Ein Proxy-View-Ansatz zur Navigation hierarchischer Diagramme

Till Kurzenberger

Bachelorarbeit
September 2022

Prof. Dr. Reinhard von Hanxleden
Arbeitsgruppe Echtzeitsysteme / Eingebettete Systeme
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
M. Sc. Maximilian Kasperowski, M. Sc. Niklas Rentz

Selbstständigkeitserklärung

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

Kiel,

Abstract

Knoten-Kanten-Diagramme werden in Wissenschaft und Wirtschaft alltäglich verwendet, um Daten und Sachverhalte übersichtlich darzustellen. In vielen Anwendungsfällen können diese Größenordnungen von über 100 Knoten und Kanten annehmen. Um Nutzern/Nutzerinnen bei der Navigation solcher Diagramme zu unterstützen, wurde der *Proxy-View* entwickelt. Im Proxy-View werden für Knoten außerhalb des Bildschirms Miniatur-Stellvertreter am Bildschirmrand angezeigt, um Nutzern/Nutzerinnen so ohne weitere Navigation des Diagramms selektiven Kontext zu geben. Auch weitere Konzepte sind im Proxy-View enthalten – darunter insbesondere eine Erweiterung für hierarchische Diagramme. Implementiert ist der Proxy-View für generelle Knoten-Kanten-Diagramme in KLighD-VS Code. Zur Validierung des Konzepts ist hierbei auch eine Implementierung spezifisch für SCCharts vorhanden.

Danksagungen

Zunächst möchte ich dem Leiter der Arbeitsgruppe Prof. Dr. Reinhard von Hanxleden danken, für die Möglichkeit meine Bachelorarbeit zu Proxy-Views zu schreiben. In dem Sinne danke ich auch besonders Sören Domrös, bei dem ich zuvor bereits in einer Seminararbeit das Thema kennenlernen konnte.

Außerdem möchte ich meinen beiden Betreuern Niklas Rentz und Maximilian Kasperowski danken für die häufige und tatkräftige Unterstützung besonders bei Fragen zu Software und Brainstorming. Auch spontan waren beide stets bereit zu helfen und gaben konstruktives Feedback und gute Denkanstöße.

Im Allgemeinen danke ich der gesamten Arbeitsgruppe für die immerzu freundliche und lockere Atmosphäre, sowie dem vielen ergiebigen Gruppenfeedback, wenn ich meinen Fortschritt vorstellen durfte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
GPU	Graphics Processing Unit
DOM	Document Object Model
ELK	Eclipse Layout Kernel
FPS	Frames per Second
UI	User Interface
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
KLighD	KIELER Lightweight Diagrams
LSP	Language Server Protocol
SCChart	Sequentially Constructive Chart
SCGraph	Sequentially Constructive Graph
SVG	Scalable Vector Graphics
CSS	Cascading Style Sheets
VS Code	Visual Studio Code

KKD	Knoten-Kanten-Diagramme
LF	Lingua Franca
KLighD-VS Code	KLighD VS Code-Extension

Kapitel 1

Einleitung

Diagramme werden alltäglich verwendet, um Daten, Sachverhalte oder Informationen grafisch darzustellen und damit einfach verständlich zu machen. Knoten-Kanten-Diagramme (KKD) bezeichnen die Klasse aller Diagramme, welche aus Knoten und Kanten bestehen – auch Verschachtelungen sind hierbei möglich. Diese sind insbesondere in Wissenschaft und Wirtschaft sehr verbreitet [Mok17]. So kommen in der Informatik zur Visualisierung von Systemen etwa Klassen- und Komponentendiagramme vor. Die im Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) entwickelten Sequentially Constructive Charts (SCCharts) [Mot17; HDM+14] ermöglichen sogar eine visuelle Programmierung komplexer synchroner Systeme. All diese Diagramme unterstützen dabei nicht nur die Veranschaulichung von Konzepten sondern sind ein in der Regel unabdinglich notwendiger Schritt zur Entwicklung und Wartung guter Software [CVD+07; SS07].

Gerade in der Softwareentwicklung können KKD jedoch schnell sehr groß werden – selbst scheinbar einfache Systeme benötigen mitunter komplexe Modellierungen. Es wurden mehrere Lösungsmöglichkeiten vorgeschlagen, um dem entgegenzuwirken. So beziehen sich etwa Klaus Bergner et al. [BRS+99] auf das Beispiel aus Abbildung 1.1 und betonen dabei das Strukturieren des Klassendiagramms nach Secondary Notation [Pet95] als mögliche Lösung. Xinli Yang et al. [YLX+16] stellen eine Möglichkeit vor, für den Nutzer/die Nutzerin

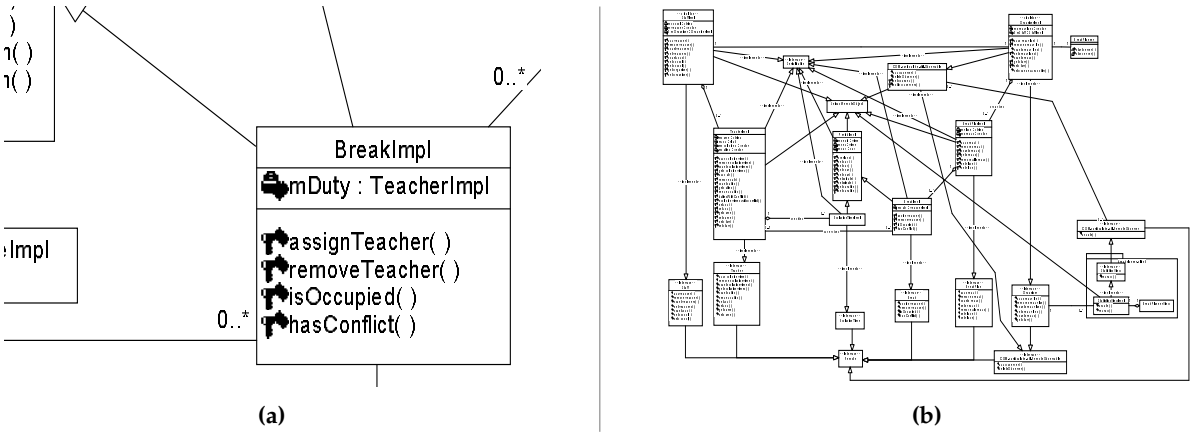
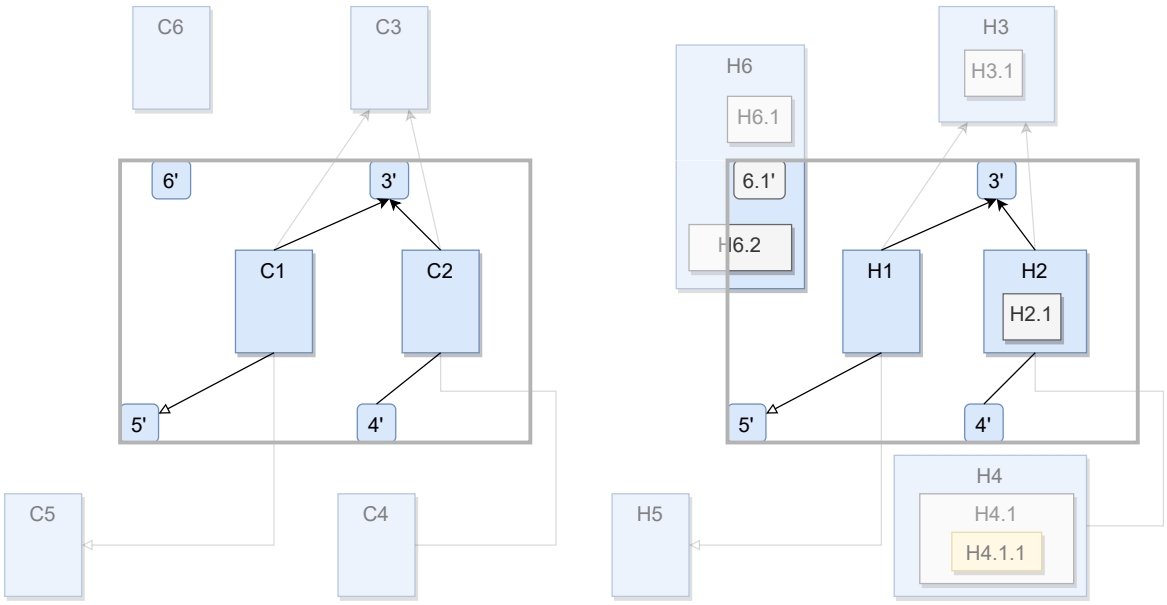


Abbildung 1.1. Beispiel eines Klassendiagramms zur Entwicklung einer einfachen Scheduling-Software. Links (a) ein lesbarer Auszug des Diagramms, rechts (b) das gesamte Diagramm. Von Klaus Bergner et al. [BRS97] erstellt.



(a) Einfacher Fall **(b)** Hierarchisches KKD

Abbildung 1.2. Links (a) ein Beispiel, wie der Proxy-View im einfachen Fall aussehen könnte. Der graue Rahmen markiert den sichtbaren Bereich. Für Knoten C3 ist ein Proxy (3′) vorhanden, der mit den beiden Knoten C1 und C2 verknüpft ist. Auch für C6 ist ein Proxy (6′) am Rand platziert, obgleich er mit keinem sichtbaren Knoten verknüpft ist. Rechts (b) die in dieser Arbeit thematisierte Erweiterung des einfachen Falls für hierarchische Diagramme. Für H3 und H4 werden jeweils nur Proxies für den äußersten Knoten dargestellt. Ferner wird für H6.1 ein Proxy (6.1′) erstellt. Der Proxy-View-Ansatz von M. Frisch und R. Dachselt [FD10] entspricht dem Beispiel von (a), wobei die Knoten hierbei Klassen darstellen. In beiden KKD könnten Filter den selektiven Kontext genauer definieren.

unwichtige Knoten zu vernachlässigen – beispielsweise indem Implementierungsdetails ausgelassen werden – und somit das KKD kompakt zu machen.

Hierbei liegt der Kern dieser Lösungsvorschläge darin, dem Nutzer/der Nutzerin nur die Knoten und Kanten anzuzeigen, die für ihn oder sie relevant sind. Der so angezeigte selektive Kontext sorgt für ein einfacheres Verständnis des Diagramms und verhindert, dass der betrachtete Knoten durch die schiere Anzahl anderer Knoten aus den Augen verloren geht. Auch Cockburn et al. [CKB09] haben hierzu Untersuchungen angestellt – im Gegensatz zu den vorher genannten Lösungsvorschlägen wurden jedoch unterschiedliche Darstellungsmöglichkeiten von KKD verglichen. Die dabei untersuchte *Cue-Based* Darstellungsvariante wurde von M. Frisch und R. Dachselt in einem *Proxy-View-Ansatz* für Klassendiagramme implementiert [FD10]:

Im Proxy-View-Ansatz hat jede Klasse außerhalb des sichtbaren Bereichs einen Miniatur-Stellvertreter – den Proxy. Dieser wird am Rand des Bildschirms platziert und Kanten zur Klasse werden stattdessen mit dem entsprechenden Proxy verbunden. Abbildung 1.2a ver-

deutlicht die Darstellung. Die Klassen C3 bis C6 sind nicht sichtbar und haben somit Proxies am Rand des Bildschirms. Es wurden außerdem mehrere Navigations-, Interaktions-, Cluster- und Filtermöglichkeiten implementiert. Besonders letztere erlauben eine feingranularere Auswahl des selektiven Kontexts.

Allerdings ist der bisherige Proxy-View-Ansatz spezifisch für Klassendiagramme entwickelt und implementiert. Es gibt noch keine Implementierung für andere Diagrammarten aus der KKD-Klasse – oder für generelle KKD an sich. Darüber hinaus existiert nach aktuellem Stand kein Konzept, einen Proxy-View für hierarchische Diagramme wie Komponentendiagramme oder SCCharts einzusetzen – Abbildung 1.2b zeigt ein Beispiel dessen für ein generelles hierarchisches Diagramm. Diese Themen bilden das Ziel dieser Arbeit.

1.1. Gliederung

Die Arbeit ist thematisch strukturiert. Zunächst wird das Konzept in Kapitel 2 näher vorgestellt und anhand einiger Mocks und Beispiele weiter erläutert. Kapitel 3 beschäftigt sich anschließend mit der Implementierung des Konzepts. Beide Kapitel gehen hierbei besonders auf hierarchische KKD ein. Schließlich werden Konzept und Implementierung in Kapitel 4 evaluiert. Kapitel 5 fasst die Ergebnisse der Arbeit zusammen und verweist auf Möglichkeiten für zukünftige Arbeit und Forschung.

1.2. Verwandte Arbeiten und Anwendungen

Der in dieser Arbeit vorgestellte Proxy-View baut direkt auf dem von M. Frisch und R. Dachselt vorgestellten Proxy-View-Ansatz auf [FD10] und erweitert dabei den einfachen Fall um hierarchische Diagramme sowie die dazugehörige Implementierung. Doch es gibt auch einige andere Ansätze, die dem Konzept des Proxy-Views ähneln. Alle im Folgenden vorgestellten Ansätze haben gemein, dass eine Art Stellvertreter am Bildschirmrand angezeigt wird – analog zum Proxy.

Der von Zellweger et al. entwickelte Ansatz *City Lights* [ZMG+03] bildet die Grundlage vieler weiterer Arbeiten. Proxies werden hier mittels dicker Linien am Rand dargestellt und deuten so die Richtung des off-screen Objekts an.

Auf der Quintessenz von City Lights weiter aufbauend wurde von P. Baudisch und R. Rosenholtz *Halo* [BR03] entwickelt. Hierbei wird vom off-screen Objekt aus ein Ring hin zum Bildschirmrand aufgespannt, sodass ein Teil des Rings als Proxy sichtbar ist. Analog zu City Lights bestehen Proxies somit aus gekrümmten Linien. Diese Krümmung erlaubt die Abschätzung des Mittelpunkts des Rings – also die Position des off-screen Objekts. Halo wurde ferner von Irani et al. [IGY06] um Interaktionsmöglichkeiten erweitert. So wurde ermöglicht, per Auswahl eines Proxys zum entsprechenden off-screen Objekt zu springen (Hopping). Eine Abwandlung dessen wurde von M. Frisch und R. Dachselt [FD10] sowie auch in dieser Arbeit implementiert. Abbildung 1.3b zeigt Halo angewandt auf einer Landkarte.

Als Alternative zu Halo entwickelten Gustafson et al. *Wedge* [GBG+08] sowie Burigat et al. das in Abbildung 1.3a dargestellte *Arrow* [BCG06]. Wie auch in Halo werden in Wedge Proxies als Linien dargestellt. Jedoch anstelle eines Rings wird vom off-screen Objekt aus ein Dreieck zum Bildschirmrand aufgespannt. Im Gegensatz dazu stellt Arrow Proxies als Pfeile dar, welche auf das entsprechende off-screen Objekt zeigen. Ein Vergleich von Halo, Arrow und Wedge in Augmented sowie Virtual Reality findet in den Arbeiten von Gruenefeld et al. [GAH+17; GAB+18] statt.

Auch außerhalb des Kontexts von KKD werden Proxies im Sinne des Proxy-Views bereits verwendet. So zeigt das von Baudisch et al. für Betriebssysteme entwickelte *Drag-and-Pop* [BCR+03] beim Ziehen von Objekten (Drag) kontextrelevante Anwendungssymbole in der Nähe des Mauszeigers an. In der selben Arbeit wurde das Konzept noch erweitert, sodass auch per Drag auf leerer Fläche mit den Anwendungssymbolen – den Proxies – interagiert werden kann (*Drag-and-Pick*). A. Bezerianos und R. Balakrishnan bauten mit *Vacuum* [BB05] den Ansatz weiter aus. Indem dem Nutzer/der Nutzerin erlaubt wurde auszuwählen, welche Proxies angezeigt werden sollen, kann der selektive Kontext und somit die Anzahl an Proxies weiter eingeschränkt werden. Somit dient Vacuum bereits als eine Art Filter.

Die Anwendung Nonograms Katana[1] bietet ebenfalls eine Variante des Proxy-Views. Es wird die Option geboten, Zahlenbalken am Rand einzurasten, wodurch diese auch bei starkem Zoom stets als Proxy am Bildschirmrand angezeigt werden. Auch Microsoft Excel[2] bietet die Funktion, Zeilen und Spalten einer Tabelle am Rand zu fixieren.

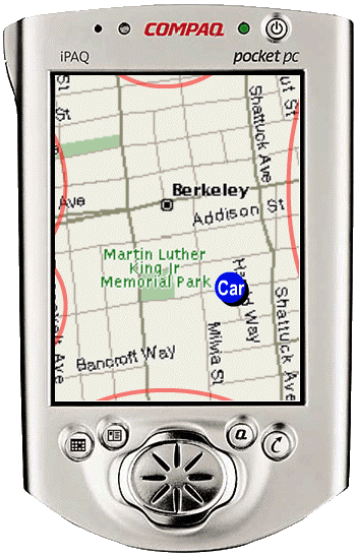
Auch in mehreren Ablegern der Assassin's Creed-Spielereihe[3] ist ein Proxy-View implementiert. Hier werden etwa dem Spieler nahe off-screen Gegner als weißes Leuchten am Bildschirmrand angedeutet – analog zu City Lights [ZMG+03]. In Abbildung 1.3d ein Beispiel hiervon.

[1] https://nonograms-katana.com

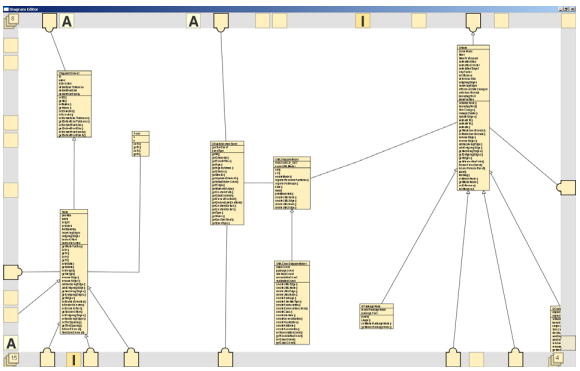
[2] https://www.microsoft.com/en-us/microsoft-365/excel

[3] https://www.ubisoft.com/en-us/game/assassins-creed

(a) Arrow



(b) Halo



(c) Proxy-View für Klassendiagramme

(d) Assassin's Creed Origins

Abbildung 1.3. Diverse Beispiele für Anwendungen des Proxy-Views. Arrow und Halo [BR03] zeigen relevante off-screen Orte auf Karten als Proxies an. In (c) die Implementierung des Proxy-Views für Klassendiagramme von M. Frisch und R. Dachselt [FD10]. Schließlich wird in (d)[a] eine Version des Proxy-Views in Assassin's Creed Origins[b] gezeigt. Vergleichbar zu City Lights [ZMG+03] werden off-screen Gegner als weißes Leuchten am Bildschirmrand angezeigt.

[a] https://youtu.be/EzlR9VCmbTE?t=346

[b] https://www.ubisoft.com/en-us/game/assassins-creed/origins

Kapitel 2

Konzept

Um in KKD intuitiv selektiven Kontext zu geben, wurde der Proxy-View-Ansatz entwickelt [CKB09; FD10]. In den nachfolgenden Abschnitten wird dessen Konzept näher erläutert: Abschnitt 2.1 definiert den einfachen Fall sowie die Grundsätze des Proxy-Views, während Abschnitt 2.2 auf hierarchische Diagramme und die dafür notwendige Erweiterung der Regeln des Proxy-Views eingeht. Schließlich wird in Abschnitt 2.3 das Konzept anhand einiger Mocks und Beispiele weiter verdeutlicht.

2.1. Einfacher Fall

KKD bezeichnen die Klasse aller Diagramme, welche aus Knoten und Kanten bestehen. Der Bildschirm gibt den sichtbaren Bereich des Diagramms an. Zur Navigation des Diagramms kann der Bildschirm verschoben und gezoomt werden (*Zoom + Pan*). Abbildung 2.1 greift auf dieser Definition auf und erweitert sie um den Proxy-View:

Im Proxy-View wird für jeden Knoten außerhalb des Bildschirms eine Miniatur erstellt und stellvertretend für den Knoten am Rand des Bildschirms platziert. Formell gilt im Proxy-View also folgende Regel für einen beliebigen Knoten k eines KKD G:

▷ Ist k off-screen, wird ein Proxy erstellt.

Durch diese *Proxies* können Informationen anderer Knoten ohne weitere Navigation des Diagramms vermittelt werden – Kontext ist somit stets vorhanden. Um nicht von zu vielen Daten überfordert zu werden, enthalten Proxies typischerweise nur die für den Nutzer/die Nutzerin wichtigsten Informationen.

Auch für Kanten können Proxies erstellt werden, hierbei wird zwischen zwei Arten unterschieden: *Kantenproxies* verbinden einen on-screen Knoten mit einem Proxy anstelle des entsprechenden off-screen Knotens. *Segmentproxies* verbinden die Schnittpunkte einer stellenweise off-screen Kante mit dem Bildschirm, sodass der off-screen Verlauf der Kante weiterhin nachvollziehbar ist.

Für das Routing eines Kantenproxys sind unterschiedliche Strategien möglich. So wird per *Straight-Edge-Routing* der Kantenproxy direkt zwischen Knoten und Proxy aufgespannt. Hierbei gehen jegliche Informationen über den Pfad der Originalkante komplett verloren. Im Gegensatz dazu versucht *Along-Border-Routing* den Pfad der Originalkante so weit wie möglich wiederzuverwenden und geht den übrigen Pfad zum Proxy entlang des Bildschirmrands. In Abbildung 2.2 werden beide Strategien beispielhaft verglichen.

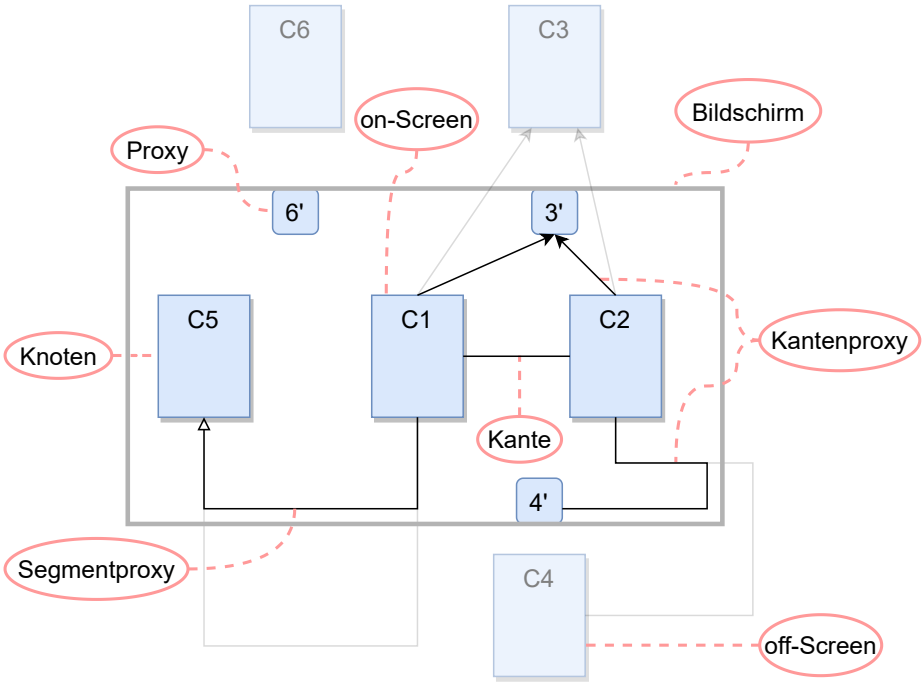
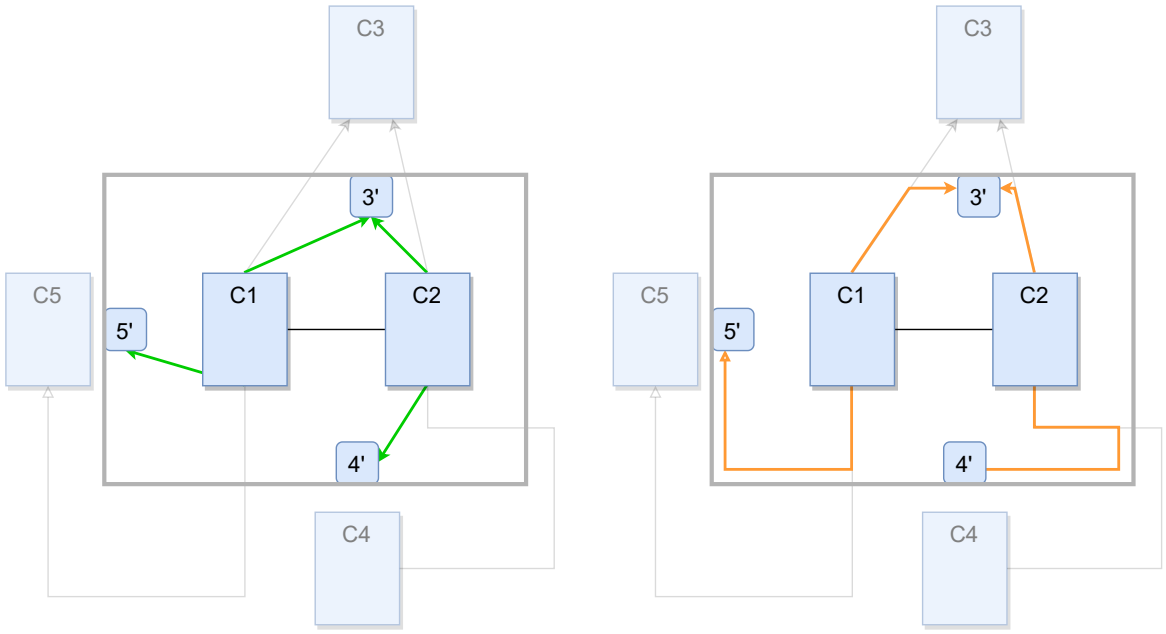


Abbildung 2.1. Definition des einfachen Falls von KKD. KKD bestehen aus Knoten und Kanten. Im Proxy-View werden diese Elemente danach unterschieden, ob sie on-screen – im Bildschirm sichtbar – oder off-screen – außerhalb des Bildschirms – sind. Für off-screen Knoten wird ein Proxy am Rand des Bildschirms platziert. Falls ein off-screen Knoten A per Kante mit einem on-screen Knoten B verbunden ist, kann ein Kantenproxy von B mit dem Proxy von A verknüpft werden. Auch für Kanten, welche stellenweise off-screen sind, kann der nicht sichtbare Verlauf mittels Segmentproxies angedeutet werden. Zur Erhaltung der Konsistenz wird die Originalkante von Kanten- und Segmentproxies verborgen – etwa per Ausblenden.

