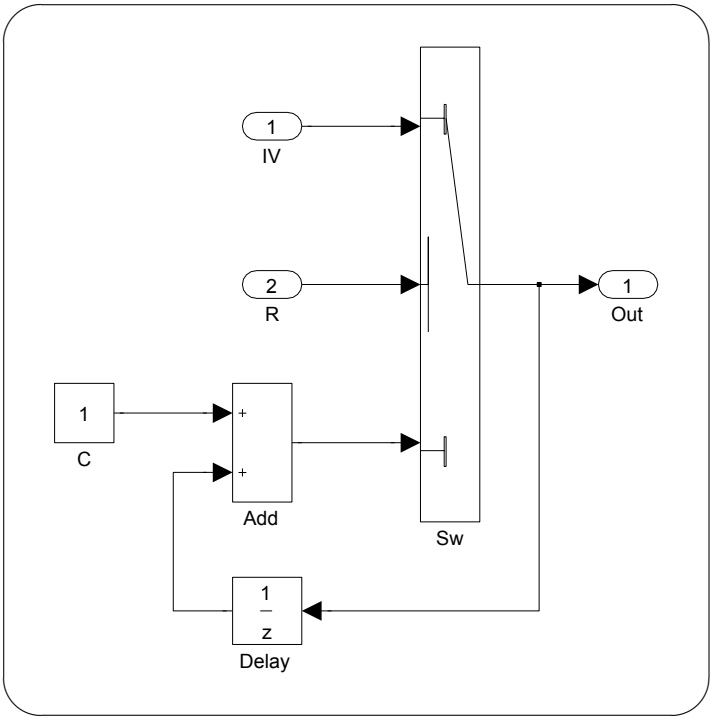
LARS KRISTIAN KLAUSKE

EFFIZIENTES BEARBEITEN VON SIMULINK MODELLEN MIT HILFE EINES SPEZIFISCH ANGEPASSTEN LAYOUTALGORITHMUS



[disi: arti:]

EFFIZIENTES BEARBEITEN VON SIMULINK MODELLEN MIT HILFE EINES SPEZIFISCH ANGEPASSTEN LAYOUTALGORITHMUS

Dipl.-Ing. Lars Kristian Klauske

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

genehmigte

DISSERTATION

Promotionsausschuss
Prof. Dr. rer. nat. Sabine Glesner (Vorsitzende)
Prof. Dr. Stefan Jähnichen (Gutachter)
Prof. Dr. rer. nat. Reinhard von Hanxleden (Gutachter)

Tag der wissenschaftlichen Aussprache:
28. August 2012

Berlin 2012

D83

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Fachbereich Softwaretechnik
Ernst-Reuter-Platz 7
D-10587 Berlin

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
AG Echtzeitsysteme und Eingebettete Systeme
Olshausenstr. 40
D-24098 Kiel

ZUSAMMENFASSUNG

Die grafische modellbasierte Entwicklung mit Werkzeugen wie Matlab / Simulink ist ein weit verbreiteter Ansatz zur Entwicklung eingebetteter Software im Automobilbereich und von zunehmender Bedeutung in Bahntechnik und Luftfahrtindustrie. Hierbei werden zunächst mit Hilfe grafischer Editoren Modelle erstellt, die Struktur und Funktion einer Softwarekomponente repräsentieren. Die Software selbst kann dann mittels automatischer Codegeneratoren direkt aus den Modellen erzeugt werden. Im Vergleich mit textueller Software gelten grafische Modelle als besser lesbar, verständlicher und leichter wartbar. Insbesondere die Lesbarkeit der Modelle hängt jedoch maßgeblich vom Layout der grafischen Darstellung ab. Da aktuelle Modelleditoren kaum Unterstützung beim Erstellen von Layouts bieten, müssen diese bisher aufwändig manuell erstellt werden.

In dieser Arbeit wird ein spezifisch an die grafischen Eigenschaften von Simulink Diagrammen angepasster Layoutalgorithmus beschrieben. Er baut auf einem Algorithmus zum hierarchischen Zeichnen gerichteter Graphen auf und ergänzt diesen sowohl um ein Verfahren zur Kantenbegradigung unter Berücksichtigung variabler Knotengrößen, als auch um einen auf zusätzlichen Hilfsknoten basierenden Ansatz zum Kantenrouting bei Zykluskanten und Kanten an oben oder unten liegenden Ports. Zudem führt er benutzerdefinierte Constraints bei der Hierarchisierung ein, die es ermöglichen, die Auswirkung der frühen Hierarchisierung bei hierarchischen Layoutverfahren auf das Layout zu kontrollieren. Aufbauend auf diesem Layoutalgorithmus werden Funktionen zur Unterstützung beim strukturellen Bearbeiten von Modellen, insbesondere das auf einer Heuristik zur Auswahl neu einzufügender Elemente basierende *kontextbasierte Modellieren*, vorgestellt.

Im Rahmen einer Erprobung konnte gezeigt werden, dass durch den Einsatz dieser Verfahren beim Bearbeiten von Simulink® (SL) Modellen in der Praxis eine deutliche Zeitersparnis und Erleichterung der Arbeit erreicht werden kann. Benutzer können sich stärker als bisher auf die funktionalen Aspekte der Modelle konzentrieren und das grafische Anordnen der Elemente automatisch durchführen lassen. Darüber hinaus wird die Entwicklung von Modellgeneratoren zum automatischen Erzeugen von Modellteilen deutlich erleichtert.

ABSTRACT

Model-based development of embedded software using tools like Matlab / Simulink is widely-used in the automotive domain and becoming equally important in other domains like aerospace and rail engineering. In model-based development, a software component is directly created using automatic code generation, using graphical models which represent the components structure and functionality as primary input. When compared to textual software development, graphical models are often considered to be more readable, thereby easier to understand and maintain. This readability is significantly influenced by the layout of a models graphical representation. To maintain it, it is often required to adjust the layout when the model is modified. As current model editors offer no automatic layouting support, this has to be done manually.

This paper presents a layout algorithm, that is specifically taylored to deal with the graphical characteristics of Simulink models. It complements an existing algorithm that creates hierarchical drawing from directed graphs by adding a method to straighten edges under the influence of variable-sized nodes based on linear programming and a method to improve edge routing for feedback edges as well as edges connected to node faces across the signal flow direction. Additionally, it reduces the original algorithms hierarchisation phases influence on the final layout by introducing user-defined contraints. Building on this layout algorithm, several functions to assist the model editing process are introduced, like *kontextual editing*, which provides automatic insertion and connection of elements selected from a heuristically generated list of most propable candidates.

Using a real-world evaluation of the methods presented in this work, it is shown that both time and effort can be reduced during the model editing process by using those methods. Also, by leaving the layout generation to the provided layout algorithm, users were reportedly able to better focus onto a models functional aspect than before. Additional benefit in terms of flexibility and reduced effort could be seen in the development of model generation tools, that had to develop individual static layouting schemas for each function before.

VERÖFFENTLICHUNGEN

Die in dieser Dissertation vorgestellte Arbeit entstand zwischen **01 2008** und **12 2011** an der Technischen Universität Berlin (Daimler Center for Automotive Information Technology Innovations). Einige Ideen und Darstellungen sind bereits in den im Folgenden aufgelisteten vorherigen Veröffentlichungen erschienen:

[56] **Lars Kristian Klauske** und Christian Dziobek: *Improving Modeling Usability: Automated Layout Generation for Simulink*, Mathworks Automotive Conference 2010 (MAC10), 06 2010, Stuttgart, Deutschland

[57] **Lars Kristian Klauske** und Christian Dziobek: *Effizientes Erstellen von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus*, 7. Dagstuhl-Workshop MBEES 2011: Modellbasierte Entwicklung eingebetteter Systeme, 02 2011, Dagstuhl, Deutschland

Die im folgenden Konferenzbeitrag vorgestellten Ideen und Darstellungen basieren bereits teilweise auf dieser Arbeit und wurden in Zusammenarbeit des Autors mit der AG Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel entwickelt:

[58] **Lars Kristian Klauske**, Christoph Daniel Schulze, Miro Spönemann und Reinhard von Hanxleden: *Improved Layout for Data Flow Diagrams with Port Constraints*, Diagrams 2012: The 7th International Conference on the Theory and Application of Diagrams, 07 2012, Canterbury, UK

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

Teil I

GRUNDLAGEN

1 EINLEITUNG UND MOTIVATION

Seit Einführung der ersten Softwarefunktionen in Fahrzeugen vor gut 30 Jahren wächst deren Umfang exponentiell. Mehr als 1000 vernetzte Softwarefunktionen mit über 10 Mio. Programmzeilen auf über 70 Steuergeräten machen heutzutage etwa 50-70% des Entwicklungsaufwandes von elektronischen Systemen in aktuellen Fahrzeugen aus [7, 93]. Da die meisten Innovationen im Automobilbereich mittlerweile von softwaregesteuerten Funktionen ausgehen, wird erwartet, dass sich diese Entwicklung auch in den kommenden 20 Jahren weiter fortsetzt [7]. Um die steigende Komplexität der Funktionen bei gleichzeitig gestrafften Entwicklungszyklen beherrschbar zu halten, wird eingebettete Software im Automobilbereich zunehmend modellbasiert entwickelt [94] – ein Trend, der auch in anderen Industriezweigen wie der Luftfahrt oder der Bahntechnik zu beobachten ist.

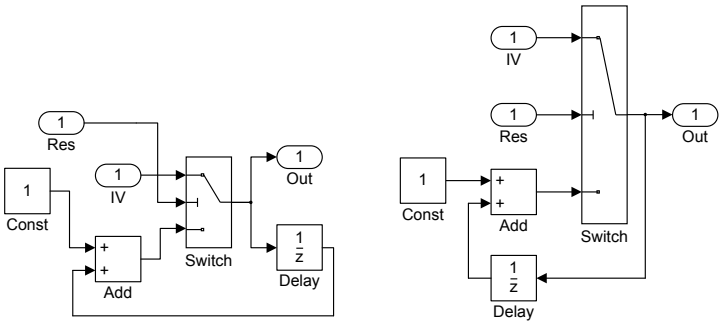
Die dabei eingesetzten Werkzeuge zur modellbasierten Softwareentwicklung wie MATLAB® (ML) / SL, das Advanced Simulation and Control Engineering Tool ® [30] (ASCET), Ptolemy [28] oder SCADE [29] ermöglichen die Beschreibung der Softwarefunktionen als grafische Funktionsmodelle in Form von Zustands- und Datenflussdiagrammen. Diese Modelle können vom Benutzer in den grafischen Editoren der Werkzeuge bearbeitet, in Simulationen validiert und mittels effizienter Codegeneratoren wie RealTime Workshop® (RTW) oder Targetlink® (TL) direkt in Steuergeräte integriert werden.

Eine bessere Lesbarkeit und Anschaulichkeit dieser grafischen Modelle im Vergleich mit textuell entwickelter Software sowie eine daraus resultierende Verbesserung der Wartbarkeit und Wiederverwendbarkeit der Software gelten als maßgebliche Gründe für eine zunehmende Verbreitung der modellbasierten Entwicklung.

1.1 PROBLEMSTELLUNG

Ausschlaggebend für die Lesbarkeit eines Diagramms ist einerseits die Anschaulichkeit der einzelnen Beschreibungselemente, andererseits das Layout des Diagramms, d.h. die Anordnung und Größe der Elemente in Relation zueinander [73]. Abbildung 1.1 verdeutlicht dies anhand zweier

möglicher Layouts des SL Diagramms eine Zählers: Unmittelbar nach dem Erstellen sind die Elemente des Diagramms in Abbildung 1.1a nur grob angeordnet, unnötige Linienknicke und -kreuzungen reduzieren die Lesbarkeit. Das Layout wird daher in einem weiteren Arbeitsschritt überarbeitetet (Abbildung 1.1b), um diese unnötigen Linienknicke und -kreuzungen zu entfernen und Knoten mit besonderer Funktion im Signalfluss (*Delay* Knoten) hervorzuheben.



(a) Neu erstelltes Diagramm. **(b)** Überarbeitetes Layout.

Abbildung 1.1: Verschiedene Layouts eines Simulink Diagramms.

Die aufgeführten Modellierungssprachen enthalten zwar die erforderlichen visuellen Elemente und Freiheitsgrade zum Erstellen lesbarer Darstellungen, ihre Editoren bieten jedoch häufig lediglich rudimentäre Unterstützung beim Erstellen der Layouts. Diese Aufgabe muss daher bisher mit großem Aufwand manuell erledigt werden: Untersuchungen beim Bearbeiten von Zustandsdiagrammen zeigen, dass bis zu 40% des Aufwandes beim Modellieren allein auf das manuelle Erstellen des Layouts entfallen [36, 64]. Basierend auf Erfahrungen aus aktuellen Projekten im Automobilbereich liegt dieser Anteil beim Erstellen von SL-Diagrammen mit etwa 25% zwar niedriger, ist jedoch immer noch signifikant. Zudem fehlen den Modelleditoren im Gegensatz zu textuellen Editoren selbst einfache Entsprechungen von Funktionen wie automatischer Einrückung und Hervorhebung von Schlüsselwörtern, kontextsensitiver Codevervollständigung oder Refactoring Operationen, die bei textuellen Editoren bereits seit vielen Jahren Stand der Technik sind.

Demgegenüber steht der aktuelle Stand der Wissenschaft: Das automatische Erstellen von Layouts für Graphen, das *Graph Drawing*, ist eine aktive Disziplin der Informatik und für viele Arten von Diagrammen wurden bereits effiziente Layoutalgorithmen entwickelt. Diese behandeln insbesondere allgemeine Formen der verschiedenen Diagrammtypen, berücksichtigen jedoch die spezifischen Eigenschaften der in praktischen

Werkzeugen verwendeten Diagrammtypen nicht oder nur unzureichend. Eine direkte Anwendung der Algorithmen auf diese Diagrammtypen ist auf Grund ihrer Eigenschaften entweder gar nicht möglich oder resultiert in Darstellungen, die von den Anwendern nicht akzeptiert werden würden.

1.2 BEITRAG UND ABGRENZUNG

In dieser Arbeit wird die Forschung des Autors zur Verbesserung der Effizienz beim Bearbeiten von SL Datenfluss-Diagrammen mit Hilfe eines spezifisch angepassten Layoutalgorithmus sowie der Automatisierung typischer Arbeitsschritte als Modellierungsunterstützung zusammengefasst.

1.2.1 LAYOUT VON SIMULINK DIAGRAMMEN

Beim manuellen Erstellen von Layouts für SL Diagramme versuchen Benutzer typischerweise Knicke in Linien zu vermeiden, da diese mit dem Auge schwerer zu verfolgen sind als gerade Linien [86]. Da Ports bei SL Diagrammen in Abhängigkeit von der Größe des jeweiligen Blocks positioniert werden, verändern die Benutzer zur Kantenbegradigung nicht nur die Position von Blöcken und Linien, sondern auch die Größe der Blöcke. Dieses Vorgehen ermöglicht im Vergleich zur reinen Positionierung von Blöcken eine deutlich bessere Vermeidung von Linienknicken.

In dieser Arbeit wird ein Algorithmus zum automatischen Erstellen von Layouts für SL Diagramme auf Basis hierarchischer Layoutverfahren vorgestellt. Dieser nutzt lineare Programmierung, um erstmals zur Begradigung von Linien nicht nur die Position von Blöcken zu verändern, sondern auch deren Größe. Zudem führt er mit den *Inverter* Knoten ein neues Verfahren für das Routing von Zyklen ein und reduziert damit im Vergleich zu bestehenden Verfahren die Anzahl der Knicke in diesen Linien. Er nutzt zusätzliche Hilfsknoten, um das Kantenrouting an Ports zu verbessern, die quer zur Ausrichtung des Diagramms liegen. Zudem beachtet er Benutzervorgaben bezüglich der Anordnung von Blöcken während der Hierarchisierung und bei der Begradigung von Kanten.

Der Algorithmus wurde in einem prototypischen Werkzeug implementiert und Probanden aus mehreren Bereichen der Forschung und Entwicklung eines Fahrzeugherstellers zur Erprobung zur Verfügung gestellt.

Die Auswertung der qualitativen Aussagen der Probanden über Nutzen und Grenzen des Werkzeugs in der Praxis sowie eine quantitative Untersuchung des Algorithmus sind ebenfalls Teil dieser Arbeit.

1.2.2 MODELLIERUNGSUNTERSTÜTZUNG

Aufbauend auf der Möglichkeit automatisch Layouts für SL Diagramme zu erzeugen, wurden in Anlehnung an Refactoring-Operationen aus der textuellen Programmierung sowie bestehenden Konzepten zur fortschrittlichen Bearbeitung von Statecharts verschiedene Unterstützungsfunktionen für SL Diagramme entwickelt:

Das *kontextbasierte Modellieren* erleichtert das Auswählen, Einfügen und Verbinden eines neuen Elements in ein bestehendes Diagramm, indem dessen Auswahl zunächst mit Hilfe einer Heuristik unterstützt und dieses dann automatisch eingefügt und verbunden wird. Das *Aufbrechen von Subsystemen* überträgt die Refactoring Operation *Inline Method* [32, Seite 117] als Modelltransformation auf SL Diagramme und integriert diese in den SL Editor. Zusätzlich wird mit dem *wahlfreien Einfügen von Ports* das Bearbeiten von Bussen, dem Äquivalent von Datenstrukturen in SL, mit Hilfe einer einfachen Modelltransformation erleichtert.

1.2.3 ABGRENZUNG

Die dieser Arbeit zugrunde liegende Forschungsarbeit konzentrierte sich insbesondere auf die Modellierungssprache des in der Automobilindustrie weit verbreiteten Werkzeugs SL sowie der darin verwendeten Datenfluss-Diagramme in praxistypischen Größen. *Statecharts*, die Zustandsdiagramme aus der SL Erweiterung *Stateflow*, werden im Rahmen dieser Arbeit nicht betrachtet.

Es ist zu erwarten, dass die entwickelten Algorithmen auch auf andere, auf Datenfluss-Diagrammen basierende, Modellierungssprachen und Werkzeuge anwendbar sind, obwohl deren besondere Eigenschaften nicht explizit betrachtet werden. Dies wird beispielsweise beim Algorithmus zur Begradigung von Kanten ersichtlich, der in der vorgestellten Form auch mit anderen, als den in SL zum Einsatz kommenden *Port Constraints* umgehen kann.

1.3 KAPITELÜBERSICHT

Die vorliegende Arbeit gliedert sich in die folgenden Kapitel:

KAPITEL 1

Das erste Kapitel führt in das Thema der Arbeit ein, motiviert diese und stellt den geleisteten Beitrag dar.

KAPITEL 2

Im nachfolgenden Kapitel werden verwandte Ansätze zum Layout von Datenfluss-Diagrammen beschrieben und ihre Grenzen bezüglich der spezifischen Eigenschaften von SL Diagrammen dargestellt. Zudem werden die in der Arbeit verwendeten Formalismen, Methoden und Techniken eingeführt.

KAPITEL 3

Dieses Kapitel stellt die zum Erstellen von Layouts für SL Diagramme entwickelten Erweiterungen des in Kapitel 2 vorgestellten hierarchischen Layoutverfahrens dar: Der neue Ansatz zur Zyklenbehandlung reduziert im Vergleich zu bestehenden Ansätzen die Kantenknicke bei Zykluskanten. Erweiterungen in der Berechnung der Hierarchisierung ermöglichen die Beeinflussung des resultierenden Layouts durch den Benutzer und die Reduktion von Kantenkreuzungen. Zusätzliche Hilfsknoten ermöglichen ein besseres Routing von Kanten bei Ports, die nicht entlang der Ausrichtung des Diagramms liegen.

Der Schwerpunkt des Kapitels ist der auf linearer Programmierung basierende Algorithmus zur Begradigung von Kanten unter Berücksichtigung spezieller Eigenschaften bezüglich der Portpositionen bei verschiedenen Typen von Datenfluss-Diagrammen. Dieser wird auf Basis einer Analyse der spezifischen Eigenschaften von SL Diagrammen und einer daraus entwickelten Linearisierung der Berechnungsvorschrift für Portpositionen in SL auf diese übertragen.

KAPITEL 4

Die im Rahmen dieser Forschungsarbeit entwickelten Funktionen zur Modellierungsunterstützung werden in diesem Kapitel vorgestellt: Mit dem *kontextbasierten Editieren* wurde ein neues Konzept zum Editieren von SL Diagrammen entwickelt, das die typischen Arbeitsschritte beim Auswählen, Einfügen und Verbinden neuer Blöcke beim Bearbeiten von

Diagrammen zusammenfasst und mit Hilfe einer Heuristik unterstützt. Zudem werden die beim *Aufbrechen von Subsystemen* und dem *wahlfreien Einfügen von Ports* aufgrund bestehender Einschränkungen des SL Editors notwendigen manuellen Arbeitsschritte mit Hilfe von Modelltransformationen automatisiert.

KAPITEL 5

Die Realisierung des in Kapitel 2 und 3 beschriebenen Algorithmus sowie die Technik zu dessen Einbindung in den SL Editor ist in diesem Kapitel beschrieben.

KAPITEL 6

Der in den Kapitel 2 und 3 vorgestellte Algorithmus in Form des in Kapitel 4 beschriebenen prototypischen Werkzeugs wurde mit Hilfe verschiedener Metriken bezüglich der resultierenden Kantenknicke, Knotengröße und Laufzeit untersucht. Zudem wurde in verschiedenen Bereichen der Vor- und Serienentwicklung eines Automobilherstellers eine Evaluierung des Werkzeugs bezüglich seiner Praxistauglichkeit durchgeführt. Die jeweils verwendeten Methoden und Metriken sowie ihre Resultate sind in diesem Kapitel dargestellt.

KAPITEL 7

Im abschließenden Kapitel wird die Arbeit zusammengefasst, die aufgezeigten Ansätze reflektiert und bezüglich ihrer Grenzen diskutiert. Zudem werden künftige Forschungsarbeiten im Themenfeld der Arbeit aufgezeigt.

2 GRUNDLAGEN UND VERWANDTE ARBEITEN

Die hier vorgestellte Arbeit konzentriert sich auf den Ansatz zur modellbasierten Entwicklung aus der SL Modellierungsumgebung. Für die darin verwendeten SL Diagramme wurde ein Layoutalgorithmus auf Basis linearer Optimierung sowie verschiedene Funktionen zur Unterstützung beim Bearbeiten der Diagramme entwickelt. In diesem Kapitel wird daher zunächst eine Einführung in die SL Modellierungsumgebung gegeben und es werden sowohl die grafischen als auch die strukturellen Eigenschaften von SL Diagrammen beschrieben. Nachfolgend werden verwandte Ansätze zum Layout von Diagrammen mit ähnlichen grafischen Eigenschaften aufgezeigt, bezüglich ihrer Eignung zum Layout von SL Diagrammen diskutiert und von der hier vorgestellten Arbeit inhaltlich abgegrenzt. Abschließend werden die im weiteren Verlauf dieser Arbeit verwendeten Techniken und Methoden beschrieben: Das Kiel Integrated Environment for Layout – Eclipse Rich Client [38] (KIELER) als technische Grundlage der Implementierung und die lineare Optimierung als mathematische Grundlage des Layoutalgorithmus.

2.1 DIE SIMULINK MODELLIERUNGSUMGEBUNG

SL ermöglicht das Modellieren, Simulieren und Analysieren von Systemen mit zeitvarianten Ausgängen [62]. Die Systeme werden dabei vorwiegend grafisch in expliziter Form mit der Simulink Modellierungssprache beschrieben, in der Funktionen als Blöcke dargestellt werden, die über durch Linien repräsentierte Signale miteinander Informationen austauschen. Diese Modelle können mit Hilfe verschiedener Simulationsalgorithmen direkt in SL simuliert werden. Ebenfalls im Werkzeug enthaltene Funktionen zur Visualisierung und Analyse der Simulationsergebnisse ermöglichen die Auswertung der Simulationen.

SL wird sowohl zur Entwicklung von Algorithmen zur Steuerung und Regelung technischer Systeme oder der Signalverarbeitung, als auch zur Beschreibung von elektrischen, mechanischen oder thermodynamischen Systemen verwendet. Auf SL aufbauende Werkzeuge wie RTW oder TL erlauben es, diese Modelle direkt in kompilierbaren Programmcode für Steuergeräte (C) oder in synthesefähige Hardwarebeschreibungen (VHDL) zu übersetzen.

2.1.1 GRAFISCHE EIGENSCHAFTEN VON SIMULINK DIAGRAMMEN

SL Diagramme bestehen aus Knoten, Kanten und Ports. Im Rahmen dieser Arbeit werden für die grafischen Elemente die in der Graphentheorie bevorzugten Bezeichnungen Knoten (*Nodes*) und Kanten (*Vertices*) verwendet. In SL selbst werden Knoten normalerweise als Blöcke (*Blocks*) und Kanten als Linien (*Lines*) bezeichnet.

Definition 1. *Ein SL Diagramm ist ein Graph* G*, bestehend aus Knoten* V*, gerichteten Kanten* $E \subset V^2$ *und Ports* $P \subset \{ p = (v, F) \mid v \in V, F \subseteq E \}$.

Alle Elemente (Knoten, Kanten, Ports) eines SL Diagramms werden anhand eines linkshändigen, zweidimenensionalen kartesischen Koordinatensystems angeordnet. Der Ursprung des Koordinatensystems liegt links oben im Diagramm, die horizontale (x) Achse verläuft also von links nach rechts, die vertikale (y) Achse von oben nach unten.

Definition 2. *Die vertikale Position der linken oberen Ecke eines Knotens* $v \in V$ *in einem SL Diagramm wird mit* $y(v)$ *bezeichnet (Abbildung 2.1), die horizontale Position mit* $x(v)$.

Alle Koordinaten müssen ganzzahlige Koordinaten sein. Zusätzlich wird die Positionierung von Elementen durch den Benutzer vom grafischen Editor in SL derart eingeschränkt, dass über die Graphical User Interface (Grafische Benutzerschnittstelle) (GUI) Koordinaten nur in 5er Schritten erreichbar sind: $0, 5, 10, \ldots$. Obwohl diese Einschränkung nur für die GUI und nicht für die Programmierschnittstelle gilt, muss sie auch hier beachtet werden, da die Diagramme sonst nur noch eingeschränkt manuell bearbeitet werden können.

Definition 3. *Alle Koordinaten in SL Diagrammen sind ganzzahlige Vielfache von 5:* $0, 5, 10, \ldots$.

Die Eigenschaften der verschiedenen grafischen Elemente werden nachfolgend beschrieben.

KNOTEN

Knoten in SL Diagrammen werden als Rechtecke, vereinzelt auch als Dreiecke oder Kreise dargestellt. Im weiteren Verlauf dieser Arbeit werden die Knoten jedoch vereinfacht als Rechtecke betrachtet, wobei sowohl dreieckige, als auch runde Knoten jeweils durch das kleinste umschließende Rechteck repräsentiert werden. Abbildung 2.2a zeigt verschiedene Knotenformen in SL, jeweils mit umschließenden Rechtecken bei nicht rechteckigen Knoten.

