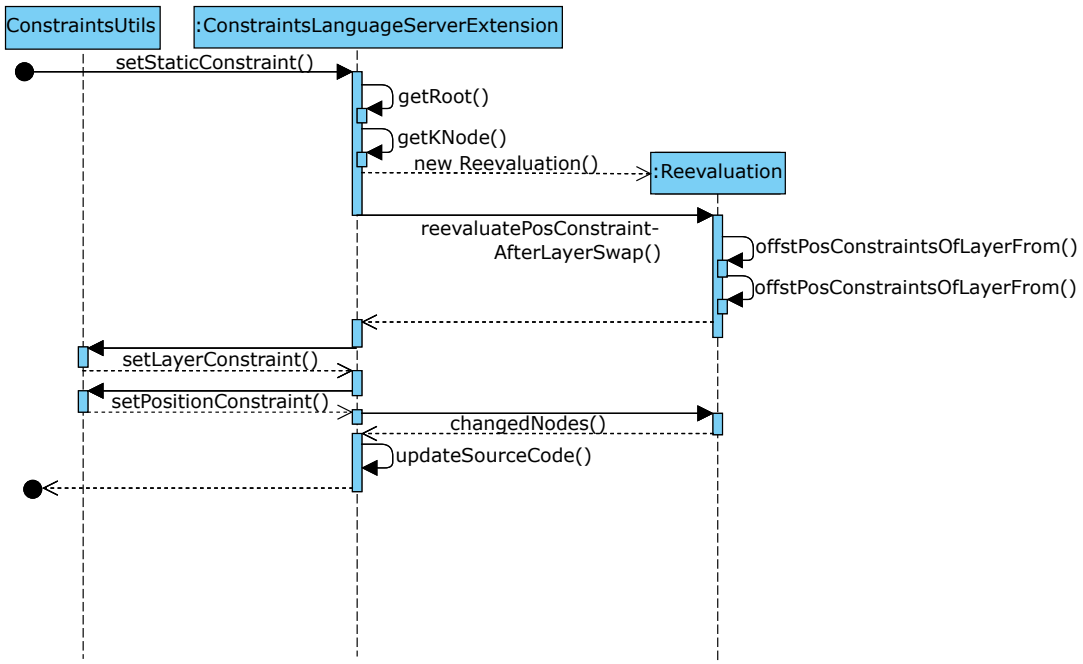


Abbildung 6.3. Ein Sequenzdiagramm zur Reevaluation für kombinierte Positions- und Layer-Änderung.

Acht gelassen, wie weitere Bestandteile des `elkt`-Formats wie zum Beispiel das `EdgeLabelSpacing` die Breite eines Layouts ändern können.

6.8. Reevaluation

Im Bereich der Reevaluation sind die grundlegenden Methoden zur Reevaluation von kombinierten Positions- und Layer-Änderungen und Positions-Änderungen implementiert, siehe Abschnitt 5.3. Die Methode `reevaluatePosConstraintsAfterLayerSwap()` für die kombinierte Positions- und Layer-Änderung lässt sich auch für eine bloße Layer-Änderung verwenden, da sie die Ziel-Position des bewegten Knotens als Argument erwartet. Das Sequenzdiagramm in Abbildung 6.3 zeigt, wann die Reevaluation erfolgt: Zunächst erhält der LS eine Instanz von `StaticConstraint` führt dann die Reevaluation durch, sammelt die Knoten, deren Constraints verändert wurden, und speichert die Änderungen inklusive dem Setzen der neuen Constraints im Modell. Wiederum ist die Reevaluation für eine Positions-Änderung im Layer in der Methode `reevaluatePosConstraintsAfterPosChangeInLayer()` implementiert. Sie wird in einem ähnlichen Ablauf gerufen wie die Reevaluationsmethode für kombinierte Positions- und Layer-Änderung. Die Methode `reevaluateAfterEmptyingALayer()` ist die Impementierung der Reevaluation bei Layerausleerung. Die Erläuterung dieser findet sich in Abschnitt 5.3.4. Sie dient dazu, um bei Layer-Ausleerung die Layer Constraints hinter dem „ausgeleerten“ Layer zu dekrementieren. Sie wird aktuell in der Implementierung nicht verwendet und sollte durch eine Option aktivier- und deaktivierbar sein.