Es ist möglich, dass Proxies unterschiedlicher off-screen Knoten sich gegenseitig überlappen. In Abbildung 2.3 wird auf diverse Strategien eingegangen, um die daraus resultierende Unordnung zu verringern. Im Normalfall wurden hierbei Proxies nach aufsteigender Reihenfolge erstellt, sodass 5′ etwa unterhalb von 9′ dargestellt ist. *Clustering* fasst überlappende Proxies in einem Cluster zusammen, welches wiederum selbst als Proxy am Bildschirmrand platziert wird – das Label des Clusters entspricht der Anzahl der darin enthaltenen Proxies. Nicht jeder anfänglich überlappende Proxy muss im Resultat geclustert worden sein. So überlappt der Proxy 8′ im Normalfall zwar mit 7′, jedoch wurde letzterer bereits mit 6′ in ein Cluster zusammengefasst – welches nun nicht mehr mit 8′ überlappt. Im Gegensatz dazu fasst *Kaskadierendes Clustering* auch transitiv überlappende Proxies zusammen. Da also 6′ mit 7′ und 7′ mit 8′ überlappen, werden alle drei Proxies einem Cluster zusammengefasst. Die Strategie *Opacity + Stacking Order* folgt einem anderen Ansatz: Überlappende Proxies werden toleriert und sollen nach passendem Relevanzmaß sortiert werden. Als allgemeines Relevanzmaß wurde hier der Abstand eines Knotens zum Bildschirm gewählt, da einander nahe

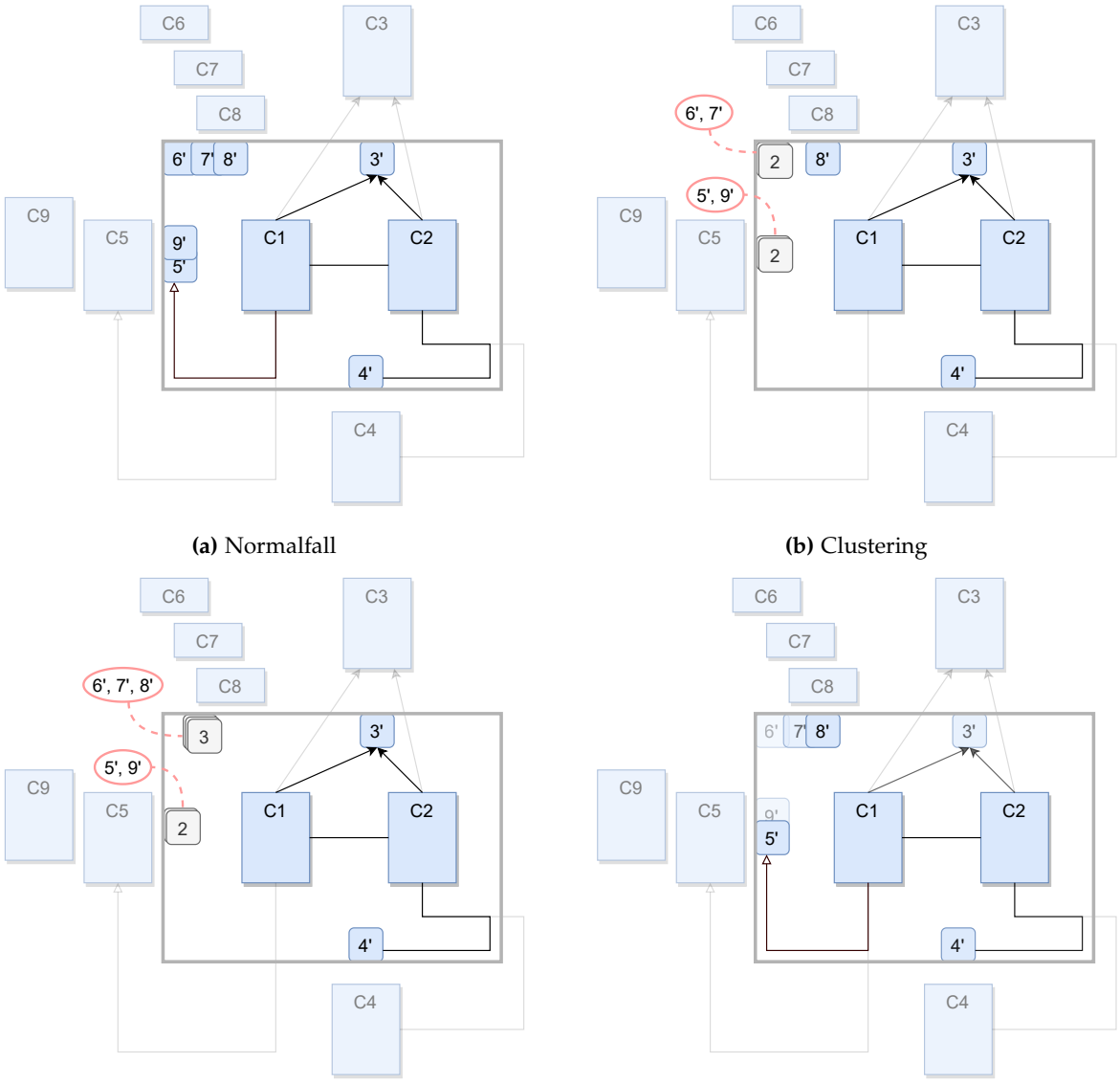


(a) Straight-Edge-Routing **(b)** Along-Border-Routing

Abbildung 2.2. Ein Vergleich verschiedener Routing-Strategien für Kantenproxies. Bei Straight-Edge-Routing (a) wird der Knoten per Gerade mit dem Proxy verbunden. Bei Along-Border-Routing (b) wird der Pfad der Originalkante so weit wie möglich übernommen.

Knoten typischerweise einen Zusammenhang aufweisen [Pet95]. Somit wird, je weiter ein Knoten vom Bildschirm entfernt ist, der entsprechende Proxy umso transparenter dargestellt. Außerdem werden dem Bildschirm nahe Knoten über weiter entfernten Knoten dargestellt – im Bild ist daher 5′ über 9′.

Auch Kantenproxies können überlappen. Hier jedoch gibt es noch keine klare Strategie zu einer optimalen Lösung dessen. Lösungsansätze stellen etwa *Edge-Bundling* (auch Edge-Concentration) [New89; ZXY+13], *Minimize-Edge-Crossings* [EMW86; EW94] oder *Edge-Coloring* [JRF+09] dar. Der NP-schwere Lösungsansatz [Lin00] Edge-Bundling bündelt jeweils mehrere ähnliche Kanten in eine einzelne und verringert so effektiv die von Kanten eingenommene Bildschirmfläche. Hingegen versucht der NP-vollständige Lösungsansatz Minimize-Edge-Crossings die Anzahl an Überkreuzungen von Kanten zu minimieren, was zur Erhöhung der Verständlichkeit beiträgt [PCJ96]. Schließlich färbt Edge-Coloring sich kreuzende Kanten ein, um diese so besser unterscheiden zu können – jedoch kann die Farbe von Kanten semantische Informationen beinhalten, weshalb der Ansatz nicht für allgemeine KKD geeignet ist. Da noch kein Ansatz das Problem bestmöglich löst und Teillösungen algorithmisch komplex – und damit nicht nutzerfreundlich – sind, wird die Überlappung von Kanten in dieser Arbeit nicht weiter behandelt.



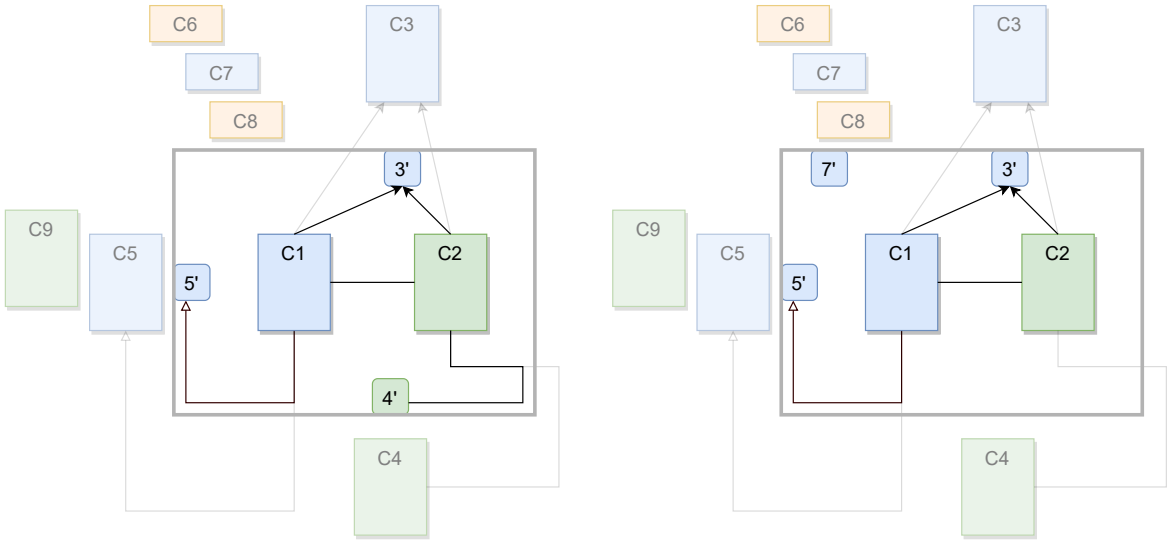
(a) Normalfall

(b) Clustering

(c) Kaskadierendes Clustering

(d) Opacity + Stacking Order

Abbildung 2.3. Ein Vergleich verschiedener Strategien zur Handhabung überlappender Proxies. Ohne dedizierte Strategie (a) können Proxies beliebig überlappen. Mittels Clustering (b) werden überlappende Proxies in Clustern zusammengefasst. Kaskadierendes Clustering (c) fasst außerdem auch transitiv überlappende Proxies in einem Cluster zusammen. Als Alternative zu Clustering hebt Opacity + Stacking Order (d) nahe Proxies hervor, indem entfernte Proxies weniger opak und unterhalb naher Proxies dargestellt werden.



(a) Struktureller Filter **(b)** Semantischer Filter

Abbildung 2.4. Beispiele für mögliche Filter. Links (a) werden Proxies gefiltert, die nicht mit on-screen Knoten verbunden sind. Rechts (b) werden Proxies nicht blauer Knoten gefiltert. Auch eine Kombination beider Filter ist möglich.

In großen KKD kann es viele off-screen Knoten geben – entsprechend nehmen viele Proxies den Bildschirmrand ein. Die zuvor genannten Strategien helfen bereits, Unordnung zu verringern. Jedoch kann bei zu vielen Clustern oder überlappenden Proxies dennoch der Kontext verloren gehen. Mithilfe von *Filtern* hingegen kann Unordnung vermieden werden, indem die Anzahl der dargestellten Proxies selbst reduziert wird: off-screen Knoten werden nach einem oder mehreren Filterkriterien geprüft – nur für den Filter erfüllende Knoten werden Proxies erstellt. Der so erhaltene *selektive Kontext* steigert die Skalierbarkeit des Proxy-Views und verdeutlicht die Eignung dessen auch für große Diagramme. Im Allgemeinen wird zwischen zwei Arten von Filtern unterschieden: *Strukturelle Filter* filtern unabhängig vom Diagrammtyp und nutzen dementsprechend nur strukturelle Informationen eines Diagramms. *Semantische Filter* hingegen filtern auch nach semantischen Informationen und sind somit für spezifische Typen von Diagrammen konstruiert. Abbildung 2.4 enthält potenzielle Filterkriterien.

2.2. Hierarchischer Fall

Hierarchische KKD erweitern die vorige Definition um eine beliebige Verschachtelung von Knoten. In Abbildung 2.5 werden die so neu entstandenen Begriffe definiert: *Hierarchische Knoten* enthalten mindestens einen Knoten und sind damit synonym zu *Elternknoten*. *Kindknoten* bezeichnen jene Knoten, welche in anderen enthalten sind.

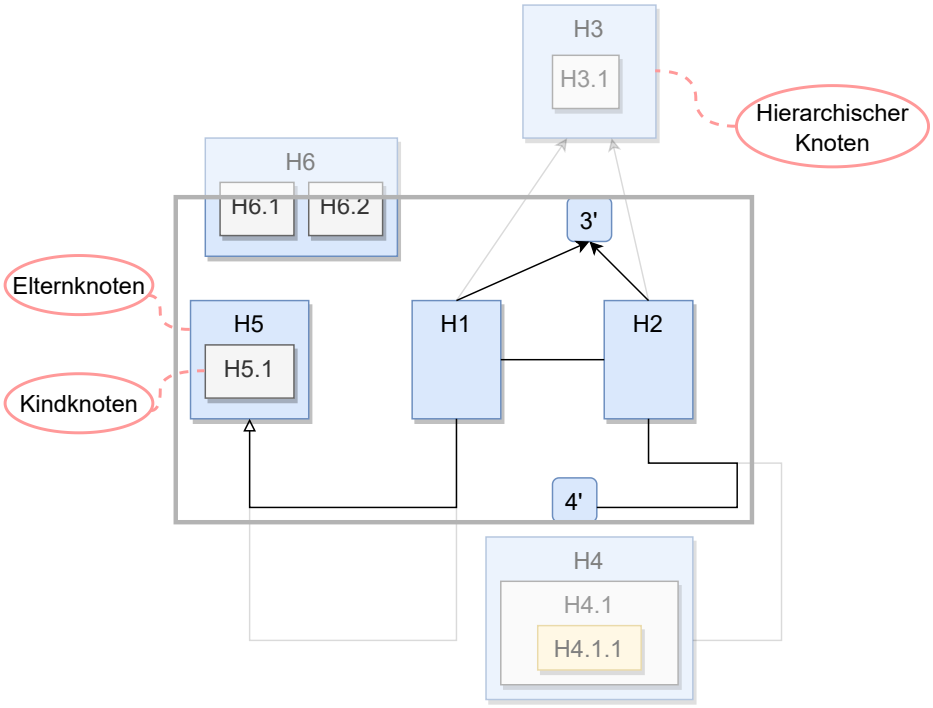
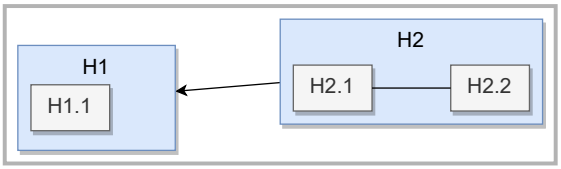
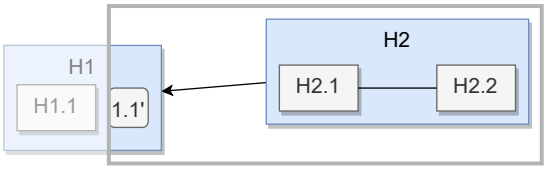


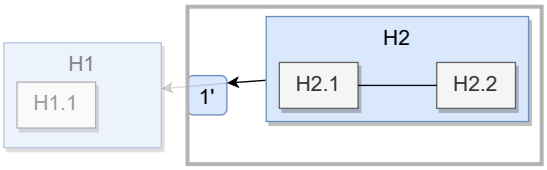
Abbildung 2.5. Erweiterung der Definition aus Abbildung 2.1 um hierarchische KKD. Hierarchische KKD erweitern den einfachen Fall um hierarchische Knoten. Hierarchische Knoten enthalten mindestens einen Knoten, sodass eine beliebige Schachtelungstiefe möglich ist. Als Synonym werden diese auch als Elternknoten bezeichnet. Kindknoten hingegen sind in anderen Knoten enthalten.



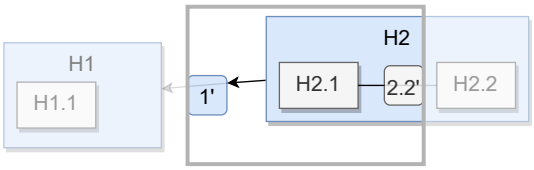
(a) Basisfall



(b) Kindproxy



(c) Elternproxy



(d) Kind- und Elternproxy

Abbildung 2.6. Erweiterung der Regeln des Proxy-Views für hierarchische KKD. In (a) der Basisfall – alle Knoten sind on-screen. Durch sukzessives Verkleinern des Bildschirms geraten Knoten jeweils off-screen.

Für den Proxy-View ergeben sich durch diese Erweiterung nun neue Regeln. Sei hierzu k ein beliebiger Knoten eines hierarchischen KKD G. Falls k keinen Elternknoten besitzt – er selbst somit kein Kindknoten ist – gilt für k die Regel des einfachen Falls:

▷ Ist k off-screen, wird ein Proxy erstellt.

Falls k hingegen einen Elternknoten e besitzt, gilt folgende Regel:

▷ Sind k off-screen und e on-screen, wird für k ein Proxy gekappt in e erstellt – dieser kann somit nicht über e hinausragen.

Insbesondere wird für k also kein Proxy erstellt, falls k und e off-screen sind – eventuell jedoch für e. Intuitiv wird demnach ein Proxy für den jeweils äußersten Knoten erstellt, welcher vollständig off-screen ist. Abbildung 2.6 verdeutlicht diese Definition: In (b) ist der Knoten H1 teilweise on-screen, nur für den off-screen Kindknoten H1.1 wird ein Proxy 1.1′ erstellt. In (c) hingegen ist H1 selbst off-screen und hat einen Proxy – 1.1′ wird somit nicht angezeigt. Schließlich wird in (d) eine Kombination beider vorangegangenen Fälle gezeigt – auch für Knoten unterschiedlicher Hierarchieebenen können Proxies dargestellt werden.

2.3. Mocks und Beispiele

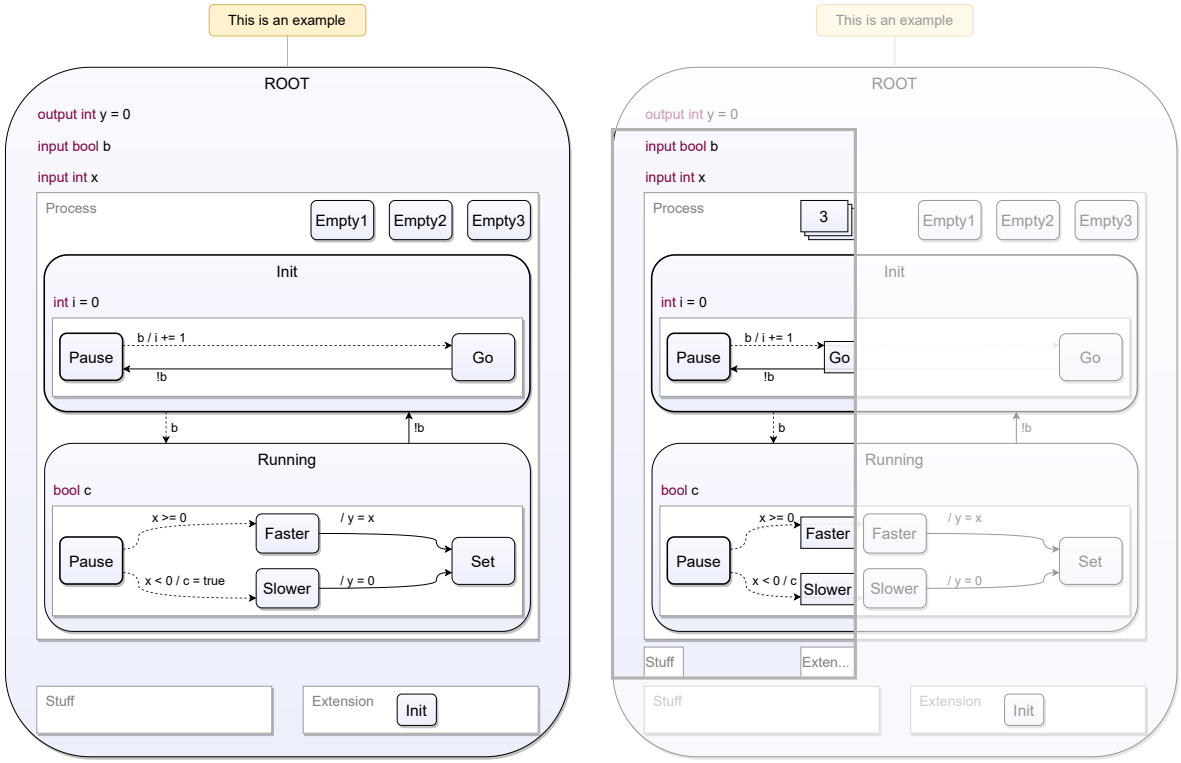
Zur Verdeutlichung, wie der Proxy-View für unterschiedliche KKD angewendet werden kann, enthält dieser Abschnitt einige Mocks und Beispiele.

So wird in Abbildung 2.7 der Proxy-View konzeptuell für SCCharts gezeigt. Die oberen Zustände *Empty1*, *Empty2* und *Empty3* haben die gleiche vertikale Position, wodurch die drei entsprechenden Proxies überlappen und in einem Cluster zusammengefasst werden. Ebenso wird für den darunter liegenden Zustand *Go* ein Proxy am Rand des Rahmens platziert und per Kantenproxy mit *Pause* verbunden. In *Running* hingegen sind nur Proxies für *Faster* und *Slower* enthalten – etwa durch ein Filterkriterium kann so vom Zustand *Set* abstrahiert werden. Am unteren Rand befinden sich die off-screen Regionen *Stuff* und *Extension*. Analog zu Zuständen werden auch für diese Knoten Proxies erstellt. Der Proxy für *Extension* zeigt schließlich eine Möglichkeit, wie mittels Kürzen des Textes zu *Exten...* mit zu langen Labels umgegangen werden kann.

Darüber hinaus ist es auch möglich, den Proxy-View für Diagramme in Lingua Franca (LF) [LMS+20; LMB+21] zu verwenden. Die Koordinierungssprache LF erlaubt die Definition von Reaktorprogrammen auf hohem Abstraktionsniveau [Fou21; DCB+21]. Abbildung 2.8 zeigt ein Mock des Proxy-Views in LF: Für den Reaktor *Door* ist ein Proxy vorhanden – dessen Kantenproxies wurden mit den entsprechenden eingehenden Ports verbunden. Auch für Reaktionen werden Proxies erstellt. So befindet sich ein Proxy für Reaktion 1 in *Cockpit* am linken Rand des Bildschirms – in diesem Beispiel wird erneut der daneben liegende Knoten für die Startaktion per Filter nicht als Proxy dargestellt. Schlussendlich ist auch ein Proxy für den Port *empty* vorhanden. Die Pfeilspitze signalisiert, dass es sich um einen Port-Proxy

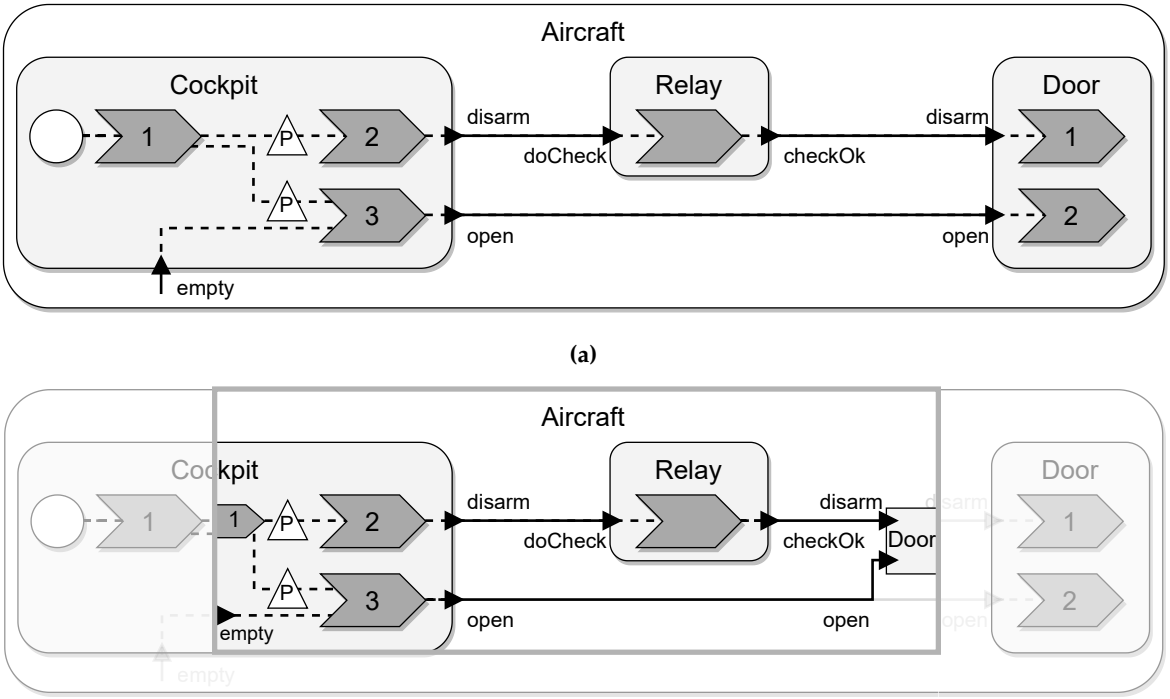
handelt. Das Label des Ports kann entweder dauerhaft angezeigt werden oder erscheint, sobald der Nutzer/die Nutzerin mit der Maus über den Proxy hovert.