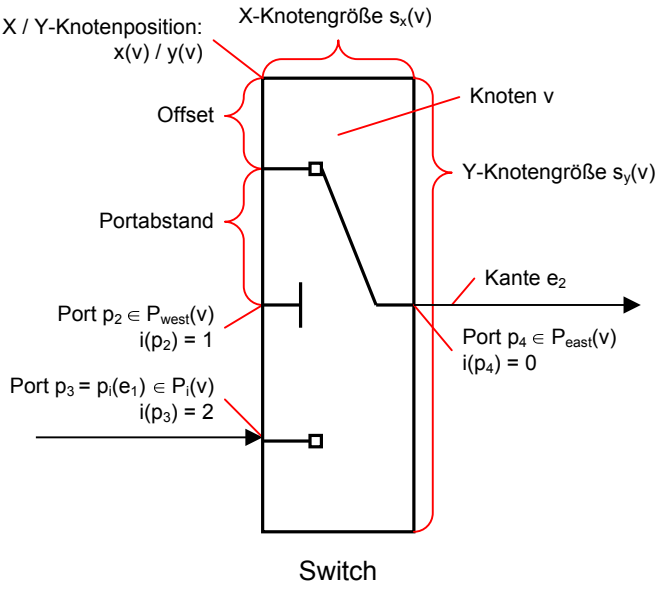


Abbildung 2.1: Beispielhafte Darstellung eines Simulink Knotens mit Beschriftungen gemäß der Definitionen aus Abschnitt 2.1.1.

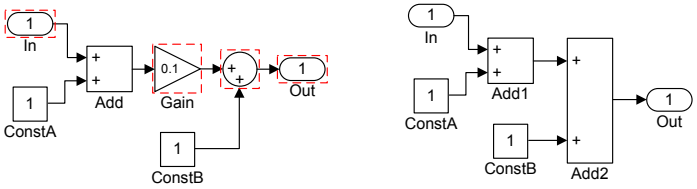
Definition 4. *Die Seiten eines Knotens werden, beginnend mit der oberen Seite, im Uhrzeigersinn als* north, east, south, west *bezeichnet. Bei nicht rechteckigen Knoten beziehen sich die Bezeichnungen auf die Seiten des umschließenden Rechtecks.*

Die Größe der Knoten ist variabel und direkt durch den Benutzer anpassbar: In Abbildung 2.2b wurde der Addierer *Add2* in seiner Größe verändert um die eingehenden Kanten ohne Knicke zeichnen zu können.

Definition 5. *Die Größe von SL Knoten $v \in V$ ist variabel, die Höhe wird als $s_y(v)$, die Breite als $s_x(v)$ bezeichnet (Abbildung 2.1).*

Ein Ansatz zur automatischen Anpassung dieser Knotengrößen zur Begradigung von Kanten ist Kernbestandteil dieser Arbeit und wird in Abschnitt 3 vorgestellt.

SL ermöglicht es, Knoten in 90-Grad-Schritten rotiert und horizontal gespiegelt darzustellen. Gespiegelte (*flipped*) Knoten werden in der Praxis häufig zur Verdeutlichung der Darstellung von Rückkopplungen verwendet, beispielsweise bei *Delay* Knoten (siehe Abbildung 2.7 in



(a) Verschiedene Knotenformen. **(b)** Verschiedene Knotengrößen.

Abbildung 2.2: Form und Größe verschiedener Knoten.

Abschnitt 2.1.3), daher werden diese im Rahmen dieser Arbeit in Abschnitt 3.1.1 zur Kennzeichnung von Rückkopplungen besonders berücksichtigt. Rotierte Knoten werden nicht gesondert betrachtet, da diese in der Praxis kaum Anwendung finden und deren Verwendung zudem meist durch Richtlininen eingeschränkt wird.

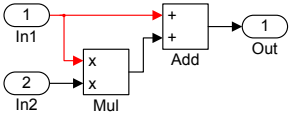
Zu den Knoten gehörende Bezeichner wie der Name des Knotens oder Informationen über Parameter werden bei SL außerhalb der Knoten dargestellt. Im Rahmen dieser Arbeit werden diese Bezeichner explizit unterstützt, wenn ihre Anordnung den in Abschnitt 2.1.4 aufgeführten Modellierungsrichtlinien entspricht und diese zentriert unterhalb der Knoten positioniert sind.

Die Erweiterung der Ansätze dieser Arbeit auf rotierte Knoten durch Verschiebung der betroffenen Ports an andere Seiten des Knotens oder auf beliebig am Knoten positionierte Bezeichner ist jedoch trivial.

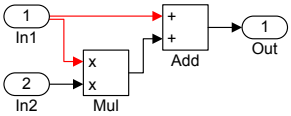
KANTEN

Eine einfache gerichtete Kante zeigt von einem Quell- zu einem Zielknoten; die auf mehrere Knoten verallgemeinerte Form einer Kante ist die Hyperkante, die einen oder mehrere Quell- mit einem oder mehreren Zielknoten verbindet. Die Kanten in SL Diagrammen sind gerichtete Hyperkanten, die genau einen Port eines Quellknotens mit einem oder mehreren Zielknoten an einem oder mehreren Ports verbindet (Abbildung 2.3a). Im Rahmen dieser Arbeit werden diese Hyperkanten durch mehrere einfache Kanten repräsentiert (siehe Abbildung 2.3b).

Definition 6. *Kanten in SL Diagrammen sind gerichtete Kanten* $e = (u, v) \in E \subseteq V^2$*. Die Menge aller Kanten mit gleichem Quellknoten und -port* p *wird als* $h(p) \subseteq E$ *bezeichnet; sie entspricht einer Hyperkante, die den Port* p *mit allen* $|h(p)|$ *Zielen der Elemente von* $h(p)$ *verbindet.*

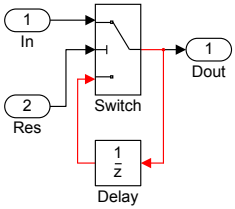


(a) Darstellung in Simulink.

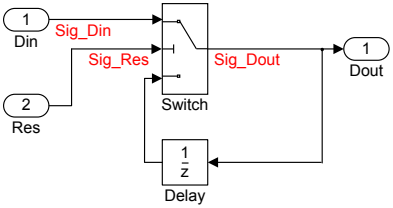


(b) Interne Repräsentation.

Abbildung 2.3: Interne Repräsentation von SL Hyperkanten.



(a) Mögliche Zykluskanten.



(b) Erlaubte Positionierung von Bezeichnern.

Abbildung 2.4: Zyklen und Bezeichner bei Simulink Kanten.

Alle Kanten in SL Diagrammen werden rechtwinklig gezeichnet. Die Begradigung dieser Kanten ist Bestandteil dieser Arbeit und wird in Abschnitt 3 genauer betrachtet.

Die Kanten in SL Diagrammen können Zyklen bilden (Abbildung 2.4a). Abschnitt 3.1.1 stellt einen neuen Ansatz zum Umgang mit Zyklen unter Berücksichtigung der SL Port Constraints vor.

Bezeichner (Signalnamen) können bei SL an verschiedenen Positionen der Kante dargestellt werden. In der Praxis kommen jedoch häufig Modellierungsrichtlinien (siehe Abschnitt 2.1.4) zum Einsatz, die die Positionierung dieser Bezeichner einschränken. Daher werden im Rahmen dieser Arbeit nur Bezeichner betrachtet, die den in Abschnitt 2.1.4 aufgeführten Richtlinien entsprechen: Die Positionierung erfolgt unterhalb der Kante, direkt an deren Anfang oder Ende (siehe Abbildung 2.4b).

PORTS

Ports sind die Verbindungspunkte zwischen Kanten und Knoten, deren Positionierung unterschiedlichen Einschränkungen (*Port Constraints*) unterliegen kann.

Definition 7. *Ein Port* $p = (F, v) \in P$ *bezeichnet den Verbindungspunkt zwischen einer oder mehrerer Kanten (*$F \subseteq E$*) und einem Knoten* $v \in V$*. Siehe auch Abbildung 2.1.*

Port in SL Diagrammen unterliegen Port Constraints, bei denen die Positionierung der Ports direkt von der Anzahl der Ports und der Größe des Knotens abhängig ist (`PORT_CALCULATED`). Eine wesentliche Eigenschaft der Berechnungsvorschrift für die Portpositionen ist die feste Reihenfolge der Ports (`PORT_ORDERED`). In Abschnitt 3.5.1 erfolgt eine Analyse der Berechnungsvorschrift von Portpositionen in SL. Die Einführung in die in dieser Arbeit verwendeten Port Constraints erfolgt in Abschnitt 2.2.1.

Definition 8. *Jeder Port ist eindeutig einer Seite* s *eines Knotens* $v \in V$ *zugeordnet:* $s \in \{north, south, west, east\}$ *Die Menge aller Ports einer Seite* s *wird als* $P_s(v)$ *(beispielsweise* $P_{west}(v)$*) bezeichnet. Die Ports einer Seite sind über einen Index* $i_s(p) \in \{0, 1, \ldots, |P_s(v)| - 1\}$ *eindeutig geordnet. Siehe auch Abbildung 2.1.*

Einen Port, der den Anfang einer Kante mit einem Knoten verbindet, bezeichnet man als Ausgangsport (*Outport*), einen Port am Ende einer Kante als Eingangsport (*Inport*).

Definition 9. *Die Menge aller Eingangsports eines Knotens* v *wird als* $P_i(v)$ *bezeichnet, die Menge der Ausgangsports als* $P_o(v)$*. Der Eingangsport am Ende einer Kante* $e \in E$ *wird mit* $p_i(e)$ *bezeichnet, der Ausgangsport am Anfang der Kante als* $p_o(e)$*. Siehe auch Abbildung 2.1.*

Ports in SL Diagrammen sind entweder Eingangsports oder Ausgangsports, niemals beides.

Definition 10. *Ein Port* $p = (F, v)$ *des Knotens* v *ist entweder Eingangsport (*$p \in P_i(v)$*) oder Ausgangsport (*$p \in P_o(v)$*). Die Mengen aller Eingangs- und Ausgangsports sind disjunkt:* $\forall\, (u, v) \in V^2 : P_i(u) \cap P_o(v) = \emptyset$*. Ein Port kann also niemals Eingangs- und Ausgangsport gleichzeitig sein.*

Die Ports sind so angeordnet, dass alle Ausgangsports an einer Seite und die Eingangsports an den anderen Seiten liegen. Die Ein- und Ausgangsports eines Knotens sind jeweils über einen Index geordnet. Bei einem nicht rotierten und nicht gespiegelten Knoten liegen die Ausgangsports an der Seite *east* und die Eingangsports an der Seite *west*. Einige spezielle Blocktypen haben zusätzliche Eingangsports an der Seite *north* (7). Bei einem gespiegelten Knoten vertauschen sich die *east* und *west* Seiten (5). Lokale Schleifen, also direkte Verbindungen eines Knoten mit sich selbst, sind dadurch zwar möglich, verbinden jedoch stets zwei verschiedene Seiten des Knotens miteinander.

Definition 11. *Die Menge aller Eingangsports eines Knotens* $P_i(v)$ *ist über einen Index* $i(p_i) \in \{0, 1 \ldots |P_i(v)| - 1\}, p_i \in P_i(v)$ *geordnet. Analoges gilt für die Menge der Ausgangsports* $P_o(v)$*. Siehe auch Abbildung 2.1.*

Die Berechnungsvorschrift für Portpositionen in SL sorgt dafür, dass alle Ports auf dem Koordinatenraster liegen.

AUSRICHTUNG

SL Diagramme sind bei unveränderter Ausrichtung der Knoten von links nach rechts ausgerichtet, so dass der Anfang einer Kante typischerweise links vom Ende der Kante liegt. Ausnahmen bilden Zykluskanten, die in Gegenrichtung ausgerichtet sind (siehe Abbildung 2.4a). Auch wenn der Editor über das freie Anordnen sowie Rotieren und Spiegeln von Knoten ein Abweichen von dieser Standardausrichtung zulässt, fordern Modellierungsrichtlinien (siehe Abschnitt 2.1.4) typischerweise diese Ausrichtung einzuhalten.

2.1.2 STRUKTURELLE EIGENSCHAFTEN VON SIMULINK MODELLEN

SL Modelle sind hierarchisch in sogenannte Subsysteme strukturiert. Subsysteme werden in der jeweils höheren Hierarchieebene als Knoten des Typs *Subsystem* repräsentiert und enthalten ihrerseits wieder Knoten (auch *Subsystem* Knoten) und Kanten. Im Rahmen dieser Arbeit wird der Inhalt genau einer Ebene eines Subsystems (oder die oberste Modellebene) als Diagramm bezeichnet.

Für die Entwicklung von Layoutalgorithmen ist es hilfreich, die Größe der darzustellenden Diagramme zu kennen. Hierzu wurden Funktionsmodelle aus dem Innenraumbereich eines Automobilherstellers, sowie mit SL R2011a ausgelieferte Demo-Modelle analysiert und miteinander verglichen. Dem sogenannten Innenraumbereich sind Fahrzeugfunktionen wie beispielsweise Innen- und Aussenlicht, Scheibenwischer, Fensterheber oder das Bordnetzmanagement zugeordnet. Abbildung 2.5 stellt die Anzahl von Knoten und Kanten der untersuchten Subsysteme grafisch dar. Der Lesbarkeit halber sind Diagramme mit mehr als 100 Knoten oder 200 Kanten nicht in Abbildung 2.5 enthalten; dies betrifft 21 Diagramme. Tabelle 2.1 führt die Ergebnisse der Analyse noch einmal tabellarisch auf.

Die im Vergleich zu den Demo-Modellen höheren Werte der realen Modelle bei den Kanten und Knoten pro Diagramm in Tabelle 2.1 entsprechen den Erwartungen. Sie sind insbesondere auf die deutlich höhere Anzahl

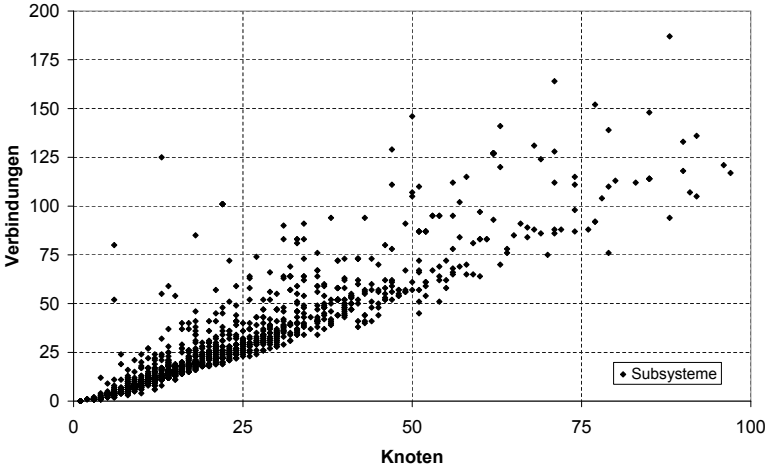


Abbildung 2.5: Anzahl der Knoten und Kanten der untersuchten Subsysteme des Innenraumbereichs.

an Signalen zurückzuführen, die von realen Systemen verarbeitet werden müssen. Schnittstellen mit mehr als 200 Signalen sind bei Modellen aus der Praxis nicht unüblich, die Schnittstellen in den Demomodellen enthalten hingegen selten mehr als 10 Signale. Da in SL jeder Port an der Schnittstelle eines Subsystems durch einen Knoten (*Inports* und *Outports*, siehe Abschnitt 2.1.3) repräsentiert wird, ergibt sich aus größeren Schnittstellen unmittelbar eine höhere Anzahl an Knoten und Kanten in den Modellen.

Einige spezielle Diagramme weisen eine besonders hohe Anzahl von Knoten und Kanten auf: In den realen Modellen sind jeweils ein oder mehrere Subsysteme zur Aufbereitung von Parametern für die Codegenerierung angelegt, damit diese im generierten Programmcode bestimmte Kriterien erfüllen. Auch die Subsysteme mit den äußeren Systemschnittstellen erreichen durch die hohe Anzahl an Signalen in diesen Schnittstellen sehr hohe Werte bei der Anzahl der Knoten oder Kanten. Die Struktur dieser Diagramme ist typischerweise sehr einfach.

Die Häufigkeit, mit der die untersuchten Subsysteme eine bestimmte Größe hatten, ist in Abbildung 2.6 anhand der Anzahl der Kanten dargestellt: 96% der untersuchten realen Subsysteme hatten weniger als 64 Knoten / 100 Verbindungen.

Kriterien		Automotive	Demo
Untersuchte Modelle		12	45
Diagramme		1796	270
Knoten		39993	2913
Kanten		51293	2866
Unterschiedliche Blocktypen		67	112
Knoten pro Diagramm	Min	1	3
	Max	547	39
	Mittelwert	22	11
	Standardabweichung	25	6
Kanten pro Diagramm	Min	0	1
	Max	545	41
	Mittelwert	29	11
	Standardabweichung	35	7

Tabelle 2.1: Statistische Daten aus Modellen des Innenraumbereichs, sowie aus den Simulink Demomodellen.

2.1.3 DER SIMULINK EDITOR

SL vereinigt sowohl den eigentlichen grafischen Editor als auch die Simulationsumgebung für SL Modelle in einem Werkzeug (siehe Abbildung 2.8). Der Editor folgt dem *What you see is what you get* (WYSIWYG) Prinzip: Der Benutzer fügt dabei Knoten per *drag & drop* aus einer erweiterbaren hierarchischen Bibliothek in das Diagramm ein und platziert diese manuell mit der Maus. Das Einfügen von Kanten erfolgt ebenfalls manuell durch Anklicken und Ziehen von einem Port zu einem anderen. Dieses Prinzip lässt dem Benutzer zwar viele Freiheiten beim Anordnen der Knoten und Kanten, bietet ihm jedoch beim Bearbeiten der Diagramme kaum Unterstützung.

BLOCKBIBLIOTHEK

Alle Knoten in SL haben einen Typ (*Blocktype*), der dessen Funktion, sowie die Anzahl und Anordnung der Ein- und Ausgänge festlegt. Der *Blocktype* wird häufig durch unterschiedliche Symbole auf dem Knoten oder durch dessen Form dargestellt. Abbildung 2.7 zeigt ein Beispieldiagramm mit Knoten unterschiedlicher Typen:

- Die *Inport* Knoten *In1*, *Select* und *In2* repräsentieren die gleichnamigen Eingangsports des aktuellen Subsystems. Analog dazu

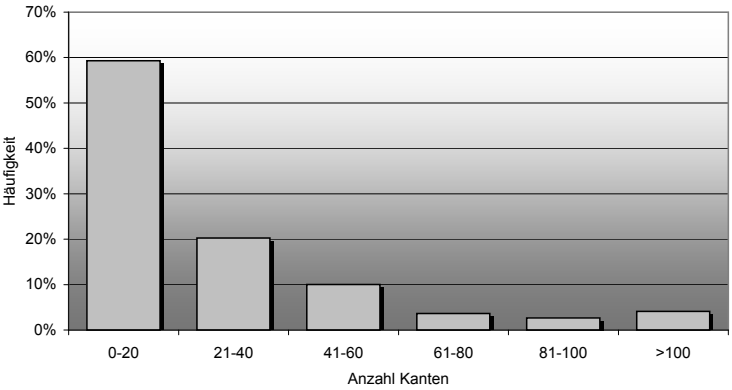


Abbildung 2.6: Anteile an Subsystemen verschiedener Größen, gemessen an der Anzahl der Kanten.

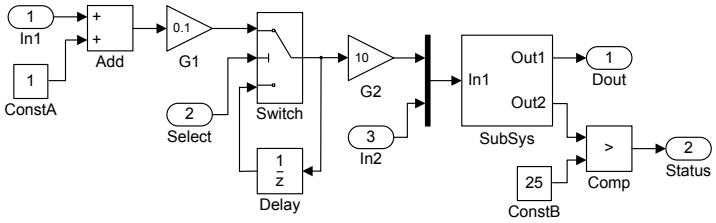


Abbildung 2.7: Beispieldiagramm mit Knoten verschiedener Typen.

stellen die *Outports Dout* und *Status* die Ausgangsports dar. Im Vergleich mit textueller Programmierung entsprechen diese Knoten den Eingangs- und Ausgangsvariablen einer Funktion.

- Der *Subsystem* Knoten *SubSys* repräsentiert ein anderes Subsystem, das wiederum Knoten und Kanten enthalten kann. Die Anzahl der Eingangs- und Ausgangsports dieses Knotens ist variabel. *Subsystem* Knoten sind vergleichbar mit Funktionen in der textuellen Programmierung. Mehr zu Subsystemen und ihren typischen Größen in der Praxis findet sich in Abschnitt 2.1.2.
- *ConstA* und *ConstB* sind *Constant* Knoten und zeigen am Ausgangsport den Wert einer Konstanten. Sie entsprechen einer Konstanten der textuellen Programmierung.
- Der *Add* Knoten vom Typ *sum* addiert die an seinen Eingangsports anliegenden Werte und gibt das Ergebnis an seinem Ausgangsport aus.

- *G1* und *G2* sind vom Typ *Gain* und repräsentieren eine Multiplikation mit einem konstanten Faktor.
- *Switch* ist ein *Switch* Knoten. Je nach Wert des am mittleren Eingangsport anliegenden Signals leitet dieser den Wert des oberen oder unteren Eingangsports an den Ausgang weiter. Er entspricht einem *if/then/else* Konstrukt aus der textuellen Programmierung.
- Der *Delay* Knoten mit der Bezeichnung *Delay* leitet den Wert des Eingangsports mit einer Verzögerung von einem Berechnungszyklus an den Ausgang weiter. Er entspricht einer Zustandsvariable in der textuellen Programmierung.
- Der *Compare* Knoten *Comp* vergleicht die Werte der beiden Eingangsports miteinander (hier: Eingang 1 > Eingang 2) und liefert das boolesche Ergebnis am Ausgang.
- Der senkrechte Block ohne Bezeichner zwischen *G2*/*In2* und *SubSys* repräsentiert ebenfalls einen Knoten: Der *BusCreator* Knoten gruppiert die Werte an seinen Eingangsports und stellt sie am Ausgang als *Bus* zur Verfügung. Dies entspricht der Zuordnung von Werten (Eingangsports) in eine Datenstruktur (Bus). Die Anzahl der Eingangsports ist daher variabel.

Die Blockbibliothek enthält bereits in der Grundausstattung[1] mehr als 240 verschiedene Blocktypen, gegliedert in über 20 Ordnern. Diese werden in typischen Entwicklungsumgebungen einerseits durch zusätzliche Ordner und Blocktypen aus Benutzerbibliotheken und Erweiterungen in ähnlichem Umfang ergänzt, andererseits durch Richtlinien in ihrer Verwendbarkeit wieder eingeschränkt, so dass Benutzer beim Modellieren typischerweise aus rund 200 verschiedenen Blocktypen wählen können. Abbildung 2.8 zeigt beispielhaft Editor und Blockbibliothek aus Simulink R2011a.

Die Bibliothek bietet eine Suchfunktion und eine Liste *häufig verwendeter Blocktypen*, darüber hinaus erhält der Benutzer jedoch keine Unterstützung beim Auswählen, Einfügen und Verbinden der Knoten. Ein Ansatz zur Unterstützung des Benutzers bei der Auswahl sowie beim Einfügen und Verbinden von Knoten wird in Abschnitt 4 dieser Arbeit vorgestellt.

LAYOUT UNTERSTÜTZUNG

Der aktuelle SL Editor[2] bietet lediglich grundlegende, aus Werkzeugen für Desktop-Publishing (DTP) oder Präsentationen bekannte Funktionen zum Anordnen von Knoten, sowie ein einfaches, statisches Kantenrouting.

1 Matlab/Simulink R2010a ohne Benutzerbibliotheken und ohne Erweiterungen
2 Matlab/Simulink R2011a

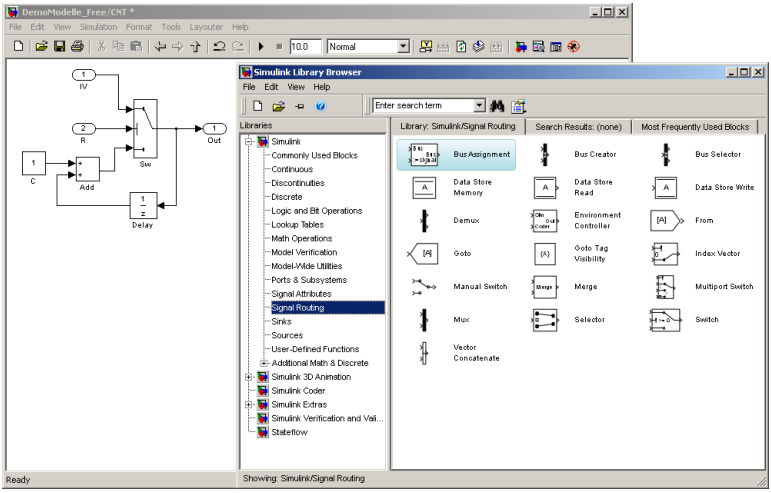
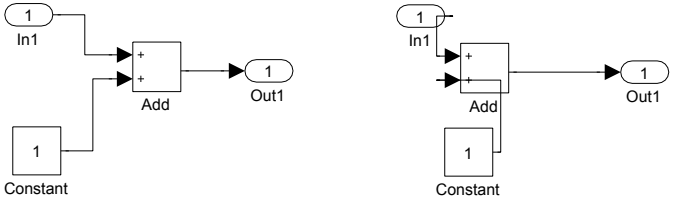


Abbildung 2.8: Editor (Hintergrund) und Blockbibliothek (Vordergrund) aus Simulink R2011a.

Selektierte Knoten können horizontal oder vertikal zentriert, links- oder rechtsbündig ausgerichtet, in gleichen Abständen verteilt oder in ihrer Größe angeglichen werden. Zusätzlich werden beim Verschieben von Knoten Hilfslinien eingeblendet, die das Ausrichten erleichtern.

Das automatische Kantenrouting (Algorithmus nicht veröffentlicht) des Editors arbeitet jedoch nur beim Erstellen neuer Kanten. Bei späteren Veränderungen am Modell (Verschiebung oder Größenänderung von Knoten, Kanten, Bezeichnern) werden lediglich die letzten Segmente einer Kante angepasst, wodurch Überschneidungen zwischen Kanten und Knoten, zusätzliche Kantenkreuzungen und sogar ungültige Kanten-



(a) Neu erstelltes Diagramm. **(b)** Nach dem Verschieben.

Abbildung 2.9: Das Kantenrouting im Simulink Editor ist beim Verschieben von Knoten nicht aktiv.

konfigurationen, beispielsweise Überschneidungen einer Kante mit ihrer eigenen Pfeilspitze, entstehen können. Abbildung 2.9 zeigt die Situation, die durch Verschieben der Knoten *Add* und *Out1* nach links und des Knotens *Constant* nach rechts ohne manuelle Anpassung des Linienroutings entsteht.

In Abschnitt 3 wird ein Verfahren zum automatischen Erstellen von Layouts für SL Diagramme unter Berücksichtigung variabler Knotengrößen vorgestellt.