Die Reevaluationen für eine Interaktion sollten alle mit der gleichen Instanz der `Reevaluation`-Klasse durchgeführt werden, da diese die durch die Reevaluationen veränderten Knoten vorhält. Wurden die Reevaluationen alle durchgeführt, lassen sie sich diese Knoten abfragen, wonach man die Änderungen auf das Modell kopieren kann.

Zudem ist die Methode `checkForCollidingPosConstraints()` eine Implementierung für die Deaktivierung kollidierender Position Constraints, siehe Abschnitt 5.6. Sie sorgt dafür, dass kollidierende Position Constraints deaktiviert werden. Sie wird bisher noch nicht genutzt. Sie sollte noch in ELK in die Klasse `ElkGraphValidator` eingegliedert werden, damit diese solche Position Constraints Warnungen zum Modell hinzufügt. Darüber hinaus geht der Abschnitt zu Future-Work, siehe Abschnitt 7.2, darauf ein, was alles noch zu implementieren ist.

Kapitel 7

Schluss

Zusammen mit Petzold [Pet19] zeigt diese Arbeit eine Möglichkeit, Intentional Layout in die IDE KEITH zu integrieren. Aufbauend auf der gemeinsamen Arbeit mit Petzold betrachtet diese Arbeit, wie auf dem Modell eingeführte Constraints zu reevaluieren sind und erarbeitet erweiterte Features des Intentional Layouts konzeptuell. Darüber hinaus klärt Abschnitt 7.2 offene Probleme und weitere Aspekte für Intentional Layout, die zukünftige Arbeiten untersuchen könnten.

7.1. Zusammenfassung

Der erste Teil dieser Arbeit ist als Projekt mit Petzold entstanden. Er befasst sich damit, Intentional Layout für den Layered Layoutalgorithmus in die IDE KEITH zu integrieren. Position Constraints und Layer Constraints sind als Layout Optionen in ELK implementiert. Interaktionen im Client von KEITH sind mit bestimmten Layout-Anpassungen assoziiert, die in Constraints übersetzt werden. Diese Constraints sendet der Client mithilfe von Nachrichten einer für das Intentional Layout ausgelegten `LanguageServerExtension` an den LS. Dieser betrachtet das Modell und reevaluiert es bei Bedarf, bevor die Constraints der Nachricht gesetzt werden. Wir erweitern nicht den Layered Layoutalgorithmus, sondern verwenden die interaktiven Layoutstrategien, die ELK für den Algorithmus bereitstellt. Diese Strategien erfüllen die Aufgabe der jeweiligen Algorithmusphase auf Basis der Knoten-Koordinaten. Daher besteht unser Layout-Prozess aus einem initialen Layout, dem Anpassen von Koordinaten und einem zweiten Layout mit gesetzten interaktiven Layout Strategien. Als Folge daraus gehen Zwischenzustände der Ausprägung von Constraints verloren. Ein Postprozessor fügt Positions-Indices und Layer-Indices zum Modell hinzu, nachdem die letzte Phase des Layered Layoutalgorithmus durchgelaufen ist. Dadurch können diese zum Setzen der Koordinaten genutzt werden.

Damit unser Intentional Layout hierarchische Graphen unterstützt, führen wir den Layout Prozess für jede Hierarchieebene eines Graphen separat durch und gehen dabei in einem Depth-First-Ansatz vor. Um Same Layer Edges zu verhindern, ohne die Möglichkeiten des Intentional Layouts einzuschränken, führen wir Shifting ein. Shifting bewegt adjazente Knoten solange aus der internen Layerstruktur, bis kein Layer mehr adjazente Knoten beinhaltet.