Abschließend werden in Abbildung 1.3 weitere Anwendungen des Proxy-Views gezeigt.



(a) **(b)**

Abbildung 2.7. Links ein Mock eines SCCharts, rechts die exemplarische Anwendung des Proxy-Views.



(a)

(b)

Abbildung 2.8. Oben ein Mock eines Lingua-Franca-Diagramms, unten die exemplarische Anwendung des Proxy-Views.

Kapitel 3

Implementierung

Das Framework KIELER Lightweight Diagrams (KLighD) erlaubt, KKD anhand textueller Modelle in Eclipse[1] dynamisch erstellen zu lassen [SSH13]. Das von TypeFox entwickelte Framework Sprotty[2] erstellt Diagramme als Scalable Vector Graphics (SVG) – welche in den meisten modernen Browsern und auch Visual Studio Code (VS Code) direkt angezeigt werden können. Aufgrund des Trends vieler Applikationen zu Webtechnologien zu migrieren, kann KLighD mittels Sprotty auch web-basiert verwendet werden [Ren18; Dom18] – etwa mittels der KLighD VS Code-Extension (KLighD-VS Code)[3].

In KLighD-VS Code wird das Language Server Protocol (LSP) angewandt. VS Code nutzt Sprotty zur Darstellung der KKD und stellt damit die Client-Seite des LSP dar. Zur Entwicklung wird die Programmiersprache TypeScript[4] genutzt, welche den geschriebenen Code in die weit verbreitete Programmiersprache JavaScript transpiliert. KLighD wiederum entspricht der Server-Seite des LSP und ist somit für die Erstellung sowie das Layout eines KKD verantwortlich. Um die textuelle Darstellung dessen in eine Graph-Struktur umzuwandeln, wird eine dem Format des KKD entsprechenden gegebene Synthese verwendet. Das Layout des resultierenden Graphen wird schließlich an den Eclipse Layout Kernel (ELK)[5] delegiert. Abbildung 3.1 verdeutlicht diesen Ablauf.

Außerdem können in Sprotty über dem gerenderten SVG auch weitere Elemente des User Interface (UI) angezeigt werden. Hierfür wird die *UI-Extension*-Schnittstelle zur Verfügung gestellt. Die in Abbildung 3.2 gezeigte Sidebar aus KLighD-VS Code ist ein Beispiel einer UI-Extension: Im Vordergrund des KKD können Nutzer/Nutzerinnen Einstellungen etwa bezüglich des Layouts oder Aussehen dessen vornehmen. Analog würden auch Minimaps wie VS Codes Code Outline[6] oder Androids Bild im Bild[7] in Sprotty Architektur UI-Extensions entsprechen. Cockburn et al. nennen weitere verwandte Beispiele [CKB09].

Implementiert wurde der Proxy-View schließlich als UI-Extension in KLighD-VS Code. Im Folgenden wird in Abschnitt 3.1 auf die allgemeine Implementierung des Proxy-Views für KKD eingegangen. Abschnitt 3.2 geht im Anschluss auf die serverseitige Erweiterung spezifisch für SCCharts ein. Abschließend werden in Abschnitt 3.3 weitere Themen der Implementierung außerhalb des Proxy-Views vorgestellt, die im Rahmen dieser Arbeit behandelt wurden.

[1] https://www.eclipse.org

[2] https://projects.eclipse.org/projects/ecd.sprotty

[3] https://github.com/kieler/klighd-vscode

[4] https://www.typescriptlang.org

[5] https://www.eclipse.org/elk

[6] https://code.visualstudio.com/docs/getstarted/userinterface#_minimap

[7] https://developer.android.com/guide/topics/ui/picture-in-picture

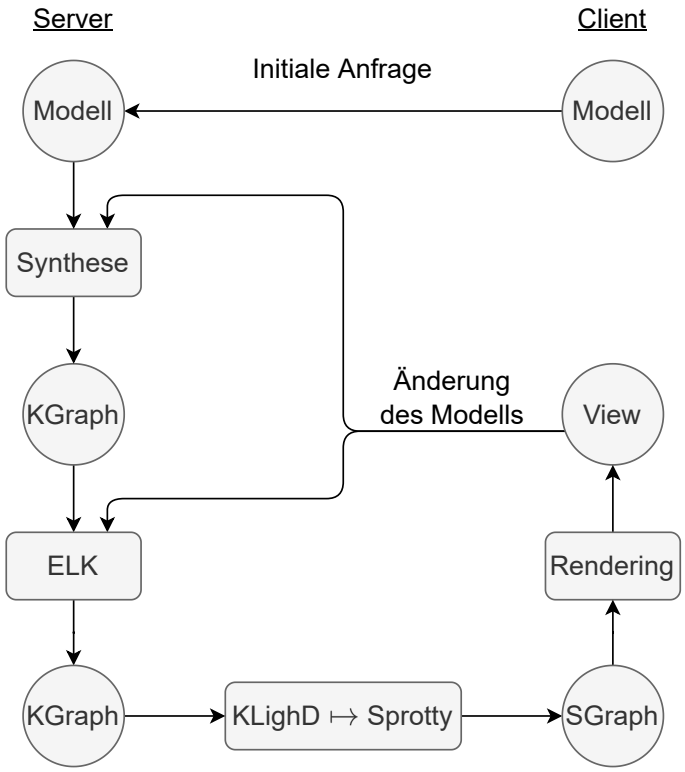


Abbildung 3.1. Veranschaulichung der Kommunikation von KLighD-VS Code mittels Language Server Protocol. Zunächst wird das textuelle Modell vom Client zum Server gesendet. Anschließend wird ein Graph synthetisiert und mittels ELK gelayouted. Schließlich wird der entstandene Graph vom KLighD-Format `KGraph` in das Sprotty-Format `SGraph` übersetzt und so gerendert. Bei Änderung eines bereits gerenderten Modells werden Synthese und/oder Layout wiederholt und der View dynamisch aktualisiert. N. Rentz geht auf diese Architektur in mehr Detail ein [Ren18].

3.1. Allgemeine Knoten-Kanten-Diagramme

Der Proxy-View wurde als UI-Extension realisiert. Proxies und weitere Elemente des Proxy-Views werden so direkt über dem tatsächlichen KKD platziert. Dieses soll insbesondere nicht vom Proxy-View verändert, sondern erweitert werden. Durch Realisierung als UI-Extension werden

- die Integrität des KKD sichergestellt,
- kognitiv belastende Kontextwechsel [CKB09] vermieden, wenn Elemente des KKD plötzlich verschwinden oder auftauchen – etwa wenn ein Knoten zu einem Proxy wird,
- eine einfache und unkomplizierte Ein- und Ausschaltung des Proxy-Views ermöglicht,
- der Proxy-View von der restlichen Applikation bestmöglich abgekapselt, sodass dieser durch die geringe externe Abhängigkeit leicht erweiterbar ist,

▷ der Proxy-View an passender Stelle in die Struktur des Document Object Model (DOM) eingefügt – auf gleicher Ebene des KKD-Views.

Der SVG-Tag im DOM wird schließlich dynamisch an die Größe des Bildschirms angepasst. Da der Proxy-View über dem KKD-View liegt, muss der Tag außerdem mittels der Cascading Style Sheets (CSS)-Eigenschaft `pointer-events: none` durchklickbar gemacht werden – so kann weiterhin mit dem KKD-View interagiert werden.

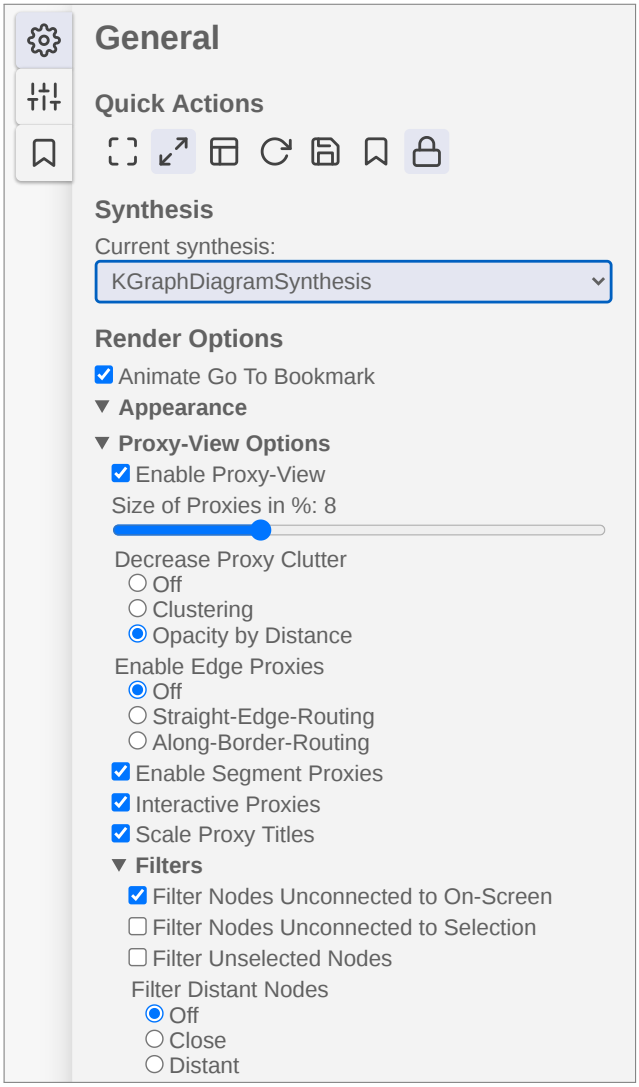
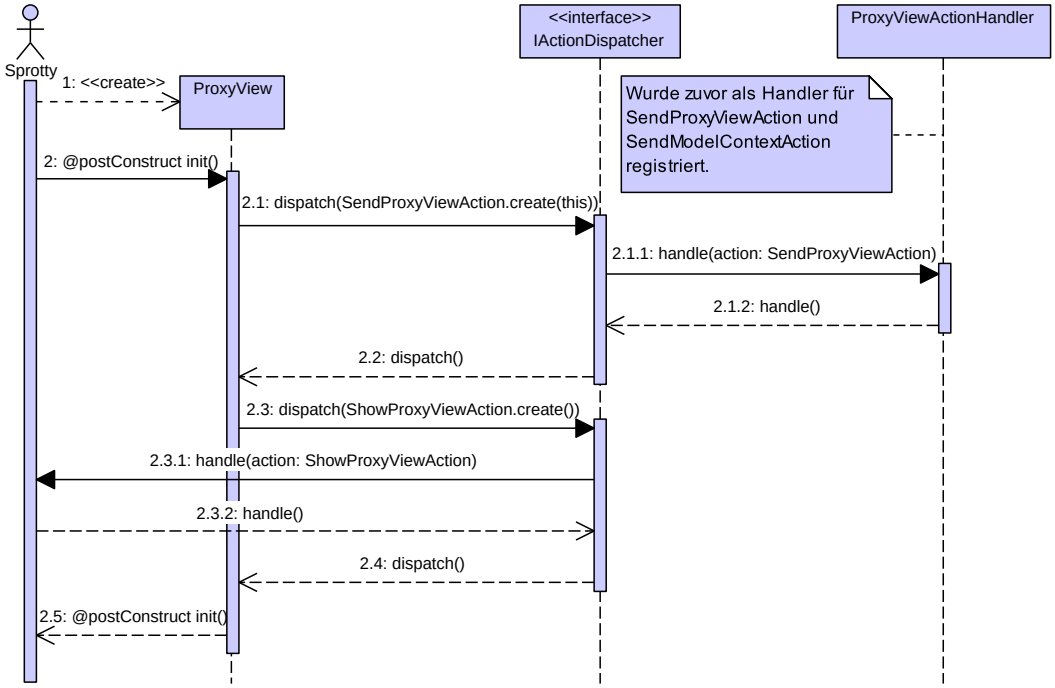
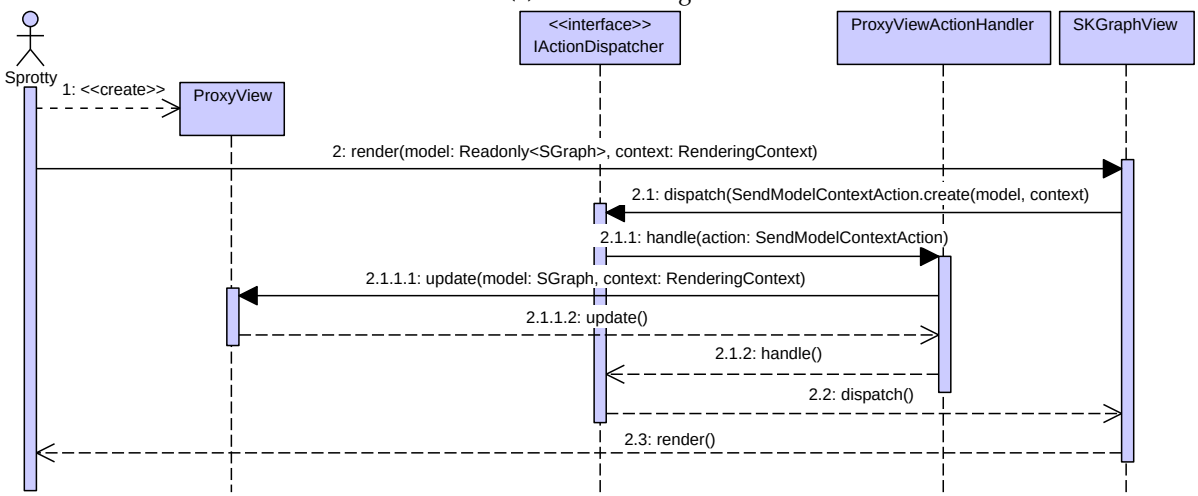


Abbildung 3.2. Die Sidebar in KLighD-VS Code. Diese ermöglicht dem Nutzer/der Nutzerin Einstellungen etwa bezüglich des Layouts oder der Darstellung des Diagramms zu ändern, welche dynamisch angewandt werden. Auch das Verhalten und Aussehen des Proxy-Views können hierüber auf viele Weisen angepasst werden.

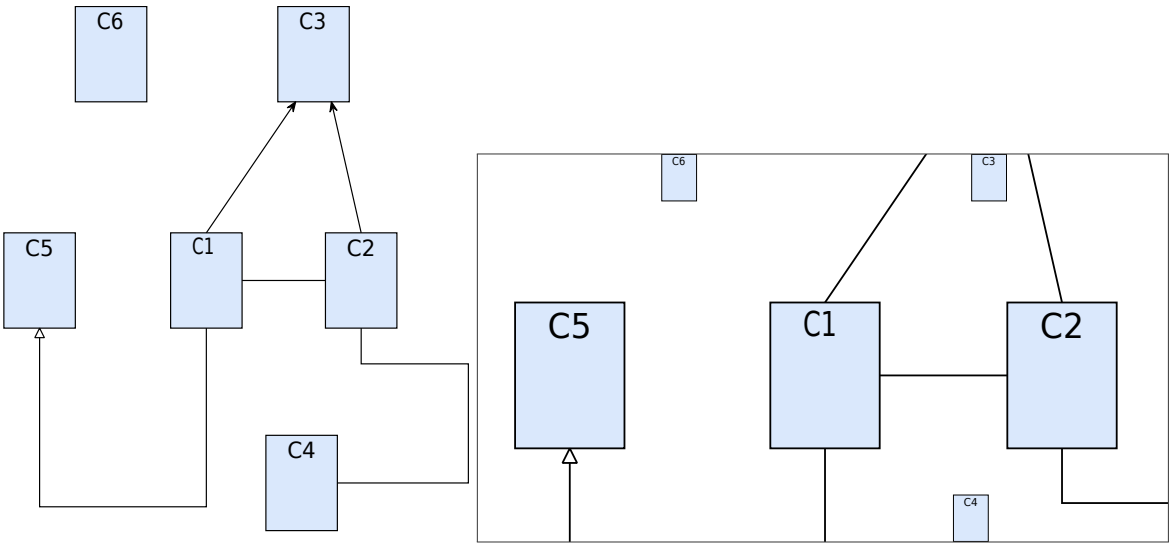


(a) Initialisierung



(b) Update

Abbildung 3.3. Eine Übersicht der Proxy-View Kommunikation mittels Sprottys `Actions`. Zur Initialisierung (a) erstellt Sprotty alle UI-Extensions – inklusive Proxy-View, welcher bei Erstellung zwei `Actions` sendet. Die erste `Action`, `SendProxyViewAction`, enthält die Instanz des Proxy-Views zur Vermeidung zyklischer Abhängigkeiten mittels `@inject`. Mit der zweiten `Action` wird Sprotty anschließend die Anweisung gegeben, den Proxy-View sichtbar zu machen. In (b) wird der Update-Schritt des Proxy-Views gezeigt: Jeder Aufruf von `SKGraphViews render()` leitet Modell und Kontext auch an den Proxy-View weiter.



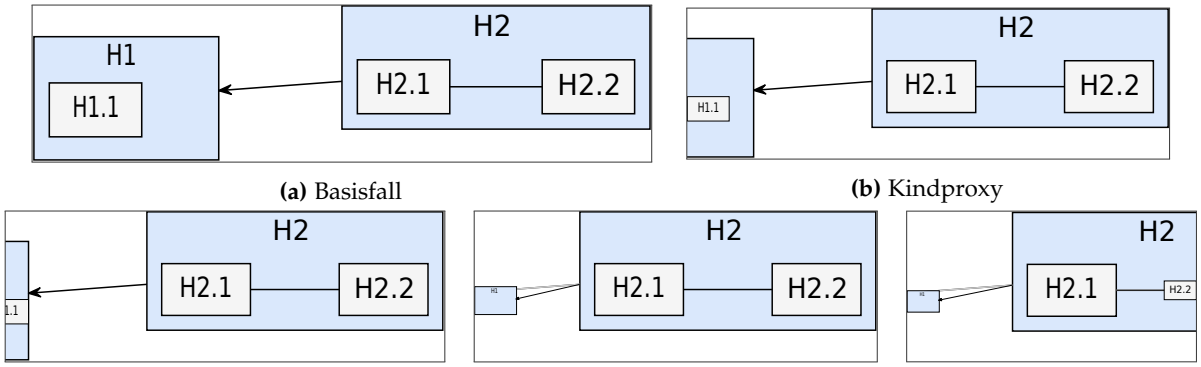
(a) Gesamtes KKD **(b)** Ausschnitt

Abbildung 3.4. In der Implementierung erstellte Proxies für einen `KGraph`. Links (a) das gesamte KKD, rechts (b) ein Ausschnitt mit angewandtem Proxy-View. Für jeden off-screen Knoten wurde ein Proxy erstellt.

Zur Erstellung von Proxies und auch für weitere Logik des Proxy-Views werden das aktuelle Modell sowie der Rendering-Kontext benötigt. Eine Injektion dieser mittels Inversify-JS[8] führt zu einer zirkulären Abhängigkeit. Um dies zu umgehen werden Sprottys `Actions` verwendet: Modell und Kontext werden vom KKD-View in jedem Rendering-Schritt verschickt und von einem entsprechenden `ActionHandler` empfangen. Auch hier jedoch kann es zur zirkulären Abhängigkeit kommen, wenn der Proxy-View die `Actions` selbst verarbeitet – somit wird ein `ProxyViewActionHandler` benötigt. Damit dieser schließlich Modell und Kontext an den Proxy-View weiterleiten kann, muss die erstellte Instanz des Proxy-View mittels einer `SendProxyViewAction` verschickt und vom Handler empfangen werden. Abbildung 3.3 verdeutlicht die beschriebene Kommunikation: Nach der Erstellung wird zunächst die `SendProxyViewAction` gesendet. Anschließend muss der Proxy-View außerdem mittels einer `ShowProxyViewAction` sichtbar gemacht werden – per Default werden UI-Extensions nicht angezeigt. Ist diese Initialisierung abgeschlossen, wird schließlich durch jeden Rendering-Schritt des KKD-Views auch ein Update-Schritt des Proxy-Views ausgelöst.

Somit wird also ermöglicht, die in Kapitel 2 genannten Konzepte des Proxy-Views umzusetzen: *Proxies* werden für jeden off-screen Knoten erstellt und an entsprechender Stelle am Rand des Bildschirms platziert – unter Berücksichtigung der Sidebar. Für allgemeine KKD wird das vorhandene Rendering des Knotens hierbei übernommen und dessen Kindknoten entfernt – nur Label des Knotens bleiben zur Zuordnung vorhanden. Abbildungen 3.4 und 3.5 zeigen die so erstellten Proxies. Insbesondere wird in Abbildung 3.5c verdeutlicht, dass ein Kindproxy

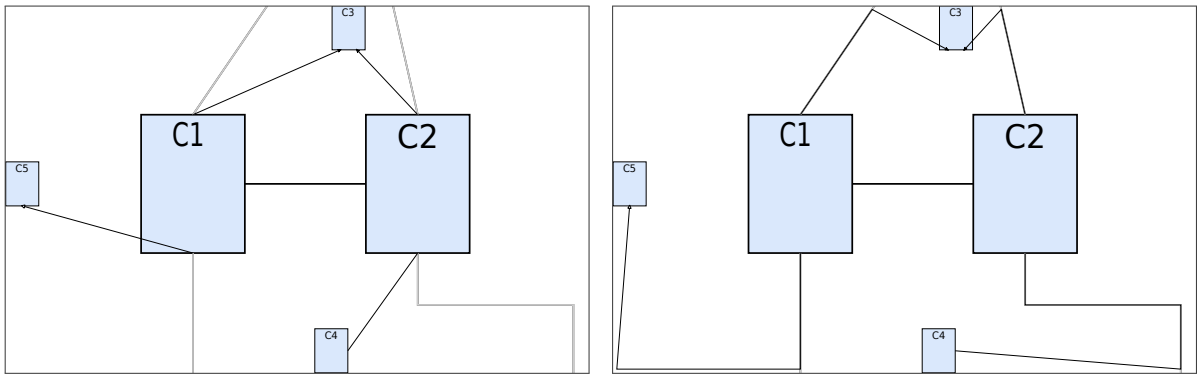
[8] `https://inversify.io`



(a) Basisfall **(b)** Kindproxy

(c) Kindproxy gekappt in Elternknoten **(d)** Elternproxy **(e)** Kind- und Elternproxy

Abbildung 3.5. Proxies in einem hierarchischen KKD. Im Basisfall (a) sind alle Knoten on-screen. Durch sukzessives Verkleinern des Bildschirms werden Proxies für Kind- und Elternknoten erstellt. Insbesondere wird in (c) gezeigt, dass ein Kindproxy den Elternknoten nicht verlassen kann. In (d) und (e) sind *H1* und *H2* per Kantenproxy verbunden.



(a) Straight-Edge-Routing **(b)** Along-Border-Routing

Abbildung 3.6. Ein Vergleich der implementierten Routing-Strategien für Kantenproxies. Straight-Edge-Routing (a) verbindet Knoten mittels Geraden mit Proxies. Along-Border-Routing (b) hingegen übernimmt möglichst viel des Pfads der Kante.

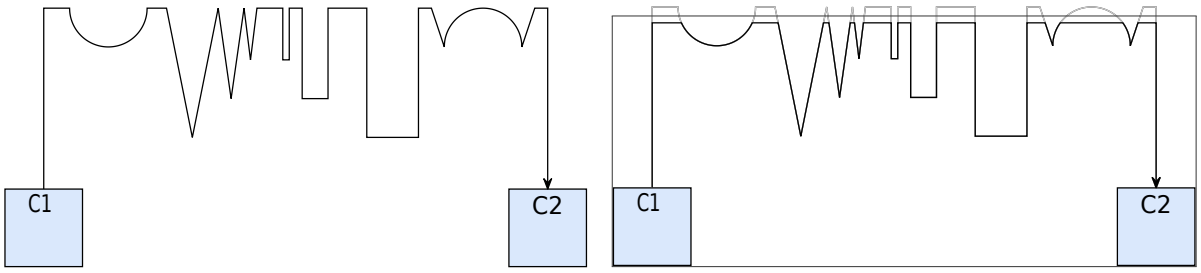
im Elternknoten gekappt bleibt – nur bei angeschalteter Debug-Option können Kindproxies auch über ihre Eltern hinausragen. Synthesen ist es außerdem möglich, das Proxy-Rendering eines Knotens mittels der Eigenschaft `KlighdProperties.PROXY_VIEW_PROXY_RENDERING` zu definieren. Ferner kann per `KlighdProperties.PROXY_VIEW_RENDER_NODE_AS_PROXY` spezifiziert werden, ob für einen Knoten ein Proxy erstellt werden soll. Diese Schnittstellen werden in Abschnitt 3.2 am Beispiel von SCCharts weiter ausgeführt. Darüber hinaus wurde auch für Sequentially Constructive Graphs (SCGraphs) ein spezifisches Proxy-Rendering definiert.