2.1.4 MODELLIERUNGSRICHTLININEN

Als Modellierungsrichtlinien werden im Rahmen dieser Arbeit Vorgaben zu Struktur, Parametrierung, Namensgebung und Aussehen von Modellen bezeichnet. Sie sollen die durch Methoden und Tools gegebenen Freiheitsgrade beim Erstellen und Bearbeiten von Modellen einschränken, um insbesondere die folgenden Aspekte zu verbessern:

1. Die Modelle sollen in Struktur, Namensgebung und Aussehen einheitlich sein, um Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit zu verbessern.
2. Technische Einschränkungen der Werkzeugketten sollen durch spezielle Strukturen und Parametrierungen vermieden werden.
3. Der aus den Modellen generierte Code soll effizient sein.

Allein für SL existiert eine Vielzahl an Modellierungsrichtlinien aus unterschiedlichen Quellen mit zum Teil ähnlichem oder gleichem Inhalt, da sowohl verschiedene herstellerübergreifende Gremien[3], Werkzeughersteller[4] selbst, als auch einzelne Anwendergruppen wie Automobilhersteller oder Zulieferer eigene Richtlinien entwickelt haben. Die jeweils anzuwendenden Richtlinien werden auf Unternehmens-, Abteilungs-, oder Projektebene typischerweise von Qualitätsmanagern ausgewählt und sind dann in der Regel für alle Beteiligten bindend.

Eine Analyse der MAAB Modellierungsrichtlinien ergab, dass die nachfolgend aufgeführten Richtlinien unmittelbar relevant für das Layout von SL Diagrammen sind. Aufgeführt ist jeweils die ID im Dokument zusammen mit dem englischen Originaltitel, sowie eine übersetzte Zusammenfassung des Inhalts. Diese Richtlinien haben die Entwicklung des Layout Algorithmus für SL Diagramme in dieser Arbeit direkt beeinflusst.

3 Herstellerübergreifende Gremien mit veröffentlichten Modellierungsrichtlinien: Mathworks Automotive Advisory Board [63] (MAAB), Motor Industry Software Reliability Association [66] (MISRA)

4 Z.B. Modellierungsrichtlinien der dSpace GmbH [18]

DB_0141: SIGNAL FLOW IN SIMULINK MODELS

Festlegung der Ausrichtung von SL Diagrammen: Der Signalfluss ist von links nach rechts zu führen; Ausnahmen bilden Zykluskanten. Sequenzielle Knoten sollen von links nach rechts, parallele Knoten von oben nach unten angeordnet werden.

DB_0042: PORT BLOCK IN SIMULINK MODELS

Inport Knoten sollen am linken, *Outport* Knoten am rechten Rand des Diagramms angeordnet werden. Fälle, in denen dadurch zusätzliche Kreuzungen entstehen, dürfen davon ausgenommen werden.

JC_0121: USE OF THE SUM BLOCK

Bei *Sum* Knoten soll die rechtwinklige Form verwendet werden. Ausnahme bilden Summen, an denen Zykluskanten beteiligt sind; in diesen Fällen sind maximal 3 Eingänge an den Positionen 90, 180, 270 Grad (*south*, *west*, *north*) erlaubt.

JM_0002: BLOCK RESIZING

Knoten sollen groß genug sein, um die Darstellung des Icons / Textes im Knoten zu gewährleisten.

JC_0061: DISPLAY OF BLOCK NAMES

Bezeichner von Knoten sollen angezeigt werden, wenn sie für das Verständnis des Modells hilfreich sind.

DB_0140: DISPLAY OF BASIC BLOCK PARAMETERS

Wichtige, von Standardeinstellungen abweichende, Parameter sollen angezeigt werden.

DB_0142: POSITION OF BLOCK NAMES

Angezeigte Bezeichner / Parameter sollen unterhalb der Knoten angeordnet werden.

DB_0097: POSITION OF LABELS FOR SIGNALS AND BUSSES

Bezeichner von Kanten sollen unterhalb horizontaler Kanten nahe des Quell- oder Zielknotens angeordnet werden und müssen eindeutig zuzuordnen sein. Sie sollen sich nicht mit Kanten oder Knoten überschneiden.

DB_0032: SIMULINK SIGNAL APPEARANCE

Kanten sollen...

- sich wenn möglich nicht überkreuzen.
- rechtwinklig gezeichnet werden.
- nicht übereinander gezeichnet werden.
- sich nicht mit Knoten überkreuzen.
- sich an einer Abzweigung nicht in mehr als zwei Segmente aufteilen.

2.2 LAYOUT VON DATENFLUSS-DIAGRAMMEN

Graphenlayout (*Graph Drawing*) stellt sich als eigene Disziplin in der Informatik mit umfassender Literatur dar [17]; Tollis *et al.* und Kaufmann und Wagner geben in ihren Büchern [55, 88] einen umfassenden Überblick über relevante Arbeiten und Ansätze. Das Layout von Datenfluss-Diagrammen als Spezialform gerichteter Graphen ist ein Teilbereich dieser Disziplin.

Die aus Sicht von Layoutalgorithmen besonders hervorzuhebenden Merkmale von SL Diagrammen, vorgestellt in Abschnitt 2.1.1, sind die in ihrer Größe veränderlichen Knoten, sowie die über spezielle Port Constraints in nicht linearer Weise daran gekoppelten Portpositionen. Weitere relevante Eigenschaften sind die Hyperkanten, das orthogonale Kantenrouting sowie Constraints bezüglich der Anordnung bestimmter Knoten. Zudem liegen alle Knoten, Kanten und Ports auf ganzzahligen Koordinaten mit einem Raster von fünf Punkten.

Die folgenden Abschnitte geben nach einer kurzen Einführung von Port Constraints einen Überblick über Ansätze des Graphenlayouts, die prinzipiell zum Anordnen von SL Diagrammen geeignet sein könnten und beschreiben, inwieweit diese bereits Diagramme mit den grafischen Eigenschaften von SL Diagrammen unterstützen.

2.2.1 PORTS UND PORT CONSTRAINTS

Als Ports bezeichnet man die Punkte, an denen Kanten und Knoten miteinander verbunden sind. Die Positionierung der Ports kann gewissen Einschränkungen unterliegen, diese bezeichnet man als Port Constraints.

Im Rahmen dieser Arbeit wird auf die nachfolgend beschriebenen Port Constraints eingegangen:

- `PORT_NONE`: Ports ohne Port Constraint werden auf dem Ursprung des Knotens, typischerweise im Mittelpunkt, platziert.
- `PORT_FREE`: Ports können frei auf dem Rand des Knotens platziert werden und haben keine feste Reihenfolge.
- `PORT_SIDE`: Ports können frei auf dem Rand einer pro Port festgelegten Seite des Knotens platziert werden, haben jedoch innerhalb einer Seite keine feste Reihenfolge.
- `PORT_ORDERED`: Ports können frei auf dem Rand einer pro Port festgelegten Seite des Knotens platziert werden, dürfen dabei jedoch die vorgegebene Reihenfolge nicht verändern.
- `PORT_FIXED`: Alle Ports sind an feste Positionen auf dem Rand des Knotens festgelegt.
- `PORT_CALCULATED`: Alle Portpositionen werden direkt aus der Knotengröße berechnet. Die Reihenfolge der Ports bleibt dabei konstant. Die Berechnung der Portpositionen für SL Diagramme wird in Abschnitt 3.5.1 behandelt.

2.2.2 KRÄFTEBASIERTE LAYOUTVERFAHREN

Dieser Ansatz basiert auf dem iterativen Optimieren einer Kostenfunktion, deren Komponenten die verschiedenen zu erfüllenden ästhetischen Kriterien repräsentieren. Die Bezeichnung "kräftebasierte Verfahren" stammt aus den ersten Varianten von Algorithmen dieser Art, bei denen die Komponenten der Kostenfunktion physikalischen Kräften zwischen Knoten und/oder Kanten nachempfunden waren. Beispielsweise wurden beim *Spring Embedder* die Kanten zwischen den Knoten durch Federn repräsentiert [20].

Die verschiedenen kräftebasierten Verfahren unterscheiden sich einerseits in der Formulierung der Kostenfunktionen und andererseits durch

ihre Optimierungsstrategien, wobei die Betrachtung der Kräfte als allgemeine Kostenfunktion zur Repräsentation der ästhetischen Kriterien insbesondere auf [15, 16] zurückzuführen ist.

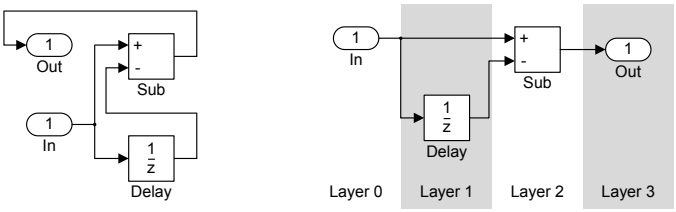
Kräftebasierte Verfahren können durch entsprechend formulierte Kostenfunktionen eine Vielzahl ästhetischer Kriterien erfüllen. Beispielsweise nutzt Lee *et al.* sie in [60], um die *Mental Map*, die geistige Vorstellung des Graphen beim Benutzer (siehe z.B. [68, 76]), bei Änderungen an dessen Struktur möglichst zu erhalten. In [19] werden sie von Dwyer *et al.* um Ansätze zum Routing gekrümmter Kanten erweitert. Noack stellt mit ihnen in [69] zusammenhängende Knotengruppen (*Cluster*) bevorzugt dar. Chuang *et al.*, Harel und Koren setzen kräftebasierte Verfahren in ihren Ansätzen [12, 46] zum Zeichnen von Graphen mit ungleich großen Knoten ein und weisen dabei insbesondere auf die Problematik hin, dass diese Verfahren bei Graphen, deren Knoten in Relation zur Gesamtfläche groß sind, typischerweise eher langsam konvergieren. Sie schlagen daher verschiedene Strategien vor, diesen Vorgang zu beschleunigen.

Obwohl die Anwendung kräftebasierter Layoutverfahren auf Blockdiagramme mit variablen Knotengrößen bei geeigneter Formulierung der Kostenfunktion prinzipiell möglich erscheint, ist eine effiziente praktische Umsetzung bisher nicht dokumentiert. Diese erste Einschätzung deckt sich mit unseren eigenen Versuchen aus der Anfangsphase dieser Forschungsarbeit, bei denen kräftebasierte Verfahren mit unterschiedlichen Kostenfunktionen auf SL Diagramme angewendet wurden.

2.2.3 Hierarchische Layoutverfahren

Gerichtete azyklische Graphen lassen sich vollständig hierarchisieren (Vgl. [91]). Beim Hierarchisieren werden die Knoten des Graphen derart geordneten *Layern* zugewiesen, dass jede Kante des Graphen von einem *Layer* mit niedriger zu einem *Layer* mit hoher Nummer führt. Abbildung 2.10 stellt dies beispielhaft für einen einfachen Graphen ohne Zyklen dar.

Basierend auf dieser Eigenschaft wurde von Sugiyama *et al.* in [85] ein mehrstufiger Algorithmus vorgeschlagen, mit dem gerichtete Graphen entlang einer Hauptrichtung gezeichnet werden können (*Sugiyama-Algorithmus*). Eventuell vorhandene Zyklen werden dabei in einer ersten Phase entfernt, um eine eindeutige Hierarchisierung des Graphen zu ermöglichen. Spätere Arbeiten erweitern, verfeinern und verbessern den Algorithmus, behalten jedoch die grundlegende Vorgehensweise im Wesentlichen bei [26, 41, 79]. Für jede der nachfolgend beschriebenen Phasen sind mittlerweile verschiedene Lösungsansätze dokumentiert. Der ursprüngliche Algorithmus ist in nur vier Phasen beschrieben; er fasst die hier getrennt aufgeführten Phasen 1 und 2 zu einer Phase zusammen.



(a) Vor der Hierarchisierung. **(b)** Nach der Hierarchisierung.

Abbildung 2.10: Hierarchisierung eines einfachen Graphen ohne Zyklus.

Der nachfolgend beschriebene hierarchische Layoutalgorithmus bildet die Basis für den in Abschnitt 3 vorgestellten Layoutalgorithmus für SL Diagramme. Der Algorithmus erwartet einen gerichteten Graphen G, der Zyklen enthalten kann.

PHASE 1: ZYKLENERKENNUNG UND -BEHANDLUNG

Die erste Phase gliedert sich in zwei Arbeitsschritte: In der Zyklenerkennung werden diejenigen Kanten erkannt, die Zyklen im Graphen erzeugen (siehe auch [88] S. 294 ff.). Diese Zykluskanten werden dann im zweiten Arbeitsschritt bisher entweder umgekehrt (Vertauschen von Quelle und Ziel), oder temporär aus dem Graphen entfernt, so dass am Ende der Phase ein gerichteter azyklischer Graph vorliegt. Bestehende Algorithmen zur Zyklenerkennung lassen sich unmittelbar auf SL Diagramme anwenden, zielen jedoch üblicherweise auf eine möglichst geringe Anzahl von Zykluskanten ab, ohne dabei eine eventuell vorhandene Semantik im Graphen zu berücksichtigen. Die beiden Ansätze zur Behandlung von Zyklen, die Umkehr oder das Entfernen von Kanten, lassen sich ebenfalls prinzipiell auf SL Diagramme anwenden, jedoch wird bei diesen Ansätzen das Kantenrouting in Phase 5 bei PORT_SIDE oder stärkeren Port Constraints erschwert.

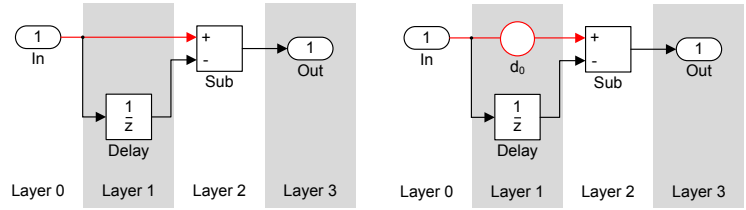
PHASE 2: HIERARCHISIERUNG

Die Knoten des Graphen werden derart auf sogenannte *Layer* verteilt, dass alle Kanten von *Layern* mit niedrigem Index zu *Layern* mit höheren Index zeigen. Diese *Layer* werden später in aufsteigender Reihenfolge entlang der Hauptrichtung des Graphen angeordnet (siehe Abbildung 2.10).

Definition 12. *Hierarchisierung: Die Abbildung der Knoten* $v \in V$ *eines Graphen* $G = (V, E, P)$ *auf einen oder mehrere* Layer $L_{\{0,1,\ldots,k-1\}} \in L$ *mit* $k \in \mathbb{N}$, *wobei* $l(v) = n$ *wenn* $v \in L_n$ *den Index des* Layers *von* v *bezeichnet. Diese Abbildung erfolgt derart, dass für alle Kanten* $e = (u, v) \in E$ *gilt:* $l(u) < l(v)$.

Typische Optimierungsziele bei der Hierarchisierung sind: Minimierung der Anzahl von *Layern*, Minimierung der Kantenlängen, Einhaltung eines vorgegebenen Höhen-/Breitenverhältnisses oder das Erreichen einer ausgeglichenen Anzahl an Blöcken pro *Layer*.

Um eine einwandfreie Hierarchisierung (*proper hierarchy*) herzustellen und dadurch die späteren Phasen des Algorithmus zu vereinfachen, werden Kanten, die mehrere *Layer* überspannen, durch das Einfügen von *Dummy* Knoten aufgespalten (siehe [85]). Diese *Dummy* Knoten werden in Phase 5 wieder entfernt. Abbildung 2.11 zeigt das Aufspalten langer Kanten durch *Dummy* Knoten.



(a) Eine lange Kante vor Einfügen des *Dummy* Knotens.

(b) Darstellung mit *Dummy* Knoten d_0 in *Layer* 1.

Abbildung 2.11: Aufspalten langer Kanten durch *Dummy* Knoten.

Definition 13. *Einwandfreie Hierarchisierung: Eine einwandfreie Hierarchisierung liegt vor, wenn für alle Kanten* $e = (u, v) \in E$ *eines Graphen* $G = (V, E, P)$ *gilt, dass* $l(u) = l(v) + 1$. *Eine einwandfreie Hierarchisierung wird hergestellt, indem so lange für jede lange Kante* $e = (u, v)$ *mit* $l(u) < l(v) + 1$ *ein neuer Knoten* d *in* V *und eine neue Kante* $f = (d, v)$ *in* E *eingefügt werden und die lange Kante mit diesem neuen Knoten an Stelle ihres alten Ziels verbunden wird* $(e = (u, d))$*, bis keine lange Kante mehr vorkommt. Die eingefügten Knoten werden als* Dummy *Knoten* $D \subset V$ *bezeichnet.*

Ein Ansatz zur Minimierung der Kantenlängen bei der Hierarchisierung wird von Gansner *et al.* [42] vorgestellt. Dabei wird das Problem als lineares (integer) Programm formuliert und mit Hilfe gängiger Verfahren optimiert. Die optimale Lösung des linearen Programms entspricht einer Hierarchisierung mit minimalen Kantenlängen. Das verwendete lineare Programm ist in Gleichung 2.1 dargestellt.

$$\text{Minimize:} \sum_{e=(u,v)\in E} (l(u) - l(v) - 1) \tag{2.1}$$

Hierbei muss $l(v) \geqslant 1$ für alle $v \in V$, sowie $l(u) - l(v) \geqslant 1$ für alle $(u, v) \in E$ gelten. Nach Lösung des linearen Programms ordnet man alle Knoten dem *Layer* mit dem durch ihr jeweiliges l gegebenen Index zu.

Bestehende Ansätze zur Hierarchisierung lassen sich prinzipiell direkt auf SL Diagramme anwenden, wobei zusätzliche Constraints aus Modellierungsrichtlinien (siehe Abschnitt 2.1.4) oder Benutzervorgaben von diesen Ansätzen bisher nicht explizit berücksichtigt wurden.

PHASE 3: KREUZUNGSREDUKTION

In dieser Phase werden die Knoten innerhalb ihres *Layers* L_n so sortiert, dass die Kantenkreuzungen (möglichst) minimal sind. Ein eindeutiger Index $i(v) \in \{0, 1, \ldots, |L_n| - 1\}$, so dass $i(u) \neq i(v) \Leftrightarrow u \neq v \; \forall \; u, v \in L_k$, repräsentiert dabei die Ordnung der Knoten innerhalb eines *Layers*. Dieses Problem ist bereits für nur zwei *Layer* NP-vollständig (siehe [22, 43]). Daher kommen zu seiner Lösung verschiedene Heuristiken zur Anwendung:

Zunächst wird das Problem für Graphen mit mehr als zwei *Layern* in mehrere Kreuzungsreduktionen für *Layer* Paare (*two-layer crossing reduction*) zerlegt und diese iterativ ausgeführt (*layer-by-layer sweeping*). Dabei kommen unterschiedliche Vorgehensweisen für die Iteration zum Einsatz ([51, 91]). Beispielsweise wird für n *Layer* zunächst $i = 0 \ldots n-1$ iteriert und dabei für alle Paare (L_i, L_{i+1}) die Reihenfolge der Knoten in L_i konstant gehalten, während die L_{i+1} Knoten geordnet werden und danach in umgekehrter Reihenfolge $i = n \ldots 1$ iteriert, wobei das *Layer* Paar (L_{i-1}, L_i) mit konstanter Reihenfolge in L_i und variabler Reihenfolge in L_{i-1} bearbeitet wird.

Zentral bei allen Varianten der Iteration bleibt das NP-vollständige Problem der Kreuzungsreduktion für *Layer* Paare (siehe oben), so dass auch hier Heuristiken zum Einsatz kommen müssen. In [51] werden hierfür bekannte Heuristiken im Hinblick auf Laufzeit und Effizienz miteinander verglichen.

In [84] wird von Spönemann *et al.* korrekterweise festgestellt, dass bisherige Algorithmen zur Kreuzungsreduktion für Graphen mit den Port Constraints PORT_ORDERED oder PORT_FIXED (Vgl. Seite 24) nicht ohne Weiteres anwendbar sind. Daher schlagen die Autoren eine Erweiterung der Barycenter-Heuristik zur Kreuzungsreduktion mit diesen Port Constraints vor, die sich auch zur Kreuzungsreduktion bei SL Diagrammen mit Port Constraints vom Typ PORT_CALCULATED eignet.

PHASE 4: KNOTENPOSITIONIERUNG

Basierend auf den Reihenfolgen innerhalb der *Layer* werden den Knoten konkrete Y-Positionen $y(v)$ zugeordnet. Es sind verschiedene Ansätze bekannt, die bei der Zuordnung der Koordinaten üblicherweise zum Ziel

haben, Kantenknicke zu minimieren [6, 8, 85]. Die Ansätze von Gansner *et al.* [42] und Sander [77] unterstützen bereits feste Kontaktpunkte zwischen Kanten und Knoten, ähnlich den Port Constraints vom Typ PORT_FIXED. Keiner der bisher dokumentierten Ansätze zur Kantenbegradigung berücksichtigt jedoch veränderliche Knotengrößen und Port Constraints vom Typ PORT_CALCULATED.

Da der Ansatz von Gansner *et al.* aus [42] die Grundlage des in Abschnitt 3.3 vorgestellten Algorithmus zur Kantenbegradigung bei Graphen mit Port Constraints vom Typ PORT_CALCULATED ist, wird er nachfolgend ausführlicher vorgestellt:

Gansner *et al.* betrachtet die vertikalen Koordinaten als Variablen einer linearen Kostenfunktion, in der die Summe der Größen aller Kantenknicke ausgedrückt wird. Diese Kostenfunktion wird mit Hilfe von Methoden der linearen Optimierung minimiert, was einer Minimierung der Kantenknicke entspricht. Constraints bezüglich der Abstände zwischen Knoten stellen sicher, dass die Knoten sich nicht überlappen. Weitere Constraints ergeben sich beispielsweise aus Beschränkungen der Größe des Layouts. Konkrete Lösungen für das lineare Programm können mit gängigen Verfahren der linearen Optimierung errechnet werden (siehe Abschnitt 2.4) und ergeben unmittelbar die gesuchten vertikalen Koordinaten aller Knoten.

Gleichung 2.2 stellt die resultierende Kostenfunktion mit den in Abschnitt 2.1.1 definierten Symbolen für alle Kanten $e = (u, v) \in E$ dar. Zusätzlich beschreibt die Konstante $\Omega(e)$ eine Gewichtung zur bevorzugten Begradigung langer Kanten in Anlehnung an [42]: $\Omega(e) = 1$ für Kanten zwischen normalen Knoten, $\Omega(e) = 2$ für Kanten, die mit einem normalen und einem *Dummy* Knoten verbunden sind, $\Omega(e) = 8$ für Kanten zwischen zwei *Dummy* Knoten. Die Konstante $\varphi(e)$ enthält die vertikale Verschiebung der Ports, mit denen e verbunden ist, relativ zu $y(u)$ und $y(v)$. $\omega(e) \in \mathbb{R}^+$ stellt eine optionale semantische Gewichtung von Kanten dar und ist ebenfalls konstant.

$$\text{Min:} \sum_{e \in E} \Omega(e)\omega(e) \mid y(v) - y(u) + \varphi(e) \mid \tag{2.2}$$

Constraints der Form $y(a) - y(b) \geqslant \delta(a, b) + s_y(v)$ stellen für benachbarte Knotenpaare $a, b \in V$, $L(a) = L(b)$ und $i(a) = i(b) + 1$, die Einhaltung von Mindestabständen $\delta(a, b)$ sicher, wobei die Knotengrößen $s_y(v)$ hier als konstant angesehen werden.

PHASE 5: KANTENROUTING

Zwischen den fertig angeordneten *Layern* wird in dieser Phase das Kantenrouting festgelegt und dabei der hierfür notwendige Platz zwischen

den *Layern* bestimmt. *Dummy* Knoten werden in Stützpunkte der jeweiligen Kanten umgewandelt.

Spönemann *et al.* beschreiben basierend auf [78] einen Ansatz zum Routing von Hyperkanten in hierarchischen Graphen, der in der dokumentierten Fassung für SL Diagramme bereits geeignet ist [84]. Er weist jedoch bei Zykluskanten im Nahbereich der Anfangs- und Endknoten zum Teil noch unnötige Kantenknicke auf. Schulze greift in seiner Arbeit [81] bereits einige der in dieser Arbeit vorgestellten Ansätze zum Kantenrouting mit Hilfe zusätzlicher *Dummy* Knoten (siehe Abschnitt 3) auf und entwickelt diese weiter.

2.2.4 VERFAHREN FÜR PLANARE GRAPHEN

Auf Planarisierung basierende Layoutverfahren wie Topology-Shape-Metrics [5, 86, 87], Topology-Shape-Metrics mit Port Constraints [25] oder Upward-Planarization [9, 45] kommen ohne explizite Hierarchisierung aus und können im Vergleich zu hierarchischen Layoutverfahren bis zu 50% der Linienkreuzungen einsparen [10]. Die für Diagramme mit vorgegebener Ausrichtung prinzipiell geeignete Upward-Planarization konnte jüngst zumindest theoretisch um Hyperkanten und Port Constraints erweitert werden [11], wobei die Implementierung und der praktische Vergleich zu hierarchischen Algorithmen mit Hyperkanten und Port Constraints wie z.B. [84] noch aussteht. Die für SL Diagramme relevanten Port Constraints von Typ PORT_CALCULATED sowie die damit zusammenhängende Möglichkeit, Knoten in der Größe zu variieren, findet keine Beachtung.

2.2.5 KOMMERZIELLE WERKZEUGE

Die kommerzielle Layout Bibliothek yFiles (yWorks GmbH) unterstützt zwar feste Kontaktpunkte zwischen Knoten und Kanten auch bei Blockdiagrammen, jedoch wird die Veränderung von Knotengrößen, speziell im Zusammenhang mit Port Constraints vom Typ PORT_CALCULATED wie bei SL Diagrammen, nicht unterstützt. Details der entsprechenden Algorithmen sind nicht veröffentlicht. Gleiches gilt für ILOG JViews [80] und Tom Sawyer Visualization [89], deren Algorithmen ebenfalls verschiedene Port-Constraints unterstützen, jedoch nicht veröffentlicht sind.

2.3 KIEL INTEGRATED ENVIRONMENT FOR LAYOUT – ECLIPSE RICH CLIENT

Das Forschungsprojekt KIELER hat zum Ziel, die (grafische) modellbasierte Entwicklung komplexer Systeme zu verbessern, indem es diese nicht nur aus Sicht der Semantik und Syntax ihrer Modelle betrachtet, sondern zusätzlich unter dem Aspekt der *Pragmatik* [35]. Diese beschreibt die praktischen Anteile der modellbasierten Entwicklung: Das Betrachten, Bearbeiten und Simulieren von Modellen. Dabei setzt KIELER insbesondere auf die konsequente Einbindung automatischer Layoutverfahren, um Konzepte wie *Focus and Context* in KIELER View Management [34] (KIVi) zur Visualisierung während der Simulation oder *Structure-Based Editing* in KIELER Structure Based Editing [65] (KSBasE) zur Unterstützung beim Bearbeiten von Modellen umzusetzen.