Der zweite Teil der Arbeit befasst sich mit Reevaluation gesetzter Constraints und erweiterten Features des Intentional Layouts. Bei unserem Intentional Layouts gibt es Unterschiede zwischen dem interaktiven Verwenden und dem Spezifizieren im Texteditor. Dies beruht darauf, dass bei Interaktionen der Server zwischen neuen Constraints und alten Constraints unterscheiden kann, beim Spezifizieren im Texteditor aber nicht. Das liegt daran, dass Modell- und Diagramm-Updates in der derzeitigen Version von KEITH nicht über Dateiunterschiede erfolgen. Reevaluationen und erweiterte Features des Intentional Layouts stützen sich auf drei Prinzipien: Zum einen soll es zu keinen unbeabsichtigten, nicht in die Semantik der Constraints inbegriffenen Layout-Änderungen kommen. Zum anderen sollen Knoten ohne Constraints keine Constraints erhalten, wenn sie nicht das Ziel einer Interaktion sind. Ansonsten wird das Modell vergrößert, was es die Verarbeitung verlangsamt. Außerdem werden mehr Knoten im Modell fixiert, als der Nutzer beabsichtigt hat, wodurch das auto-

matische Layout weniger Einfluss auf das Layout hat. Schließlich sollen sich die Werte von Constraints nicht ändern, sofern ihre Knoten nicht von einer Interaktion betroffen sind. Bei Interaktionen, die zu Layer-Änderungen, Positionsänderungen oder einer kombinierten Änderung von Layer und Position führen, kann es neben der Änderung des Layers und der Position zu unbeabsichtigten Änderungen in der Knotenreihenfolge kommen. Da es sich bei Shifting um Layer-Änderungen handelt, kommt es bei Shifting ebenso zu unbeabsichtigten Änderungen in der Knotenreihenfolge. In dieser Arbeit gezeigte Reevaluationen verhindern diese. Darüber hinaus zeigt diese Arbeit, wie man Layer am linken Ende, zwischen den Layern und am rechten Ende einfügen kann. Am linken Ende ist dies über negative Layer Constraints möglich. Am rechten Ende lösen dies Layer Constraints, die einen Index haben, der größer ist, als die Anzahl an Layern im Layout. Sogenannte Constraint Layer Constraints, die Listen von ganzen Zahlen als Werte erwarten, können dazu dienen, Layer zwischen zwei existenten Layern aus dem initialen Layout zu erstellen. Ebenso wird untersucht, was das interaktive Einführen von Knoten und Kanten für einen Einfluss auf das Modell haben kann. Dabei ergibt sich, dass Knoten sich ohne größere Widersprüche mit den Prinzipien interaktiv einführen lassen. Kanten lassen sich hingegen nur unter Verletzung der Prinzipien einführen. Abschließend präsentiert diese Arbeit erste Überlegungen zu relativen Constraints: Sie stellt heraus, dass der Layer beziehungsweise die Position eines Knoten, auf dem ein relativer Constraint gesetzt ist, von dem Knoten abhängt, der ihm von dem relativen Constraint zugewiesen wird. Weiterhin lassen sich relative Constraints in Formeln übersetzen. Dies legt nah, dass es sich beim Evaluieren und dem Finden eines gültigen Intentional Layout mit relativen Constraints um ein Problem handelt, das sich durch Constraint-Programmierung lösen lässt.

7.2. Future Work

Diese Arbeit konzentriert sich auf die konzeptuelle Seite von Intentional Layout, weshalb einige Konzepte nicht vollständig oder gar nicht implementiert sind. So gibt es noch Nachholbedarf bei der Reevaluation von Shifting, der Implementierung der Validierung von Position Constraints. Zudem sollte die Validierung für Position Constraints in die Struktur für Fehlerannotation im Modell eingefügt werden. Die Validierung von Layer Constraints muss ebenso noch implementiert werden. Die Konzepte für negative Layer Constraints, Constraint Layer Constraints und das Einführen von neuen Knoten sind nicht implementiert worden.