Kantenproxies verbinden on-screen Knoten mit Proxies anstelle der entsprechenden off-screen Knoten. Die Originalkante wird zur Übersichtlichkeit nur verblasst angezeigt. In Abbildung 3.6 werden die hierfür implementierten Routing-Strategien verglichen. *Straight-*

Edge-Routing spannt Kantenproxies direkt zwischen Knoten und Proxy auf. Dem gegenüber folgt *Along-Border-Routing* dem Pfad der Originalkante so weit wie möglich und geht den übrigen Pfad zum Proxy entlang des Bildschirmrands. Vom Konzept abweichend bleibt in der Implementierung jedoch die Verbindungsstelle des Knotens auch beim Proxy erhalten – eventuelle semantische Informationen werden somit erhalten. Zwar nicht implementiert, kann der Pfad zur Verbindungsstelle wie in Abbildung 4.4b jedoch entlang des Proxys geroutet werden.

Mittels *Segmentproxies* können off-screen Segmente von Kanten verständlich dargestellt werden. Hierbei werden je zwei Schnittpunkte zwischen Kante und Bildschirm verbunden, wodurch der off-screen Verlauf der Kante angedeutet wird. Abbildung 3.7 enthält ein Beispiel eines Segmentproxys: Auch hier wird die Originalkante, kurz bevor diese den Bildschirm verlässt, nur blass angezeigt. Diese Stellen werden vom Segmentproxy miteinander verbunden – auch der Pfad beliebiger Formen bleibt so nachvollziehbar.

Abbildung 3.9 enthält eine Übersicht diverser implementierter Strategien zur Handhabung überlappender Proxies: Gegenüber dem Normalfall fasst *Clustering* überlappende Proxies in Clustern zusammen – das Label gibt hierbei Informationen über die Anzahl der darin



(a) Gesamtes KKD **(b)** Segmentproxy

Abbildung 3.7. Ein Beispiel eines in der Implementierung erstellten Segmentproxys. Off-screen Segmente der Kante werden beim Segmentproxy weiterhin dargestellt.

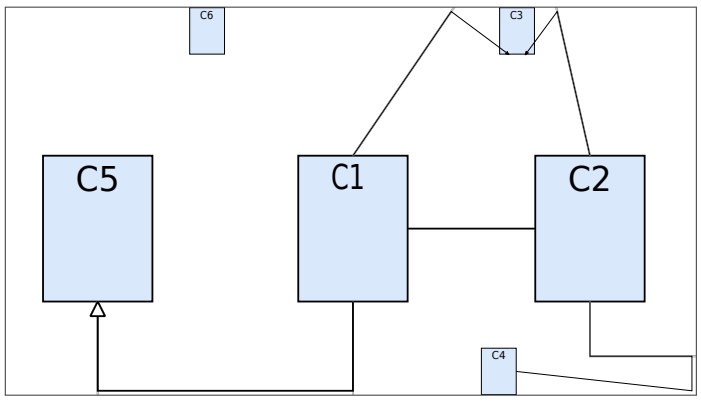
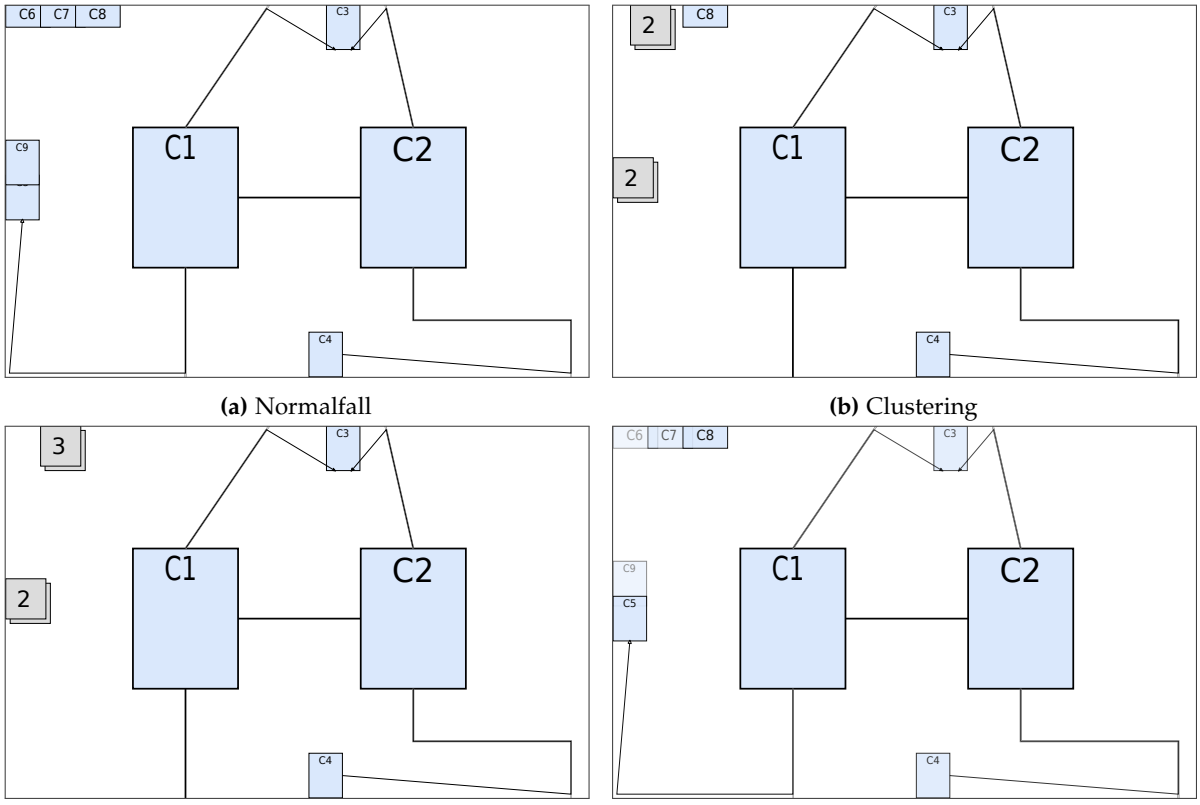


Abbildung 3.8. Eine umfassende Übersicht der Implementierung analog zur konzeptuellen Übersicht aus Abbildung 2.1.

enthaltenen Proxies. Nicht jeder anfänglich überlappende Proxy muss in einem Cluster enthalten sein – so wurde im Beispiel C8 etwa nicht geclustert. Dahingegen fasst *Kaskadierendes Clustering* auch transitiv überlappende Proxies zusammen. Schließlich werden mittels *Opacity + Stacking Order* überlappende Proxies toleriert und mit steigendem Abstand zum Bildschirm transparenter dargestellt. Außerdem werden nahe Knoten über entfernten Knoten dargestellt – entgegen dem Normalfall wird C5 somit über C9 dargestellt.

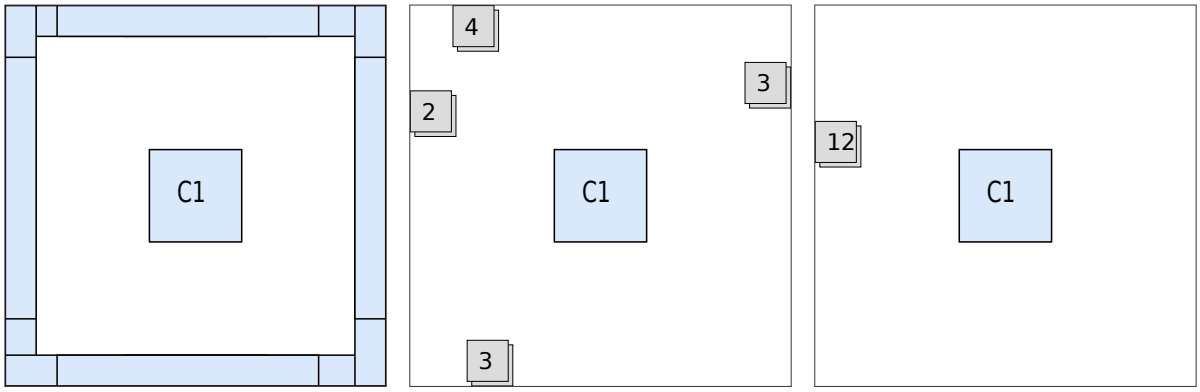
Jedoch kann kaskadierendes Clustering bei einer hohen Anzahl an Proxies zu für den Nutzer/die Nutzerin willkürlich erscheinenden Positionen der Cluster führen. In Abbildung 3.10 werden Clustering-Verfahren in einem solchen Extremfall verglichen: Überlappende Proxies füllen hierbei den gesamten Bildschirmrand aus. Mittels Clustering werden diese in mehreren Clustern zusammengefasst und jeweils an geeigneter Stelle platziert. Kaskadierendes Clustering hingegen fasst alle Proxies in einem Cluster zusammen – die Position dessen gibt jedoch



(a) Normalfall **(b)** Clustering

(c) Kaskadierendes Clustering **(d)** Opacity + Stacking Order

Abbildung 3.9. Ein Vergleich der implementierten Strategien zur Handhabung überlappender Proxies. Ohne eingeschaltete Strategie (a) können Proxies beliebig überlappen. Clustering (b) hingegen fasst überlappende Proxies in Clustering zusammen. Kaskadierendes Clustering (c) erweitert dies, indem auch transitiv überlappende Proxies in einem Cluster zusammengefasst werden. Dementgegen hebt Opacity + Stacking Order (d) nahe Proxies hervor, indem entfernte Proxies transparenter und unterhalb von nahen Proxies dargestellt werden.



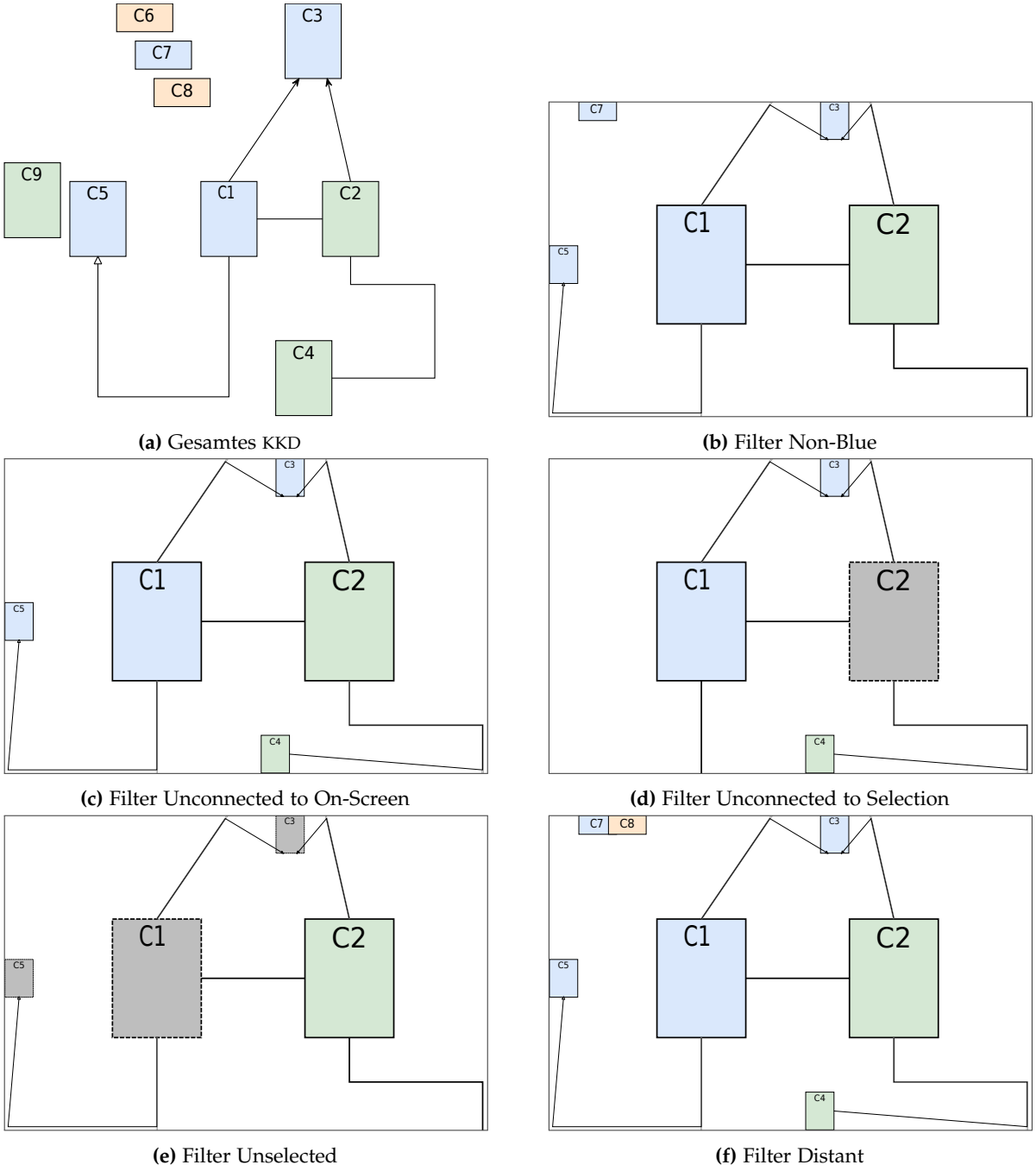
(a) Normalfall **(b)** Clustering **(c)** Kaskadierendes Clustering

Abbildung 3.10. Ein Vergleich der Clustering-Verfahren im Extremfall. Im Normalfall (a) überlappen Proxies am ganzen Bildschirmrand. Clustering (b) fasst die Proxies in insgesamt vier Clustern diverser Größen zusammen, welche an jeweils entsprechenden Stellen platziert werden. Kaskadierendes Clustering (c) erstellt nur ein Cluster bestehend aus allen Proxies. Dieses würde als Mittelwert der Proxy-Positionen über C1 platziert werden, zur Vermeidung frei schwebender Proxies wird das Cluster jedoch an den Rand des Bildschirms gekappt.

keinen Rückschluss darauf, dass auch dem Cluster gegenüberliegende Proxies darin enthalten sind. Darüber hinaus kann unter Verwendung eines Sweep-Line-Ansatzes [SH76; Sou05] ein erheblicher rechnerischer Aufwand entstehen. Tabelle 4.3 vergleicht die durchschnittlichen Rechenzeiten der Verfahren für diverse Anzahlen an Proxies. Aus diesen Gründen wurde kaskadierendes Clustering schließlich als Debug-Option implementiert.

Filter verringern anhand eines oder mehrerer Filterkriterien die Anzahl angezeigter Proxies. In der Sidebar können diese an- und ausgeschaltet und somit auch beliebig kombiniert werden. Abbildung 3.11 enthält Beispiele für die vier implementierten strukturellen Filter: Mittels *Filter Unconnected to On-Screen* werden alle Proxies gefiltert, die nicht mit mindestens einem on-screen Knoten verbunden sind. Daher sind im Beispiel für die Knoten C6 – C9 keine Proxies vorhanden. *Filter Unconnected to Selection* verschärft das vorige Kriterium: Hier werden nur Proxies angezeigt, welche mit selektierten Knoten verbunden sind. Dementsprechend wird für C5 kein Proxy angezeigt – der Knoten ist nicht mit der Selektion verbunden. Hingegen zeigt *Filter Unselected* nur Proxies selektierter Knoten an. Auf diese Weise kann der Nutzer/die Nutzerin den *selektiven Kontext* frei definieren. Den letzten implementierten strukturellen Filter stellt *Filter Distant* dar. Dieser filtert Proxies abhängig von der Distanz des Knotens zum Bildschirm – da, Secondary Notation [Pet95] entsprechend, nah zusammenliegende Knoten typischerweise einen Zusammenhang aufweisen. Im Beispiel sind C6 und C9 zu weit vom Bildschirm entfernt und werden somit nicht angezeigt. Auch in der Situation von Abbildung 3.10 wären Filter hilfreich, da die Verringerung der Gesamtanzahl an Proxies etwaige Clustering-Verfahren entlasten würde.

Es ist möglich, weitere strukturelle Filter zu definieren – so könnten etwa auch den Kontext betreffende Outlier erkannt und gefiltert werden [HA04]. Hierfür definiert der Proxy-View



(a) Gesamtes KKD

(b) Filter Non-Blue

(c) Filter Unconnected to On-Screen

(d) Filter Unconnected to Selection

(e) Filter Unselected

(f) Filter Distant

Abbildung 3.11. Eine Übersicht implementierter struktureller Filter und möglicher semantischer Filter. In (a) das gesamte KKD. Als Beispiel eines semantischen Filters werden in (b) alle nicht blauen Knoten gefiltert. (c)–(f) hingegen entsprechen strukturellen Filtern: In (c) werden Proxies gefiltert, die nicht mit on-screen Knoten verbunden sind. Analog werden in (d) Proxies gefiltert, welche nicht mit selektierten Knoten verbunden sind – die Selektion wird hier grau hinterlegt dargestellt. In (e) wiederum werden Proxies nicht selektierter Knoten direkt gefiltert. Schließlich zeigt (f) einen Filter nach Distanzmaß.

eine Filter API: Das Interface `ProxyFilterArgs` definiert Argumente, welche dem Filter bei der Berechnung, ob der Proxy angezeigt werden soll, zur Verfügung stehen. Diese Argumente beinhalten

- den Knoten, für den ein Proxy erstellt werden soll,
- eine Liste aller off-screen Knoten,
- eine Liste aller on-screen Knoten,
- den Bildschirm definierende Attribute – Größe, Breite, Position und Zoomstufe,
- die Distanz des Knotens zum Bildschirm.

Auch weitere Informationen können anhand dieser Argumente einfach erlangt werden. Der Typ `ProxyFilter` ist definiert als `(args: ProxyFilterArgs) => boolean` und bezeichnet somit jede Funktion, welche die gegebenen Argumente auf einen booleschen Wert abbildet – per Rückgabewert `false` wird der Proxy herausgefiltert. Schließlich können die definierten `ProxyFilter` mittels der Methode `registerFilters()` des Proxy-Views registriert werden. Hierfür wird per Dokumentation empfohlen, die Filter bei der Registrierung zu ordnen – primär nach Stärke des Filterkriteriums, sekundär nach Kosten der Überprüfung. Somit wird sichergestellt, dass

1. Proxies bereits früh herausgefiltert werden und infolgedessen pro Proxy möglichst wenige Filterkriterien geprüft werden müssen,
2. weniger rechenintensive Filter zuerst geprüft werden und daher teurere Filter potenziell vermieden werden können.

`ProxyFilter`, die in Abhängigkeit anderer Eigenschaften zur Laufzeit umdefiniert werden, können mittels `unregisterFilters()` die Registrierung rückgängig machen oder sich direkt erneut registrieren und damit die vorige Definition überschreiben.

Abbildung 3.11b zeigt außerdem ein Beispiel eines semantischen Filters: Nur für blaue Knoten werden Proxies angezeigt – in Klassendiagrammen könnte dies etwa einem Filter nach abstrakten Klassen entsprechen. Die mit der Arbeitsgruppe gemeinsam entwickelte API für semantische Filter wird auch vom Proxy-View unterstützt: In Synthesen können Regeln für semantische Filter definiert werden, bestehend aus booleschen und numerischen Operationen, welche clientseitig ausgewertet werden. Diese werden in KLighD-VS Code im Interface `Filter` gekapselt, welches

- einen optionalen Namen des Filters,
- einen optionalen Defaultwert – bestimmend ob der Filter standardmäßig angeschaltet sein soll – sowie
- eine Funktion `filterFun(el: SKGraphElement): boolean`

definiert. Auch hier ist es somit möglich, einen Knoten auf einen booleschen Wert abzubilden. Die Klasse `ProxySemanticFilterHandler` wandelt nun vorhandene `Filter` in `ProxyFilter` um und registriert diese zur Laufzeit in der Sidebar und dem Proxy-View – Name und Defaultwert des Filters werden hierbei ebenfalls berücksichtigt. Abschnitt 3.2 enthält einige für SCCharts erstellte semantische Filter und in Abschnitt 3.3 wird näher auf deren Funktionsweise eingegangen.

Tabelle 3.1 fasst die implementierten Konzepte in einer Übersicht zusammen. Anhand dessen kann der Ablauf des Update-Schritts konstruiert werden: Die Methode `update()` dient als Schnittstelle und aktualisiert das DOM, jegliche Logik des Proxy-Views wird an `createAllProxies()` delegiert. In `createAllProxies()` wiederum wird die Logik in Teilprobleme unterteilt, welche durch entsprechende sukzessive Methodenaufrufe gelöst werden. Abbildung 3.12 enthält einen Abhängigkeitsgraphen dieser Aufrufe sowie die daraus folgende topologische Sortierung [HLC09]. Hierbei zeigt jede Ebene des Abhängigkeitsgraphen voneinander unabhängige Prozesse:

I Zunächst wird das Modell von `getOffAndOnScreenNodes()` in off- und on-screen Knoten unterteilt. Für off-screen Knoten werden potenziell Proxies erstellt, hingegen dienen on-screen Knoten als Argument für Filter – etwa *Filter Unconnected to On-Screen* – und sind für das erstellen von Kantenproxies notwendig. Simultan können durch `connectEdgeSegments()` bereits Segmentproxies erstellt werden, da diese unabhängig von off- und on-screen Knoten sind.

Tabelle 3.1. Übersicht der implementierten Konzepte des Proxy-Views. Die Implementierung von Along-Border-Routing weicht leicht vom Konzept ab – die Verbindungsstelle des Knotens bleibt beim Proxy erhalten und wird somit nicht verschoben. Kaskadierendes Clustering wurde als Debug-Option implementiert.

Konzept	Implementiert
Proxy	✓
Kantenproxy	✓
Straight-Edge-Routing	✓
Along-Border-Routing	✓*
Segmentproxy	✓
Clustering	✓
Kaskadierendes Clustering	(✓)
Opacity + Stacking Order	✓
Strukturelle Filter	✓
Semantische Filter	✓

II Im nächsten Schritt können mittels `applyFilters()` registrierte Filter angewandt werden. `calculateOpacity()` und `orderNodes()` sind etwa für *Opacity + Stacking Order* relevant und können ebenfalls zeitgleich ablaufen. Mittels `getSynthesisProxyRendering()` wird das Default-Rendering des Knotens mit dem von der Synthese definierten Proxy-Rendering überschrieben – falls vorhanden. Proxies werden hier jedoch noch nicht gerendert, lediglich Daten wie Größe, Schriften und interne Positionen des finalen Renderings werden so umdefiniert.

III Vom Synthese-Rendering abhängig werden Größe und Positionen der Proxies mit `map(getTransform())` berechnet – diese können vom Default-Rendering abweichen.

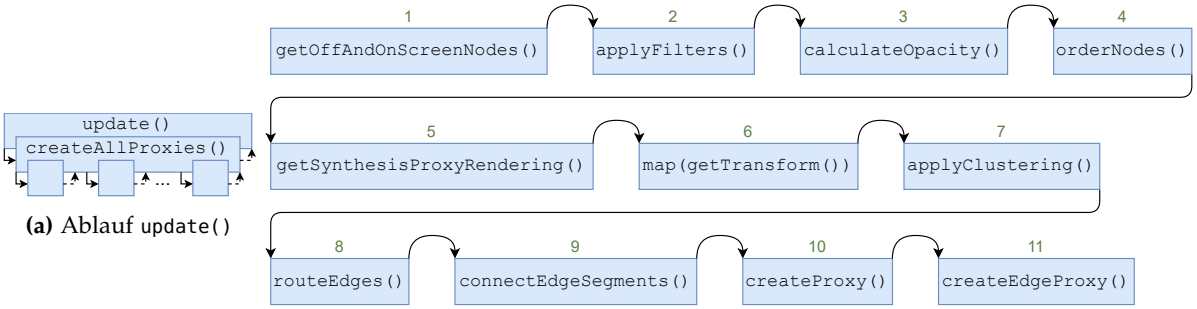
IV Wiederum abhängig von den Positionen der Proxies wird in `applyClustering()` das gewählte Clustering-Verfahren angewendet. Dieses nutzt außerdem die bereits gefilterten off-screen Knoten, da zuvor überlappende Proxies eventuell herausgefiltert wurden.

V Nun ist es möglich, die resultierenden Proxies und Cluster mit `createProxy()` zu rendern. Gleichzeitig können durch `routeEdges()` Kantenproxies geroutet werden. Diese sind abhängig von Clustering, da keine Kantenproxies zu in Clustern enthaltenen Proxies erstellt werden.

VI Schlussendlich werden mit `createEdgeProxy()` alle Kanten- und Segmentproxies gerendert.