KIELER ist eine Weiterentwicklung von Kiel Integrated Environment for Layout [75] (KIEL), das sich seinerzeit ausschließlich auf Statecharts beschränkte. Es integriert sich in die *Eclipse* Entwicklungsumgebung [24]. KIELER ist modular aufgebaut und an die *Eclipse* Module zur graphischen Modellierung angebunden: Eclipse Modeling Framework [50] (EMF), Graphical Modeling Framework [48] (GMF) und Graphical Editing Framework [49] (GEF). Die verschiedenen Teilprojekte sind jeweils als *Eclipse* Plugins realisiert. Die in KIELER enthaltenen Implementierungen stehen unter der Eclipse Public License [31] (EPL).

2.3.1 KIELER LAYOUTERS FOR LAYERED GRAPHS

KIELER Layouters for Layered Graphs [37] (KLay Layered) behandelt die Entwicklung von hierarchischen Layoutalgorithmen zur Verwendung in KIELER. Das Modul hat seinen Ursprung in einem regelmäßigen Austausch der KIELER Forschungsgruppe mit dem Autor dieser Arbeit, aus der insbesondere die KLay Layered Datenstruktur selbst entstand und in deren Rahmen weitere Entwicklungen angeregt werden konnten (beispielsweise [58, 81]).

KLay Layered bildet den in Abschnitt 2.2.3 beschriebenen Algorithmus modular ab: Jede der fünf Phasen ist als eigenes, austauschbares Modul umgesetzt, so dass je nach Konfiguration unterschiedliche Algorithmen oder Implementierungen zum Einsatz kommen können. Grundlage der in KLay Layered umgesetzten Algorithmen war zunächst die Arbeit von Spönemann *et al.* [84], die später um weitere Implementierungen ergänzt wurden. Zwischen den Phasen können bei KLay Layered zusätzliche Berechnungsschritte (*Intermediate Processors*) eingebunden werden, die eine flexible Erweiterung der Algorithmen aller Phasen ermöglichen. Diese

Erweiterung des hierarchischen Layoutalgorithmus geht insbesondere auf die Arbeit von Schulze [58, 81] zurück.

In den Datenstrukturen von KLay Layered wird ein Graph als *LayeredGraph* formuliert, der sich aus geordneten *Layern* (*Layer*) zusammensetzt. Diesen sind die pro *Layer* geordneten Knoten (*LNode*) des Graphen zugeordnet. Die Knoten enthalten geordnete Ports (*LPort*) mit optional festgelegter Seite (*PortSide*), die über gerichtete Kanten (*LEdge*) miteinander verbunden sind.

Die in Abschnitt 5 vorgestellte Implementierung der Algorithmen zum Layout von SL Diagrammen aus Abschnitt 3 nutzt ebenfalls die KLay Layered Datenstruktur und bindet für einige Phasen (Kantenrouting, Zyklenerkennung) direkt Implementierungen aus KLay Layered ein.

2.3.2 STRUCTURE-BASED EDITING

Das *Structure-Based Editing* ist eine Funktion zur Modellierungsunterstützung, die mit der automatischen Vervollständigung von Schlüsselworten aus der textuellen Programmierung vergleichbar ist. Es wurde von Prochnow auf Basis von KIEL für Statecharts vorgestellt [74] und von Matzen als KSBasE auf KIELER übertragen [64].

Das Ziel beim *Structure-Based Editing* ist es, die bei der Bearbeitung notwendigen manuellen Arbeitsschritte zu reduzieren. Anstatt per *Drag and Drop* einzelne Zustände in ein Zusandsdiagramm einzufügen und zu platzieren, werden dem Benutzer komplexere Arbeitsschritte wie *Folgezustand einfügen* angeboten und mit Hilfe von Modelltransformationen und automatischen Layoutverfahren umgesetzt. Mittels vergleichender Experimente wurde gezeigt, dass sich durch derartige Funktionen eine deutliche Zeitersparnis von bis zu 40% beim Modellieren ergeben kann [36, 64, 74].

Für SL Diagramme erscheint eine direkte Übertragung des *Structure-Based Editing* in der ursprünglichen Form als nicht sinnvoll. Dem Benutzer müssten allein für das Einfügen neuer Blöcke wegen der durch Blocktypen und Ports bedingten vielen möglichen Variationen bereits eine große Anzahl an Transformationen angeboten werden, so dass im Vergleich zum manuellen *Drag and Drop* Verfahren keine Verbesserung bei der Benutzbarkeit zu erwarten wäre. Stattdessen erfolgt die Übertragung auf SL Diagramme als *kontextbasiertes Modellieren* mit Hilfe einer Heuristik zur Auswahl geeigneter Transformationen. Diese Heuristik wurde im Rahmen dieser Arbeit entwickelt und wird in Abschnitt 4.1 vorgestellt.

2.4 LINEARE OPTIMIERUNG

Lineare Optimierung, auch als lineare Programmierung bezeichnet, betrachtet die Optimierung einer linearen Zielfunktion unter Berücksichtigung von Einschränkungen (Constraints) durch lineare Gleichungen oder Ungleichungen. Die lineare Optimierung, eingeführt von Kantorowitsch [53], wurde 1947 mit Veröffentlichung des Simplex-Verfahrens zur Lösung linearer Programme durch Dantzig [13] praktisch anwendbar gemacht. Das Simplex-Verfahren ist bis heute das meistgenutzte Lösungsverfahren für lineare Programme, da es zwar im schlechtesten Fall exponentiellen Aufwand hat, in der Praxis jedoch zumeist das schnellste Lösungsverfahren ist [40, 59]. Weitere Lösungsverfahren sind die Ellipsoidmethode sowie verschiedene Innere-Punkte-Verfahren, mit denen lineare Programme theoretisch in Polynomialzeit gelöst werden können.

Als lineares Programm bezeichnet man die Kombination aus Zielfunktion und Constraints eines gegebenen (linearen) Problems. Die Standardform eines solchen linearen Programms ist in Gleichung 2.3 dargestellt. Gegeben sind eine Matrix $A \in \mathbb{R}^{m,n}$ sowie zwei Vektoren $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$; gesucht ist eine zulässige Lösung des Lösungsvektors $x \in \mathbb{R}^n$. $c^T x = c_1 x_1 + \ldots + c_n x_n$ bezeichnet dabei das Skalarprodukt zwischen c (transponiert) und x. Dieses ist dann durch geeignete Auswahl von x zu maximieren, wobei die Elemente von x nicht negativ sein dürfen und die durch die Komponenten von $Ax \leqslant b$ gegebenen Constraints $a_i x \leqslant b_i$ eingehalten werden müssen.

$$\text{Max: } c^T x \mid Ax \leqslant b, x \geqslant 0 \tag{2.3}$$

Die Anwendung von Optimierungsverfahren auf reale Probleme macht es häufig erforderlich, dass die Lösung der Optimierung ganzzahlig ist (beispielsweise soll ein optimierter Produktionsprozess ganze Werkstücke aus ganzen Bauteilen herstellen). Die (gemischt) ganzzahlige lineare Optimierung fordert daher zusätzlich für Teile des gesuchten Lösungsvektors, dass diese ganzzahlig sind: $x_i \in \mathbb{Z}$ für einige $i \in \{1, \ldots, n\}$. Die in der Praxis relevanten Lösungsverfahren für ganzzahlige lineare Programme basieren auf einer iterativen Lösung ähnlicher nicht ganzzahliger linearer Programme. Ihre Lösung ist aus komplexitätstheoretischer Sicht NP-schwer[5]. Wie schnell sich ganzzahlige Programme in der Praxis tatsächlich lösen lassen, hängt jedoch wesentlich stärker von der Struktur des jeweiligen Problems und seiner Formulierung ab, als von der reinen Anzahl an Variablen und Constraints im Programm.

Lineare Programme können je nach ihren Eigenschaften normalerweise genau eine, keine, oder unendlich viele optimale Lösungen haben.

5 *NP-hard* wird auch als NP-hart übersetzt. Der Autor zieht jedoch die Übersetzung NP-schwer vor (siehe beispielsweise [92]).

Ganzzahlige lineare Programme (auch gemischt ganzzahlige lineare Programme) können zudem auch eine endliche Anzahl optimaler Lösungen haben.

2.4.1 UMFORMUNGEN LINEARER PROGRAMME

Bei der Formulierung linearer Programme helfen einige grundlegende Umformungen (siehe beispielsweise [72]), die zum Teil von aktuellen Solvern automatisch angewandt werden. Im weiteren Verlauf dieser Arbeit, insbesondere in Abschnitt 3, werden verschiedene Umformungen an linearen Programmen vorgenommen. Diese werden daher hier vorgestellt:

GLEICHHEITS- UND GRÖSSER-GLEICH-CONSTRAINTS

Zur Darstellung von Gleichheits-Constraints wird $a_i x = b_i$ durch zwei Constraints der Form $a_i x \leqslant b_i$ und $a_i x \geqslant b_i$ ersetzt, so dass diese zusammen genommen die Gleichheit $a_i x = b_i$ sicherstellen. Größer-Gleich-Constraint werden durch Multiplikation der entsprechenden Zeile des Constraints in A und b mit -1 ausgedrückt.

ÄNDERUNG DER OPTIMIERUNGSRICHTUNG

Soll die Zielfunktion nicht maximiert, sondern minimiert werden, multipliziert man c mit -1.

NEGATIVE LÖSUNGEN

Soll x auch negative Werte annehmen können, ersetzt man x durch $x' - x''$, wobei $x', x'' \geqslant 0$ sind.

DARSTELLUNG VON BETRÄGEN

Da die Betragsfunktion $f(x) = |x|$ keine lineare Funktion ist, kann sie nicht unmittelbar als Zielfunktion eines linearen Programms verwendet werden. Sie lässt sich jedoch in zwei lineare Segmente aufteilen, nämlich in $|x| = x$ für $x \geqslant 0$ und $|x| = -x$ für $x < 0$. Mit den passenden Constraints lassen sich diese dann ersatzweise in die Zielfunktion einsetzen:

$$\text{Max: } \sum_{i=1}^{n} c_i |x_i| \Leftrightarrow \text{Max: } \sum_{i=1}^{n} c_i x_i^+ - c_i x_i^- \tag{2.4}$$

$x_i^+, x_i^- >= 0$ sind die Hilfsvariablen, mit deren Hilfe der Wertebereich von x über die Constraints $x - x_i^+ + x_i^- = 0$ und $x_i^+ - x_i^- = 0$ aufgeteilt wird. Diese Umformung ist dann gültig, wenn die Koeffizienten c_i nicht positiv sind [47, 82].

SKALIERUNG GANZZAHLIGER VARIABLEN

Bei ganzzahligen linearen Programmen wird eine ganzzahlige Lösung gefordert: $x_i \in \mathbb{Z}$. Soll die Lösungsmenge zusätzlich auf eine Schrittweite $\lambda_i \in \mathbb{N}$ eingeschränkt werden ($x_i \in \{\lambda_i \cdot k \mid k \in \mathbb{Z}\}$), ist es erforderlich diesen Faktor als Skalierung in das lineare Programm zu integrieren. Dazu wird im gesamten linearen Programm $x_i = \lambda_i \cdot x_i^s$ mit $x_i^s \in \mathbb{Z}$ ersetzt:

$$\text{Max: } \sum_{i=1}^{n} c_i x_i \mid x_i \in \{\lambda_i \cdot k \mid k \in \mathbb{Z}\} \Leftrightarrow \text{Max: } \sum_{i=1}^{n} c_i \lambda_i x_i^s \tag{2.5}$$

Die Constraints sind analog dazu anzupassen. Aus der Lösung des linearen Programms für x^s lässt sich dann durch Einsetzen die eigentlich gesuchte Lösung für x errechnen.

2.4.2 WERKZEUGE ZUM LÖSEN LINEARER PROGRAMME

In diesem Abschnitt werden die im Rahmen dieser Arbeit verwendeten Werkzeuge zur Lösung gemischt ganzzahliger linearer Programme (LP-Solver) vorgestellt. Die Auswahl basierte insbesondere auf den von Mittelmann durchgeführten Benchmarks [67] mit verschiedenen Werkzeugen. Die Werkzeuge sollten darüber hinaus in einer kostenlosen akademischen Lizenz verfügbar sein und mit vertretbarem Aufwand in die prototypische Implementierung der Layoutalgorithmen (siehe Abschnitt 5) eingebunden werden können. Wenigstens eines der Werkzeuge sollte für die Praxiserprobung (siehe Abschnitt 6.1) auch im kommerziellen Umfeld ohne Lizenzkosten einsetzbar sein.

GUROBI OPTIMIZER

Der *Gurobi Optimizer* [44] ist ein kommerzielles Werkzeug zum Lösen von gemischt ganzzahligen linearen und quadratischen Programmen. Er unterstützt verschiedene Lösungsverfahren sowohl zum Lösen der eigentlichen linearen Programme (primal simplex, dual simplex, parallel barrier) als auch zur Behandlung ganzzahliger Variablen (branch-and-bound, cutting planes und zusätzliche Lösungsheuristiken). Nach Angaben des Herstellers sowie der Benchmarks von Mittelmann ist der *Gurobi*

Optimizer das schnellste verfügbare Werkzeug zum Lösen gemischt ganzzahliger linearer Programme. Das Werkzeug ist für akademische Zwecke unter einer speziellen Lizenz kostenlos verfügbar.

SCIP

Solving Contraint Integer Programming [2] (SCIP) wurde von Achterberg ursprünglich im Rahmen seiner Promotion [1] erstellt und am Konrad-Zuse-Zentrum für Informationstechnik Berlin weiterentwickelt. Es verwendet SoPlex [95], eine Implementierung des Simplex Algorithmus, zur Lösung linearer Programme und branch-cut-and-price zur Behandlung ganzzahliger Variablen. SCIP hat die Benchmarks von Mittelmann als schnellstes nicht kommerzielles Werkzeug abgeschlossen. Es ist für die akademische Verwendung frei verfügbar und wird vom Konrad-Zuse-Zentrum als *nicht kommerzieller LP Solver* vertrieben. Für die kommerzielle Verwendung ist eine kostenpflichtige Lizenz erforderlich.

LP_SOLVE

lp_solve [27] ist ein frei verfügbares Werkzeug zum Lösen gemischt ganzzahliger linearer Programme. Es basiert auf dem revised simplex Algorithmus [14] und verwendet branch-and-bound zur Behandlung von ganzzahligen Variablen. lp_solve steht unter der Open Source Lizenz GNU Lesser General Public License [33] (LGPL). Diese erlaubt nach eigener Einschätzung die kostenlose Verwendung von lp_solve als Bibliothek auch im Zusammenhang mit Software, die selbst nicht unter einer Open Source Lizenz steht, solange die Bibliothek selbst gemäß den Richtlinien der Lizenz gehandhabt wird.

Teil II

VISUALISIEREN UND BEARBEITEN

3 LAYOUT VON SIMULINK DIAGRAMMEN

Im vorherigen Kapitel wurden die grafischen Eigenschaften von SL Diagrammen beschrieben und Beispiele für layoutrelevante Modellierungsrichtlinien gegeben. Es wurden dort verschiedene Layoutverfahren betrachtet und bezüglich ihrer Eignung zum Layout von SL Diagrammen untersucht. Dabei wurde festgestellt, dass diese die besonderen grafischen Eigenschaften von SL Diagrammen nicht oder nur unzureichend unterstützen. Insbesondere finden Knoten variabler Größe mit Port Constraints vom Typ PORT_SIMULINK bisher keine Beachtung. Daher wurde der in Abschnitt 2.2.3 vorgestellte hierarchische Layoutalgorithmus umfassend erweitert. Die modulare Struktur des Algorithmus ermöglichte es, nur diejenigen Phasen anzupassen, die bisher nicht oder nur unzureichend für SL Diagramme geeignet waren:

Eine Erweiterung der Zyklenerkennung ermöglicht es Benutzern, mit Hilfe weniger gewohnter Arbeitsschritte, Informationen über die semantische Bedeutung bestimmter Knoten an den Layoutalgorithmus zu übergeben, um damit die Zyklenerkennung zu verbessern. Die Zyklenbehandlung mit *Inverter* Knoten ersetzt bisherige Vorgehensweisen zum Routing von Kanten in unmittelbarer Nähe der Quell- und Zielknoten und hilft dabei, die dort entstehenden Kantenknicke zu vermeiden.

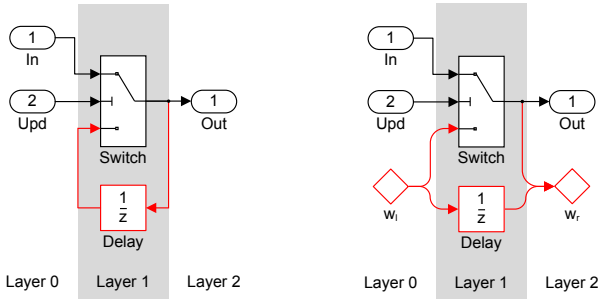
Die in Abschnitt 2.2.3 vorgestellte Hierarchisierung mittels linearer Programmierung wurde um zusätzliche Constraints zur Beachtung von Modellierungsrichtlinien bezüglich der Anordnung bestimmter Knoten ergänzt. Benutzerdefinierte Constraints helfen dabei, die durch die frühe Hierarchisierung entstehenden Probleme in der Kreuzungsreduktion und Kantenbegradigung in hierarchischen Layoutverfahren mit vertretbarem Aufwand zu vermeiden.

Den Fokus dieses Kapitels bildet die Kantenbegradigung für SL Diagramme. Ausgehend von der Arbeit von Gansner *et al.* [42] wird schrittweise ein Algorithmus zur Kantenbegradigung mit verschiedenen Port Constraints und für Knoten variabler Größe entwickelt. Für diesen Algorithmus werden verschiedene Ansätze zur Begradigung von Hyperkanten sowie ein Verfahren zur Vermeidung von Kantenknicken an oben oder unten liegenden Ports beschrieben. Abschließend wird der Algorithmus mit Hilfe einer Linearisierung der SL Portpositionierung und einer spezifischen Skalierung von Variablen auf SL Diagramme übertragen.

3.1 ZYKLENERKENNUNG- UND BEHANDLUNG

Bestehende Methoden zur Erkennung von Zyklen lassen sich bereits unmittelbar auf SL Diagramme anwenden (siehe Abschnitt 2.2.3). Auf Grund der semantischen Bedeutung einiger Zyklen in SL Diagrammen (beispielsweise bei Rückkopplungen) ist jedoch nicht jede Kante gleichermaßen als Zykluskante geeignet. Stattdessen sind die Zykluskanten in solchen Fällen durch Modellierungsrichtlinien (siehe Abschnitt 2.1.4) oder gängige Praxis vorgegeben. Dabei werden üblicherweise einzelne Knoten *flipped*, also links-rechts gespiegelt, dargestellt (siehe Abschnitt 2.1), so dass sie als Teil des Zyklus erscheinen (siehe Abbildung 3.1a).

Diese Information wird verwendet, um die vom Benutzer über *flipped* Knoten markierten Zykluskanten bereits vor der eigentlichen Zyklenerkennung festzuhalten. Falls erforderlich können weitere Zykluskanten durch bestehende Algorithmen erkannt werden; im Rahmen dieser Arbeit wurde hierfür die *GreedyCycleBreaker* Implementierung des Algorithmus von Eades *et al.* [23] aus KLay Layered verwendet. So kann der Benutzer die Zyklenerkennung direkt beeinflussen, ohne dabei von seiner gewohnten Arbeitsweise abweichen zu müssen.



(a) Ein *flipped* Knoten vor Einfügen der *Inverter* Knoten.

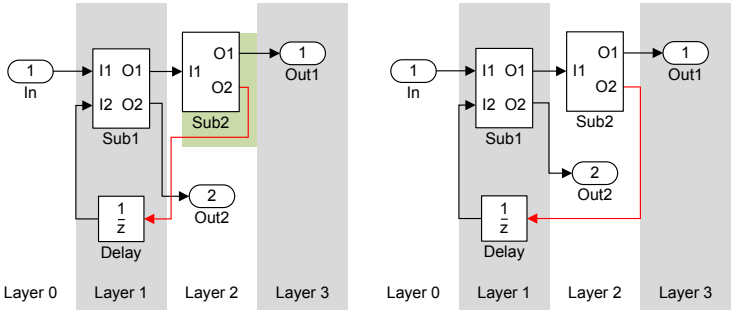
(b) Der *flipped* Knoten nach Einfügen der *Inverter* w_l und w_r.

Abbildung 3.1: Einfügen von *Inverter* Knoten in die Kanten eines als *flipped* markierten Knotens.

3.1.1 ZYKLENBEHANDLUNG MIT INVERTER KNOTEN

Bei der Behandlung von Zyklen machen es die Port Constraints in SL Diagrammen bei bisherigen Ansätzen (siehe Abschnitt 2.2.3) notwendig,

bei der späteren Knotenpositionierung in der Nähe des Start- und Endknotens zusätzlichen Platz vorzuhalten, in welchem die Zykluskante um den Knoten herum geführt werden kann. Es entsteht ein Routing der Form: Nahbereich Startknoten → Weg zum Zielknoten → Nahbereich Zielknoten. Hierbei kann es an der Schnittstelle zwischen den drei Sektionen zu zusätzlichen Kantenknicken kommen. Abbildung 3.2a zeigt das im Bereich um *Sub2* entstehende Routing der Kante *Sub2* → *Delay*.



(a) Kantenrouting im reservierten Nahbereich um *Sub2* erzeugt zusätzliche Kantenknicke und Kreuzungen.

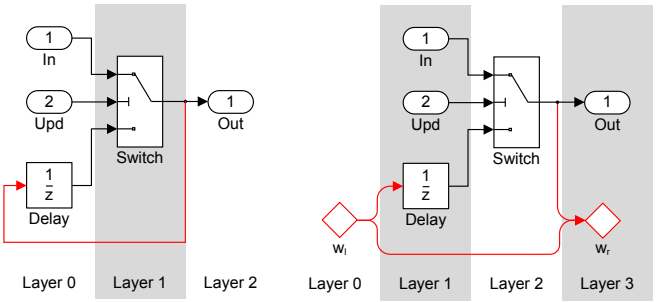
(b) Ideales Routing erzeugt weniger Kantenknicke und keine Kreuzung.

Abbildung 3.2: Routing von Zykluskanten mit und ohne explizit reservierten Platz im Nahbereich von *Sub2*.

Diese Knicke können durch einen neuen Ansatz zur Kantenumkehr in vielen Fällen vermieden werden: Zykluskanten werden durch das Einfügen zusätzlicher virtueller Knoten, sogenannter *Inverter* Knoten, umgekehrt. Sei $e = (u, v)$ eine solche Kante, dann wird e aus G entfernt und stattdessen zusätzliche (markierte) Knoten w_l, w_r und Kanten f_l, f_m, f_r eingefügt, so dass nach der Transformation für die Kanten $f_l = (w_l, v)$, $f_m = (w_l, w_r)$, $f_r = (u, w_r)$ und für die Ports $p_i(f_l) = p_i(e)$ und $p_o(f_r) = p_o(e)$ gilt.

Definition 14. *Die Menge aller* Inverter *Knoten wird als* $I \subset V$*, die Menge aller Knoten die weder* Inverter *noch* Dummy *Knoten sind als* $V_R = V \setminus (D \cup I)$ *und die Menge aller Kanten, die nicht mit* Inverter *Knoten verbunden sind, als* $E_R = \{e = (u, v) \mid e \in E$ *und* $u, v \notin I\}$ *bezeichnet.*

Abbildung 3.3 zeigt das Ersetzen einer Zykluskante durch die *Inverter* Knoten und Hilfskanten, Abbildung 3.2 stellt das Kantenrouting mit *Inverter* Knoten im Vergleich zum herkömmlichen Routing im Nahbereich dar.



(a) Eine Zykluskante vor Einfügen der *Inverter* Knoten.

(b) Ersatzdarstellung durch die *Inverter* Knoten w_l und w_r.

Abbildung 3.3: Einfügen von *Inverter* Knoten in eine Zykluskante.

Inverter Knoten dienen in den späteren Phasen Kreuzungsreduktion und Knotenpositionierung als Stützpunkte und machen so ein gesondertes Routing im Nahbereich der Knoten unnötig. Sie werden am Ende von Phase 5 des Algorithmus entfernt und in echte Stützpunkte für die Kanten umgewandelt.

3.2 HIERARCHISIERUNG MIT CONSTRAINTS

Bestehende Ansätze zur Hierarchisierung von gerichteten Graphen lassen sich direkt auf SL Diagramme anwenden. Die Hierarchisierung von SL Diagrammen kann jedoch, über die in Abschnitt 2.2.3 genannten Optimierungsziele hinaus, zusätzlichen Anforderungen aus Modellierungsrichtlinien (siehe Abschnitt 2.1.4) unterliegen. Zudem erschwert das frühe Festlegen der Hierarchie bei den hierarchischen Layoutverfahren die Kreuzungsreduktion und Kantenbegradigung.

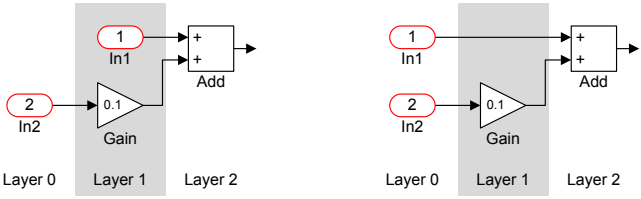
Nachfolgend wird der in Abschnitt 2.2.3 vorgestellte Ansatz zur Minimierung der horizontalen Kantenlängen mittels ganzzahliger linearer Optimierung von Gansner *et al.* [42] um Constraints zur Erfüllung von Modellierungsrichtlinien bezüglich der Anordnung bestimmter Knoten angereichert und um eine Nachbearbeitungsphase zur Behandlung von *Inverter* Knoten erweitert. Zusätzlich wird es dem Benutzer ermöglicht, die eigene Kenntnis über die Struktur des Diagramms, also benutzerdefinierte Constraints, mit in die Hierarchisierung einzubringen. Die so erweiterte Hierarchisierung verbessert in vielen Situationen die Lesbarkeit des resultierenden Layouts.

Zwischen zwei Knoten $(u, v) \in V^2$ kommt dabei jeweils eines der folgenden Constraints, formuliert mit Hilfe der Definitionen aus Abschnitt 2.2.3, zum Einsatz:

1. u und v liegen im gleichen *Layer*: $l(u) = l(v)$,
2. u liegt links von v: $l(u) < l(v)$,
3. u liegt rechts von v: $l(u) > l(v)$.

3.2.1 Anordnung von Inports und Outports

Typische Anforderungen aus Modellierungsrichtlinien bezüglich der Anordnung bestimmter Knoten sind beispielsweise in *db_0141* der MAAB Guidelines (siehe Abschnitt 2.1.4) gegeben. Diese Richtlinie schreibt vor, SL *Inport* Knoten möglichst am linken und *Outport* Knoten möglichst am rechten Rand anzuordnen. Abbildung 3.4 stellt diese Anforderung an einem einfachen Beispiel dar.



(a) Anordnung von *Inport* Knoten ohne zusätzliche Anforderungen.

(b) Anordnung am linken Rand.

Abbildung 3.4: Unterschiedliche horizontale Anordnung von Knoten.