Im derzeitigen Konzept und in der momentanen Implementierung verwenden wir die interaktiven Layoutstrategien von ELK. Als Folge setzen wir das Layer und die Position von Knoten indirekt über das Berechnen und Setzen passender Koordinaten, siehe Abschnitt 4.4. Dies ist problematisch bei hierarchischen Graphen, siehe Abschnitt 6.7. Eine Alternativlösung ist die Entwicklung eigener Layoutstrategien für das Intentional Layout, die direkt das Angeben von Layer und Position der Knoten erlauben. Dabei lässt sich auf dem bestehenden Konzept aufbauen. Zudem bietet sich für zukünftige Arbeiten an, andere Shifting-Strategien zu implementieren.

Darüber hinaus stellt sich die Frage, wie sich Intentional Layout für andere Algorithmen in ELK wie den Mr. Tree Layoutalgorithmus oder den Force Directed Layoutalgorithmus von ELK umsetzen lässt. Hierfür müssten eigene Constraint-Typen eingeführt werden: Für Force Directed Layoutalgorithmen bieten sich absolute Positions Constraints an, die einem Knoten feste horizontale und vertikale Koordinaten zuweisen. Bei Tree Layoutalgorithmen ist beispielsweise ein Constraint-Typ denkbar, der den Unterbaum eines Knoten als den i-ten Unterbaum eines anderen Knoten festlegt.

7.2.1. Intentional Layout in SCCharts

Bisher ist unser Intentional Layout auf Graphen im `elkt`-Format ausgelegt. Unser Konzept für Intentional Layout ist jedoch ebenso für die synchrone Programmiersprache SCCharts von Interesse, die auch Teil des KIELER-Projekts ist. SCCharts' grafische Repräsentation baut auf derselben Modellstruktur für Graphen wie `elkt` auf, weshalb sich eine Unterstützung der visuellen Programmiersprache anbietet. Insbesondere im Hinblick auf die Interaktionen müssten mehrere Aspekte neu bewertet werden [Pet19]: So kann sich die Ausrichtung des Layouts bei SCCharts je nach Hierarchielevel unterscheiden. Wir gehen bisher jedoch von einem Layout von links nach rechts aus. Ebenfalls müsste man entscheiden, ob Interaktionen und das Setzen von Constraints auf allem möglich sein sollte, was als Knoten in SCCharts definiert ist. Beispielsweise stellt SCCharts Regionen in weißen Boxen dar. In diesen Regionen platziert SCCharts wiederum Graphen. Dabei ist zu beachten, dass es sich bei diesen Regionen um Knoten handelt. Also gilt es zu betrachten, inwiefern Regionen bewegbar sein sollen.

Eine weitere Eigenheit ist, dass für Kanten-Label eigene Layer eingeführt werden. Es gilt also zu entscheiden, wie mit diesen Zusatz-Layern umzugehen ist. Wahrscheinlich sollte es nicht möglich sein, Knoten in solche Layer über Constraints zu platzieren. Dies legt nahe, dass diese Layer in der Constraint-Auswertung ignoriert werden sollten. Dabei muss Sorge getragen werden, dass Knoten mit Layer Constraint nicht versehentlich in diesen Label-Layern platziert werden. Abschließend betrachtet diese Arbeit relative Constraints, die sich als Ungleichungen über die Position und Layer von Knoten darstellen lassen. Daher entspricht das Überprüfen der relativen Constraints auf Validität und das Finden eines sie erfüllenden Layouts einem Problem, das sich durch Constraint-Programmierung lösen lässt.