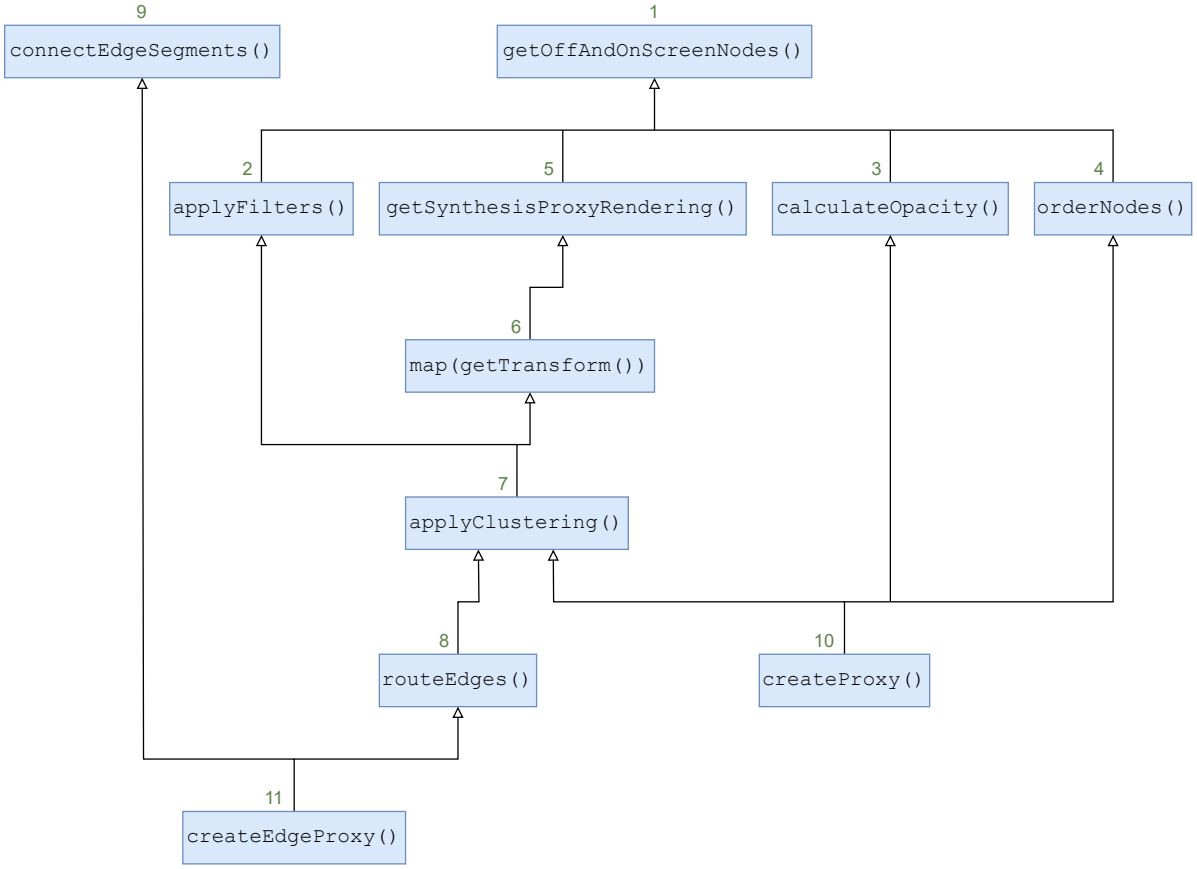
Für die gewählte topologische Sortierung gibt es neben den direkten Abhängigkeiten jedoch noch weitere beachtenswerte Eigenschaften. So ist es sinnvoll, in II zuerst Filter anzuwenden, da so die Anzahl an Proxies verringert und damit redundanter Rechenaufwand vermieden werden kann. Des Weiteren werden für eine bessere Übersichtlichkeit im Code Segmentproxies und Kantenproxies nacheinander erstellt – und so von den übrigen Methoden für Proxies abgegrenzt. Analog findet das Rendern von Proxies sowie Segment- und Kantenproxies zum Schluss statt. Insbesondere wird durch diese Modularisierung die Erweiterbarkeit des Proxy-Views gefördert: Weitere Methoden können problemlos eingefügt werden und lokale Optimierungen des Programms erfordern möglichst wenig zusätzliche Aufmerksamkeit. Insgesamt ergibt sich als Reihenfolge:

1. `getOffAndOnScreenNodes()`
2. `applyFilters()`
3. `calculateOpacity()`
4. `orderNodes()`
5. `getSynthesisProxyRendering()`
6. `map(getTransform())`
7. `applyClustering()`
8. `routeEdges()`
9. `connectEdgeSegments()`
10. `createProxy()`
11. `createEdgeProxy()`



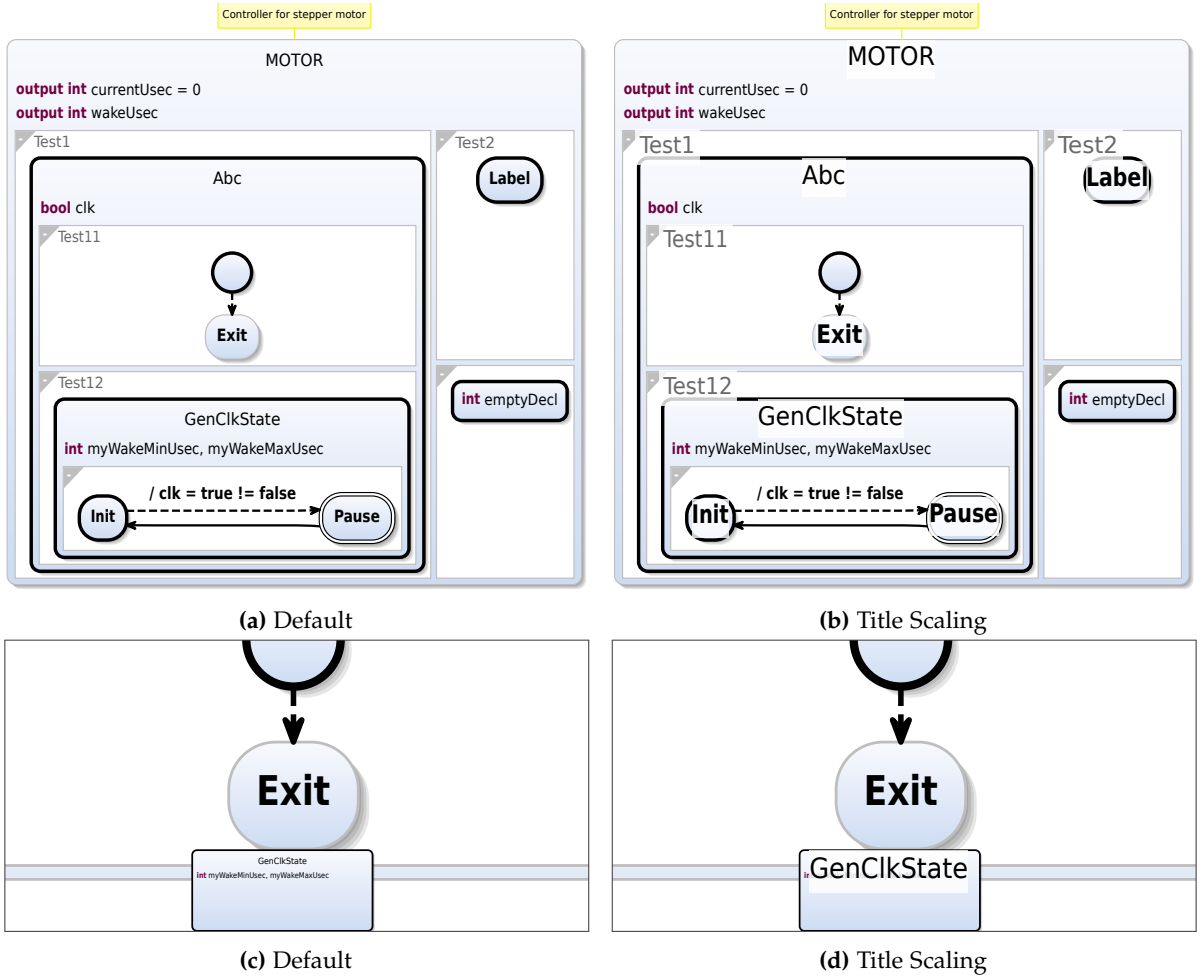
(a) Ablauf `update()`

(b) Abfolge der Aufrufe in `createAllProxies()`



(c) Abhängigkeitsgraph der Aufrufe in `createAllProxies()`

Abbildung 3.12. Ablauf vom Update-Schritt des Proxy-Views (a). Die Schnittstelle `update()` delegiert an `createAllProxies()`, welches wiederum aus einer Abfolge von Methodenaufrufen besteht – jede Methode löst einen Aufgabenteil. In (c) der Abhängigkeitsgraph der Methodenaufrufe, transitive Abhängigkeiten wurden bereits ausgelassen. Jede Ebene entspricht theoretisch parallelisierbarer Prozesse – praktisch würde hierbei jedoch ein erheblicher Aufwand durch das Zusammenführen der Ergebnisse entstehen. Schließlich zeigt (b) die tatsächlich verwendete topologische Sortierung.

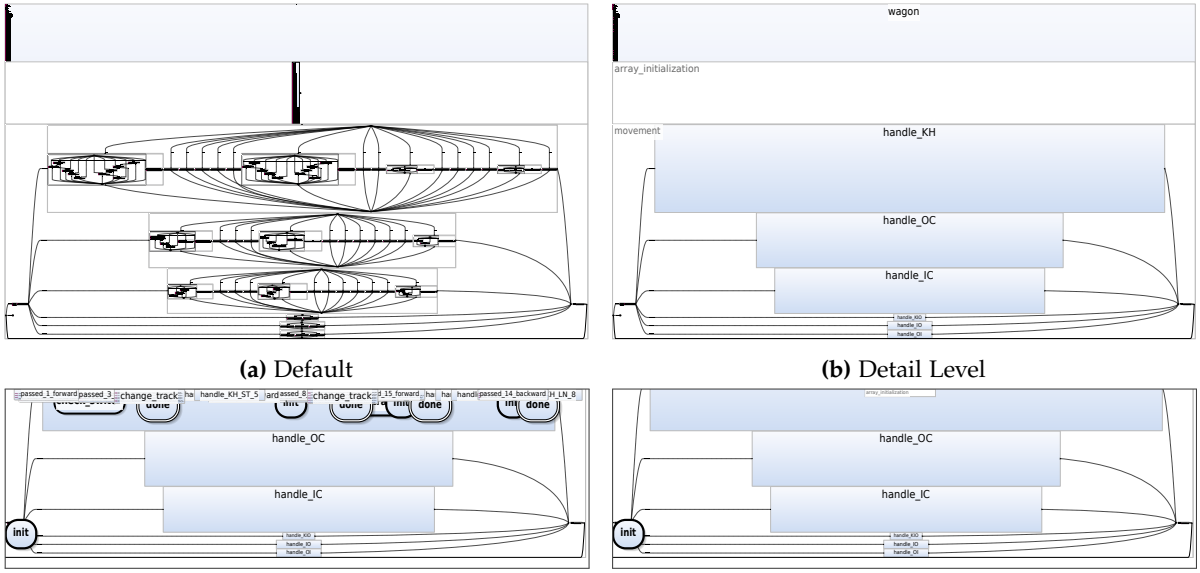


(a) Default **(b)** Title Scaling

(c) Default **(d)** Title Scaling

Abbildung 3.13. Ein Vergleich des Default-Renderings von Proxies mit und ohne Title Scaling. Die Titel von Knoten werden entsprechend der Zoomstufe hochskaliert. Oben Title Scaling in einem ganzen SCChart, unten die Anwendung auf Proxies in einem Ausschnitt des selben SCCharts.

Auch über das Konzept hinaus wurden dem Proxy-View weitere Optionen hinzugefügt. Wie von M. Frisch und R. Dachselt [FD10] evaluiert, steigert Interaktivität mit Proxies die Nutzerfreundlichkeit. Besonders das Springen zu einem off-screen Knoten per Klick auf den entsprechenden Proxy (Hopping) – analog zur Arbeit von Irani et al. [IGY06] – wurde von Nutzern/Nutzerinnen häufig verwendet. Die daraus resultierende Möglichkeit, das KKD Schritt für Schritt zu erkunden, wurde als besonders gemocht betont – ebenso kann mittels hin und zurück springen rasch Kontext erhalten werden. Daher wurde Hopping auch in dieser Arbeit implementiert: Zum entsprechenden off-screen Knoten wird mittels einer kurzen Zoom + Pan Animation gesprungen, welche weiter zur Nutzerfreundlichkeit beiträgt [WN04]. Es ist außerdem möglich, auch zusätzliche Interaktionen unkompliziert hinzuzufügen – etwa per Rechtsklick, Hover oder Drag and Drop, auch für Cluster, Kanten- oder Segmentproxies.



(a) Default **(b)** Detail Level

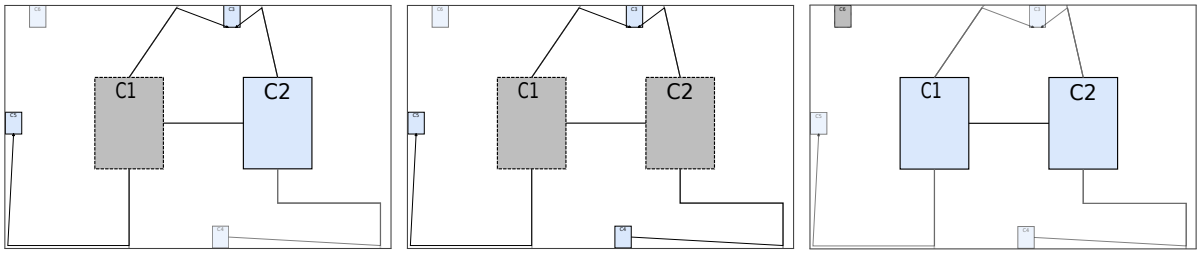
(c) Default **(d)** Berücksichtigung des Detail Level

Abbildung 3.14. Verhalten des Proxy-Views im Bezug auf Smart Zooms Detail Level. Kinder von Knoten werden entsprechend der Zoomstufe nicht angezeigt – zur Erhöhung der Übersichtlichkeit und Effizienz. Oben ein SCChart vor und nach Verwendung des Detail Levels, unten der Proxy-View vor und nach Berücksichtigung des Detail Levels.

In KLighD-VS Code erhöht die Option *Smart Zoom* die Verständlichkeit von KKD auf geringeren Zoomstufen. So können mittels eines Features die Titel von Knoten in Abhängigkeit der Zoomstufe hochskaliert werden, sodass auch bei geringem Zoom die Titel lesbar bleiben. Abbildung 3.13 verdeutlicht dieses *Title Scaling* und zeigt die Anwendung im Proxy-View: Für den Knoten *GenClkState* ist ein Proxy im Default-Rendering vorhanden – dessen Titel ist jedoch kaum erkennbar. Bei angeschaltetem Title Scaling kann auch hier der Titel hochskaliert werden und ist nun lesbar.

Ein weiteres Feature von Smart Zoom stellt das variable *Detail Level* dar: Auch hier wird in Abhängigkeit der Zoomstufe die Übersichtlichkeit aber auch Performanz des Diagramms erhöht. Bei geringem Zoom wird von gewissen Kindknoten abstrahiert – der entsprechende Elternknoten wird kinderlos angezeigt. In Abbildung 3.14 wird das Verhalten des Proxy-Views im Bezug auf Detail Level gezeigt: Per default werden auch für Knoten mit niedrigem Detail Level – deren Kinder somit nicht im KKD angezeigt werden – Proxies für Kindknoten erstellt. Unter Berücksichtigung des Detail Levels hingegen werden diese Knoten behandelt als hätten sie keine Kinder, was zur Konsistenz beiträgt. Im Beispiel enthält der zunächst mit Kindproxies überfüllte obere Rand nur noch einen weiterhin angezeigten Proxy bei Beachtung des Detail Levels.

Es gibt diverse Möglichkeiten, den Inhalt des selektiven Kontexts zu bestimmen. Besonders Filter erlauben eine Definition dessen in unterschiedlichen Granularitätsstufen. Es ist allerdings kein seltener Anwendungsfall, zwischen mehreren Kontexten hin und her zu



(a) (b) (c)

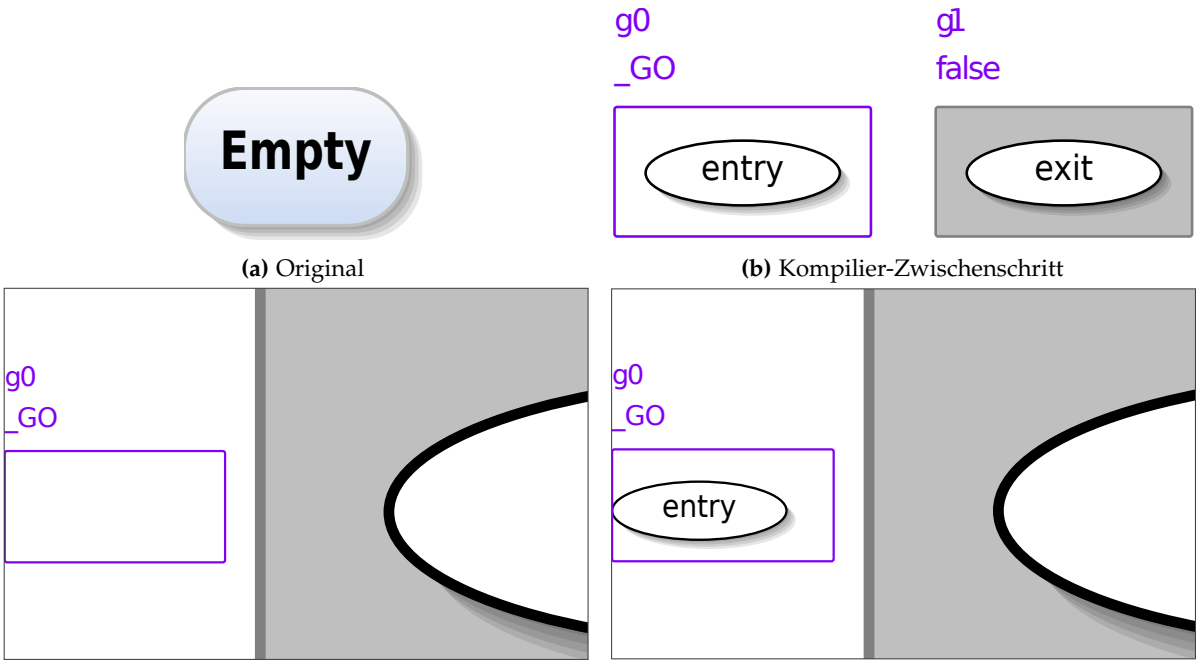
Abbildung 3.15. Ein Beispiel für das Hervorheben von Proxies anhand der Selektion. Nicht mit der Selektion verbundene Proxies werden transparenter dargestellt.

wechseln – etwa wenn in einem Klassendiagramm die Kommunikation zweier Klassen je für sich betrachtet wird. Per An- und Ausschalten von Filtern sind diese Kontextwechsel bereits möglich, jedoch kann so ein erheblicher zusätzlicher Aufwand pro Wechsel entstehen. Um diesen Aufwand zu vermeiden, wurde die Möglichkeit eingebaut, selektierte Knoten und damit zusammenhängende Proxies hervorzuheben. So kann durch Ändern der Selektion leicht der erwünschte selektive Kontext in den Fokus genommen werden. Um dies zu erreichen, wird die Transparenz nicht verbundener Proxies erhöht und selbige werden unterhalb der übrigen angezeigt – vergleichbar zu Opacity + Stacking Order. Auf diese Weise verschwinden Proxies nicht plötzlich, sind aber dennoch nicht mehr Teil des Hauptaugenmerks – ein Hin-und-Her-Wechseln der Selektion ist somit sanft. In Abbildung 3.15 wird ein Beispiel hierfür gezeigt.

In gewissen hierarchischen KKD haben Elternknoten nur geringe Aussagekraft – stattdessen dienen sie primär als Wrapper für enthaltene Kindknoten. Abbildung 3.16 enthält ein Beispiel eines solchen Diagrammtyps, welcher als Zwischenschritt des Kompilierens von SCCharts zu C-Code generiert wird. Gemäß dem Konzept wird standardmäßig nur für den Elternknoten *g0_GO* ein Proxy erstellt – wirklich relevant ist jedoch dessen Kindknoten. In diesem Sinne wurde die API um *Hierarchical Depth* erweitert: Synthesen ist es nun möglich, mittels der Eigenschaft `KlighdProperties.PROXY_VIEW_HIERARCHICAL_OFF_SCREEN_DEPTH` einen Wert festzusetzen, wie tief auch für Kinder von off-screen Elternknoten Proxies erstellt werden sollen. Dem Konzept entsprechend ist der Wert null als Standard festgelegt – nur off-screen Elternknoten werden als Proxies angezeigt. Werte größer null definieren die Tiefe, für die Kindproxies erstellt werden können. Im Beispiel ist diese Tiefe eins, sodass *entry* als Proxy dargestellt wird – beim Wert zwei wären auch für dessen Kinder wiederum Proxies vorhanden. Schließlich ist es auch möglich mit Werten kleiner null festzulegen, dass für alle off-screen Knoten Proxies erstellt werden dürfen – unabhängig vom jeweiligen Elternknoten.

3.2. SCCharts

Der Proxy-View für allgemeine KKD wurde generisch implementiert, um so auch möglichst allgemeingültig für alle Diagrammtypen zu sein. Diese generische Implementierung kann



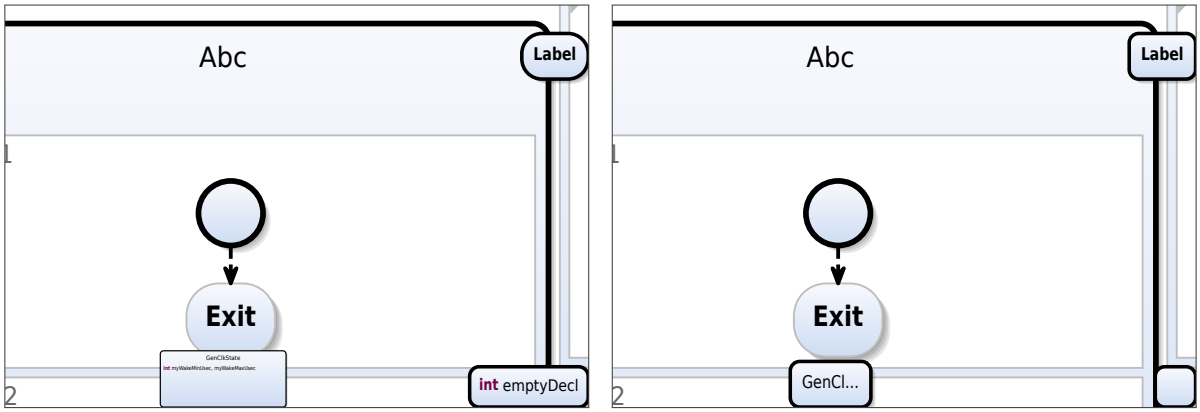
(a) Original **(b)** Kompilier-Zwischenschritt

(c) Default **(d)** Hierarchical Depth

Abbildung 3.16. Motivation zur Erweiterung des Proxy-Views um Hierarchical Depth. In manchen KKD können Elternknoten eine geringe Aussagekraft für den Kontext haben. In (b) ein solches KKD, welches als Zwischenschritt des Kompilierens vom SCChart aus (a) zu C-Code entsteht. Unten der Proxy-View vor und nach Erweiterung um Hierarchical Depth.

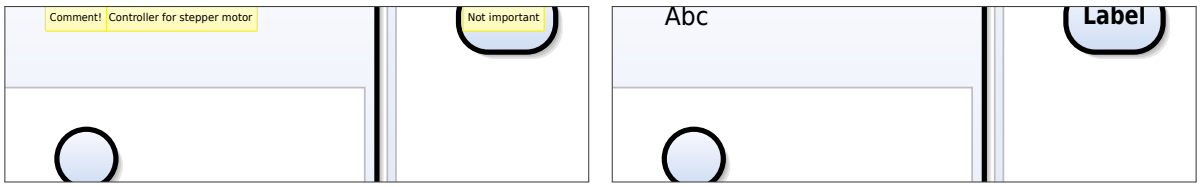
jedoch für spezifische Diagrammtypen konkretisiert werden: Synthesen können je Knoten mehrere Eigenschaften definieren, welche den Proxy-View besser auf das jeweilige Diagramm zuschneiden. Diese Erweiterungen sind somit rein serverseitig und erfordern keine weiteren clientseitige Änderungen. Anhand einer Beispielimplementierung für SCCharts wird diese API erläutert:

So ist es möglich, mittels der Eigenschaft `KlighdProperties.PROXY_VIEW_PROXY_RENDERING` das Default-Rendering zu überschreiben. In Abbildung 3.17 werden Default- und Synthese-Rendering miteinander verglichen: Der Knoten *Label* besteht nur aus einem Titel – er enthält keine Kinder oder Deklarationen. Beide Renderings stellen dessen Proxy größtenteils gleich dar – im Synthese-Rendering hat dieser eine eckigere Erscheinung. Zur Einheitlichkeit werden Proxies im Synthese-Rendering allgemein eher eckig dargestellt. Außerdem werden bis auf den Titel keine weiteren Labels in Proxies angezeigt. Somit ist der im Default-Rendering nur aus Deklarationen bestehende Proxy im Synthese-Rendering nun leer. Der Proxy *GenClkState* verdeutlicht schließlich den Hauptgrund für Synthese-Renderings: Im Default-Rendering ist nicht erkennbar, welchen off-screen Knoten der Proxy darstellt. Der Titel ist unlesbar und im Proxy ist viel leerer, ungenutzter Platz vorhanden – ein typisches Problem von Proxies hierarchischer Knoten. Im Synthese-Rendering hingegen ist der Platz voll ausgenutzt und der



(a) Default-Rendering **(b)** Synthese-Rendering

Abbildung 3.17. Ein Vergleich von Default- und Synthese-Renderings von Proxies in SCCharts. Der Proxy *Label* ändert sich kaum – lediglich die Form wurde eckiger. Für den leeren Proxy wurde im Synthese-Rendering von der Deklaration abstrahiert. *GenClkState* hat im Synthese-Rendering eine kompaktere Erscheinung, sodass der Titel nun lesbar ist.