Die Anforderung wurde zunächst verallgemeinert, in elementare Anforderungen aufgeteilt und schließlich in Constraints für die Hierarchisierung der Knoten aus V übertragen:

1. Alle Knoten einer vorgegebenen Menge $U \subseteq V$ sollen im gleichen *Layer* liegen: $l(u) = l(v)\ \forall\ v \in U \setminus u$, mit $u \in U$.
2. Soll die Menge dieser Knoten U links angeordnet werden, müssen die Knoten in einem *Layer* liegen, dessen Index kleiner als der aller anderen Knoten aus V: $l(v) < l(w)\ \forall\ (u, v) \in U \times (V \setminus U)$ ist. Analoges gilt beim Anordnen der Knoten auf der rechten Seite: $l(v) > l(w) \ldots$.

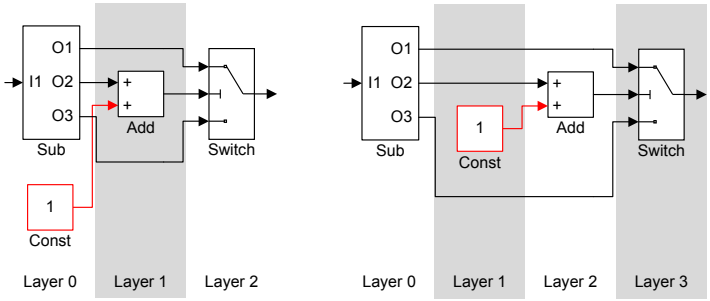
Bei Anwendung beider Constraints lässt sich das zweite Constraint vereinfachen: Da bereits alle Knoten in U durch die Constraints 1 im gleichen

Layer liegen, muss nur noch für einen einzelnen Knoten aus $u \in U$ gefordert werden, dass dieser links anzuordnen ist $l(u) < l(w)\ \forall\ v \in V \setminus U$. Für die anderen Knoten aus U folgt die Anordnung dann aus Constraint 1. Analoges folgt für rechts angeordnete Knoten.

Zur Erfüllung der Anforderungen der oben genannten Richtlinie (links angeordnete *Inport* und rechts angeordnete *Outport* Knoten) sind zwei Gruppen solcher Constraints einzufügen. Beide Gruppen basieren auf dem ersten Constraint: Für die erste Gruppe enthält U alle *Inport* Knoten und im zweiten Constraint wird gefordert, dass $l(u) < l(w)$. Für die zweite Gruppe enthält U die *Outport* Knoten; die zweite Forderung lautet dann auf $l(u) > l(w)$.

3.2.2 ZUSÄTZLICHE LAYER FÜR INVERTER KNOTEN

Durch die frühe Hierarchisierung kann es, insbesondere bei Quellen und Senken wie *Constant*, *Inport* und *Outport* Knoten, auch bei optimaler Kreuzungsreduktion zu zusätzlichen Kantenkreuzungen kommen. Beispielsweise legt die bezüglich der Kantenlänge optimale Hierarchisierung in Abbildung 3.5a die Knoten *Const* und *Sub* in einen *Layer*, so dass eine Kantenkreuzung entsteht, die durch eine Hierarchisierung mit längeren Kanten vermeidbar wäre (Abbildung 3.5b).



(a) *Const* Knoten mit vermeidbarer Kantenkreuzung.

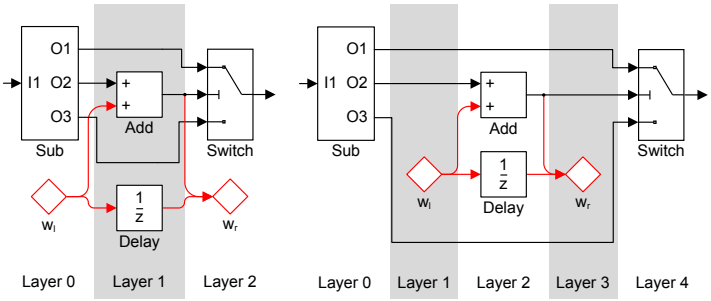
(b) *Const* Knoten mit eigenem *Layer* vermeidet die Kantenkreuzung.

Abbildung 3.5: Einfügen zusätzlicher *Layer* zum Verhindern von Kantenkreuzungen.

Dieser Effekt tritt auch bei den in der Zyklenbehandlung eingeführten *Inverter* Knoten auf. Abbildung 3.6a zeigt eine solche Situation: Bedingt durch die Positionierung der *Inverter* Knoten im gleichen *Layer* wie *Sub* und *Switch* führt die Positionierung des *Delay* Knotens unterhalb der

Kante zunächst zu einer minimalen Anzahl Kreuzungen. Nach dem Entfernen der *Inverter* Knoten verbleiben jedoch unnötige Kantenkreuzungen im Diagramm.

Für *Inverter* Knoten werden daher unmittelbar nach der Hierarchisierung zusätzliche *Layer* eingefügt. Da die so entstehenden *Layer* ausschließlich temporäre Knoten (*Inverter* und *Dummy* Knoten) enthalten, erhalten sie eine Breite von 0 und bleiben im resultierenden Layout unsichtbar. Abbildung 3.6 zeigt dieses Vorgehen für w_l und w_r in Zusammenhang mit dem *Delay* Knoten.



(a) *Inverter* Knoten ohne eigene *Layer*.

(b) *Inverter* Knoten mit eigenen *Layern*.

Abbildung 3.6: Einfügen zusätzlicher *Layer* bei *Inverter* Knoten.

3.2.3 BENUTZERDEFINIERTE CONSTRAINTS

Das pauschale Anlegen zusätzlicher Layer für alle Quellen, Senken, oder Knoten eines bestimmten Typs, wie bei den *Inverter* Knoten, ist jedoch im Allgemeinen nicht zielführend: Die so entstehenden Layer sind wegen der enthaltenen echten Knoten auch dann sichtbar, wenn sie eigentlich gar nicht benötigt werden (siehe Abbildung 3.7).

Vergleichbare Situationen treten bei SL Diagrammen aus der Praxis nicht nur bei einfach Quellen und Senken, sondern insbesondere auch im Bereich von Knoten mit vielen Ports, wie *BusCreator*, *BusSelector* oder *Subsystem* Knoten, auf und betreffen nicht nur einzelne Knoten. Abbildung 3.8 zeigt eine Situation, in der die Knoten *In1*, *In2* und *Out1* eigentlich zwischen zwei Knoten mit vielen Ports (*Select*, *Create*) "gefangen" sind, durch die Hierarchisierung in Abbildung 3.8a jedoch den äußeren *Layern* zugeordnet werden, so dass viele Kantenkreuzungen entstehen. Bei Diagrammen aus der Praxis haben die äußeren Knoten zudem

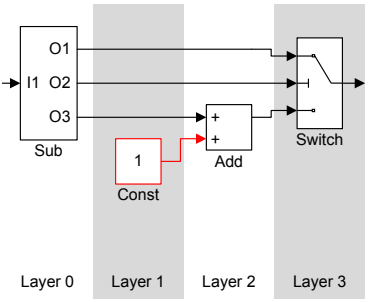
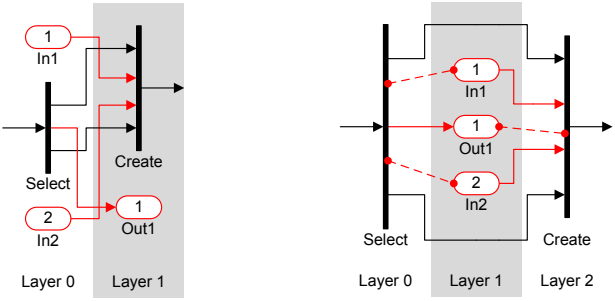


Abbildung 3.7: Unnötig eingefügter Layer.

teilweise mehr als 50 Ports und schließen an Stelle der einzelnen Knoten *In1*, *In2* und *Out1* ganze Gruppen von Knoten ein. Solche Situationen lassen sich bisher nicht einwandfrei automatisch erkennen.

Für Benutzer ist die Erkennung solcher Situationen jedoch in vielen Fällen trivial. Zudem hat sich aus der Praxiserprobung (siehe Abschnitt 6.1) ergeben, dass Benutzer durchaus bereit sind, einem Layoutalgorithmus zusätzliche Informationen, beispielsweise über Situationen, die für die Hierarchisierung problematisch sind, zur Verfügung zu stellen.



(a) Hierarchisierung mit minimaler horizontaler Kantenlänge.

(b) Hierarchisierung mit benutzerdefinierten Constraints.

Abbildung 3.8: Benutzerdefinierte Constraints bei der Hierarchisierung können Kantenkreuzungen vermeiden und zur Verbesserung der Lesbarkeit beitragen.

Daher erhalten sie die Möglichkeit, Informationen zur Hierarchisierung über zusätzliche Constraints in der zu Beginn dieses Abschnitts genannten Form *Knoten u liegt links/rechts/im* Layer *von Knoten v* in das Diagramm einzubringen, so dass problematische Situationen aufgelöst

und semantische Zusammenhänge bei der Hierarchisierung besser berücksichtigt werden können. Abbildung 3.8b zeigt, wie die Situation aus Abbildung 3.8a mit Hilfe benutzerdefinierter Constraints (gestrichelt) aufgelöst und die Kantenkreuzungen zu Gunsten der Lesbarkeit vermieden werden können[1].

3.2.4 AUSWAHL DER CONSTRAINTS

Das ursprüngliche lineare Programm zur Hierarchisierung mit minimalen Kantenlängen von Gansner *et al.* (siehe Abschnitt 2.2.3) ist für gerichtete azyklische Graphen stets lösbar. Bei ungünstiger Wahl zusätzlicher Constraints kann das lineare Programm jedoch leicht unlösbar werden. Würde beispielsweise in Abbildung 3.8b das Constraint *In1 liegt rechts von Create* durch den Benutzer eingefügt werden, würde dies dem durch die Kante zwischen den Knoten gegebenen Constraint widersprechen und das lineare Programm würde unlösbar werden.

Das Berechnen der maximalen widerspruchsfreien Teilmenge der Constraints ist bei der gegebenen Struktur der Constraints NP-schwer: Betrachtet man in Anlehnung an die Problematik der Hierarchisierung den Abhängigkeitsgraphen der Constraints, in dem die Constraints und die Kanten des Diagramms als Kanten und die Knoten des Diagramms als Knoten repräsentiert werden, entspricht die Suche nach der maximalen widerspruchsfreien Teilmenge von Constraints dem NP-schweren Problem des maximalen zyklusfreien Teilgraphen (*feedback arc set problem* [54]), wobei hier nur die Constraints zur Bildung des zyklusfreien Teilgraphen entfernt werden dürfen.

Daher werden die zusätzlichen Constraints schrittweise hinzugefügt, es ergibt sich das folgende Vorgehen:

1. Erzeugen des linearen Programms nach Gansner *et al.*.
2. Für jedes benutzerdefinierte Constraint:
   a) Überprüfe, ob das Constraint im bestehenden Programm einen Widerspruch erzeugen würde.
   b) Wenn ein Widerspruch entsteht wird das Constraint nicht hinzugefügt.
   c) Sonst wird das Constraint hinzugefügt.
3. Lösen des linearen Programms ergibt die Hierarchisierung.

1 Anmerkung: Die Situation wird bereits durch jedes der drei Constraints für sich vollständig aufgelöst, die Darstellung der drei Constraints erfolgt nur zur Verdeutlichung möglicher Optionen.

4. Nachbearbeitung der Hierarchisierung (*Layer* für Inverter Knoten).

Für eine kleine Anzahl benutzerdefinierter Constraints und Kanten kann an Stelle des schrittweisen Vorgehens in Arbeitsschritt 2 auch die maximale Teilmenge widerspruchsfreier Constraints, beispielsweise durch Permutation, ermittelt werden.

3.3 KANTENBEGRADIGUNG MIT PORT CONSTRAINTS

In Abschnitt 2.2.3 wurde der Ansatz zur Kantenbegradigung auf Basis linearer Optimierung von Gansner *et al.* [42] vorgestellt. Dieser berechnet vertikale Positionen für alle Knoten, so dass die gewichtete Summe der Länge aller vertikalen Kantensegmente minimal ist. Eine unterschiedliche Gewichtung der Segmente sorgt dabei insbesondere für eine bevorzugte Begradigung langer Kanten. Bisher werden von diesem Ansatz jedoch lediglich die zwei einfachen Port Constraints PORT_NONE und PORT_FIXED unterstützt (siehe Abschnitt 2.2.1).

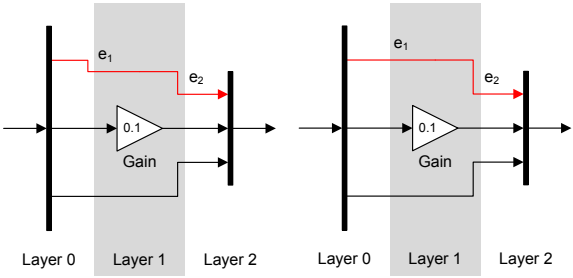
In den folgenden Abschnitten wird das lineare Programm von Gansner *et al.* [42] zunächst schrittweise verfeinert und danach zur Anwendung auf weitere Port Constraints und für Knoten variabler Größe erweitert.

3.3.1 VERFEINERUNGEN DES LINEAREN PROGRAMMS

In diesem Abschnitt wird die ursprüngliche Formulierung des linearen Programms von Gansner *et al.* [42] (siehe Abschnitt 2.2.3) zur Berücksichtigung bisher nicht betrachteter Aspekte erweitert. Dabei werden zusätzliche Komponenten in das Programm eingefügt, die die Diagrammgröße gegenüber der Kantenbegradigung gewichten, die Anzahl von Kantenknicken bei der Begradigung langer Kanten reduzieren und eine Priorisierung zwischen einander kreuzenden Kanten vornehmen.

GEWICHTUNG DER DIAGRAMMGRÖSSE

Gansner *et al.* merkt in [42] zurecht an, dass die durch das darin beschriebene lineare Programm berechneten Knotenpositionen leicht zu (vertikal) sehr großen Diagrammen führen können. Um diesem Effekt entgegenzuwirken, wird hier ein zusätzlicher Term zur Gewichtung der Kompaktheit des Diagramms gegenüber der Begradigung von Linien in



(a) $\Omega(e_1) = \Omega(e_2)$ **(b)** $\Omega_L(e_2) > \Omega_L(e_1)$

Abbildung 3.9: Begradigung langer Kanten mit verschiedenen $\Omega(e)$.

das lineare Programm eingefügt: Dieser gewichtet die Summe der Positionen der unteren Knoten in jedem *Layer* L_n und damit die vertikale Größe des Diagramms gegenüber dem Hauptterm zur Kantenbegradigung über einen Faktor $\kappa \in \mathbb{R} \geqslant 0$. Die Menge der jeweils unteren Knoten aus allen *Layern* $L_n \in L$ wird mit Hilfe von Def. 12 als $V_L = \{w \mid i(w) = |L_n| - 1\}$ bezeichnet, wobei *Inverter* Knoten nicht berücksichtigt werden ($w \notin I$).

BEGRADIGUNG LANGER KANTEN

Beim Begradigen langer Kanten kann es zu Situationen kommen, in denen die ursprüngliche Formulierung aus Gleichung 3.3 nicht ausreicht, um eindeutig entscheiden zu können, wie die einzelnen Knicke der gleich gewichteten Segmente einer langen Kante verteilt sein sollen. In Abbildung 3.9a ist ein solcher Fall dargestellt: Die Abschnitte e_1 und e_2 werden mit gleicher Gewichtung ($\Omega(e_{1,2}) = 2$ für die Enden langer Kanten) begradigt. Es kann daher ohne Benutzervorgabe in $\omega(e)$ nicht eindeutig entschieden werden, welcher der beiden Knicke bevorzugt entfernt werden soll.

Die ursprüngliche Formulierung von $\Omega(e)$ wird daher um einen kleinen Faktor erweitert, um für eine eindeutige Gewichtung zu sorgen: $\Omega_L(e) = \max\left(0, \Omega(e) - \epsilon\frac{l(u)}{k}\right)$, wobei $e = (u, v)$, k die Anzahl an *Layern* und $l(u)$ der Index des *Layers* von u ist. $0 < \epsilon << 1$ sorgt dafür, dass der zusätzliche Faktor lediglich eine leichte Veränderung der Gewichtung bewirkt.

In Abbildung 3.9b ist die Situation durch die Verwendung von $\Omega_L(e_{1,2})$ eindeutig gelöst: e_1 wird gegenüber e_2 bevorzugt begradigt.

GEWICHTUNG VON KREUZENDEN KANTEN

Ebenfalls nicht nicht berücksichtigt wurde bisher, dass zwei kreuzende Kanten nicht gleichzeitig ohne Knicke gezeichnet werden können; wobei Kanten, deren Ein- oder Ausgangsports auf der selbe Position liegen, hier als "nicht kreuzend" betrachtet werden. Um diesen Umstand zu berücksichtigen, setzt ein zusätzlicher Faktor $\chi(e) \in \mathbb{R} \geqslant 0$ die Gewichtung derjenigen Kanten herab, die für Kantenkreuzungen verantwortlich sind. Normale Kanten erhalten $\chi(e) = 1$, Kreuzungskanten $\chi(e) = \chi_k < 1$; in der aktuellen Implementierung des Algorithmus ist $\chi_k = 0$. Dadurch können die anderen Kanten leichter vollständig begradigt und die Anzahl der Kantenknicke reduziert werden.

Die Bestimmung der $\chi(e)$ entspricht dem NP-vollständigen Problem der Berechnung des maximalen, kreuzungsfreien Teilgraphen [21]. In der Praxis ist bereits ein einfacher *Greedy* Algorithmus ausreichend. Dieser berücksichtigt auch die Benutzervorgaben aus $\omega(e)$ zum Auflösen von Konflikten:

1. Für alle *Layer* L_n ...
2. Sei $E_n \subseteq E_R$ die Menge derjenigen Kanten, die in L_n beginnen: $e = (u, v) \in E_n, u \in L_n$.
3. Initialisiere $\chi(e) = 1$ für alle $e \in E_n$.
4. Berechne die Anzahl an Kreuzungen an denen jedes $e \in E_n$ beteiligt ist als $c(e)$, beispielsweise mit [4].
5. Entferne alle e mit $c(e) = 0$ aus E_n.
6. Wenn $E_n = \oslash$, fahre fort mit dem nächsten *Layer* bei 1.
7. Sortiere die restlichen e absteigend nach $\frac{c(e)}{1+(\Omega_L(e)\omega(e))}$. Bei Gleichheit gelten die folgenden sekundären Kriterien in absteigender Priorität:
   a) Sortiere aufsteigend nach $\omega(e)$,
   b) sortiere aufsteigend nach $\Omega_L(e)$,
   c) sortiere aufsteigend nach der Summe der globalen Port Indizes beider Seiten,
   d) sortiere aufsteigend nach dem Port Index von $p_i(e)$.
8. Wähle die erste Kante e_0, setze $\chi(e_0) = \chi_k$ und entferne e_0 aus E_n.
9. Fahre fort bei 4.

VERFEINERTES LINEARES PROGRAMM

Diese Erweiterungen lassen sich unmittelbar in das ursprüngliche lineare Programm einbringen, wodurch sich die Formulierung in Gleichung 3.2 ergibt. Da die Länge der vertikalen Segmente der Kanten in den folgenden Abschnitten für unterschiedliche Port Constraints jeweils anders formuliert sein muss, wird sie zunächst zusammenfassend als $s_y(e)$ bezeichnet.

$$\text{Min:} \sum_{e \in E} (\Omega_L(e)\omega(e)\chi(e)s_y(e)) + \kappa \sum_{w \in V_L} y(w) \tag{3.1}$$

Über die ursprüngliche Formulierung $\Omega(e)$ von Gansner *et al.* [42] hinaus sorgt $\Omega_L(e)$ hier nicht nur für eine bevorzugte, sondern auch für eine eindeutige Begradigung langer Kanten. $\chi(e)$ verbessert die Begradigung von einander kreuzenden Kanten und ein zusätzlicher Term gewichtet die Diagrammgröße mit κ gegenüber der Kantenbegradigung.

Benutzervorgaben bezüglich der Gewichtung einzelner Kanten werden wie in der ursprünglichen Formulierung durch den Faktor $\omega(e) \in \mathbb{R} \geqslant 0$ ausgedrückt: Für Kanten ohne Vorgaben sei $\omega(e) = 1$, für Kanten, die bevorzugt begradigt werden sollen, sei $\omega(e) > 1$ und für weniger relevante Kanten sei $\omega(e) < 1$. Da Kanten, die mit *Inverter* Knoten verbunden sind, nicht begradigt werden müssen, wird deren Gewichtung auf $\Omega(e) = 0$ gesetzt, was sie effektiv aus dem linearen Programm entfernt.

Der Lesbarkeit halber wird das pro Kante konstante Produkt aus den Faktoren $\Omega_L(e)$, $\omega(e)$ und $\chi(e)$ im weiteren Verlauf als Gewichtung der Kante $w(e) = \Omega_L(e)\omega(e)\chi(e)$ bezeichnet:

$$\text{Min:} \sum_{e \in E} (w(e)s_y(e)) + \kappa \sum_{v \in V_L} y(v) \tag{3.2}$$

3.3.2 PORT CONSTRAINTS BEI KONSTANTEN KNOTENGRÖSSEN

Zur Berücksichtigung von Port Constraints über PORT_NONE und PORT_FIXED hinaus muss die Länge der vertikalen Segmente der Kanten im linearen Programm aus Gleichung 3.2 nicht mehr wie im ursprünglichen Ansatz auf Basis der Knotenpositionen als Variablen und mit einem optionalen Offset zur Repräsentation der relativen Verschiebungen der Ports als Konstante, sondern mit Hilfe der Portpositionen $y(p_i(e))$ und

$y(p_o(e))$ formuliert werden. Gleichung 3.3 zeigt diese Erweiterung des linearen Programms aus Gleichung 3.2 durch Anwendung von Def. 9 und Einsetzen von $s_y(e) = |y(p_i(e)) - y(p_o(e))|$.

$$\text{Min: } \sum_{e \in E} (w(e)\,|y(p_i(e)) - y(p_o(e))|) + \kappa \sum_{v \in V_L} y(v) \tag{3.3}$$

Durch Einsetzen unterschiedlicher Funktionen für die Portpositionen $y(p_i(e))$ und $y(p_o(e))$ sowie der Constraints, lassen sich mit diesem linearen Programm Kanten unter Berücksichtigung verschiedener Port Constraints begradigen. Die dafür notwendigen Formulierungen sind nachfolgend für verschiedene Port Constraints aufgeführt:

PORT_NONE UND PORT_FIXED

Portpositionen werden als konstante Offsets zur jeweiligen Knotenposition ausgedrückt. Fallen alle Offsets auf den Ursprung des Knotens (Offset= 0), entspricht dies dem Fall PORT_NONE. Diese Varianten wurden bereits von Gansner *et al.* in [42] beschrieben, siehe auch Abschnitt 2.2.3.

PORT_SIDE

$y(p_i(e))$ und $y(p_o(e))$ werden dem LP als eigene Variablen hinzugefügt. Diese $n = |P_{side}|$ Variablen werden durch die Länge der jeweiligen Seite (ausgedrückt über Knotenposition und -größe) beschränkt, was $2n$ Constraints erfordert.

Um zwischen den Ports einer Seite (P_{side}) einen Mindestabstand δ sicherzustellen, müssen Constraints der Form $|y(p_1) - y(p_2)| \geqslant \delta \;\forall\; p_1, p_2 \in P_{side}$ für alle Port Paare $p_1 \neq p_2$ eingehalten werden. Dadurch ergeben sich bei $n = |P_{side}|$ Ports $\frac{1}{2}n\,(n-1)$ zusätzliche Constraints.

Insgesamt ergeben sich durch diese Art der Portpositionierung $2n + \frac{1}{2}n\,(n-1)$ Constraints. Je nach Gewichtung der einzelnen Kanten könnten sich durch Änderung der Reihenfolge zwischen den Ports neue Kantenkreuzungen ergeben. Um das zu vermeiden ist es sinnvoll, die Ports während der Kreuzungsreduktion zu ordnen und zur Kantenbegradigung als `PORT_ORDERED` zu betrachten.

PORT_ORDERED

Für PORT_ORDERED werden $y(p_i(e))$ und $y(p_o(e))$ ebenfalls als eigenständige Variablen formuliert.

Die Positionen des ersten Ports p_1 und letzten Ports p_n einer Seite werden wie bei PORT_SIDE durch die jeweilige Größe des Knotens v und einen Mindestabstand oben $\delta \geqslant 0$ beschränkt. Für die vertikale Seite *east* lassen sich die Constraints beispielsweise als $y(p_1) \geqslant y(v) + \delta$ und $y(p_n) \leqslant y(v) + s_y(v) - \delta$ formulieren.

Zwischen benachbarten Ports müssen Constraints zur Einhaltung des Minimalabstands eingefügt werden, so dass die Reihenfolge der Ports erhalten bleibt: $y(p_i) + \delta \leqslant y(p_j) \forall p_i, p_j \in P_{side}$ bei denen $i_{side}(p_i) = i_{side}(p_j) + 1$.

Insgesamt ergeben sich für n Ports $2 + (n-1) = n+1$ Constraints für die Portpositionen.

PORT_CALCULATED

Um in diesem Fall ein lineares Programm formulieren zu können, muss die Berechnungsvorschrift der Portpositionen aus der Knotengröße als lineare Funktion formulierbar sein. Diese wird dann an Stelle der Portpositionen in das lineare Programm eingesetzt. Typischerweise ergeben sich dabei keine zusätzlichen Constraints für die Portpositionen. Die Berechnungsvorschrift für Portpositionen in SL Diagrammen, sowie ihre Linearisierung, wird in Abschnitt 3.5 besprochen.

3.3.3 VARIABLE KNOTENGRÖSSEN

Bisher wurde die Knotengröße bei der Begradigung von Kanten als konstante Größe angenommen. Viele Modellierungswerkzeuge bieten jedoch die Möglichkeit, die Größe von Knoten zu variieren. Dieser zusätzliche Freiheitsgrad lässt sich bei Verwendung der Port Constraints PORT_FREE, PORT_SIDE, PORT_ORDERED und PORT_SIMULINK zur Begradigung von Kanten verwenden. Nachfolgend sind die hierzu nötigen Erweiterungen am Algorithmus aus Abschnitt 3.3.2 dargestellt:

Zunächst müssen die Knotengrößen $s_y(v)$ derjenigen Knoten, die in ihrer Größe verändert werden dürfen, als Variablen formuliert werden. Die Größe von *Dummy* und *Inverter* Knoten bleibt dabei in jedem Fall konstant: $s_y(v) = 0 \; \forall \; v \in (D \cup I)$. Die Menge der Knoten mit variabler Größe wird als $V_S \subseteq V \backslash (D \cup I)$ bezeichnet. Constraints der Form $0 \leqslant \text{minSize}_y(v) \leqslant s_y(v) \leqslant \text{maxSize}_y(v)$ stellen sicher, dass sich die Anpassung der Knotengrößen in einem vorgegebenen Rahmen bewegt.

Insbesondere $\text{minSize}_y(v)$ muss dabei in Abhängigkeit von der Port Anzahl an den jeweiligen Seiten des Knotens festgelegt werden, um sicherzustellen, dass eventuell vorhandene Constraints für einen Mindestabstand zwischen Ports eingehalten werden können.