7.2.2. Relative Constraints

In Abschnitt 5.10 finden erste konzeptuelle Betrachtungen von relativen Constraints für unser Intentional Layout statt. Zukünftige Arbeiten sollten diese ausdehnen und intensivieren: Dabei sollte unter anderem die Übersetzung aller relativen Constraint-Typen in Formeln vorgenommen werden. Dazu kommen Ungleichungen für Zusatzbedingungen wie das Verbot von adjazenten Knoten im gleichen Layer. Darüber hinaus gilt es zu untersuchen, ob sich weitere Constraint-Typen anbieten. Ebenso sollten zukünftige Arbeiten untersuchen, welchen Einfluss relative Constraints auf die Mental Map der Nutzenden haben und wie sich Interaktionen für sie definieren lassen. Letzteres betrachtet Petzold [Pet19]. Abschnitt 5.10 stellt heraus, dass die Validität und das Finden eines die relativen Constraints erfüllenden Layouts sich als ein durch Constraint-Programmierung lösbares Problem ausdrücken lässt. Es ist eine offene Frage, ob dies zuverlässig funktioniert und alle Anforderungen an das Intentional Layout erfüllt. Wie relative Constraints sich als Layoutoptionen umsetzen lassen, ist eine wichtige Frage, um sie ins Intentional Layout zu integrieren. Damit kein eigener Constraint-Löser entwickelt werden muss, bietet sich die Verwendung einer Open Source-Bibliothek für Constraint-Programmierung in Java wie Choco [1] an. Dafür muss geprüft werden, ob sich Choco in KEITH integrieren lässt, und ob tatsächlich die Überprüfung der Constraint-Gleichungen und das Finden einer Lösung mit Choco durchführbar ist.

[1] `http://www.choco-solver.org/`

Anhang A

Anhang

A.1. Liste von Abkürzungen

DSL Domain Specific Language

ELK Eclipse Layout Kernel

GRL Graph Representation Language

IDE Integrated Development Environment

JSON JavaScript Object Notation

RPC Remote Procedure Call

KEITH Kiel Environment Integrated in Theia

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KlighD Kieler Light Weight Diagrams

LS Language Server

LSP Language Server Protocol

MDE Model-Driven Engineering

SFK Smalltalk Frame Kit

SVG Scalable Vector Graphics

Literatur

[93] "'EDGE: An extendible graph editor'". In: *The Design of an Extendible Graph Editor* (1993), pp. 133–151. DOI: 10.1007/BFb0019379. URL: https://link.springer.com/chapter/10.1007/BFb0019379 (visited on 09/21/2019).

[Böh89] K.-F. Böhringer. „‚Stabilität von Algorithmen für Graphenumbruch'". Diplom thesis. University of Karlsruhe, Department of Computer Science, Juli 1989.

[BP90] Karl-Friedrich Böhringer und 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: New York, NY, USA. ACM, 1990, S. 43–51. ISBN: 978-0-201-50932-8. DOI: 10.1145/97243.97250. URL: http://doi.acm.org/10.1145/97243.97250 (besucht am 29. 05. 2019).

[DG02] Stephan Diehl und Carsten Görg. „‚Graphs, they are changing - dynamic graph drawing for a sequence of graphs'". In: *Proc. 10th Int. Symp. Graph Drawing (GD 2002), number 2528 in Lecture Notes in Computer Science, LNCS*. Springer-Verlag, 2002, S. 23–31.

[DMS+08] T. Dwyer, K. Marriott, F. Schreiber, P. Stuckey, M. Woodward und M. Wybrow. „‚Exploration of Networks using overview+detail with Constraint-based cooperative layout'". In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (Nov. 2008), S. 1293–1300. DOI: 10.1109/TVCG.2008.130.

[DMW09a] Tim Dwyer, Kim Marriott, and Michael Wybrow. "'Dunnart: A Constraint-Based Network Diagram Authoring Tool'". In: *Graph Drawing*. Ed. by Ioannis G. Tollis and Maurizio Patrignani. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 420–431. ISBN: 978-3-642-00219-9.

[DMW09b] Tim Dwyer, Kim Marriott und Michael Wybrow. „‚Topology Preserving Constrained Graph Layout'". In: *Graph Drawing*. Hrsg. von Ioannis G. Tollis und Maurizio Patrignani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 230–241. ISBN: 978-3-642-00219-9.

[Dom18] Sören Domrös. „‚Moving Model Driven Engineering from Eclipse to Web Technologies'". Master's Thesis. Universität zu Kiel, Nov. 2018. 102 S.