(a) Default **(b)** In Synthese definiert

Abbildung 3.18. Ein Vergleich von Default und synthese-definierten Eigenschaften für Proxies in SCCharts. Per default wird für jeden Knoten ein Proxy erstellt – einschließlich Kommentaren. In der SCChart-Synthese kann definiert werden, dass für Kommentare keine Proxies erstellt werden sollen.

Titel lesbar. Vom Proxy kann so direkt auf den entsprechenden off-screen Knoten geschlossen werden.

Um dies für alle Proxies sicherzustellen, wird einfache *Label Truncation* angewandt: Lange Titel werden auf ihre ersten fünf Buchstaben gekürzt – entsprechend der durchschnittlichen Länge englischer Wörter [BSS12]. Damit für Nutzer/Nutzerinnen erkenntlich bleibt, dass ein Titel gekürzt wurde, wird dieser am Ende durch ... markiert. Auch weitere Strategien zur Kürzung langer Titel sind denkbar. Analog zur Implementierung von M. Frisch und R. Dachselt [FD10] könnte der Titel nur den Typen des jeweiligen Knotens angeben. Eine Art der Zusammenfassung des Inhalts vom Knoten wäre ebenfalls möglich [MP21].

Außerdem kann in Synthesen definiert werden, für welche Knoten Proxies erstellt werden sollen. So sollten – abweichend vom Default – in SCCharts keine Proxies für Kommentare vorhanden sein. Die Eigenschaft `KlighdProperties.PROXY_VIEW_RENDER_NODE_AS_PROXY` erlaubt also je Knoten festzulegen, ob ein Proxy erstellt werden soll. In Abbildung 3.18 ein Vergleich eines SCCharts mit und ohne Kommentar-Proxies.

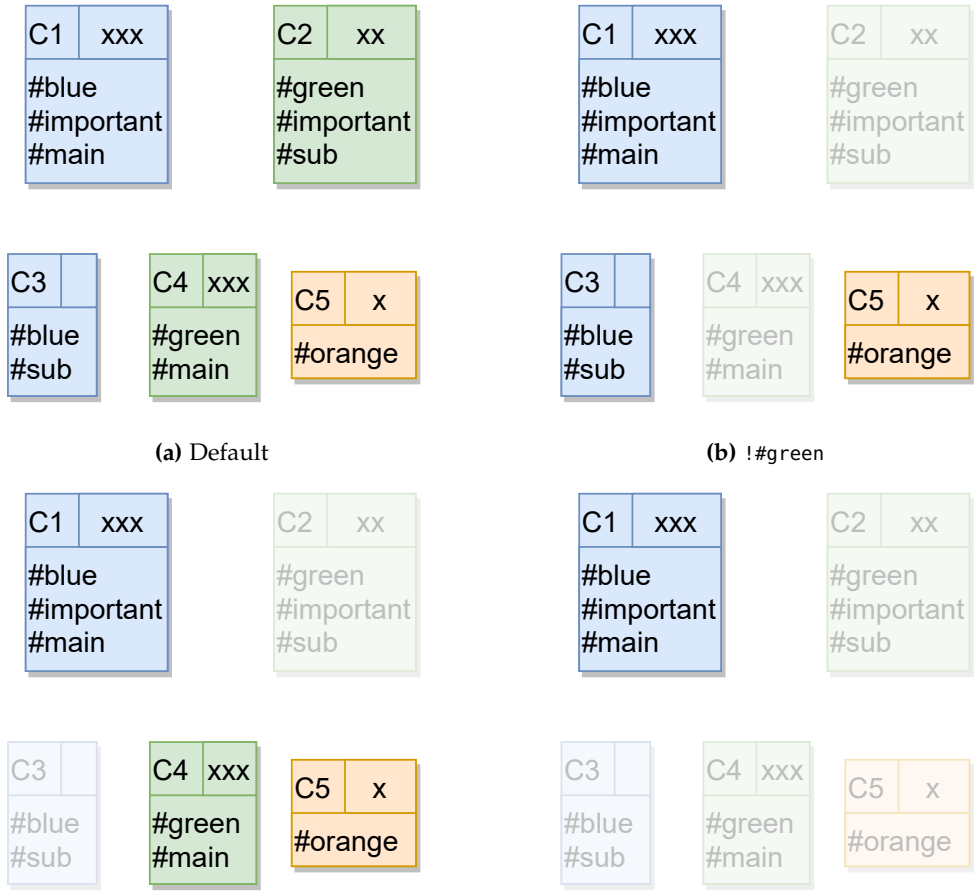
Abbildung 3.19. Eine Übersicht der implementierten semantischen Filter für SCCharts.

Darüber hinaus wurde der Proxy-View anhand dieser Eigenschaften auch für SCGraphs, welche als Zwischenschritt der Kompilation von SCCharts zu C-Code entstehen, genauer definiert.

Auch für die, zusammen mit der Arbeitsgruppe, entwickelte API für semantische Filter wurden einige SCChart-Beispielfilter implementiert – welche ebenso im Proxy-View angewendet werden können. Abbildung 3.19 zeigt ebendiese Filter, welche vom Proxy-View zur Laufzeit in die Sidebar eingebaut wurden: Es ist möglich nach Typen von Knoten zu filtern – etwa können bei Belieben nur Regionen als Proxies angezeigt werden. Auch weiter definierende Eigenschaften wie, ob ein Knoten initial ist, sind filterbar. Ferner können diese Kriterien auf beliebige Weise miteinander logisch verknüpft werden – so wird ermöglicht nur Knoten anzuzeigen, die entweder initial oder final sind. Schließlich können auch in Zahlen ausdrückbare Eigenschaften – so etwa die Anzahl an Deklarationen – geprüft und mit numerischen Operationen miteinander verknüpft werden. Auf diese Weise kann eine semantische Größenmetrik für Knoten definiert werden, wie bereits von M. Frisch und R. Dachselt [FD10] als praktisches Filterkriterium für Proxies vorgeschlagen. Zwar entsprechen nicht alle implementierten Filter tatsächlich für SCCharts sinnvollen Filterkriterien, allerdings erforschen diese die Grenzen und zeigen die Möglichkeiten der API auf. Im nachfolgenden Abschnitt wird auf die genaue Funktionsweise semantischer Filter näher eingegangen.

3.3. Abseits des Proxy-Views

Auch außerhalb des Proxy-Views wurden mehrere APIs erweitert: Darunter etwa die in Abbildung 3.2 gezeigte Sidebar. Damit diese effektiv für diverse Einstellungen des Proxy-



(a) Default

(b) `!#green`

(c) `#main || #orange`

(d) `$x > 2 && #important`

Abbildung 3.20. Einige Beispiele semantischer Filter in der dafür entwickelten Sprache. Knoten bestehen aus einem Titel links oben, daneben einer Anzahl an Kreuzen sowie darunter den gesetzten Tags. Tags sind zur Übersicht mit einem #-Präfix versehen.

Views verwendet werden kann, wurde neben einiger Optimierungen etwa die Möglichkeit hinzugefügt, per default unsichtbare Debug-Optionen einzufügen.

Vom Proxy-View ab ging die meiste Arbeit jedoch in die mit der Arbeitsgruppe entwickelte API für semantische Filter: Diese erlaubt Synthesen semantische Informationen an beliebige Elemente des Diagramms anzufügen – darunter etwa Knoten, Kanten oder Ports. Hierfür werden *Tags* verwendet – ein Tag ist ein String. Semantische Filter entsprechen nun *Regeln*, welche aus Tags und logischen Verknüpfungen bestehen – die Auswertung dieser Regeln erfolgt clientseitig. Eine einfache Regel kann bereits durch einen Tag selbst definiert werden: Alle Elemente, auf denen der Tag gesetzt ist, erfüllen die Regel – die übrigen Elemente werden herausgefiltert. Im Beispiel aus Abbildung 3.20 würde die Regel `#blue` alle nicht blauen Knoten entfernen – dies entspricht exakt dem Filter aus Abbildung 3.11b. Logische Operationen

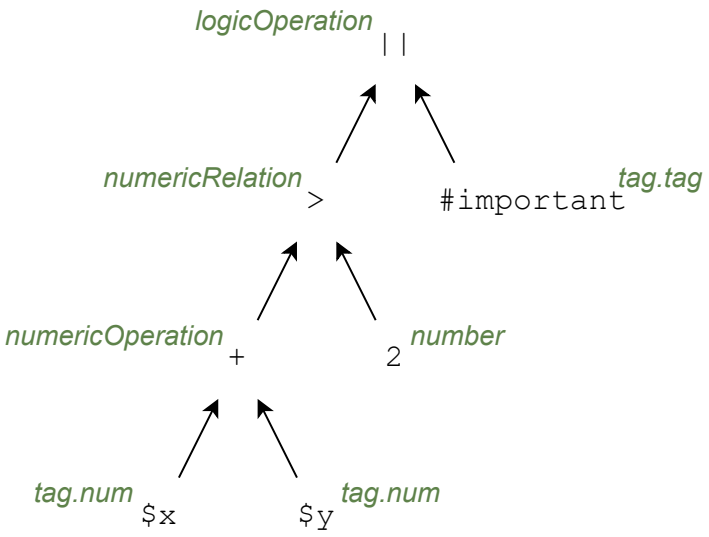


Abbildung 3.21. Einzelne Bestandteile eines semantischen Filters in Baumstruktur. Der Filter `$x + $y > 2 || #important` wurde in der dafür entwickelten Sprache geschrieben. Grüne Anmerkungen referenzieren die jeweils entsprechende Bezeichnung in kontextfreier Grammatik.

wie `!` (Negation), `&&` (Und) etc. können ebenfalls verwendet werden, um komplexere Regeln darzustellen.

Darüber hinaus können Regeln auch numerische Relationen und Operationen beinhalten. Tags werden hierfür um einen Zahlenwert erweitert, welcher standardmäßig stets auf null gesetzt ist. Auf diese Weise können auch wie in Abbildung 3.20d gezeigte Regeln definiert werden: Hier werden alle Knoten, welche höchstens zwei Kreuze besitzen oder nicht `#important` sind, herausgefiltert. Numerische Relationen stellen somit etwa > oder = dar. Darüber hinaus können diese Zahlen mittels numerischer Operationen wie + oder $*$ miteinander verknüpft werden. Für eine nutzerfreundliche Definition semantischer Filter wurde hierfür eine Sprache entwickelt.

Regeln müssen immer einen booleschen Wert ergeben – das Interface `NumericResult` garantiert diese Typsicherheit. Als kontextfreie Grammatik für semantische Filter ergibt sich so:

$$
\begin{aligned}
\mathit{numericOperation} &:= \mathit{number} \mid \mathit{binaryNumericOperation} \\
\mathit{numericResult} &:= \mathit{tag.num} \mid \mathit{numericOperation} \\
\mathit{logicOperation} &:= \mathit{nullaryLogicOperation} \mid \mathit{unaryLogicOperation} \mid \\
&\quad\ \mathit{binaryLogicOperation} \mid \mathit{ternaryLogicOperation} \\
\mathit{numericRelation} &:= \mathit{binaryNumericRelation} \\
\mathit{rule} &:= \mathit{tag.tag} \mid \mathit{logicOperation} \mid \mathit{numericRelation}
\end{aligned}
$$

Hierbei gibt *binaryNumericOperation* die Menge aller numerischen Operationen an, welche zwei Argumente benötigen – übrige Operationen und Relationen sind analog definiert.

In Abbildung 3.21 werden die einzelnen Bestandteile einer Regel mit Referenz auf den jeweiligen Ausdruck der kontextfreien Grammatik gezeigt. Weitere Beispielregeln für SCCharts werden in Abbildung 3.19 gezeigt.

Kapitel 4

Evaluation

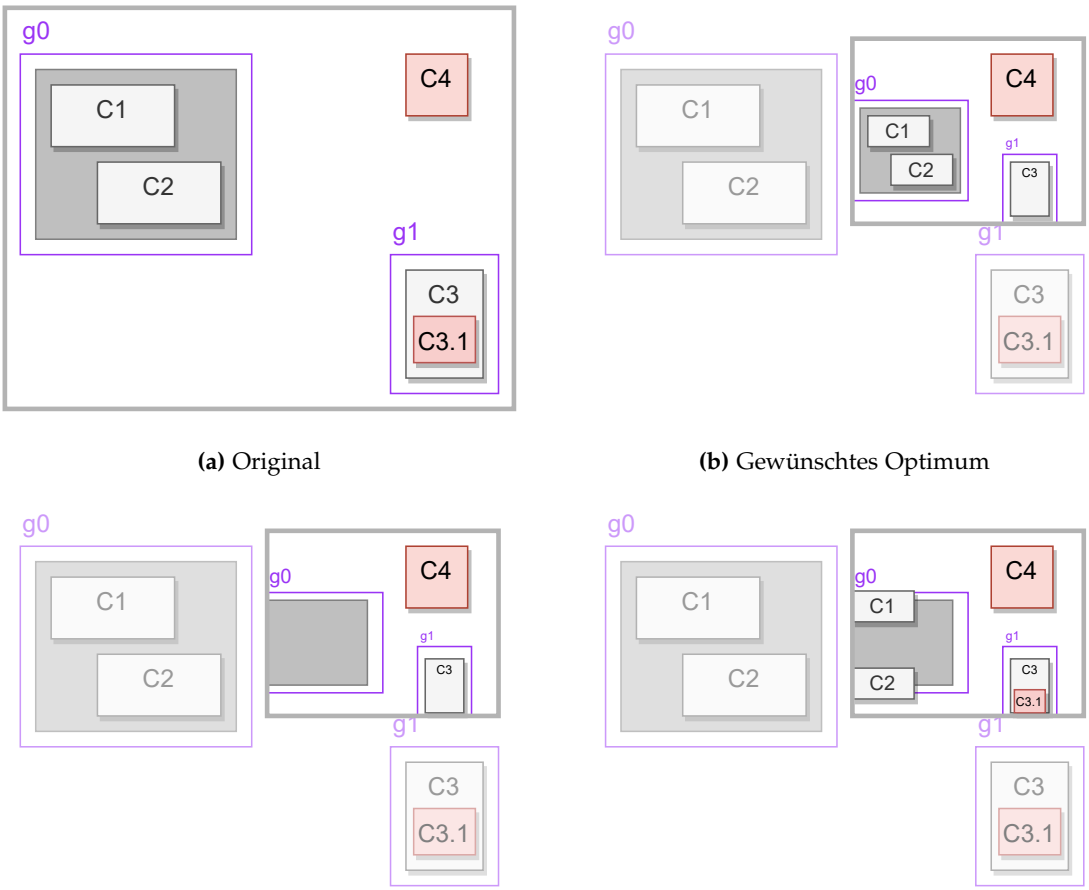
Im Laufe dieser Arbeit wurde regelmäßig arbeitsgruppeninternes Feedback eingeholt. Anhand dessen sowie weiterer Auffälligkeiten behandelt Abschnitt 4.1 zunächst die Evaluation des Konzepts. Zusätzlich dazu wird die Implementierung in Abschnitt 4.2 auch anhand diverser Qualitätsmerkmale ausgewertet.

4.1. Evaluation Konzept

Viele potenzielle Probleme des Proxy-Views wurden im Konzept bereits bedacht. Allerdings gibt es auch weitere Anwendungsfälle, die konzeptuell nicht ausreichend abgedeckt sind: Darunter etwa die in der Implementierung ergänzte Hierarchical Depth. Anhand dieser wird versucht zwischen Eltern- und Kindknoten bei der Erstellung von Proxies weniger streng zu unterscheiden, indem sowohl Eltern- als auch Kindproxies bis zu einer fest gesetzten Tiefe erstellt werden. Das in Abbildung 3.16 gezeigte KKD dient hierbei als Motivation. Jedoch ist auch in diesem Fall Hierarchical Depth noch keine optimale Lösung. Da die Tiefe global gesetzt wird, dient Hierarchical Depth nur als statische Lösung – wie in Abbildung 4.1 gezeigt, können jedoch auch dynamische Tiefen notwendig sein. Hier wird der angestrebte Optimalfall mit den tatsächlichen Möglichkeiten verglichen. Die statischen Möglichkeiten stellen einen Kompromiss aus entweder zu wenigen oder zu vielen gezeigten Kindproxies dar. Eine dynamische Lösung – etwa mittels gesetzter Tiefe je Knoten – würde dieses Problem beheben. Auch die Positionen der Kindproxies weichen vom Optimalfall ab – alle Proxies sind einheitlich am Rand platziert. Diese Grenze der Hierarchical Depth kann ohne Weiteres jedoch

Tabelle 4.1. Vergleich der Lösungen für simultane Darstellungen von Kind- und zugehörigen Elternproxies. Erklärung der Spalten von links nach rechts: Lösungsvorschlag, simultane Darstellung von Kind- und Elternproxies, simultane Darstellung von Kind- und Elternproxies mit variabler Tiefe, korrekte Positionen der Kinder im Elternproxy, Komplexität der Lösung.

Lösung	Simultan	Variabel	Positionen	Komplexität
Statische Hierarchical Depth	✓	✗	✗	*
Dynamische Hierarchical Depth	✓	✓	✗	**
Darstellung von Kindern in Proxies	✓	✓	✓	***
Abstraktion von Elternknoten	✓	✓	✓	*



(a) Original

(b) Gewünschtes Optimum

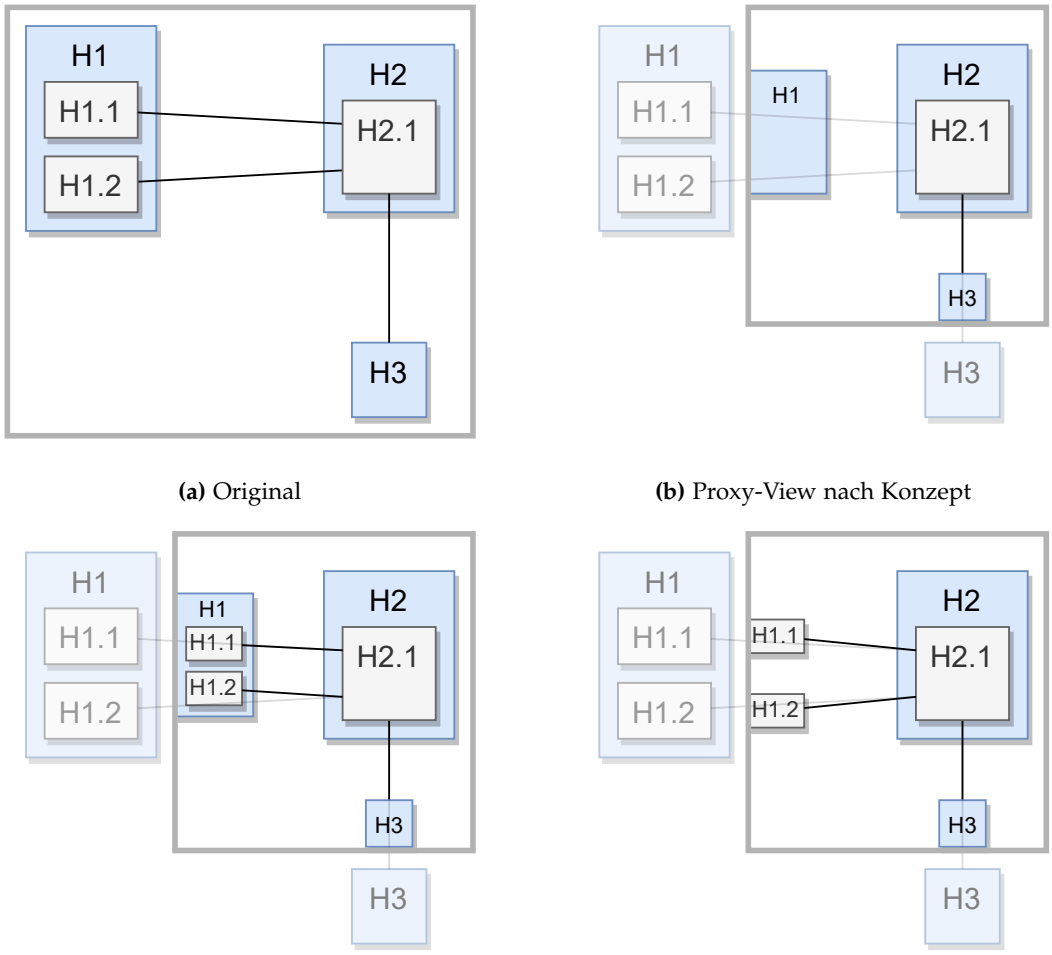
(c) Hierarchical Depth = 1

(d) Hierarchical Depth = 2

Abbildung 4.1. Die Grenzen von Hierarchical Depth. Durch die KKD-weite Definition von Hierarchical Depth kann das gewünschte Optimum (b) nicht erreicht werden. Unten die Ergebnisse fest gesetzter Tiefen. Auch die Positionierungen von Kindproxies weichen von Optimum ab.

nicht überwunden werden. Als Alternative kann die Darstellung von Kindern in Proxies auch dieses Problem überwinden. Hierbei wird für jeden Proxy definiert, welche Kinder angezeigt werden sollen – über die serverseitige Schnittstelle `KlighdProperties.PROXY_VIEW_PROXY_RENDERING` kann dies bereits erreicht werden. Jedoch kann so zusätzlicher Aufwand bei Definition der Eigenschaft und Rendering entstehen – auch können Proxies schnell unübersichtlich werden. Schließlich ist es auch möglich, völlig von gewissen Elternproxies zu abstrahieren und an deren Stelle die jeweiligen Kinder anzuzeigen – vergleichbar zu Hierarchical Depth. Tabelle 4.1 vergleicht die genannten Lösungen.

Hierarchical Depth zeigt außerdem, dass Kindproxies nicht immer streng im Elternknoten gekappt sein müssen. Auch davon ab gibt es Fälle, in denen Kindproxies auch über den Elternknoten hinaus angezeigt werden sollten: So wird in Abbildung 4.2 etwa ein KKD



(a) Original

(b) Proxy-View nach Konzept

(c) Gewünschtes Optimum 1

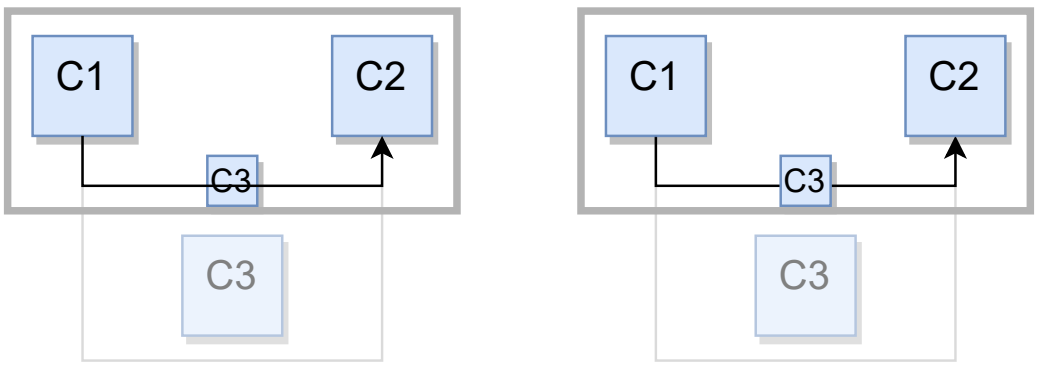
(d) Gewünschtes Optimum 2

Abbildung 4.2. Die Grenzen des Proxy-Views für Kanten zwischen Knoten unterschiedlicher Hierarchieebenen. Der in (b) gezeigte Proxy-View entspricht der Definition des Konzepts – es können keine Kantenproxies zu den Kindern eines Elternproxys erstellt werden. Unten zwei Möglichkeiten für das gewünschte Optimum: Kantenproxies können auch über Hierarchieebenen hinweg stets gezeichnet werden.

gezeigt, in dem Kanten über Hierarchieebenen hinweg verlaufen – wie auch bei Verteilungsdiagrammen der Fall. Dem Konzept entsprechend werden keine Kantenproxies zu Kindern eines Elternproxys erstellt. Optimalerweise sollten jedoch auch in diesem Fall Kantenproxies dargestellt werden können – Abbildung 4.2d zeigt ein Beispiel hierfür, in dem Kindproxies nicht im Elternknoten gekappt sind. Diagramme, die eine solche Art von Kanten erlauben, wurden in dieser Arbeit nicht betrachtet.