Um zu vermeiden, dass der Platz zwischen benachbarten Knoten mit Knotenfläche aufgefüllt und nicht leer gelassen wird, erhält die Knotengröße in der Kostenfunktion ein zusätzliches Gewicht von $\sigma > 0$ gegenüber Leerflächen ($\sigma = 4$ lieferte bisher gute Ergebnisse). Zusätzlich muss die Knotengröße der unteren Knoten bei der Gewichtung der Diagrammgröße gegenüber der Begradigung berücksichtigt werden (erster κ-Term), um tatsächlich die untere Kante der unteren Knoten bei der Gewichtung zu erfassen.

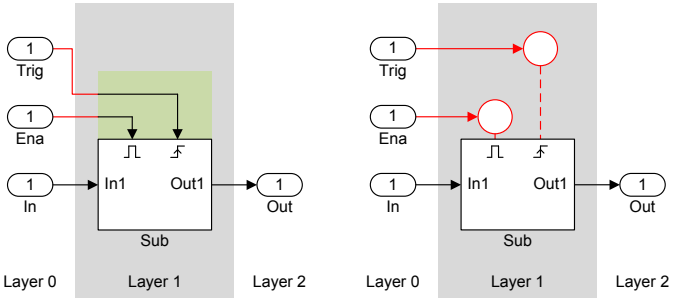
$$\text{Min: } \sum_{e \in E} w(e) s_y(e) + \kappa \sum_{v \in V_L} (y(v) + s_y(v)) + \sigma\kappa \sum_{u \in V_S} s_y(u) \tag{3.4}$$

Knoten mit fester Größe werden wie `PORT_FIXED` Port Constraints behandelt; dort werden die aktuellen Portpositionen als Konstanten eingesetzt. Sie werden in Gleichung 3.4 nicht gesondert betrachtet.

3.3.4 BEGRADIGUNG VON KANTEN BEI OBEN ODER UNTEN LIEGENDEN PORTS

Beim Begradigen von Kanten, die mit Ports an den Seiten *north* und *south* verbunden sind, muss die Knotengröße nicht zwingend angepasst werden, um Kantenknicke zu entfernen. Stattdessen kommt es dabei auf das Routing der Kanten im Nahbereich des Knotens an. Bisherige Ansätze, beispielsweise von Spönemann *et al.* [84], basieren auf einem vor der eigentlichen Kantenbegradigung festgelegten Routing im Nahbereich des Knotens. Für die Begradigung selbst bleibt dieser Bereich unveränderlich, als würden die Ports nicht am Knoten selbst, sondern am Rand des Bereichs liegen. Dieses Vorgehen gleicht also dem bisherigen Vorgehen beim Routing von Rückkopplungen (siehe Abschnitt 3.1.1). Abbildung 3.10a zeigt das aus diesem Ansatz resultierende Layout sowie die durch die Begradigung beeinflussbaren Bereiche der Kanten: Während die Kante vom *Ena* Knoten wegen der geringen Knotengrößen in *Layer* 1 tatsächlich vollständig begradigt werden kann, weist die Kante vom *Trig* Knoten zusätzliche Knicke auf, da ihre rechte Hälfte bei der Begradigung nicht verändert werden kann.

Um diese zusätzlichen Knicke zu entfernen, wurde der folgende Ansatz entwickelt: In die mit Ports an der *north* oder *south* Seite verbundenen



(a) Statisches Routing im Nahbereich des Knotens.

(b) Routing mit verschiebbaren Hilfsknoten.

Abbildung 3.10: Routing von Kanten an oben liegenden Ports.

Kanten werden zusätzliche Knoten im *Layer* des jeweiligen Knotens eingefügt. Die Anordnung dieser Knoten entspricht dabei der Anordnung der Ports während der Kreuzungsreduktion. Damit diese Hilfsknoten möglichst nahe am eigentlichen Knoten liegen, wird ihr Abstand zum eigentlichen Knoten der Zielfunktion des linearen Programms hinzugefügt. Dieser Abstand wird mit einen Faktor $\kappa\theta$ zwischen Leerfläche und Knotenfläche gewichtet, so dass dieser im Bereich $0 < \theta < \sigma$ liegt. Sie unterliegen ansonsten den selben Constraints wie andere *Dummy* Knoten. Abbildung 3.10b zeigt die zusätzlichen Hilfsknoten als Kreise und die für die Kostenfunktion relevanten Abstände als gestrichelte Linien. In der Darstellung sind die Hilfsknoten der Übersicht halber nicht direkt übereinander platziert; für die Reihenfolge und Mindestabstände ist nur relevant, dass beide sich im gleichen *Layer* befinden.

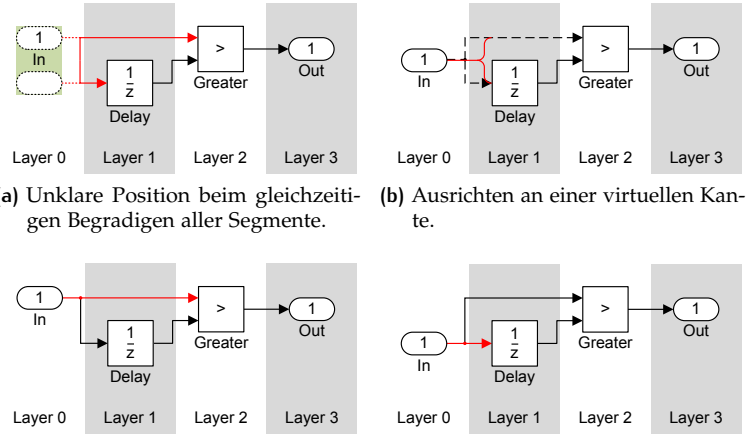
Durch diesen Ansatz können zusätzliche Knicke, die bisher durch das statische Routing im Nahbereich des Knotens entstanden sind, vermieden werden. Im Rahmen der Zusammenarbeit mit dem KLay Layered Projekt konnte der Ansatz bereits durch Schulze [81] weiter verfeinert werden, indem die zusätzlichen Dummy Knoten während der Kreuzungsreduktion gesondert berücksichtigt werden.

3.4 BEGRADIGUNG VON HYPERKANTEN

Hyperkanten stellen für die Kantenbegradigung eine besondere Herausforderung dar, da die einzelnen Segmente einer Hyperkante im Allgemeinen nicht gleichzeitig ohne Knick gezeichnet werden können. Ausnahmen bilden Graphen, bei denen zwei gegebene Ports durch mehrere Kanten

verbunden werden können oder Darstellungen, bei denen mehrere Ports auf der gleichen Position liegen dürfen. Diese sind in der Praxis, insbesondere für SL Diagramme, ohne Relevanz und werden hier daher nicht gesondert betrachtet.

Für die Begradigung von Hyperkanten werden nachfolgend drei Strategien erläutert: Das gleichzeitige Begradigen aller Segmente als eigenständige Kanten, das Begradigen einer virtuellen Kante, die im Schwerpunkt aller Segmente liegt und die Auswahl und Begradigung eines einzelnen Segments. Als Grundlage dient das lineare Programm aus Gleichung 3.4. Eine Hyperkante wird für diesen Abschnitt analog zu Abschnitt 2.1.1 als eine Menge von Kanten betrachtet: $h \subseteq E$.



(a) Unklare Position beim gleichzeitigen Begradigen aller Segmente.

(b) Ausrichten an einer virtuellen Kante.

(c) Begradigung des oberen Segments.

(d) Begradigung des unteren Segments.

Abbildung 3.11: Verschiedene Ansätze zum Begradigen von Hyperkanten.

3.4.1 FESTE GEWICHTUNG DER SEGMENTE

Setzt man die Segmente einer Hyperkante als eigenständige Kanten in das lineare Programm ein, werden diese gleichzeitig begradigt, stehen jedoch im Konflikt miteinander. Abbildung 3.11a zeigt diesen Konflikt für die Hyperkante $h = \{e_1, e_2\}$ am Ausgang vom *In* Knoten: Der Abstand zwischen der oberen Kante und dem *Delay* Knoten verhindert das Begradigen der beiden Segmente e_1 und e_2. Zudem geht aus dem Term zur Begradigung der beiden Segmente (Min: $w(e_1) \cdot \text{offset}(e_1) + w(e_2) \cdot \text{offset}(e_2)$) nicht hervor, welches der beiden Segmente bevorzugt zu begradigen ist. Jede

im markierten Bereich liegende Position des *In* Knotens entspricht prinzipiell einer optimalen Lösung dieses Teils des linearen Programms.

Diese Mehrdeutigkeit wird teilweise durch weitere Kriterien aufgelöst: Unterschiedliche Gewichte $w(e)$ der einzelnen Segmente können eindeutige Lösungen ermöglichen. Dies ist jedoch nicht immer der Fall, wie beispielsweise bei einer Hyperkante $h = \{e_1, e_2, e_3\}$ mit den Gewichten $w(e_1) = 1, w(e_2) = 1, w(e_3) = 2$. Der Kompaktheitsterm $\kappa \sum_{v \in V_L} y(v) y_s(v)$ im linearen Programm kann ebenfalls zu eindeutigen Lösungen führen, beispielsweise wäre dann die oberste Position für den *In* Knoten optimal. Dieser greift jedoch nicht, wenn sich im gleichen *Layer* unterhalb des fraglichen Knotens noch wenigstens ein weiterer Knoten befindet, dessen Kante zu begradigen ist. Sollten diese Kriterien nicht zu einer eindeutigen Lösung führen, wählt der LP Solver eine Lösung aus, meistens ist dies die zuerst gefundene (optimale) Lösung. In der Praxis werden die Knoten dabei typischerweise auf die oberste Position (kleinstes $y(v)$) im Lösungsraum gesetzt.

3.4.2 GEWICHTETES ZENTRIEREN

Beim gewichteten Zentrieren wird versucht, eine symmetrische Darstellung von Hyperkanten zu erreichen. Dazu werden Hyperkanten im linearen Programm nicht, wie bei der gleichzeitigen Begradigung aller Segmente, durch mehrere Kanten repräsentiert, sondern durch eine einzelne virtuelle Kante, deren Enden im gewichteten Mittelpunkt zwischen den Ports der jeweiligen Seite liegen.

Um den Offset der zusammengefassten Hyperkante zu berechnen, muss die Position je eines virtuellen Eingangs- und Ausgangsports definiert werden. Beim Ausgangsport wird dies als gewichtetes Mittel der Positionen der mit den Segmenten verbundenen Ausgangsports definiert: $y(h_i) = \frac{1}{|h|} \cdot \sum_{e \in h} w(e)\ y(p_i(e))$; die Definition der Position des zusammengefassten Ausgangsports $y(h_o)$ erfolgt analog. Daraus folgt die Definition des Offsets für Hyperkanten beim gewichteten Zentrieren als $s_y(h) = |y(h_i) - y(h_o)|$. Das Gewicht dieser zusammengefassten Hyperkante entspricht dabei der Summe der $w(e)$ ihrer Segmente: $w(h) = \sum_{e \in h} w(e)$.

Diese $w(h)$ und $s_y(h)$ werden im linearen Programm an Stelle des Terms für die Offsets der Kanten eingesetzt. Abbildung 3.11b zeigt die resultierende Positionierung von *In*, sowie die virtuelle Kante bei gleicher Gewichtung des oberen und unteren Segments. In der Regel führt das gewichtete Zentrieren zu einer Darstellung von Hyperkanten, bei denen alle Segmente mit Knicken gezeichnet werden.

3.4.3 AUTOMATISCHES GEWICHTEN DER SEGMENTE DURCH VARIANTENBILDUNG

Für Diagramme mit wenigen Hyperkanten, bei denen die Anzahl der Kantenknicke minimal gehalten werden soll, bietet sich das automatische Gewichten der Segmente durch Variantenbildung an. Dazu werden mehrere Begradigungen durchgeführt, wobei den Segmenten der einzelnen Hyperkanten jeweils unterschiedliche Gewichte zugeordnet werden, so dass für jede Hyperkante jeweils ein Segment pro Seite eindeutig bevorzugt begradigt wird. Hyperkanten, bei denen bereits eindeutige Prioritäten zwischen den Segmenten beispielsweise durch den Benutzer festgelegt sind, werden wie in Verfahren 1 behandelt und beim Erzeugen der Varianten nicht betrachtet. Ausgewählt wird dann die Variante, bei der die Zielfunktion des linearen Programms im Vergleich zu den anderen Varianten minimal ist. Abbildung 3.11c und Abbildung 3.11d zeigen die beiden Varianten der Hyperkante im dargestellten Diagramm.

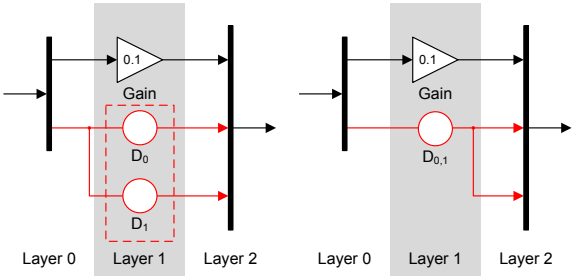
Da bei diesem Verfahren jede Hyperkante $h \subseteq E$ die Anzahl an Variationen um den Faktor $|h|$ erhöht, ergeben sich für ein Diagramm mit mehreren Hyperkanten $H = \{ h \mid h \subseteq E \text{ und } h_1 \cap h_2 = \emptyset \; \forall \; h_1 \neq h_2 \}$ bereits $v = \prod_{h \in H} |h|$ Varianten. Es ist daher insbesondere für kleine Diagramme geeignet, bei denen nicht mehr als 10 bis 20 Varianten entstehen. Bei Diagrammen, aus denen viele Varianten erzeugt werden würden, empfiehlt sich entweder die Anwendung eines der anderen Verfahren zur Begradigung oder ein hybrider Ansatz mit einer oberen Schranke für die Laufzeit T_{max}:

1. Erzeuge eine Lösung mit Hilfe eines der anderen Verfahren: *Variante 0*.
2. Wenn die aktuelle Laufzeit $t \geqslant T_{max}$ ist, fahre fort bei 4. Ansonsten löse zunächst die Variante, bei der aus jeder Hyperkante das jeweils obersten Segment ausgewählt ist. Dies basiert auf der Beobachtung, dass diese Variante häufig optimal ist.
3. Solange $t < T_{max}$ und weitere Varianten vorhanden sind: Erzeuge und löse eine weitere Variante.
4. Auswahl der besten Variante (niedrigster Wert der Kostenfunktion).

Die Verwendung der Zeitschranke sorgt jedoch dafür, dass dieser Ansatz durch Schwankungen in der realen Laufzeit des Algorithmus nicht deterministisch ist. In Stufe 3 kann daher durchaus auch eine zufällige Auswahl der zu erzeugenden Variante vorgenommen werden, so dass die Varianten mit der gleichen Wahrscheinlichkeit auftreten.

3.4.4 BEGRADIGUNG LANGER HYPERKANTEN

Während die einzelnen Segmente von Hyperkanten für die Kreuzungsreduktion noch jeweils als einzelne Kanten betrachtet werden konnten, sorgt diese Herangehensweise bei der späteren Berechnung der Positionen von benachbarten Segmenten der gleichen Hyperkante für unnötigen Platzbedarf, da zwischen jedem Segment der Kanten ein Abstand gelassen wird, so als gehörten sie zu verschiedenen Kanten. In Abbildung 3.12a ist eine solche Situation dargestellt: Die Hyperkante wird bis zum Ziel als zwei getrennte Kanten mit eigenen *Dummy* Knoten behandelt. Dadurch entsteht ein Abstand zwischen den beiden Endsegmenten um D_0 und D_1.



(a) Lange Hyperkante mit benachbarten Dummy Knoten.

(b) Benachbarte Dummy Knoten wurden zusammengefasst.

Abbildung 3.12: Zusammenfassen benachbarter Dummy Knoten bei langen Hyperkanten.

Um das zu vermeiden, werden benachbarte *Dummy* Knoten in Hyperkanten miteinander vereinigt. Dazu werden *Layer* für *Layer* zunächst von links nach rechts und von oben nach unten benachbarte *Dummy* Knoten gesucht, deren eingangsseitige Kanten den selben Quellport haben, die also Segmente der selben Hyperkante sind. Diese *Dummy* Knoten werden zusammengefasst und die Suche mit dem nächsten *Layer* fortgesetzt. Danach wird analog dazu für Rückkopplungen von rechts nach links vorgegangen, wobei hier auf Gleichheit der Zielports der ausgangsseitigen Kanten geprüft werden muss. Das Ergebnis ist beispielhaft in Abbildung 3.12b dargestellt.

3.5 KANTENBEGRADIGUNG IN SIMULINK DIAGRAMMEN

In SL Diagrammen sind die Positionen der Ports eines Knotens von dessen Größe abhängig (`PORT_SIMULINK`). Verändert man die Größe eines Knotens, passen sich die Positionen der Ports automatisch an und beeinflussen dadurch Knicke von Kanten, die mit diesen Ports verbunden sind. Diese Eigenschaft machen sich viele Benutzer zu Nutze, indem sie die Größe von Knoten gezielt anpassen, um Kantenknicke zu minimieren.

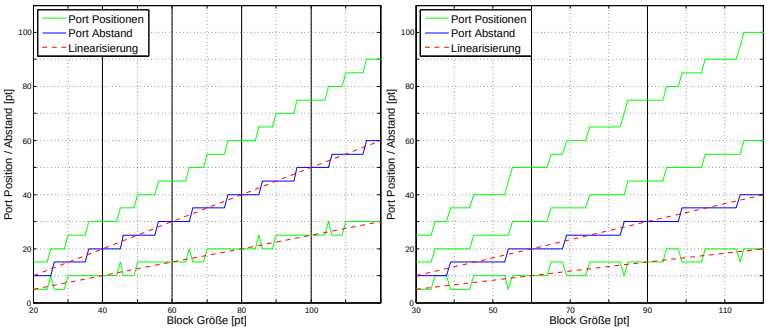
Nachfolgend wird zunächst der Zusammenhang zwischen Knotengröße und Portpositionen bei SL Diagrammen untersucht, eine lineare Approximation dieses Zusammenhangs erstellt und ein Algorithmus zur automatischen Anpassung der Knotengrößen bei der Minimierung von Kantenknicken auf Basis von Abschnitt 3.3.3 vorgestellt.

3.5.1 PORTPOSITIONEN BEI SIMULINK

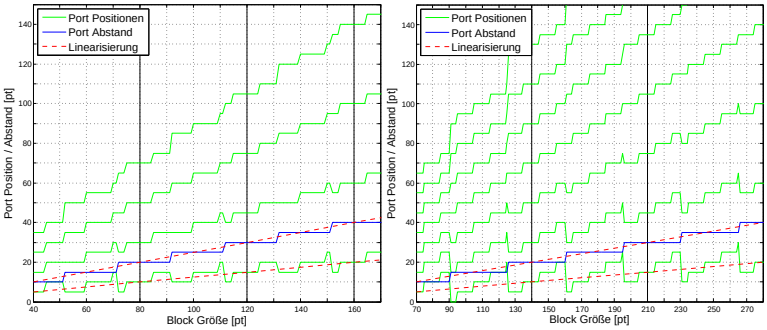
Bei SL hat die Größe eines Knotens unmittelbar Einfluss auf die Position der Ports (Port Constraint `PORT_SIMULINK`). Verändert man beispielsweise die vertikale Größe eines Knotens, verschieben sich nicht nur die unteren Ports zusammen mit der unteren Seite (*south*, siehe Def. 4), sondern es verändert sich auch die Verteilung der Ports an den Seiten *east* und *west*, in Abhängigkeit der vertikalen Größe des Knotens.

Da die genaue Berechnungsvorschrift für die Positionierung der Ports nicht öffentlich gemacht ist, wurde die Abhängigkeit zwischen Knotengröße und Portpositionen experimentell für Knoten mit einer unterschiedlichen Anzahl Ports ermittelt. Abbildung 3.13 stellt die Portpositionen für Knoten mit 2, 3, 4 und 7 Ports zu ihrer jeweiligen Seite dar.

Der Abstand zwischen zwei benachbarten Ports war bei gegebener Knotengröße und Port Anzahl für Paare benachbarter Ports einer Seite gleich; er ist ebenfalls in Abbildung 3.13 dargestellt. Die relative Position eines Ports $y_{rel}(p)$ an seiner Seite lässt sich daher mit Hilfe der Position des ersten Ports als Offset und einem Vielfachen des Port Abstands darstellen. Unter Nutzung der Definitionen in Abschnitt 2.1.1 ergibt sich für $p \in P_{east}(v)$ eine vertikale Position von $y_{rel}(p) = y_{offset} + i(p) \cdot y_{dist}$, wobei $y_{offset} = y_{rel}(p_0)$ die Position des obersten Ports der Seite ist $i(p_0) = 0, p_0 \in P_{east}$. Die horizontale Position bleibt unverändert, die Portpositionen an den anderen Seiten folgen analog und werden nicht gesondert behandelt.



(a) 2 Ports **(b)** 3 Ports



(c) 4 Ports **(d)** 7 Ports

Abbildung 3.13: Portpositionen in Abhängigkeit der Knotengröße für unterschiedlich viele Ports.

LINEARISIERUNG DER PORTPOSITIONEN

Mit Hilfe der Knotengröße $s_y(v)$ und der Port Anzahl der jeweiligen Seite $|P_{east}(v)|$, lassen sich y_{offset} und y_{dist} näherungsweise linear darstellen (siehe Gleichung 3.5 und Gleichung 3.6). Beide Linearisierungen sind in Abbildung 3.13 eingezeichnet.

$$y_{offset} = \frac{s_y(v)}{2 \cdot |P_{east}(v)|} \tag{3.5}$$

$$y_{dist} = \frac{s_y(v)}{|P_{east}(v)|} \tag{3.6}$$

Daraus ergibt sich die linearisierte Form für $y_{rel}(p)$ in Gleichung 3.7:

$$y_{rel}(p) = \frac{1 + 2i(p)}{2 \cdot |P_{east}(v)|} \cdot s_y(v) \tag{3.7}$$

GÜLTIGKEIT DER LINEARISIERUNG

Wie Abbildung 3.13 zeigt, entspricht die linearisierte Form von y_{offset} und y_{dist} nicht für jede Knotengröße den tatsächlichen Werten. Die Knotengrößen, an denen die mit der Linearisierung berechneten Portpositionen mit den tatsächlichen Portpositionen übereinstimmen, sind abhängig von der Anzahl der Ports. An diesen Stellen wird die Linearisierung nachfolgend als gültig bezeichnet.

Lässt man einen Minimalabstand zwischen den Ports von 10 pt, ist y_{dist} für die Seite *east* gültig linearisiert, wenn $s_y(v) = 5\ n\ |P_{east}(v)|$, $n \in \mathbb{N} > 0$. Die Linearisierung von y_{offset} ist nur halb so oft gültig, nämlich für $s_y(v) = 10\ n\ |P_{east}(v)|$, $n \in \mathbb{N} > 0$. Diese Knotengrößen wurden in Abbildung 3.13 markiert.

Die Menge der Knotengrößen, an denen die Linearisierung der Portpositionen für eine Seite $s \in \{north, south, west, east\}$ gültig ist, wird als Λ_s bezeichnet. Sie ist in Gleichung 3.8 gezeigt und entspricht der Menge an gültigen Positionen der Offsets.

$$\Lambda_s = \{\lambda \,|\lambda = 10\ n\ \ |P_s(v)|\},\ n \in \mathbb{N} > 0 \tag{3.8}$$

Die Größe eines Knotens kann in zwei Richtungen verändert werden, horizontal und vertikal, wobei jede der beiden Richtungen die Portpositionen an jeweils zwei Seiten beeinflusst: horizontale Änderungen beeinflussen Ports an der *north* und *south* Seite, vertikale Änderungen beeinflussen die Ports der *west* und *east* Seiten. Gültige Linearisierungen der Portpositionen liegen nur dann für eine dieser Richtungen vor, wenn die Linearisierungen beider betroffenen Seiten gültig sind. Gleichung 3.9 zeigt die Menge der vertikalen Knotengrößen mit gültiger Linearisierung Λ_y.

$$\begin{aligned} \Lambda_y &= \Lambda_{west} \cap \Lambda_{east} \\ &= \{\, \lambda \,|\lambda = 10n \cdot \mathrm{kgV}(\ |P_{west}(v)|,\ |P_{east}(v)|\ )\ \} \end{aligned} \tag{3.9}$$

Für Seiten ohne Ports ist keine eigene Linearisierung notwendig, diese werden als $|P_{west}(v)| = 1$ angenommen.

3.5.2 KANTENBEGRADIGUNG FÜR SIMULINK

Mit Hilfe der in Abschnitt 3.5.1 linearisierten Abhängigkeit zwischen Portpositionen und Knotengröße von SL lässt sich ein Algorithmus zur Kantenbegradigung formulieren, der in vertikaler Richtung nicht nur die Positionierung von Knoten variieren, sondern auch deren Größe verändern kann, um zusätzliche Kantenknicke zu entfernen. Als Grundlage dient das in den vorherigen Abschnitten verfeinerte lineare Programm zur Kantenbegradigung unter Berücksichtigung von Port Constraints:

$$\text{Min: } \sum_{e \in E} w(e) s_y(e) + \kappa \sum_{w \in V_L} (y(w) + s_y(w)) + \sigma\kappa \sum_{u \in V_S} s_y(u) \quad (3.10)$$

In dieses wurde als Kantenoffset $s_y(e)$ die lineare Formulierung der Portpositionen aus Gleichung 3.8 eingesetzt:

$$s_y(e) = \left| \left(y(v) + \frac{1 + 2i(p_i(e))}{2 \ |P_{west}(v)|} \cdot s_y(v) \right) - \left(y(u) + \frac{1 + 2i(p_o(e))}{2 \ |P_{east}(u)|} \cdot s_y(u) \right) \right| \quad (3.11)$$

Daraus ergibt sich das unskalierte lineare Programm zu Kantenbegradigung mit variablen Knotengrößen in SL Diagrammen (Gleichung 3.12):

$$\begin{aligned} \text{Min: } & \sum_{e \in E} \left(w(e) \cdot \left| \left(y(v) + \frac{1 + 2i(p_i(e))}{2 \ |P_{west}(v)|} \cdot s_y(v) \right) \right. \right. \\ & \left. \left. - \left(y(u) + \frac{1 + 2i(p_o(e))}{2 \ |P_{east}(u)|} \cdot s_y(u) \right) \right| \right) \\ & + \kappa \sum_{l \in V_L} y(l) + \kappa\sigma \sum_{r \in V_R} s_y(r) \end{aligned} \quad (3.12)$$

SONDERFÄLLE

Die Knotengröße $s_y(v)$ von Knoten, die an der *west* und *east* Seite jeweils höchstens einen Port haben, wird als konstant angenommen, wobei sie entweder die aktuelle Größe oder die minimale Knotengröße erhalten. Knoten, die vom Benutzer entsprechend markiert wurden, haben ebenfalls eine konstante Größe: $\text{minSize}_y(v) = \text{maxSize}_y(v)$. In der praktischen Umsetzung wird das lineare Programm direkt mit konstanten Größen und Portpositionen für diese Knoten erzeugt.