[Ead84] Peter Eades. „‚A heuristic for graph drawing'". In: *Congressus numerantium* 42 (1984), S. 149–160.

[FH10] Hauke Fuhrmann und Reinhard von Hanxleden. „‚Taming Graphical Modeling'". In: *Model Driven Engineering Languages and Systems*. Hrsg. von Dorina C. Petriu, Nicolas Rouquette und Øystein Haugen. Bd. 6394. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 196–210. ISBN: 978-3-642-16144-5 978-3-642-16145-2. DOI: 10.1007/978-3-642-16145-2_14. URL: http://link.springer.com/10.1007/978-3-642-16145-2_14 (besucht am 23. 08. 2019).

[FR92] Dietrich H. Fischer und Lieutenant-Colonel Michael Rostek. „‚SFK: a smalltalk frame kit-concepts and use'". In: 1992.

[Han18] Rheinard von Hanxleden. „‚Automatic Graph Drawing'". Universitätsvorlesung. 2018.

[KKR96] Thomas Kamps, Joerg Kleinz und John Read. „‚Constraint-based spring-model algorithm for graph layout'". In: *Graph Drawing*. Hrsg. von Franz J. Brandenburg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, S. 349–360. ISBN: 978-3-540-49351-8.

[Kla12] Lars Kristian Klauske. „‚Effizientes Bearbeiten von Simulink-Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus‘“. Diss. Berlin Institute of Technology, 2012. URL: http://opus.kobv.de/tuberlin/volltexte/2012/3696/.

[NE02] Hugo A. D. Nascimento und Peter Eades. „‚User Hints for Directed Graph Drawing‘“. In: *Graph Drawing*. Hrsg. von Petra Mutzel, Michael Jünger und Sebastian Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 205–219. ISBN: 978-3-540-45848-7.

[New88] F. J. Newbery. „‚An interface description language for graph editors‘“. In: *[Proceedings] 1988 IEEE Workshop on Visual Languages*. Okt. 1988, S. 144–149. DOI: 10.1109/WVL.1988.18022.

[Pet19] Jette Petzold. „‚Intentional Layout in Sprotty Diagrams: Defining User Interaction‘“. 2019.

[PHG07] Helen C. Purchase, Eve Hoggan, and Carsten Görg. “‘How Important Is the “Mental Map”? – An Empirical Investigation of a Dynamic Graph Layout Algorithm’”. In: *Graph Drawing*. Ed. by Michael Kaufmann and Dorothea Wagner. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 184–195. ISBN: 978-3-540-70904-6.

[PT90] F. Newberry Paulisch und W. F. Tichy. „‚EDGE: An Extendable Graph Editor‘“. In: *Softw. Pract. Exper.* 20.S1 (Juni 1990), S. 63–88. ISSN: 0038-0644. DOI: 10.1002/spe.4380201307. URL: http://dx.doi.org/10.1002/spe.4380201307.

[Pur97] Helen Purchase. „‚Which aesthetic has the greatest effect on human understanding?‘“ In: *Graph Drawing*. Hrsg. von Giuseppe DiBattista. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, S. 248–261. ISBN: 978-3-540-69674-2.

[Ren18] Niklas Rentz. „‚Moving Transient Views from Eclipse to Web Technologies‘“. Master’s Thesis. Universität zu Kiel, Nov. 2018. 103 S.

[SSH13] Christian Schneider, Miro Spönemann und Reinhard von Hanxleden. „‚Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice‘“. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC ’13)*. event-place: San Jose, CA, USA. Sep. 2013, S. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSH14] Christoph Daniel Schulze, Miro Spönemann und Reinhard von Hanxleden. „‚Drawing Layered Graphs with Port Constraints‘“. In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), S. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[STT81] K. Sugiyama, S. Tagawa und M. Toda. „‚Methods for Visual Understanding of Hierarchical System Structures‘“. In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (Feb. 1981), S. 109–125. ISSN: 0018-9472. DOI: 10.1109/TSMC.1981.4308636.