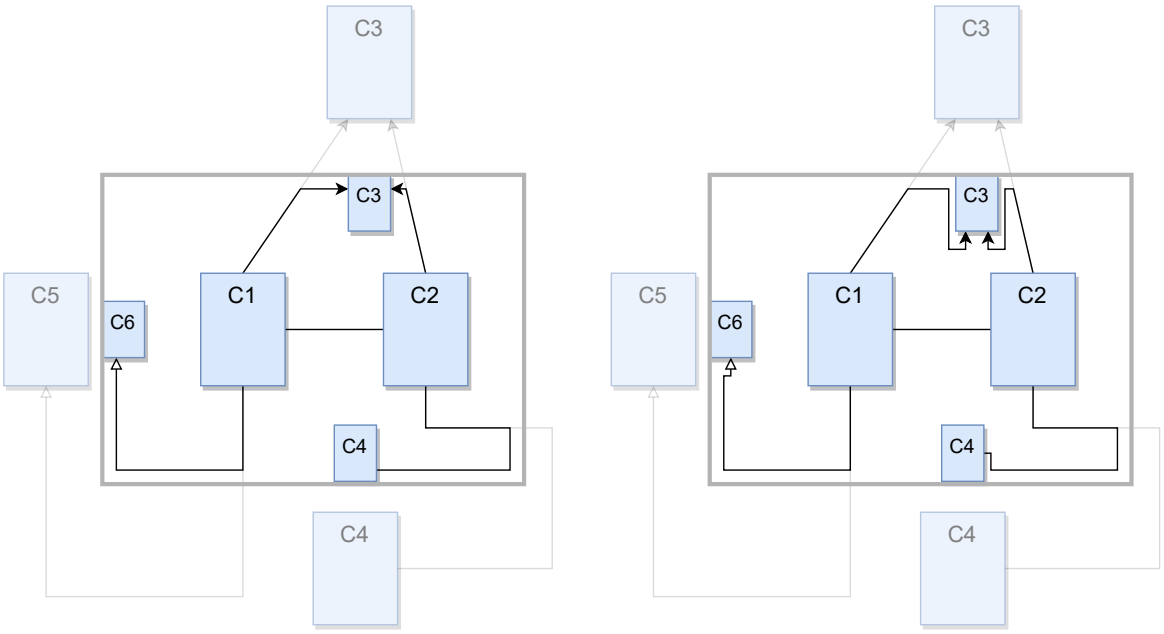
Ebenfalls nicht betrachtet wurde die Überlappung von Kanten- und Segmentproxies. Besonders per Along-Border-Routing können Kanten schnell überlappen – die Erweiterung des Konzepts um Lösungsansätze wie Edge-Bundling [New89; ZXY+13] oder Minimize-Edge-Crossings [EMW86; EW94] stellt ein Thema zukünftiger Arbeiten dar. Jedoch können Kantenproxies nicht nur untereinander überlappen, sondern auch durch Proxies hindurch gehen. Je nach Einstellung werden diese Kanten über oder unter Proxies angezeigt – wie in Abbildung 4.3 angedeutet. Werden Kanten über Proxies angezeigt, werden letztere verdeckt und können so nicht mehr einfach identifiziert werden, was den Nutzen des Proxy-Views immens einschränkt. Werden Kanten unter Proxies angezeigt, kann der Pfad einer jeweiligen Kante nicht immer nachvollziehbar sein. Für dieses Problem sind noch keine Lösungsstrategien vorhanden – da weniger kritisch werden Kanten somit unterhalb von Proxies dargestellt.

Darüber hinaus wurde im Konzept vernachlässigt, dass Verbindungsstellen zwischen Kanten und Knoten eine semantische Bedeutung haben können. Konzeptuell wird diese Verbindungsstelle bei Along-Border-Routing passend verschoben – in der Implementierung bleibt die Stelle jedoch erhalten, wie in Abbildung 3.6b gezeigt. Anders als in der Implementierung kann der Weg vom Bildschirmrand zur Verbindungsstelle auch ästhetischer gestaltet werden. Abbildung 4.4 zeigt ein Beispiel hierfür. Nachdem der Weg zum Proxy entlang des



(a) Kante über Proxy **(b)** Kante unter Proxy

Abbildung 4.3. Ein Vergleich der Überlappungsmöglichkeiten von Kanten und Proxies. Kanten können über und unter Proxies dargestellt werden. Je nach Einstellung können Informationen der Kante oder des Proxys verloren gehen.



(a) Verbindungsstelle nicht beachtet

(b) Verbindungsstelle beachtet

Abbildung 4.4. Along-Border-Routing vor und nach Beachtung der Verbindungsstelle zwischen Kante und Knoten. In (b) wurde der Weg zur Verbindungsstelle entlang des Proxys geroutet.

Bildschirmrands geroutet wurde, wird der Weg zur Verbindungsstelle nun entlang des Proxys geroutet.

Schließlich kann kaskadierendes Clustering bei einer hohen Anzahl an Proxies zu willkürlich erscheinenden Positionen der Cluster führen. Abbildung 3.10c zeigt ein solches Beispiel in der Implementierung: Proxies füllen den kompletten Bildschirmrand und überlappen dabei jeweils miteinander. Kaskadierendes Clustering erstellt ein Cluster für alle Proxies – für dieses existiert keine intuitiv gute Position. Das Cluster wurde nun am linken Bildschirmrand platziert und gibt keinen Aufschluss darüber, dass auch gegenüberliegende Proxies darin enthalten sind.

4.2. Evaluation Implementierung

Das Konzept wurde in der Implementierung bereits an mehreren Stellen ergänzt. Allgemein entsteht durch die Erweiterung von KLighD-VS Code um den Proxy-View nur ein geringer zusätzlicher Aufwand. Tabelle 4.2 enthält eine Übersicht der Performanz des KKD-Views und des Proxy-Views für mehrere KKD pro Update-Schritt. In Abbildung 4.5 werden die aus den Daten resultierenden Plots gezeigt. Die Zeiten wurden auf einem Nutzerprofil der Terminal-

Tabelle 4.2. Übersicht der Performanz des KKD-Views und des Proxy-Views pro Update-Schritt in diversen Diagrammen. Alle Zeiten sind in Millisekunden angegeben – zur Berechnung der statistischen Messgrößen wurde aufgrund des zentralen Grenzwertsatzes [LM17] von einer Normalverteilung dieser ausgegangen. In der Messung wurden für alle off-screen Knoten Proxies erstellt, keine Filter oder Clustering-Verfahren angewandt und keine Kanten- oder Segmentproxies erstellt. Inhalt waren die Erstellung des jeweiligen KKD sowie eine anschließende Navigation dessen. Caches speichern je Knoten die Distanz zum Bildschirm, die absolute Position sowie das Rendering zwischen. Erklärung der Spalten von links nach rechts: das betrachtete Modell, der View, ob Caches verwendet wurden, arithmetischer Mittelwert μ der gemessenen Zeiten sowie Standardabweichung σ, Minimum und Maximum.

Modell	View	Cache	μ	σ	min	max
`motor.sctx`	KKD-View		4,84	4,17	2,3	47,1
	Proxy-View	✗	4,79	2,4	0,1	19,8
		✓	4,22	2,07	0,1	18,2
`wagon.sctx` mit Smart Zoom	KKD-View		8,59	17,16	2,7	145,8
	Proxy-View	✗	3,97	9,21	0,1	109,8
		✓	2,47	9,76	0,1	111,5
`wagon.sctx` ohne Smart Zoom	KKD-View		45,33	17,97	29,7	147
	Proxy-View	✗	17,94	11,85	1,8	60,4
		✓	10,27	12,1	1,3	109,7

Server der Christian-Albrechts-Universität zu Kiel[1] gemessen. Die Erstellung des Diagramms sowie dessen anschließende Navigation mittels eines beliebigen Bewegungsablaufs stellten hierbei die Bestandteile der Messung dar. Die Messung fand ohne Filter, Clustering-Verfahren, Kanten- oder Segmentproxies statt – somit wurde als Kern des Proxy-Views die Erstellung von Proxies aller off-screen Knoten zeitlich geprüft. Im Durchschnitt benötigte der Proxy-View stets weniger Rechenzeit als der KKD-View. Auch Caches werden im Proxy-View verwendet: Nach der ersten Berechnung werden für jeden Knoten die Distanz zum Bildschirm, dessen absolute Position sowie das Rendering zwischengespeichert und wiederverwendet. In allen Fällen kann bei Verwendung von Caches eine geringere durchschnittliche Rechenzeit beobachtet werden – im Folgenden werden somit stets die Werte mit Caches verglichen. Besonders auffällig ist, dass der Aufwand des Proxy-Views in großen Diagrammen relativ zum Aufwand des KKD-Views geringer ist: Im größten geprüften Diagramm `wagon.sctx` ohne Smart Zoom, erzeugt der Proxy-View einen zusätzlichen Aufwand von etwa 22,7% – im kleinsten geprüften Diagramm `motor.sctx` hingegen beträgt der zusätzliche Aufwand circa 87,1%. Außerdem spiegelt das Minimum der Werte jeweils den Best Case wieder – keine Knoten sind off-screen und die Rechenzeit bleibt somit minimal. Im Gegensatz dazu entspricht das Maximum nicht dem Worst Case, da dieses stets initial bei Öffnung des Diagramms erreicht wurde. Eine

[1] `https://www.inf.uni-kiel.de/de/service/technik-service/dienste/terminalserver`

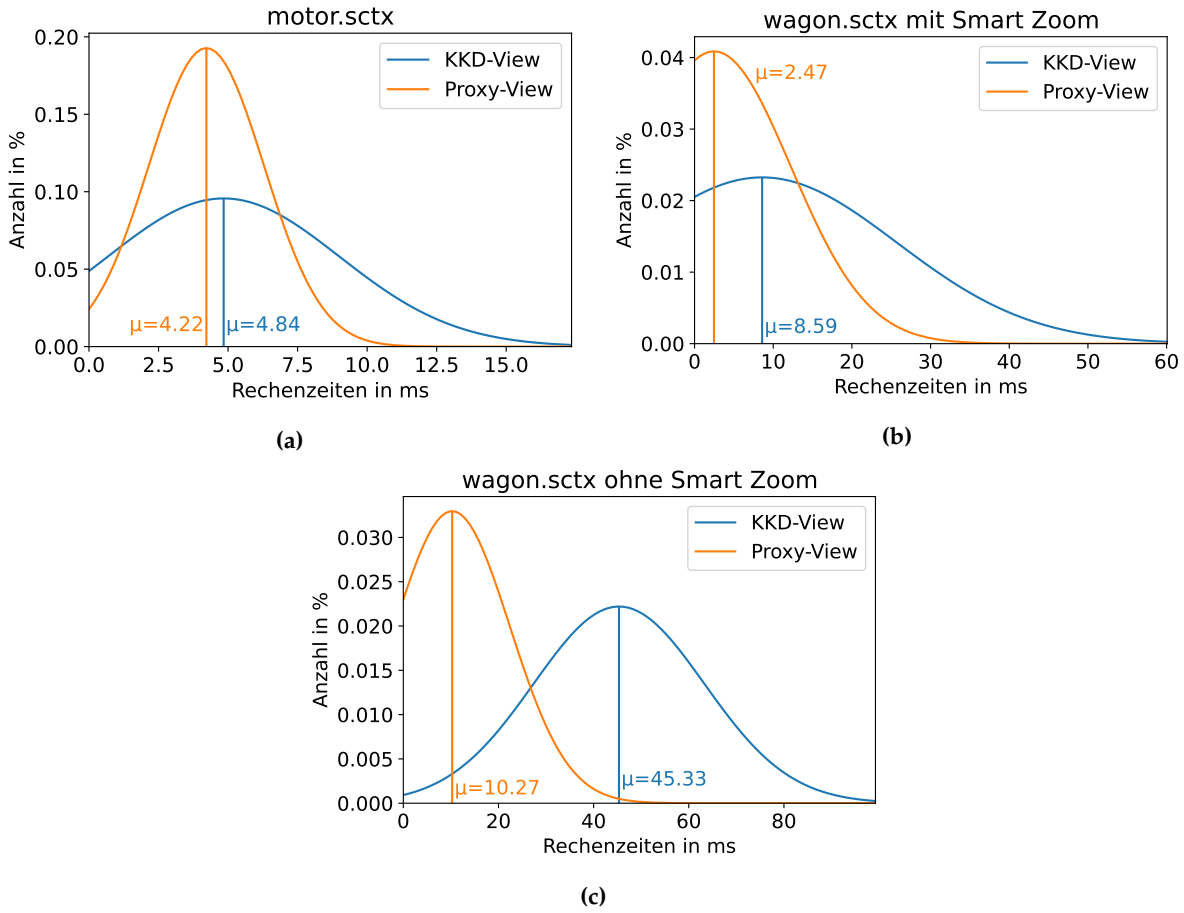


Abbildung 4.5. Plots der Normalverteilungen aus Tabelle 4.2. Für den Proxy-View wurden jeweils die Werte mit Cache verwendet.

Bildwiederholungsrate von mindestens 30 Frames per Second (FPS) ermöglicht eine flüssige Navigation des Diagramms. Um diese sicherzustellen, darf pro Update-Schritt höchstens eine Rechenzeit von 33,33 Millisekunden erreicht werden. Lediglich in `wagon.sctx` ohne Smart Zoom dauerte die durchschnittliche Rechenzeit länger – wobei der Proxy-View jedoch nur einen zusätzlichen Aufwand von etwa 10,27 Millisekunden verursachte. Insgesamt wurden durch den Proxy-View so keine Einbrüche der FPS oder anderweitige Einschränkungen der Nutzererfahrung festgestellt.

Des Weiteren wurden auch Zeiten von Clustering sowie kaskadierendem Clustering isoliert gemessen. Insbesondere wurde untersucht, ob ein Sweep-Line-Ansatz [SH76; Sou05] die Effizienz steigern könnte. Der für Clustering verwendete Algorithmus wird in Algorithmus 1 dargestellt. Dieser ruft, falls aktiviert, den Sweep-Line-Ansatz aus Algorithmus 2 auf – ansonsten wird die einfache Variante aus Algorithmus 3 aufgerufen. Tabelle 4.3 zeigt schließlich einen Vergleich der Clustering-Verfahren. Gemessen wurden die Zeiten in einer JavaScript-

Tabelle 4.3. Übersicht der Performanz von Clustering und kaskadierendem Clustering in diversen Simulationen. Alle Zeiten sind in Millisekunden angegeben. In der Messung wurden Proxies mit zufälliger Höhe und Breite zwischen 1 und 10 an zufälligen Randkoordinaten der gegebenen Verteilung generiert und anschließend geclustert. Gemessen wurde die Dauer des Clustering-Verfahrens sowie die Anzahl der resultierenden Proxies und Cluster – zur statistischen Signifikanz wurde dies je 100 mal ausgeführt, sodass eine Normalverteilung der Rechenzeiten angenähert wurde. Erklärung der Spalten von links nach rechts: Anzahl der generierten Proxies vor Anwendung des Clustering-Verfahrens, Verteilung der Proxies auf x- und y-Achsen sowie die Größe des Bildschirms, das verwendete Clustering-Verfahren, arithmetischer Mittelwert der Anzahl an Proxies und Clustern nach Anwendung des Clustering-Verfahrens, ob ein Sweep-Line-Ansatz angewandt wurde, arithmetischer Mittelwert μ der gemessenen Zeiten sowie Standardabweichung σ, Minimum und Maximum.

# in	Verteilung	Verfahren	# out	SL	μ	σ	min	max
100	[0, 100]	Clustering	33,7	✗	2,05	3,09	1	31,72
				✓	14,36	2,71	9,2	29,16
		Kaskad. Clustering	21,9	✗	2,07	1,42	1,35	14,66
				✓	135,48	21,66	93,03	203,23
	[0, 1000]	Clustering	86,4	✗	0,69	0,17	0,4	1,51
				✓	9,88	3,06	4,81	29,74
		Kaskad. Clustering	85,9	✗	0,75	0,27	0,39	2,62
				✓	21,16	7,61	6,54	53,18
1000	[0, 100]	Clustering	44,7	✗	151,69	18,35	112,89	189,59
				✓	1224,64	155,75	892,04	1682
		Kaskad. Clustering	4,2	✗	75,6	5,64	69,13	119,57
				✓	75089,44	5533,78	60594,42	93136,33
	[0, 1000]	Clustering	353,6	✗	559,38	28,14	431,98	636,41
				✓	25863,94	3049,9	18970,03	39672,68
		Kaskad. Clustering	228,3	✗	304,43	15,77	272,56	349,06
				✓	229245,86	60279,74	82387,4	494624,85

Erweiterung für Google Colab[2] mit aktivierter GPU-Beschleunigung. Jeweils 100 mal wurden Proxies an zufälligen Randkoordinaten generiert und anschließend geclustert. Für, relativ zur betrachteten Bildschirmgröße, viele Proxies bestätigen die Daten die in Abbildung 3.10 gezeigte Situation: Kaskadierendes Clustering resultiert konsistent in einer geringen Anzahl an Proxies und Clustern. Das Verhältnis von ausgehenden zu eingehenden Proxies betrug in den Beobachtungen bis zu 0,42% – von anfänglich 1000 Proxies blieben im Schnitt nur etwa 4 vorhanden. Ebenfalls auffällig ist, dass durch den Sweep-Line-Ansatz in keinem der betrachteten Fälle eine Steigerung der Performanz erreicht wurde. Tatsächlich resultierte der Sweep-Line-Ansatz in allen Simulationen in einer höheren Rechenzeit – verglichen mit dem einfachen Algorithmus. Dieser zusätzliche Aufwand entsteht durch das Sortieren aller Proxies je Iteration. So war etwa bei 100 eingehenden, zufällig zwischen 0 und 1000 verteilten Proxies ein 28 mal höherer Aufwand von kaskadierendem Clustering messbar. Mittels intelligentem Einfügen der Cluster ist es jedoch möglich, nur initial alle Proxies zu sortieren. In zukünftigen Arbeiten kann diese Optimierung die Effizienz des Sweep-Line-Ansatzes erhöhen. Allgemein ist kaskadierendes Clustering stärker vom zusätzlichen Rechenaufwand des momentanen Sweep-Line-Ansatzes betroffen als Clustering. In den jeweils extremsten Fällen dauerte Clustering 46 mal länger, kaskadierendes Clustering hingegen benötigte eine bis zu 993 mal längere

[2] http://colab.to/js

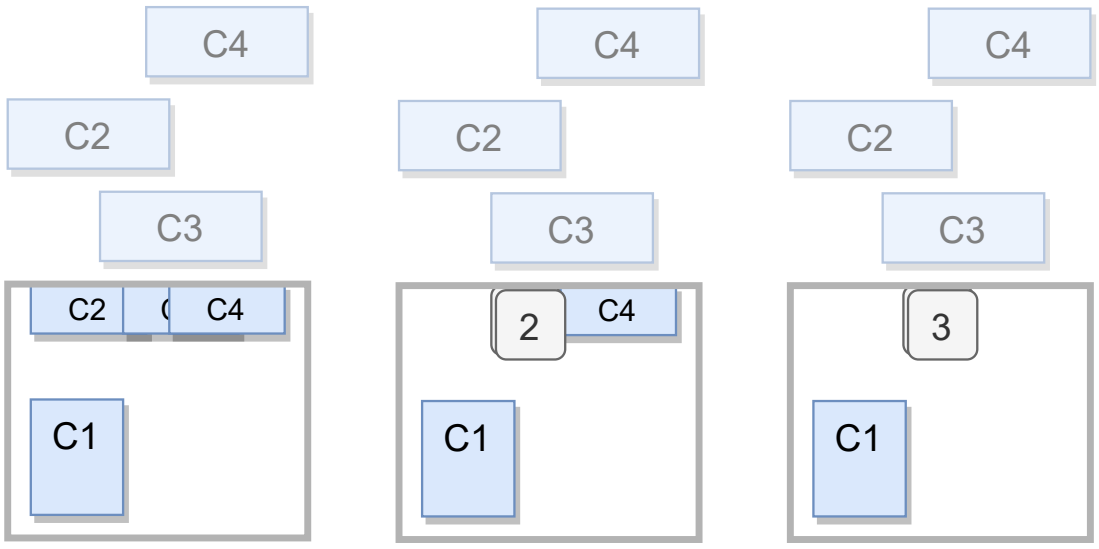


Abbildung 4.6. Eine Darstellung von Chaining in Clustering. Initial überlappen die Proxies *C2* und *C3* sowie *C3* und *C4*. Clustering fasst zuerst die Proxies *C2* und *C3* in ein Cluster zusammen. Dieses überlappt jedoch ebenfalls mit *C4* – im nächsten Schritt wird nun auch C4 geclustert. Im Gegensatz dazu erstellt kaskadierendes Clustering ohne Zwischenschritte ein Cluster für alle Proxies.

Rechenzeit. Auch die Verteilung von Proxies beeinflusst die Dauer der Clustering-Verfahren: Je weniger Proxies überlappen, desto schneller terminiert das Verfahren – außer es entsteht *Chaining* [CK02; Nie16]. Chaining kann bei Clustering in großen KKD auftreten und beschreibt, dass je Iteration ein weiterer Proxy mit einem zuvor hinzugefügten Cluster überlappt. Da Clustering versucht, Proxies optimistisch in Cluster zusammenzufassen – also jeweils nur direkt überlappende Proxies clustert und Überlappung nur für nachfolgende Proxies überprüft – ist das Verfahren anfällig für Chaining. Kaskadierendes Clustering hingegen fasst Proxies pessimistisch in Cluster zusammen und ist somit resistenter gegenüber Chaining. Abbildung 4.6 verdeutlicht diese Situation: Zunächst überlappen die Proxies *C2* und *C3* sowie *C3* und *C4*. *C2* und *C3* werden mittels Clustering nun in ein Cluster zusammengefasst - *C4* wird nicht geclustert, da der damit überlappende Proxy *C3* bereits geclustert wird und die Überlappung so optimistischerweise gelöst wird. Jedoch überlappt das Cluster nun ebenfalls mit *C4* – im nächsten Schritt sind somit alle Proxies in einem Cluster enthalten. Dieser Effekt kann über mehrere Iterationen hinweg auftreten und dadurch für eine höhere Rechenzeit sorgen. So fällt auf, dass kaskadierendes Clustering in sehr großen KKD von 1000 Proxies konsistent performanter als Clustering war. Für die in KLighD-VS Code typischen Diagramme bleibt Clustering jedoch effizienter, da Chaining nur ein geringes Ausmaß annehmen kann – besonders mit aktivierten Filtern und Smart Zoom.

Schließlich wurde die Performanz des Proxy-Views insgesamt auch per Developer Tools[3] in KLighD-VS Code gemessen. In diesem Werkzeug ist auch der Call-Stack einzelner Funktionen sowie die zur Berechnung benötigte Zeit aufgeführt. Bisherige Beobachtungen konnten so weiter bestätigt werden – besonders fiel jedoch auf, dass die Berechnung der Positionen von Proxies in großen KKD wie `wagon.sctx` ohne Smart Zoom verhältnismäßig lange brauchen kann. Grund hierfür ist die Berücksichtigung der Sidebar: Damit Proxies nicht hinter dieser angezeigt werden, wird die Sidebar als Teil des Bildschirmrands aufgefasst und Proxies entsprechend verschoben. Entsprechend wird die Sidebar aus dem DOM selektiert, um die Position und Maße dynamisch zu erhalten. Diese Selektion kann für einen – relativ zu anderen ausgeführten Funktionen – großen zusätzlichen Aufwand sorgen. Beobachtet wurde eine maximale Rechenzeit von circa 10 Millisekunden. Hier hätte etwa mit Caching gearbeitet werden können, damit die Sidebar pro Update-Schritt nur ein mal im DOM gesucht werden muss.

Auch Along-Border-Routing kann auf diverse Arten verbessert werden. Vom Konzept ab wurde bereits beachtet, dass Verbindungsstellen zwischen Kanten und Knoten eine semantische Bedeutung haben können. Jedoch kann der Pfad zum Proxy ästhetischer gestaltet werden: Abbildung 3.6b zeigt den Stand der Implementierung. In Abbildung 4.4 ein Beispiel, wie diese per Routing entlang des Proxys verbessert werden kann. Außerdem unterstützt Along-Border-Routing bisher ausschließlich Polylines vollständig. Die auch in SCCharts verwendeten Splines sind zwar zumeist funktional, können jedoch selten auftretende Inkonsistenzen beim Pfad aufweisen.

[3] `https://developer.chrome.com/docs/devtools`

Nur wenige Animationen wurden in dieser Arbeit implementiert. Insbesondere ein animierter, reibungsloser Übergang zwischen off-screen Knoten und entsprechendem Proxy kann zu einem besseren Verständnis sowie zur Konsistenz des KKD beitragen [WN04]. In diesem Sinne wurden Debug-Optionen implementiert, mit denen bereits nur teilweise off-screen Knoten in Proxies übergehen. Für einen flüssigen Übergang haben Proxies die selbe Größe des Knotens – so entsteht der Effekt, dass der Knoten am Bildschirmrand haftet. Je nach off-screen Anteil des Knotens, könnte der Proxy auf seine eigentliche Größe schrumpfen. Die Implementierung hierfür ist jedoch noch nicht voll ausgereift und ist somit Gegenstand zukünftiger Arbeiten. Denkbar wäre auch, Knoten für einen flüssigen Übergang in das Proxy-Rendering umzuwandeln – etwa mittels Image-Morphing-Strategien [Wol98; LCH+96; Wol96].