Inverter Knoten, sowie Kanten von und zu *Inverter* Knoten müssen keinen Beitrag zur Zielfunktion ($\Omega_L = 0$) oder zu den Constraints ($s_y = 0$)

leisten, da ihre konkrete Position nicht benötigt wird. In der praktischen Umsetzung werden diese vollständig aus dem linearen Programm entfernt.

3.5.3 SKALIERUNG DER VARIABLEN

Alle grafischen Elemente in SL Diagrammen liegen auf Koordinaten, die ganzzahlige Vielfache von 5 sind: $y(v), s_y(v) \in \{0, 5, 10, \ldots, \sim 32765\}$. Die direkt beeinflussbaren Variablen (Positionen von Knoten und Linien, Größen von Knoten) lassen sich mit Hilfe der in Abschnitt 2.4.1 vorgestellten Technik auf Koordinaten beschränken, die diese Anforderungen erfüllen. Für die Knoten Positionen $y(v)$ ergibt sich damit ein Skalierungsfaktor von $\lambda_0 = 5$. Da jedoch auch die Portpositionen auf diesem Raster liegen müssen und diese zudem den Einschränkungen der Linearisierung der Positionsberechnung aus Abschnitt 3.5.1 unterliegen, muss jeder Knoten für seine Größe einen eigenen Skalierungsfaktor $\lambda_y(v)$ erhalten. Dieser leitet sich direkt aus den gültigen Punkten der Linearisierung Λ_y in Gleichung 3.9 ab und führt zu Gleichung 3.13 für den Skalierungsfaktor:

$$\lambda_y(v) = 10 \cdot \mathrm{kgV}\left(\max(1, |P_{west}(v)|),\ \max(1, |P_{east}(v)|) \right) \tag{3.13}$$

Daraus ergibt sich dann die skalierte Form der Zielfunktion des linearen Programms zur Kantenbegradigung in SL:

$$\begin{aligned} &\text{Min:} \\ &\sum_{e \in E} \left(w(e) \cdot \left| \left(\lambda_0 y(v) + \lambda_y(v) \frac{1 + 2i(p_i(e))}{2\ |P_{west}(v)|} \cdot s_y(v) \right) \right.\right. \\ &\left.\left. - \left(\lambda_0 y(u) + \lambda_y(u) \frac{1 + 2i(p_o(e))}{2\ |P_{east}(u)|} \cdot s_y(u) \right) \right| \right) \\ &+ \kappa\lambda_0 \sum_{l \in V_L} y(l) + \kappa\sigma \sum_{r \in V_R} \lambda_y(r) s_y(r) \end{aligned} \tag{3.14}$$

Die Skalierungsfaktoren müssen auch in den Constraints des linearen Programms auf die jeweiligen Variablen angewendet werden.

UNGÜLTIGE KNOTENGRÖSSEN

Bei Knoten mit fester Größe, deren ursprünglichen Größe nicht in das SL Raster (5 Punkte) fallen, ist eine korrekt skalierte, ganzzahlige Darstellung

nicht möglich. Daher werden diese für die Dauer der Kantenbegradigung auf die nächste gültige Größe vergrößert und nach Abschluss des Vorgangs wieder auf ihre ursprünglichen Abmessungen zurückgesetzt. Prominentestes Beispiel für diese Gruppe sind *Input* und *Output* Knoten, die normalerweise mit einer vertikalen Größe von 14 Punkten dargestellt werden.

GRENZEN DER SKALIERUNG

Aus Gleichung 3.13 ergeben sich im Allgemeinen gute Skalierungen für Knoten, die zumindest auf einer der beiden Seiten (*east*, *west*) nur wenige Ports haben. Dies ist für einen Großteil der Knoten aus SL Diagrammen der Fall: Die meisten Knoten der Standard-Bibliotheken haben zumindest auf einer (*east*, *west*) Seite nur einen oder zwei Ports. Eine Ausnahme bilden hier insbesondere die Subsysteme, da deren Port Anzahl vom Benutzer festgelegt wird, so dass durchaus Kombinationen wie 150 Eingangs- und 7 Ausgangsports vorkommen können ($\lambda(v) = 10500$), auch wenn solche Fälle selten sind.

In solchen Situationen ist die Linearisierung der Portpositionen zwar weiterhin gültig, jedoch für die praktische Anwendung unzureichend. Der tatsächlich verwendete Skalierungsfaktor wird daher in mehreren Schritten bestimmt:

1. Bestimmung von $\lambda_y(v)$ gemäß Gleichung 3.13. Akzeptanz, wenn $\lambda_y(v) \leqslant \lambda_{max}$, sonst gehe zu 2.
2. Es wird toleriert, dass die Linearisierung für die Seite mit weniger Ports ungültig ist, wodurch an dieser Seite Knicke entstehen können:
   $\lambda_y(v) = 10 \cdot \max(1, |P_{west}(v)|, |P_{east}(v)|)$. Akzeptanz, wenn $\lambda_y(v) \leqslant \lambda_{max}$, sonst gehe zu 3.
3. Es konnte keine sinnvolle Skalierung gefunden werden. Die Größe des Knotens wird als konstant angenommen, es werden die Portpositionen aus dem entsprechenden Knoten im SL Diagramm verwendet (`PORT_FIXED`).

3.5.4 LINEARES PROGRAMM ZUR KANTENBEGRADIGUNG IN SIMULINK DIAGRAMMEN

Die bisher vorgestellten linearen Programme zur Kantenbegradigung können von den in Abschnitt 2.4.2 vorgestellten LP Solvern in ihrer aktuellen Form noch nicht gelöst werden, da sie Beträge enthalten. In diesem Abschnitt wird mit Hilfe der Umformung aus Abschnitt 2.4.1

das vollständige lineare Programm zur Begradigung von Kanten in SL Diagrammen mit Constraints dargestellt:

ZIELFUNKTION

Die Zielfunktion des linearen Programms ist der Lesbarkeit halber nicht in Standardform, sondern als Minimierung formuliert, da diese Form durch Ausschreiben der Summen bereits von aktuellen LP Solvern verarbeitet werden kann. Für die Beträge in den Portpositionen müssen jedoch gemäß der Umformung aus Abschnitt 2.4.1 jeweils zwei Hilfsvariablen α und β für jede Kante eingesetzt und minimiert werden, während sich die Berechnung der Portpositionen in die Constraints verlagert. Es ergibt sich die vollständige, von LP Solvern lösbare Zielfunktion des linearen Programms: Gleichung 3.15.

Minimize:

$$\begin{aligned} &\sum_{e \in E} \left(\Omega_L(e)\omega(e)\chi(e) \cdot (\alpha(e) + \beta(e)) \right) \\ &+ \kappa\lambda_0 \sum_{l \in V_L} y(l) + \kappa\sigma \sum_{r \in V_R} \lambda_y(r) s_y(r) \end{aligned} \tag{3.15}$$

CONSTRAINTS

Durch die Umformung der Beträge wird die Begradigung der Kanten $e = (u, v) \in E_R$ in den Constraints gefordert:

$$\begin{aligned} &\left(\lambda_0 y(v) + \lambda_y(v) \frac{1 + 2i(p_i(e))}{2\ |P_{west}(v)|} \cdot s_y(v) \right) + \alpha(e) \\ &- \left(\lambda_0 y(u) + \lambda_y(u) \frac{1 + 2i(p_o(e))}{2\ |P_{east}(u)|} \cdot s_y(u) \right) - \beta(e) = 0 \end{aligned} \tag{3.16}$$

Die Einhaltung der minimalen und maximalen Knotengrößen für alle Knoten v variabler Größe wird gefordert durch:

$$\text{minSize}_y(v) \leqslant \lambda(v) \cdot s_y(v) \leqslant \text{maxSize}_y(v) \tag{3.17}$$

Zudem werden für alle Knoten w mit konstanter Größe $\text{constSize}(w)$ feste Knotengrößen gefordert. Für diese Knoten kann zur Vereinfachung stets $\lambda(w) = 1$ angenommen werden, da eine Skalierung hier nicht benötigt wird.

$$s_y(w) = \text{constSize}(w) \tag{3.18}$$

Mindestabstände zwischen benachbarten Knoten sind einzuhalten:

$$\lambda_0 y(v) - \delta_{north}(v) - \lambda_0 y(u) - \lambda(u) s_y(u) - \delta_{south}(u) \geqslant 0 \tag{3.19}$$

VARIABLEN UND KONSTANTEN

Im linearen Programm werden die folgenden Variablen verwendet:

$$\begin{aligned} y(v), s_y(v) &\in \mathbb{Z} \geqslant 0 \\ \alpha(e), \beta(e) &\in \mathbb{R} \geqslant 0 \end{aligned} \tag{3.20}$$

Alle anderen Werte sind Konstanten.

4 MODELLIERUNGS-UNTERSTÜTZUNG

Aktuelle Werkzeuge zur textuellen Softwareentwicklung bieten etliche Funktionen zur Erleichterung häufig wiederkehrender Arbeiten: *Refactoring* Funktionen unterstützen beim Umstrukturieren von Programmabschnitten wie dem Ausgliedern von Programmteilen in eigene Klassen oder beim Ändern von Bezeichnern, Typen oder Signaturen von Attributen, Datenstrukturen und Methoden. Funktionen zur automatischen Vervollständigung von Schlüsselwörtern oder Bezeichnern bei Attributen und Methoden unterstützen den Benutzer unmittelbar bei der Eingabe des Programms (*Auto Completion*).

In der grafischen modellbasierten Entwicklung fehlen vergleichbare Funktionen zur Unterstützung des Benutzers beim Erstellen und Bearbeiten von Modellen bisher fast völlig. Ein wesentlicher Grund hierfür war bisher die fehlende Layoutunterstützung: Da jede strukturelle Änderung an einem Modell auch Anpassungen am Layout des Diagramms erforderlich macht, muss die Verfügbarkeit einer adäquaten Layoutunterstützung für die praktische Anwendbarkeit solcher Funktionen vorausgesetzt werden. Der in Abschnitt 3 beschriebene Layoutalgorithmus für SL-Diagramme ermöglicht eine solche Layoutunterstützung. Darauf aufbauend können daher in diesem Kapitel verschiedene Funktionen vorgestellt werden, die das grafische Modellieren mit SL unterstützen sollen:

Das kontextbasierte Modellieren ist eine der *Auto Completion* ähnliche Funktion für SL auf Basis des *structural editing* (Abschnitt 2.3.2). Zusätzliche Erweiterungen des SL Editors werden mit dem *Aufbrechen von Subsystemen* als ein einfaches strukturelles *Refactoring*, sowie dem *wahlfreien Einfügen von Ports* in *Bus Creator* Blöcke als Unterstützung beim Bearbeiten von Datenstrukturen vorgestellt.

Obwohl diese Funktionen vorrangig für SL Diagramme entwickelt und umgesetzt wurden, lassen sie sich prinzipiell auch auf andere Modellierungssprachen wie ASCET, Ptolemy oder SCADE anwenden.

4.1 KONTEXTBASIERTES MODELLIEREN

Während eine direkte Übertragung des *Structural Editing* auf SL Diagramme zunächst nicht sinnvoll erscheint (siehe Abschnitt 2.3.2), ist eine Funktion zur Erleichterung der Arbeitsschritte beim Einfügen und Verbinden neuer Blöcke aus Sicht der Praxis durchaus wünschenswert. Solch eine Funktion sollte dem Benutzer möglichst genau diejenigen Blöcke zum Einfügen anbieten, die im Kontext des aktuell angewählten Blocks gerade sinnvoll sind. Bestehende Lösungen bieten hierzu lediglich globale Listen mit zuletzt oder häufig verwendeten Blöcken, ohne den aktuellen Kontext zu berücksichtigen.

4.1.1 HEURISTIK

Basierend auf der grundlegenden Annahme, reale Modelle der gleichen oder einer verwandten Domäne würden sich vorwiegend aus häufig wiederkehrenden Strukturen zusammensetzen, wurde die folgende Heuristik zur Auswahl möglicher Kandidaten zum automatischen Einfügen und Verbinden im Kontext eines gewählten Blocks entwickelt:

Es sollen einem Benutzer im Kontext eines Blocks mit einem bestimmten Blocktyp diejenigen Blöcke zum Einfügen und Verbinden angeboten werden, die in bereits bestehenden Referenzmodellen besonders häufig im Kontext des gewählten Blocktyps vorkommen. Erforderlich hierzu ist lediglich die Kenntnis der Verbindungsstruktur aus bestehenden Modellen als Referenz.

Ausgenommen von dieser Heuristik bleibt eine Liste bestimmter Ports ausgewählter Blocktypen wie *SubSystem* Blöcke (alle Ports), *BusCreator* Blöcke (Eingänge) oder *BusSelector* Blöcke (Ausgänge), die an diesen Seiten auf Grund ihrer Funktionsweise fast beliebige Nachbarblöcke aufweisen können. Ebenfalls ausgenommen sind Blöcke aus projektspezifischen Bibliotheken, da diese in Modellen anderer Projekte nicht mehr zur Verfügung stehen würden.

4.1.2 REFERENZDATEN

Zum Erstellen der Referenzdaten für die Heuristik wurde eine Analyse der Struktur der bereits in Abschnitt 2.1.2 betrachteten Modelle aus dem Fahrzeug-Innenraumbereich mit insgesamt 51293 Verbindungen zwischen 39993 verschiedenen Blöcken durchgeführt. Erfasst wurde, wie oft Blöcke bestimmter Typen an ihren jeweiligen Ports miteinander verbunden sind.

Es hat sich bestätigt, dass diese Annahme zumindest für die gegebenen Modelle bei Paaren von direkt miteinander verbundenen Blöcken grundsätzlich korrekt ist. Für die meisten Blocktypen gilt, dass sie an einen gegebenen Port besonders häufig mit Blöcken eines bestimmten (anderen oder gleichen) Blocktyps an bestimmten Ports verbunden sind. In Tabelle 4.1, Tabelle 4.2 und Tabelle 4.3 sind beispielhaft die jeweils häufigsten Blocktypen und Ports aufgeführt, die mit dem Ausgang von *Constant*, *Sum* und *Logic* Blöcken verbunden waren.

Blocktyp	Port	Häufigkeit
Relational Operator	2	25%
Switch	1	19%
Switch	3	11%
Relational Operator	1	6%
Sum	2	6%
Product	2	5%
Logic	1	4%
Data Type Conversion	1	3%
Sum	1	3%
Product	1	3%
Andere	–	15%

Tabelle 4.1: Häufige ausgangsseitige Nachbarblöcke von *Constant* Blöcken (insgesamt 3498 Verbindungen).

Blocktyp	Port	Häufigkeit
Switch	1	22%
Sum	2	16%
Outport	1	12%
Relational Operator	2	11%
Product	1	9%
Andere	–	30%

Tabelle 4.2: Häufige ausgangsseitige Nachbarblöcke von *Sum* Blöcken (insgesamt 603 Verbindungen).

Blocktyp	Port	Häufigkeit
Logic	1	30%
Logic	2	25%
Outport	1	18%
Switch	2	12%
Logic	3	5%
Andere	–	10%

Tabelle 4.3: Häufige ausgangsseitige Nachbarblöcke von *Logic* Blöcken (insgesamt 4643 Verbindungen).

4.1.3 REALISIERUNG

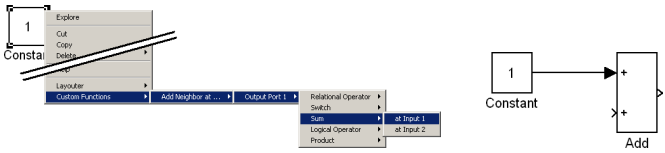
Die prototypische Umsetzung der Heuristik erfolgte in der Matlab eigenen Skriptsprache M. Mit Hilfe einer Erweiterung des Kontextmenüs der Blöcke lässt sich die Funktion zum Einfügen von Nachbarblöcken unmittelbar am aktuell selektierten Block aufrufen. Die Auswahl, welcher Block an welchem Port eingefügt werden soll, erfolgt dabei in den Schritten zwei bis vier:

1. Auswahl des Eintrags im Kontextmenü zum Einfügen von Nachbarblöcken.
2. Auswahl des lokalen Ports (Ein-/Ausgang und Portnummer).
3. Auswahl des einzufügenden Blocktyps.
4. Auswahl des zu verbindenden Ports am neuen Block.

Beispielsweise bietet das Kontextmenü eines *Constant* Blocks an, *Relational Operator*, *Switch*, *Sum*, *Product* oder *Logic* Blöcke direkt einzufügen und mit dem gewählten Port zu verbinden. Abbildung 4.1 zeigt die angebotenen Nachfolgeblöcke für einen *Constant* Block, die Auswahl zum Einfügen eines Addierers am Ausgang, sowie die Situation nach deren Ausführung mit anschließendem Layout: *Custom Functions* → *Add Neighbor at...* → *Output Port 1* → *Sum* → *at Input 1*.

4.1.4 ZUSAMMENFASSUNG

Das kontextbasierte Modellieren im Zusammenspiel mit dem in Abschnitt 3 vorgestellten Layoutverfahren fasst beim Erstellen häufig genutzter Konstrukte das Nachschlagen in der Blockbibliothek, sowie das manuelle Einfügen, Verbinden und Platzieren der Blöcke zu einem einzigen Arbeitsschritt zusammen. Das automatische Anordnen der neu



(a) Kontextmenü zur Auswahl des einzufügenden Blocks.

(b) Ergebnis nach dem Einfügen.

Abbildung 4.1: Kontextbasiertes Modellieren.

eingefügten Elemente im Diagramm wird erst nach der eigentlichen strukturellen Modifikation des Modells vom Layoutalgorithmus vorgenommen. Ohne diesen müssten die neuen Elemente mit Hilfe statisch implementierter Muster angeordnet werden, was insbesondere im Umfeld bereits bestehender Diagrammteile problematisch ist.

Obwohl eine umfassende Erprobung noch aussteht, deuten erste Rückmeldungen aus der Praxis darauf hin, dass der Ansatz das Erstellen von SL Diagrammen tatsächlich deutlich erleichtern und beschleunigen kann.

4.2 Transformationen für Simulink Diagramme

Während das kontextbasierte Modellieren insbesondere das Problem des Erstellens von Modellen betrachtet, erscheint mit Verfügbarkeit eines Layoutverfahrens für SL Diagramme auch die Übertragung einiger typischer Refactoring Operationen sinnvoll, um typische Arbeitsschritte beim Bearbeiten bestehender Modelle zu unterstützen.

Im Rahmen dieser Arbeit wurden für zwei Refactoring Operationen aus der textuellen Softwareentwicklung beispielhaft Transformationen für SL Diagramme entwickelt, um diese sinngemäß auf SL Diagramme übertragen. Diese Transformationen sind nachfolgend beschrieben.

4.2.1 Aufbrechen von Subsystemen

Der SL Editor bietet bereits eine einfache Modelltransformation zur Unterstützung des Benutzers beim strukturellen Editieren von Modellen: Das *Erzeugen von Subsystemen*. Diese Funktion verschiebt eine Gruppe

selektierter Blöcke in ein neu erstelltes Subsystem. Dabei wird gewährleistet, dass Verbindungen zu anderen Blöcken über *Inports* und *Outports* des Subsystems geleitet und der vorherigen Semantik entsprechend verbunden werden.

Das *Aufbrechen von Subsystemen* ist die Umkehr dieser Funktion, bei der ein Subsystem-Block durch den Inhalt des von ihm repräsentierten Subsystems ersetzt wird. Sie entspricht sinngemäß der *Inline Method* Refactoring Operation aus der textuellen Softwareentwicklung [32, Seite 117], bei der ein Methodenaufruf durch den Inhalt der Methode ersetzt wird.

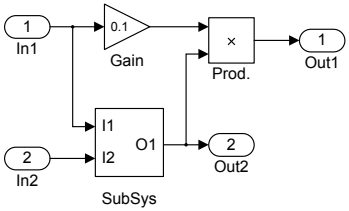
MANUELLES VORGEHEN

Das manuelle Vorgehen beim Aufbrechen von Subsystemen umfasst die folgenden Schritte, deren Erklärung anhand des Beispiels in Abbildung 4.2 erfolgt. Im Beispiel zeigt Abbildung 4.2b das im Block *SubSys* eingebettete Subsystem, während das umgebende Subsystem in Abbildung 4.2a dargestellt wird. Dieses eingebettete Subsystem soll direkt in das umgebende Subsystem integriert werden:

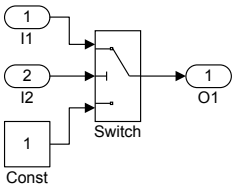
Hierzu sind im Wesentlichen die folgenden manuellen Arbeitsschritte erforderlich:

1. Kopiere den Inhalt des eingebetteten Subsystems in das umgebende Subsystem (siehe Abbildung 4.2c).
2. Für jeden *Inport* und *Outport* Knoten der kopierten Knoten (grauer Bereich in Abbildung 4.2c):
    a) Entferne den aktuellen *Inport* oder *Outport* Knoten.
    b) Verbinde die mit dem entsprechenden Port am *SubSys* Knoten verbundene Kante mit derjenigen Kante, die zuvor mit dem entfernten Knoten verbunden war.
3. Entferne den *SubSys* Knoten (rot).
4. Entferne überschüssige Kantensegmente.
5. Erstelle ein neues Layout (Abbildung 4.2d).

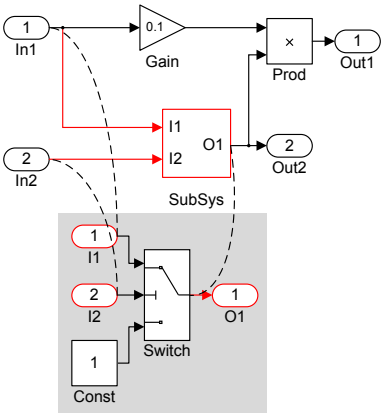
Da der SL Editor beim Einfügen von *Inport* und *Outport* Blöcken dafür sorgt, dass deren Index innerhalb eines Subsystems einzigartig ist, verändert sich bei den eingefügten Blöcken der angezeigte Index, so dass es sehr leicht zu Verwechslungen bei der Zuordnung der Ports kommen kann. In der Abbildung ist diese Anpassung der Indices der Übersichtlichkeit halber nicht dargestellt.



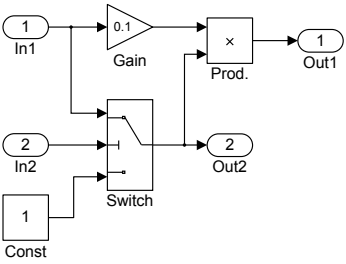
(a) Äußeres Subsystem G.



(b) Inhalt des Subsystems, das durch den Block *SubSys* in G repräsentiert wird: G_s.



(c) Durchführen der Transformation.



(d) G′ nach der Transformation.

Abbildung 4.2: Beispiel zur Transformation *Aufbrechen von Subsystemen*.

AUTOMATISCHE TRANSFORMATION

Eine Transformation zum automatischen *Aufbrechen von Subsystemen* lässt sich mit den Definitionen aus Abschnitt 2.1.1, sowie mit Hilfe von Abbildung 4.2 wie folgt darstellen:

Eingabe sind zwei SL Diagramme, repräsentiert durch $G = (V, E, P)$ und $G_s = (V_s, E_s, P_s)$, wobei G_s das innere Subsystem ist, das durch den Knoten $v_s \in V$ in G repräsentiert wird. Dargestellt werden diese beispielhaft in Abbildung 4.2a und Abbildung 4.2b, wobei *SubSys* im Beispiel den Knoten v_s darstellt.

Die Knoten $V_i \in V_s$ seien die im Subsystem enthaltenen *Inport* Blöcke und $V_o \in V_s$ die *Outport* Blöcke. Diese repräsentieren jeweils eine Verbindung zu einem Port des Knotens v_s im übergeordneten Subsystem, wobei sie

jeweils den Index des zugehörigen Ports $i(p)$ als Attribut $i(v)$ tragen: $i(v) = i(p)$ mit $v \in V_i$ wenn $p \in P_i(v_s)$ und $v \in V_o$ wenn $p \in P_o(v_s)$. In der Darstellung sind dies die Knoten *I1*, *I2*, *O1* sowie die gleichnamigen Ports des Knotens *SubSys*.

Ausgabe der Transformation ist $G' = (V', E', P')$ (siehe Abbildung 4.2d).

Die Knoten V' der Ausgabe setzen sich aus den Knoten des äußeren v und inneren Subsystems V_s zusammen, abzüglich des Subsystem Knotens v_s selbst, sowie der Knoten für die *Inports* und *Outports* im inneren Subsystem. Die in V' nicht enthaltenen Knoten sind in Abbildung 4.2c rot gezeichnet, das ursprüngliche G_s ist grau hinterlegt.

$$V' = (V \setminus v_s) \cup (V_s \setminus (V_i \cup V_o)) \tag{4.1}$$

Die Menge der Kanten E' setzt sich aus vier Teilen zusammen:

1. Diejenigen Kanten aus E, die nicht mit v_s verbunden sind: $E'_1 = \{ e = (u, v) \mid u \neq v_s \ \wedge \ v \neq v_s \}$.

2. Die Kanten aus E_s, die nicht mit *Inport* oder *Outport* Knoten verbunden sind: $E'_2 = \{ e = (u, v) \mid u \notin (V_i \cup V_o) \ \wedge \ v \notin (V_i \cup V_o) \}$.

3. Eine Menge veränderter Kanten E_3, die sich aus der Verbindung der vormalig mit den Eingangsports von v_s verbundenen Kanten mit denen ergeben, die in V_s mit den *Inport* Knoten verbunden waren: Das heißt, für jede Kante $e = (u, v_s) \in F$ die mit dem Eingangsport $p = (F, v_s) \in P_i(v_s)$ an v_s verbunden ist, sei F_s die Menge der Kanten in V_s die mit dem Ausgangsport $p_o = (F_s, w) \in P_s$ des zugehörigen *Inport* Knotens $w \in V_s$ mit gleichem Index $i(w) = i(p)$ verbunden sind. (*Inport* Knoten haben nur einen Port, eine weitere Differenzierung ist also unnötig.) Jede dieser Kanten $f = (w, b) \in F_s, b \in V_s$ wird in eine neue Kante $f' = (u, b) \in E_3$ (u ist der Ausgangsknoten von e) überführt, die mit dem Ausgangsport von u verbunden ist $f' \in F$. Nach Überführung aller Kanten aus F_s wird die aktuelle Kante e aus dem Port p entfernt.

   In der Darstellung ist beispielsweise die Kante von *In1* nach *SubSys.I1* ein Kandidat für e. Mit diesem e wäre dann *SubSys.I1* der Port p mit dem Index $i(p) = 1$, was wiederum bedeutet, dass w der Knoten *I1* ist ($i(w) = i(p) = 1$). Die Menge F_s enthält in diesem Fall nur die eine Kante f von *I1* zum ersten Port von *Switch*. Diese wird zu f' verändert (gestrichelte Verbindung zu *In1*), so dass sie *In1* mit dem ersten Port von *Switch* verbindet.

4. Eine Menge veränderter Kanten E_4, die sich aus der Verbindung der vormalig mit den Ausgangsports von v_s verbundenen Kanten mit denen ergeben, die in V_s mit den *Outport* Knoten verbunden waren. Die Überführung der Kanten erfolgt analog zu E_3.