Wie bereits von M. Frisch und R. Dachselt [FD10] evaluiert, steigert Interaktivität mit Proxies die Nutzerfreundlichkeit. Dementsprechend wurde Hopping [IGY06] implementiert: Per Klick auf einen Proxy wird ein animierter Sprung zum entsprechenden off-screen Knoten ausgelöst. Auch weitere derartige Interaktionen können problemlos ergänzt werden – etwa per Rechtsklick, Hover oder Drag and Drop.

Schließlich fällt ebenfalls auf, dass semantische Filter vom Proxy-View zwar in der Sidebar eingebaut werden, deren Zustand jedoch nicht übernommen werden kann. Jede Änderung des KKD führt dazu, dass die Filter auf ihren initialen Zustand zurückgesetzt werden – anders als in den meisten Anwendungsfällen erwartet. Hierfür existiert noch kein finaler Lösungsansatz, da semantische Filter weder im selben Diagramm noch in unterschiedlichen Diagrammen eine konsistente einzigartige ID besitzen müssen. So wäre es bei naiven Lösungsansätzen möglich, dass unterschiedliche semantische Filter clientseitig als der selbe Filter interpretiert werden würden. Voraussichtlich wäre hierfür eine Überarbeitung der API für semantische Filter notwendig.

Kapitel 5

Schluss

Wir stellten das Proxy-View-Konzept von M. Frisch und R. Dachselt [FD10] vor und erweiterten dieses für allgemeine Knoten-Kanten-Diagramme (KKD). Im Proxy-View wird für jeden off-screen Knoten ein Stellvertreter – der Proxy – als Miniatur am Rand angezeigt. Proxies vermitteln eine kontextuelle Übersicht über sonst nicht sichtbare Informationen. In diesem Sinne wurden auch diverse Erweiterungen konzeptuell vorgestellt. Abbildung 2.1 enthält eine umfassende Übersicht des Proxy-Views im einfachen Fall. Jedoch ist auch in hierarchischen KKD eine Anwendung des Proxy-Views möglich – das Konzept wurde hierfür als Teil dieser Arbeit erweitert.

Wir implementierten das erarbeitete Konzept in KLighD-VS Code[1] für allgemeine KKD. Auch Erweiterungen und Definitionen für spezifische Diagrammtypen sind per Schnittstelle möglich. In Abbildung 3.8 die gleiche umfassende Übersicht des Proxy-Views in der Implementierung.

Wir evaluierten Konzept und Implementierung anhand regelmäßigen, arbeitsgruppeninternen Feedbacks sowie weiterer Qualitätsmerkmale. Abschnitt 5.1 fasst diese Evaluation knapp zusammen. In Abschnitt 5.2 wird auf zukünftige Arbeiten verwiesen.

5.1. Fazit

Konzeptuell wurde für den Proxy-View bereits vieles bedacht. Gewisse Anwendungsfälle zeigten jedoch die Grenzen des Konzepts auf: Wir stellten fest, dass die strikte Eingrenzung Proxies nur für die äußersten off-screen Elternknoten anzuzeigen, in gewissen Fällen nicht ausreicht und ergänzten das Konzept so um Hierarchical Depth und schlugen weitere Alternativen vor. Auch evaluierten wir sowohl konzeptionell als auch qualitativ in der Implementierung die Tücken von kaskadierendem Clustering – sodass dieses keinen praktikablen Teil des Proxy-Views darstellt.

Qualitativ fanden wir ferner heraus, dass durch den Proxy-View nur ein verhältnismäßig geringer zusätzlicher Aufwand entsteht. Ein Vergleich der Performanz in Diagrammen unterschiedlicher Größen deutet auf eine gute Skalierbarkeit hin. So konnten keine Einbrüche der FPS oder anderweitige Einschränkungen der Nutzererfahrung festgestellt werden. Schließlich wiesen wir auf einige Stellen hin, welche verbesserungswürdig und leicht erweiterbar sind – darunter etwa Ästhetikanmerkungen zu Along-Border-Routing oder Vorschläge für Animationen und Interaktionen.

[1] https://github.com/kieler/klighd-vscode

5.2. Zukünftige Arbeiten

Über die Arbeit hinweg wurde bereits vermehrt auf mögliche Themen zukünftiger Arbeiten verwiesen. So ergab sich in der Evaluation aus Kapitel 4, dass Animationen für das Erscheinen und Verschwinden von Proxies einen wichtigen Bestandteil der Nutzererfahrung ausmachen. Durch flüssige Animationen kann ein verständnisvoller sowie konsistenter Übergang zwischen Knoten und Proxies gewährleistet werden [WN04]. Wir schlugen vor, Knoten etwa je nach off-screen Anteil der Größe des entsprechenden Proxys anzupassen. Auch Image-Morphing-Strategien sind jedoch denkbar [Wol98; LCH+96; Wol96]. Für hierarchische Diagramme wäre eine weitere Möglichkeit, nur Kindknoten als Proxies anzuzeigen, wobei das Proxy-Rendering und dessen Größe exakt dem des Kindknotens entsprechen.

Außerdem können die bisher implementierten Interaktionen erweitert werden: Etwa könnte per Hover über einen Proxy weitere Informationen eingeblendet werden oder mittels eines Doppelklicks auf Cluster die Inhalte angezeigt werden. Hier gibt es noch viele Möglichkeiten, die Nutzererfahrung zu verbessern und eine intuitive Navigation zu erleichtern.

In dieser Arbeit wurde ferner lediglich Überlappung von Proxies betrachtet und Lösungsmöglichkeiten hierfür präsentiert. Auch Segmentproxies und Kantenproxies können jedoch überlappen – besonders bei aktiviertem Along-Border-Routing. Wir stellten diverse Lösungsansätze hierfür vor, die weiterer Forschung bezüglich der Anwendung im Proxy-View bedürfen – darunter Edge-Bundling [New89; ZXY+13], Minimize-Edge-Crossings [EMW86; EW94] und Edge-Coloring [JRF+09].

Auch die Behandlung weiterer Diagrammtypen ist Thema zukünftiger Arbeiten. Etwa kann der Proxy-View um Regeln für KKD, welche Kanten zwischen Knoten unterschiedlicher Hierarchieebenen erlauben, erweitert werden. In Abbildung 4.2 wird ein Vergleich des momentanen Stands und möglichen Optima gezeigt. In diesem Sinne können auch Hierarchical Depth und alternative Lösungsansätze weiter erforscht und evaluiert werden – Tabelle 4.1 enthält eine Übersicht möglicher Lösungen.

Für die Implementierung wurde eine orthogonale Projektion der Proxies an den Bildschirmrand verwendet. Die Implementierung von radialer Projektion sowie Along-Edge-Projection [FD10] könnte eine intuitive Navigation weiter verstärken.

Mittels effizienten Datenstrukturen – wie balancierten binären Suchbäumen [And99] – ist es außerdem möglich, den für Clustering verwendeten Sweep-Line-Ansatz [SH76; Sou05] zu optimieren. Bereits ein intelligentes Einfügen von Clustern könnte die Performanz des Ansatzes erhöhen, da so nur einmalig eine Sortierfunktion aufgerufen werden muss.

Darüber hinaus können weitere Clustering-Verfahren untersucht werden: Als Analoga zu Opacity + Stacking Order können mehrere Ansätze entwickelt werden. Etwa würde mittels Size + Stacking Order die Größe von Proxies mit steigender Entfernung zum Bildschirm verringert werden. Fernab des Proxy-Views könnte eine Erweiterung der API für semantische Filter ein semantisches Clustering ermöglichen. Geclustert werden Proxies hierbei etwa nach Knotentyp oder Vererbungshierarchien in Klassendiagrammen [FD10] – beliebige Kriterien sind hierfür möglich. Bei Abstraktion der Projektion könnte so ermöglicht werden, semantische Cluster als eine Art des Overview + Detail [CKB09; FD10] zu verwenden, indem diese

geordnet am oberen Rand des Bildschirms platziert sind und per Interaktion eine Übersicht enthaltener Proxies anzeigen – worüber etwa Hopping [IGY06; FD10] angewendet werden könnte.

Abschließend würde eine Evaluation von Konzept und Implementierung des Proxy-View mittels Umfrage weitere Erkenntnisse des aktuellen Stands aufzeigen. Da der primäre Sinn des Proxy-Views ist, Nutzer und Nutzerinnen zu bei der Navigation von KKD unterstützen, sollte Nutzerfeedback eingeholt und sowohl Konzept als auch Implementierung dementsprechend angepasst werden.

Anhang A

Algorithmen

Algorithmus 1: Clustering

Problem: Alle überlappenden Proxies sollen geclustert werden bis keine Überlappung mehr vorhanden ist
Input: Liste aller Proxies in, Boolean useSweepLine, Boolean cascading
Output: Liste nicht überlappender Proxies und Cluster res
Nachbedingung: res.length $\leq$ in.length

```
Clustering():
1  res = in
   // Kann in einen approximativen Algorithmus umgewandelt werden, indem etwa
    eine maximale Anzahl an Iterationen gesetzt ist oder sobald ein anderes
    approximatives Qualitätskriterium erfüllt ist
2  while true
       // Liste aller Gruppen überlappender Indizes von Proxies
3      groups = []
       /* Finde Überlappungen                                              */
4      if useSweepLine
5          groups = GetGroupsSweepLine(res, cascading)
6      else
7          groups = GetGroupsSimple(res, cascading)
       // Keine Überlappung detektiert
8      if groups ist leer
9          break
10     if cascading
11         Füge alle Gruppen aus groups zusammen, die mindestens einen gleichen Index
            beinhalten (transitive merging)
       /* Erstelle Cluster und entferne darin enthaltene Proxies           */
12     Erstelle für jede Gruppe in groups ein Cluster und füge dieses am Ende von res an
13     Entferne alle Proxies aus res deren Index in einer Gruppe von groups vorhanden ist
14 return res
```

Algorithmus 2: GetGroupsSweepLine

Problem: Finde Gruppen überlappender Indizes von Proxies
Input: Liste aller Proxies res, Boolean cascading
Output: Liste aller Gruppen überlappender Indizes groups

```
GetGroupsSweepLine():
 // Liste aller Gruppen überlappender Indizes von Proxies
1 groups = []
2 Sortiere res aufsteigend primär nach x- und sekundär nach y-Koordinaten
3 for i = 0 to res.length −1
    // Verhindere redundantes Clustering falls nicht kaskadierend
4   if (nicht cascading) und (i in einer Gruppe von groups)
5     continue
    // Gruppe aller mit res[i] überlappenden Indizes
6   currGroup = []
7   for j = 0 to res.length −1
      // Verhindere redundantes Clustering
8     if (i == j) oder (j in einer Gruppe von groups)
9       continue
      // Früheres Abbruchkriterium des Sweep-Line-Ansatzes
10    if (res[j].x > res[i].x) oder (res[j].x == res[i].x und res[j].y > res[i].y)
        // Alle mit res[i] überlappenden Proxies gefunden
11      break
12    if res[i] überlappt mit res[j]
13      Füge j in currGroup ein
14  if currGroup ist nicht leer
      // res[i] überlappt mit anderen Proxies
15    Füge i in currGroup ein
16    Füge currGroup in groups ein
17 return groups
```

Algorithmus 3: GetGroupsSimple

Problem: Finde Gruppen überlappender Indizes von Proxies
Input: Liste aller Proxies res, Boolean cascading
Output: Liste aller Gruppen überlappender Indizes groups

```
GetGroupsSimple():
 // Liste aller Gruppen überlappender Indizes von Proxies
1 groups = []
2 for i = 0 to res.length −1
     // Verhindere redundantes Clustering falls nicht kaskadierend
3    if (nicht cascading) und (i in einer Gruppe von groups)
4       continue
     // Gruppe aller mit res[i] überlappenden Indizes
5    currGroup = []
6    for j = i + 1 to res.length −1
7       if res[i] überlappt mit res[j]
8          Füge j in currGroup ein
9    if currGroup ist nicht leer
        // res[i] überlappt mit anderen Proxies
10      Füge i in currGroup ein
11      Füge currGroup in groups ein
12 return groups
```

Literatur

[And99] Arne Andersson. „‚General Balanced Trees‘“. In: *Journal of Algorithms* 30.1 (1999), S. 1–18. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1998.0967.

[BB05] Anastasia Bezerianos und Ravin Balakrishnan. „‚The Vacuum: Facilitating the Manipulation of Distant Objects‘“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. Portland, Oregon, USA: Association for Computing Machinery, 2005, S. 361–370. ISBN: 1581139985. DOI: 10.1145/1054972.1055023.

[BCG06] Stefano Burigat, Luca Chittaro und Silvia Gabrielli. „‚Visualizing Locations of Off-Screen Objects on Mobile Devices: A Comparative Evaluation of Three Approaches‘“. In: *Proceedings of the 8th Conference on Human-Computer Interaction with Mobile Devices and Services*. MobileHCI '06. Helsinki, Finland: Association for Computing Machinery, 2006, S. 239–246. ISBN: 1595933905. DOI: 10.1145/1152215.1152266.

[BCR+03] Patrick Baudisch, Ed Cutrell, Daniel Robbins, Mary Czerwinski, Peter Tandler, Benjamin Bederson und Alex Zierlinger. „‚Drag-and-Pop and Drag-and-Pick: Techniques for Accessing Remote Screen Content on Touch- and Pen-Operated Systems‘“. In: *Human-Computer Interaction–INTERACT '03*. IOS Press, Jan. 2003, S. 57–64. URL: https://www.microsoft.com/en-us/research/publication/drag-and-pop-and-drag-and-pick-techniques-for-accessing-remote-screen-content-on-touch-and-pen-operated-systems/.

[BR03] Patrick Baudisch und Ruth Rosenholtz. „‚Halo: A Technique for Visualizing off-Screen Objects‘“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '03. Ft. Lauderdale, Florida, USA: Association for Computing Machinery, 2003, S. 481–488. ISBN: 1581136307. DOI: 10.1145/642611.642695.

[BRS+99] K. Bergner, A. Rausch, M. Sihling und A. Vilbig. „‚Structuring and refinement of class diagrams‘“. In: *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences. 1999. HICSS-32. Abstracts and CD-ROM of Full Papers*. Bd. Track6. 1999, 10 ff. DOI: 10.1109/HICSS.1999.772616.

[BRS97] Klaus Bergner, Andreas Rausch und Marc Sihling. Using UML for Modeling a Distributed Java Application. Techn. Ber. 1997.

[BSS12] Vladimir Bochkarev, Anna Shevlyakova und Valery Solovyev. „‚Average word length dynamics as indicator of cultural changes in society‘“. In: *Social Evolution and History* 14 (Aug. 2012), S. 153–175.

[CK02] Michal Chalamish und Sarit Kraus. „‚Learning Users Interests for Providing Relevant Information‘“. In: (Jan. 2002). https://www.researchgate.net/publication/246671507_Learning_Users_Interests_for_Providing_Relevant_Information.

[CKB09] Andy Cockburn, Amy Karlson und Benjamin B. Bederson. „‚A Review of Overview+detail, Zooming, and Focus+context Interfaces‘“. In: *ACM Comput. Surv.* 41.1 (Jan. 2009). ISSN: 0360-0300. DOI: 10.1145/1456650.1456652.

[CVD+07] Mauro Cherubini, Gina Venolia, Rob DeLine und Amy J. Ko. „‚Let's Go to the Whiteboard: How and Why Software Developers Use Drawings‘“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. San Jose, California, USA: Association for Computing Machinery, 2007, S. 557–566. ISBN: 9781595935939. DOI: 10.1145/1240624.1240714.

[DCB+21] Julien Deantoni, João Cambeiro, Soroush Bateni, Shaokai Lin und Marten Lohstroh. „‚Debugging and Verification Tools for Lingua Franca in Gemoc Studio‘“. In: *2021 Forum on specification Design Languages (FDL)*. 2021, S. 01–08. DOI: 10.1109/FDL53530.2021.9568383.

[Dom18] Sören Domrös. „‚Moving Model-Driven Engineering from Eclipse to Web Technologies‘“. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Nov. 2018.

[EMW86] Peter D. Eades, B. D. McKay und Nicholas Charles Wormald. „‚On an Edge Crossing Problem‘“. In: *Proceedings of the Ninth Australian Computer Science Conference*. Australian National University, 1986, S. 327–334.

[EW94] Peter Eades und Nicholas C. Wormald. „‚Edge crossings in drawings of bipartite graphs‘“. In: *Algorithmica* 11.4 (Apr. 1994), S. 379–403. ISSN: 1432-0541. DOI: 10.1007/BF01187020.

[FD10] Mathias Frisch und Raimund Dachselt. „‚Off-Screen Visualization Techniques for Class Diagrams‘“. In: *Proceedings of the 5th International Symposium on Software Visualization*. SOFTVIS '10. Salt Lake City, Utah, USA: Association for Computing Machinery, 2010, S. 163–172. ISBN: 9781450300285. DOI: 10.1145/1879211.1879236.

[Fou21] Clément Fournier. „‚A Rust Backend for Lingua Franca‘“. https://cfaed.tu-dresden.de/publications?pubId=3245. Diploma thesis. Technische Universität Dresden, Dez. 2021.

[GAB+18] Uwe Gruenefeld, Abdallah El Ali, Susanne Boll und Wilko Heuten. „‚Beyond Halo and Wedge: Visualizing out-of-View Objects on Head-Mounted Virtual and Augmented Reality Devices‘“. In: *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*. MobileHCI '18. Barcelona, Spain: Association for Computing Machinery, 2018. ISBN: 9781450358989. DOI: 10.1145/3229434.3229438.

[GAH+17] Uwe Gruenefeld, Abdallah El Ali, Wilko Heuten und Susanne Boll. „‚Visualizing Out-of-View Objects in Head-Mounted Augmented Reality‘“. In: *Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services*. MobileHCI '17. Vienna, Austria: Association for Computing Machinery, 2017. ISBN: 9781450350754. DOI: 10.1145/3098279.3122124.

[GBG+08] Sean Gustafson, Patrick Baudisch, Carl Gutwin und Pourang Irani. „‚Wedge: Clutter-Free Visualization of off-Screen Locations‘“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '08. Florence, Italy: Association for Computing Machinery, 2008, S. 787–796. ISBN: 9781605580111. DOI: 10.1145/1357054.1357179.

[HA04] Victoria Hodge und Jim Austin. „‚A Survey of Outlier Detection Methodologies‘“. In: *Artificial Intelligence Review* 22.2 (Okt. 2004), S. 85–126. ISSN: 1573-7462. DOI: 10.1023/B:AIRE.0000045502.10941.a9.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer und Owen O'Brien. „‚SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications: HW/SW-Synthesis for a Conservative Extension of Synchronous Statecharts‘“. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, S. 372–383. ISBN: 9781450327848. DOI: 10.1145/2594291.2594310.

[HLC09] Chung-Yang (Ric) Huang, Chao-Yue Lai und Kwang-Ting (Tim) Cheng. „‚CHAPTER 4 - Fundamentals of algorithms‘“. In: *Electronic Design Automation*. Hrsg. von Laung-Terng Wang, Yao-Wen Chang und Kwang-Ting (Tim) Cheng. Boston: Morgan Kaufmann, 2009, S. 173–234. ISBN: 978-0-12-374364-0. DOI: 10.1016/B978-0-12-374364-0.50011-4.

[IGY06] Pourang Irani, Carl Gutwin und Xing Dong Yang. „‚Improving Selection of Off-Screen Targets with Hopping‘“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. Montréal, Québec, Canada: Association for Computing Machinery, 2006, S. 299–308. ISBN: 1595933727. DOI: 10.1145/1124772.1124818.

[JRF+09] Radu Jianu, Adrian Rusu, Andrew Fabian und David Laidlaw. „‚A Coloring Solution to the Edge Crossing Problem‘“. In: Juli 2009, S. 691–696. DOI: 10.1109/IV.2009.66.

[LCH+96] SEUNG-YONG LEE, KYUNG-YONG CHWA, JAMES HAHN und SUNG YONG SHIN. „‚Image Morphing Using Deformation Techniques‘“. In: *The Journal of Visualization and Computer Animation* 7.1 (1996), S. 3–23. DOI: 10.1002/(SICI)1099-1778(199601)7:1<3::AID-VIS131>3.0.CO;2-U.

[Lin00] Xuemin Lin. „‚On the computational complexity of edge concentration‘“. In: *Discrete Applied Mathematics* 101.1 (2000), S. 197–205. ISSN: 0166-218X. DOI: 10.1016/S0166-218X(99)00207-3.

[LM17] Tatjana Lange und Karl Mosler. „‚Normalverteilung und zentraler Grenzwertsatz‘“. In: *Statistik kompakt: Basiswissen für Ökonomen und Ingenieure*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 55–62. ISBN: 978-3-662-53467-0. DOI: 10.1007/978-3-662-53467-0_6.

[LMB+21] Marten Lohstroh, Christian Menard, Soroush Bateni und Edward A. Lee. „‚Toward a Lingua Franca for Deterministic Concurrent Systems‘“. In: *ACM Trans. Embed. Comput. Syst.* 20.4 (Mai 2021). ISSN: 1539-9087. DOI: 10.1145/3448128.

[LMS+20] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon und Edward A. Lee. „‚A Language for Deterministic Coordination Across Multiple Timelines‘“. In: *Proc. Forum on Specification and Design Languages (FDL '20)*. Kiel, Germany, Sep. 2020.

[Mok17] Amirouche Moktefi. „‚Diagrams as scientific instruments‘“. In: *Virtual Reality–Real Visuality* (2017), S. 81–89.

[Mot17] Christian Motika. SCCharts — Language and Interactive Incremental Compilation. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Kiel University, Germany. Department of Computer Science, 2017. ISBN: 9783746009391. DOI: 10.21941/kcss/2017/02.

[MP21] Mirza Alim Mutasodirin und Radityo Eko Prasojo. „‚Investigating Text Shortening Strategy in BERT: Truncation vs Summarization‘“. In: *2021 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. 2021, S. 1–5. DOI: 10.1109/ICACSIS53237.2021.9631364.

[New89] F. J. Newbery. „‚Edge Concentration: A Method for Clustering Directed Graphs‘“. In: *Proceedings of the 2nd International Workshop on Software Configuration Management*. SCM '89. Princeton, New Jersey, USA: Association for Computing Machinery, 1989, S. 76–85. ISBN: 0897913345. DOI: 10.1145/72910.73350.

[Nie16] Frank Nielsen. „‚Hierarchical Clustering‘“. In: *Introduction to HPC with MPI for Data Science*. Cham: Springer International Publishing, 2016, S. 195–211. ISBN: 978-3-319-21903-5. DOI: 10.1007/978-3-319-21903-5_8.

[PCJ96] Helen C. Purchase, Robert F. Cohen und Murray James. „‚Validating graph drawing aesthetics‘“. In: *Graph Drawing*. Hrsg. von Franz J. Brandenburg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, S. 435–446. ISBN: 978-3-540-49351-8.

[Pet95] Marian Petre. „‚Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming‘“. In: *Commun. ACM* 38.6 (Juni 1995), S. 33–44. ISSN: 0001-0782. DOI: 10.1145/203241.203251.

[Ren18] Niklas Rentz. „‚Moving Transient Views from Eclipse to Web Technologies‘“. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Nov. 2018.

[SH76] Michael Ian Shamos und Dan Hoey. „‚Geometric intersection problems‘“. In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 1976, S. 208–215. DOI: 10.1109/SFCS.1976.16.

[Sou05] Diane Souvaine. „‚Line segment intersection using a sweep line algorithm‘“. In: *Tufts University* (2005). https://www.cs.tufts.edu/comp/163/notes05/seg_intersection_handout.pdf.

[SS07] Jiri Soukup und Martin Soukup. „‚The Inevitable Cycle: Graphical Tools and Programming Paradigms‘“. In: *Computer* 40.8 (2007), S. 24–30. DOI: 10.1109/MC.2007.293.

[SSH13] Christian Schneider, Miro Spönemann und Reinhard von Hanxleden. „‚Just model! — Putting automatic synthesis of node-link-diagrams into practice‘“. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 2013, S. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[WN04] J.J. van Wijk und W.A.A. Nuij. „‚A model for smooth viewing and navigation of large 2D information spaces‘“. In: *IEEE Transactions on Visualization and Computer Graphics* 10.4 (2004), S. 447–458. DOI: 10.1109/TVCG.2004.1.

[Wol96] G. Wolberg. „‚Recent advances in image morphing‘“. In: *Proceedings of CG International '96*. 1996, S. 64–71. DOI: 10.1109/CGI.1996.511788.

[Wol98] George Wolberg. „‚Image morphing: a survey‘“. In: *The visual computer* 14.8 (1998), S. 360–372.

[YLX+16] Xinli Yang, David Lo, Xin Xia und Jianling Sun. „‚Condensing Class Diagrams With Minimal Manual Labeling Cost‘“. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Bd. 1. 2016, S. 22–31. DOI: 10.1109/COMPSAC.2016.83.

[ZMG+03] Polle T. Zellweger, Jock D. Mackinlay, Lance Good, Mark Stefik und Patrick Baudisch. „‚City Lights: Contextual Views in Minimal Space‘“. In: *CHI '03 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '03. Ft. Lauderdale, Florida, USA: Association for Computing Machinery, 2003, S. 838–839. ISBN: 1581136374. DOI: 10.1145/765891.766022.

[ZXY+13] Hong Zhou, Panpan Xu, Xiaoru Yuan und Huamin Qu. „‚Edge bundling in information visualization‘“. In: *Tsinghua Science and Technology* 18.2 (2013), S. 145–156. DOI: 10.1109/TST.2013.6509098.