Damit lässt sich E' darstellen als:

$$E' = E'_1 \cup E'_2 \cup E'_3 \cup E'_4 \tag{4.2}$$

In der Zusammensetzung der Menge der Ports P' sind nur diejenigen Ports in P' enthalten, deren Knoten und Kanten auch in V' und E' enthalten sind, wobei diese teilweise durch die Transformation der Kanten verändert wurden.

$$P' := \{\, p = (F, w) \mid F \subseteq E' \text{ und } w \in V' \,\} \tag{4.3}$$

4.2.2 WAHLFREIES EINFÜGEN UND ENTFERNEN VON PORTS

In SL können mehrere verschiedene Signale mit Hilfe von *BusCreator* Blöcken zu sogenannten Bussen gruppiert werden. Bezüglich ihrer Anwendung sind Busse in SL mit einfachen Strukturen in textuellen Programmen vergleichbar. Auf die Elemente eines solchen Busses wird mit *BusSelector* Blöcken zugegriffen, wobei die Signale anhand ihrer Namen ausgewählt werden. Dabei entspricht die Reihenfolge der jeweiligen Eingangsports am *BusCreator* genau der Reihenfolge, in der die Signale beim *BusSelector* zur Auswahl angezeigt werden. Aus diesem Grund werden Signale häufig, insbesondere bei Bussen mit vielen Signalen, bereits am *BusCreator* nach bestimmten Kriterien gruppiert, um die spätere Auswahl zu erleichtern. Beispielsweise kommen in realen Modellen durchaus Busse mit 200 oder mehr Signalen vor, die zumeist nach semantischen Kriterien sortiert sind: Statusinformationen liegen neben zugehören Daten, funktional zusammengehörige Signale liegen beieinander. Obwohl Busse in SL selbst auch wieder Busse enthalten können, was eine hierarchische Strukturierung ermöglichen würde, ist das Gruppieren anhand der Reihenfolge der Eingangsports, auch innerhalb der Hierarchieebenen, gängige Praxis.

Der SL Editor bietet bis zur aktuellen Version (Matlab/Simulink R2011a) keinen Weg, Ports mit beliebigem Index in *BusCreator* Blöcke einzufügen oder zu entfernen; es sind lediglich Änderungen an der letzten Position möglich: das Hinzufügen eines Ports an letzter Position, oder das Entfernen des letzten Ports. Ähnliche Einschränkungen bestehen auch beim Bearbeiten der Ports bei anderen Blocktypen, beispielsweise bei *Multiplexer* oder *Subsystem* Blöcken. Sowohl das manuelle Vorgehen, als auch die Automatisierung erfolgt analog zu *BusCreator* Blöcken.

Die Transformation lässt sich unter Verwendung der Definitionen aus Abschnitt 2.1.1 wie folgt darstellen: Sei v der *BusCreator* Block mit seinen Eingangsports $P_i(v)$, bei dem an Position $i(p_n) <= |P_i(v)|$ der neue

Eingangsport p_n eingefügt werden soll. Dazu wird der Index aller Ports $p \in P_i(v)$ mit einem Index über $i(p_n)$ um 1 erhöht und der neue Port mit $i(p_n)$ in $P_i(v)$ eingefügt.

Das tatsächliche Vorgehen in SL weicht von der vereinfachten Beschreibung der Transformation ab, da der Index eines Ports eines *BusCreator* Blocks nicht verändert werden kann. Anstatt den Index aller Ports p mit $i(p) >= i(p_n)$ einfach zu erhöhen, müssen die mit dem Port verbundenen Kanten neu verbunden werden. Der Algorithmus dazu ist trivial:

1. Sei n der Index an dem ein freier Port eingefügt werden soll.
2. Füge einen neuen Port p mit $i(p) = |P_i(v)|$ zu $P_i(v)$ hinzu ($|P_i(v)|$ wächst dadurch um eins).
3. Für alle $k \in \mathbb{N}$ von $k = |P_i(v)| - 2$ bis $k = n$ in Schritten von -1, schiebe alle Kanten einen Port nach "unten":
   a) Sei $p_k = (F_k, v) \in P_i(v)$ der Port mit $i(p_k) = k$,
   b) $p_l = (F_l, v) \in P_i(v)$ der Port mit $i(p_l) = k + 1$
   c) und somit F die Menge von Kanten, die mit p_k verbunden sind (in SL enthält diese Menge maximal eine Kante bei Eingangsports),
   d) dann verbinde diese Kante stattdessen mit p_l: $F_l = F_k$, wodurch $F_k = \{\}$ wird.

Das Ergebnis ist ein Port ohne verbundene Kanten an Position n; das Entfernen von Ports erfolgt analog dazu.

Wegen des in Abschnitt 3.5.1 genauer betrachteten Zusammenhangs zwischen Blockgröße, Portanzahl und Portposition bei SL Blöcken, resultiert die Veränderung der Portanzahl unmittelbar in einer Veränderung der Portpositionen. Dieser Umstand, zusammen mit dem Neuverbinden der Kanten in der Transformation, macht Anpassungen am Layout zum Erhalt der Lesbarkeit unmittelbar erforderlich.

Bisher mussten sowohl die Transformation selbst, als auch die Anpassung des Layouts, manuell vorgenommen werden. Selbst unter der Annahme, dass das Neuverbinden der Kanten und die notwendigen Layoutanpassungen zusammen lediglich 10 Sekunden pro verändertem Port benötigen, würde das Einfügen eines Ports in die Mitte eines *BusCreator* Blocks mit 40 Eingangsports bereits über drei Minuten dauern - bei realen Modellen kommen durchaus *BusCreator* Blöcke mit über 200 Ports zum Einsatz.

Da der Großteil dieses Aufwands auf die abschließenden Layoutanpassungen entfällt, ist die Automatisierung der Transformation erst im Zusammenhang mit dem in Abschnitt 3 vorgestellten Algorithmus zum

automatischen Layout von SL Diagrammen wirklich sinnvoll: Für typische Diagramme dauert die Transformation einschließlich Layout weniger als eine Sekunde.

Teil III

REALISIERUNG UND ERPROBUNG

5 REALISIERUNG

Der in Abschnitt 3 vorgestellte Layoutalgorithmus wurde auf Basis der Datenstruktur von KLay Layered (Abschnitt 2.3.1) implementiert und in eine eigenständige Applikation, den sogenannten *Layout Server*, eingebunden. Ein in der Matlab Skriptsprache *M* [62] implementierter *Layout Client* nimmt Benutzerkommandos über die SL GUI entgegen, liest die benötigten Informationen aus dem Modell aus und übermittelt diese zum *Layout Server*. Dieser erstellt das geforderte Layout und gibt die Layoutinformationen zurück. Die Übertragung der Layoutinformationen und Parameter zwischen *Layout Client* und *-Server* erfolgt jeweils XML kodiert über eine Transmission Control Protocol [52] (TCP) *Stream Socket* Verbindung. Nach dem Empfang aktualisiert der *Layout Client* das Modell mit dem neuen Layout. Siehe dazu Abbildung 5.1.

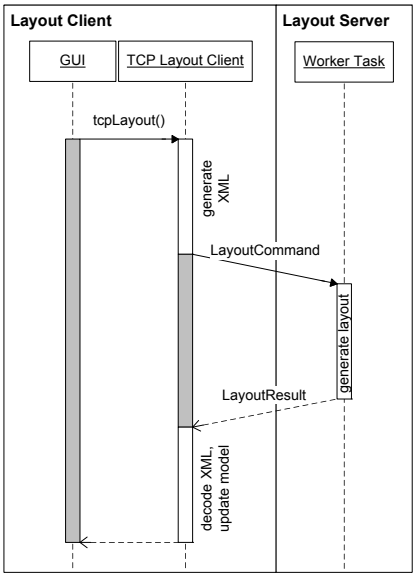


Abbildung 5.1: Sequenzdiagramm eines Layoutvorgangs.

In den nachfolgenden Abschnitten werden der *Layout Client* und *Layout Server* sowie die Kommunikation zwischen diesen beschrieben.

5.1 CLIENT–SERVER KOMMUNIKATION

Die Kommunikation zwischen *Layout Client* und *Layout Server* erfolgt über TCP *Stream Sockets*. Diese stellen verlässliche, bidirektionale serielle Kanäle zur Übertragung von Zeichen-Datenströmen zur Verfügung. Für jeden Layoutvorgang wird vom *Layout Client* eine neue Verbindung aufgebaut und nach Abschluss des Vorgangs wieder geschlossen. Ein vorzeitiger Abbruch der Verbindung (beispielsweise durch den Benutzer) von Seiten des *Layout Client*s setzt den *Layout Server* zurück, ein Verbindungsabbruch des *Layout Server*s signalisiert einen Fehler – der *Layout Client* führt dann keine Veränderungen am Modell durch.

Zum Übermitteln der Struktur und des aktuellen Layouts des Diagramms sowie zum Übertragen des fertigen Layouts, wurden zwei XML Datenpakete definiert: Die *Layoutanforderung*, mit der der *Layout Client* ein neues Layout beim *Layout Server* anfragt, und die *Layoutantwort*, mit dem der *Layout Server* das neue Layout zurücksendet. Die Struktur der Pakete ist nachfolgend anhand von Unified Modeling Language [70] (UML) Klassendiagrammen beschrieben.

5.1.1 LAYOUTANFORDERUNG

Jede *Layoutanforderung* (*LayoutCommand*, siehe Abbildung 5.2) enthält drei wesentliche Komponenten: Die Art der Anforderung (*cmd*) als Enumeration, eine optionale Liste mit Serverparametern, sowie die vom *Layout Server* benötigten Informationen über den Graphen. Parameter werden durch ihre Bezeichnung identifiziert und enthalten die zugewiesenen Werte als Zeichenkette kodiert. Der Graph besteht bei der *Layoutanforderung* lediglich aus einer Menge von Knoten (*Blocks*). Kanten werden hier nicht explizit als eigene Objekte dargestellt, sie sind implizit in der Beschreibung der Ports enthalten. Jeder Knoten ist durch eine BlockID eindeutig gekennzeichnet. Die wichtigsten Felder der Knoten sind nachfolgend beschrieben:

POSITION, AUSRICHTUNG UND GRÖSSE

Position und Größe eines Knotens werden als X-Y Koordinaten in den Feldern *pos* und *size* gespeichert. Die Ausrichtung des Knotens (*flipped* Status, siehe Abschnitt 3.1.1) wird im Feld *reversed* als Boolean übernommen.

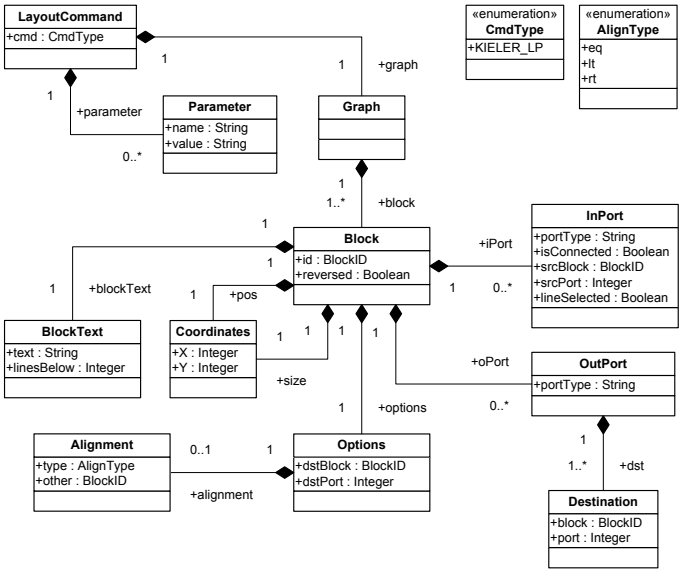


Abbildung 5.2: Klassendiagramm eines Layoutkommandos.

PORTS

Eingangs- (*InPort*) und Ausgangsports (*OutPort*) werden geordnet in den Feldern *iPort* und *oPort* abgelegt. Der SL *PortType* wird im Feld *portType* übergeben. Er enthält bei normalen Ports den Index, bei besonderen Ports wie beispielsweise *Trigger* oder *Enable* Ports ein Schlüsselwort. Ausgangsports enthalten zudem im Feld *dst* eine Liste mit verbundenen Zielknoten (*dst.block*) und -ports (*dst.port*). Da Eingangsports bei SL mit höchstens einer Kante verbunden sind (*isConnected*), reicht hier die Angabe eines einzelnen Quellknotens (*srcBlock*) und -ports (*srcPort*). Das Feld *lineSelected* gibt darüber hinaus bei Eingangsports Aufschluss darüber, ob die mit dem Port verbundene Kante in SL angewählt war oder nicht.

LAYOUTOPTIONEN

Im Feld *options* werden die zusätzlichen Benutzervorgaben zum Layout abgelegt. Benutzerdefinierte Constraints zur horizontalen Anordnung des Knotens (siehe Abschnitt 3.2.3) werden als Liste im Feld *alignment* unter den Optionen abgelegt. Die Elemente dieser Liste enthalten jeweils die Art des Constraints (siehe Tabelle 5.1) im Feld *type* und die ID des anderen Knotens, auf den sich das Constraint bezieht, im Feld *other*.

AlignType	Bedeutung
eq	Der Knoten soll im *Layer* des anderen Knoten liegen.
lt	Der Knoten soll links vom anderen Knoten liegen.
rt	Der Knoten soll rechts vom anderen Knoten liegen.

Tabelle 5.1: Bedeutung des Enumerate AlignType.

5.1.2 LAYOUTANTWORT

Die *Layoutantwort* (*LayoutResult*) beinhaltet insbesondere die neuen Layoutinformationen im Feld *graph*. Zusätzlich werden im Feld *stats* Informationen über den *Layout Server* übertragen, wobei in der aktuellen Implementierung lediglich die für das aktuelle Layout benötigte Laufzeit in Milisekunden enthalten ist. Diese Layoutinformationen enthalten die Informationen zu den Knoten als Menge im Feld *block*, zu den Kanten als Menge im Feld *line*:

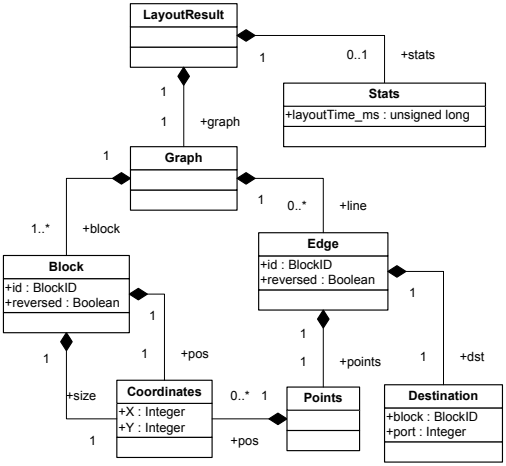


Abbildung 5.3: Klassendiagramm einer Layoutantwort.

KNOTEN

Zu jedem Knoten wird analog zur Layoutanfrage eine neue Position (im Feld *pos*) und Größe (im Feld *size*) als X-Y Koordinaten übertragen. Die eindeutige Zuordnung zwischen Layoutinformation und Knoten ermöglicht die aus der Layoutanfrage übernommene eindeutige *BlockID*.

KANTEN

Der Layouter betrachtet Kanten so, wie sie in Abschnitt 2.1.1, Def. 6 beschrieben sind. Für jede dieser Kanten wird eine (geordnete) Liste von X-Y Koordinaten im Feld *points* übertragen, die die Stützpunkte der Kante bilden. Da sich eine Kante in dieser Form eindeutig durch ihren Zielport und -knoten identifizieren lässt, werden diese im Feld *dst* an den *Layout Client* übergeben.

5.2 LAYOUT SERVER

Der *Layout Server* wurde als eigenständige Applikation in Java realisiert. Dadurch konnten bestehende Algorithmen, wie die ebenfalls in Java umgesetzten Implementierungen aus KLay Layered, direkt eingebunden werden. Gleichzeitig konnte die Anbindung externer Werkzeuge, wie die verschiedenen LP-Solver, im Vergleich zu einer Umsetzung innerhalb von SL einfach gehalten werden. Zusätzlich erleichterte die Verwendung bewährter Entwicklungswerkzeuge wie Eclipse sowohl Entwicklung als auch Debugging.

Der *Layout Server* besteht aus vier wesentlichen Komponenten: Die Grundlage bildet die eigentliche Server Komponente in *com.dcaiti.layouter.server*. Diese stellt die Kommunikation mit SL zur Verfügung, dekodiert *Layoutanforderungen* und Parameter und kodiert *Layoutantworten*. Das eigentliche Erstellen des neuen Layouts wird von der Komponente *Layoutalgorithmus* übernommen. Die Anbindung an die verschiedenen LP-Solver erfolgt über den *Solver Adapter*, die Parametrierung und Statusanzeige über die GUI (*com.dcaiti.layouter.gui*).

5.2.1 LAYOUTALGORITHMUS

Die Implementierung des Layoutalgorithmus in *com.dcaiti.layouter.kieler* setzt auf der in Zusammenarbeit mit der KIELER Arbeitsgruppe entwickelte Datenstruktur KLay Layered auf. Abbildung 5.4 zeigt ein vereinfachtes UML Klassendiagramm dieser Datenstruktur.

Sie setzt die in Abschnitt 2.2.3 beschriebenen Phasen des Algorithmus gemäß Struktur in KLay Layered jeweils als Klassen mit vorgegebenen Schnittstellen um:

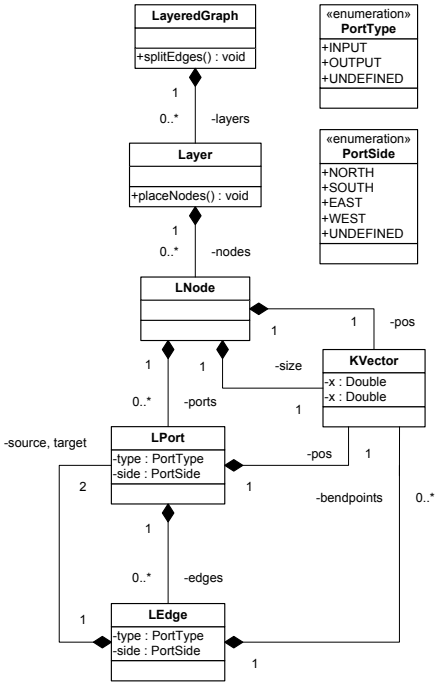


Abbildung 5.4: Vereinfachte Darstellung der KLay Datenstruktur.

PHASE 1: ZYKLENERKENNUNG UND -BEHANDLUNG

Die wesentliche Funktionalität dieser Klasse wird durch die Methode *breakCycles* (Listing 5.1) umgesetzt.

Die Zyklenerkennung erfolgt dabei in drei Schritten:

1. Erkennung von Zykluskanten im Umfeld gespiegelter (*flipped*) Knoten.
2. Erkennung weiterer Zykluskanten mit Hilfe der Zyklenerkennung aus KLay Layered in *de.cau.cs.kieler.klay.layered.impl.GreedyCycleBreaker*.
3. Erkennung von Kanten, die einen Knoten v mit sich selbst verbinden: $E_{selfloop} = \{ e = (v, v) \in E \}$.

Diese Kanten werden dann mittels *Inverter* Knoten umgekehrt.

```
1 public void breakCycles(Collection<LNode> nodes) {
2     handleReversed(nodes);
3     cycleEdges = findCyclesGreedy(nodes);
4     addSelfloopEdges(cycleEdges, nodes);
5     IEdgeInverter edgeInverter = new
          InverterNodeEdgeInverter();
6     edgeInverter.invertEdges(cycleEdges, nodes);
7 }
```

Listing 5.1: InverterNodeCycleBreaker.breakCycles

```
1 public void layer(Collection<LNode> nodes, LayeredGraph
      layeredGraph) {
2     calculateLayers(nodes, layeredGraph);
3     if (null != postProcessor)
4         postProcessor.postProcessLayers(layeredGraph);
5 }
```

Listing 5.2: EdgeLengthLayerer.calculateLayers

PHASE 2: HIERARCHISIERUNG

Die Hierarchisierung wurde als Minimierung der horizontalen Kantenlängen mit den in Abschnitt 3.2 beschriebenen Erweiterungen umgesetzt. Die eigentliche Hierarchisierung mit Constraints erfolgt in der Methode *calculateLayers*, das Einfügen zusätzlicher *Layer* für *Inverter* Knoten (siehe Abschnitt 3.2.2) schließt sich als optionale Nachbearbeitung an (Listing 5.2).

PHASE 3: KREUZUNGSREDUKTION

Für die Kreuzungsreduktion konnte die in KLay Layered vorliegende Implementierung des Algorithmus von Spönemann *et al.* [84] eingebunden werden (*de.cau.cs.kieler.klay.layered.impl.LayerSweepCrossingMinimizer*).

PHASE 4: KNOTENPOSITIONIERUNG

In dieser Phase ist die in Abschnitt 3.5.2 beschriebene Kantenbegradigung für SL in der Klasse *LpNodePlacer* umgesetzt (Listing 5.3).

Die eigentliche Berechnung der Kantenbegradigung wird in Zeile 2, *straighten(layeredGraph)*, ausgeführt; die Ergebnisse des linearen Programms werden dann in Zeile 3 in die Größen und Positionen der Knoten

```
1 public void placeNodes(LayeredGraph layeredGraph) {
2    LinearProgram best = straighten(layeredGraph);
3    updateNodesY(best, layeredGraph);
4    updatePortPositions(layeredGraph);
5    updateGraphAndLayerSizes(layeredGraph);
6 }
```

Listing 5.3: LpNodePlacer.placeNodes

übertragen. In Zeile 4 werden die Portpositionen für die neuen Knotengrößen aktualisiert, im letzten Schritt können dann die Größen der *Layer* und des Graphen aktualisiert werden.

PHASE 5: KANTENROUTING

Da die Portpositionen bereits in der vorherigen Phase gemäß der (linearisierten) Berechnungsvorschrift für die Portpositionierung in SL Diagrammen aktualisiert wurden, konnte für das abschließende Kantenrouting direkt die Implementierung eines rechtwinkligen Kantenroutings aus KLay Layered verwendet werden:
de.cau.cs.kieler.klay.layered.impl.OrthogonalEdgeRouter.

5.2.2 SOLVER ADAPTER

Zum Lösen der linearen Programme stehen verschiedene LP-Solver zur Verfügung. Die Solver Adapter Komponente *com.dcaiti.layouter.solver* des *Layout Server* stellt eine einheitliche Schnittstelle zu den in Abschnitt 2.4.2 aufgeführten LP-Solvern (SCIP 1.2.0 [2], Gurobi 3.0.3 [44] und lp_solve 5.5 [27]) zur Verfügung. Die Anbindung der LP-Solver erfolgt über die Schnittstelle *SolverAdapter* (siehe Listing 5.4).

Die Grundlage dieser Schnittstelle bildet die Datenstruktur zur Speicherung linearer Programme (*com.dcaiti.layouter.solver.LinearProgram*), dargestellt in Abbildung 5.5. Darin wird das lineare Programm als eine Menge von Variablen und Constraints sowie einer Zielfunktion dargestellt. Beim Erstellen des linearen Programms wird zu jeder Variablen gespeichert, welche Skalierung sie hat, ob es sich um eine Integer Variable handelt und welche oberen und unteren Schranken sie einhalten muss. Nach der Optimierung liegen dort zusätzlich die optimierten Werte der Variablen vor. Constraints und Zielfunktion setzen sich jeweils aus einem oder mehreren Koeffizienten zusammen, die einer Variablen jeweils einen Faktor zuweisen. Die Zielfunktion entspricht der Summe der mit den Faktoren

```
1  public abstract class SolverAdapter {
2     public enum SolverState
3        {INTEGER, OPTIMAL, INFEASIBLE, UNKNOWN, UNSOLVED};
4     public enum SolverType { LPSOLVE, GUROBI, SCIP };
5
6     public abstract void solve(LinearProgram lp);
7     public abstract boolean canSolveInteger();
8     public static SolverAdapter getSolverAdapter(SolverType
          type) {
9        switch(type) {
10       case LPSOLVE: return new SolverAdapterLPSOLVE();
11       case GUROBI: return new SolverAdapterGUROBI();
12       case SCIP: return new SolverAdapterSCIP();
13       default:
14          return getSolverAdapter(SolverType.values()[0]);
15       }
16    }
17 }
```

Listing 5.4: com.dcaiti.layouter.solver.SolverAdapter

gewichteten Variablen, die Constraints enthalten zusätzlich einen Vergleichsoperator sowie eine Konstante. Nach Abschluss der Optimierung werden in der Datenstruktur die vom LP-Solver gelieferten Informationen zum Zustand der Optimierung gespeichert.

5.3 LAYOUT CLIENT

Der *Layout Client* ist der in der ML Skriptsprache *M* implementierte SL-seitige Teil des Layouters. Er besteht aus den Layout Funktionen selbst sowie der Einbindung in die SL GUI, die auch den Zugriff auf die in Abschnitt 4 beschriebenen Funktionen zur Modellierungsunterstützung ermöglicht.

5.3.1 LAYOUT FUNKTIONEN

Der Ablauf eines Layoutvorgangs wurde bereits am Anfang dieses Abschnitts in Abbildung 5.1 dargestellt. Die folgenden Arbeitsschritte werden dabei vom *Layout Client* übernommen: Das Auslesen von Struktur, Optionen und derzeitigem Layout aus dem Modell, die Kommunikation mit dem *Layout Server* und das Aktualisieren des Modells mit dem neuen Layout.

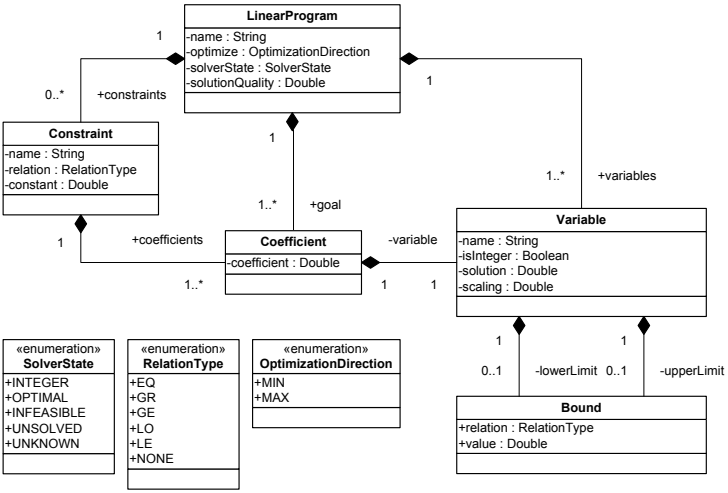


Abbildung 5.5: Datenstruktur zur Speicherung Linearer Programme.

Der *Layout Client* stellt zudem eine eigene Undo-Funktion für Layoutänderungen zur Verfügung, da die SL-eigene Undo-Funktion programmatisch veränderte Modelleigenschaften nur teilweise rückgängig machen kann. Die Undo-Funktion des *Layout Client* ermöglicht das Zurücksetzen des Layouts in den Zustand vor dem letzten Aufruf des Layouters.

Sowohl die Layout Funktionen als auch die Undo-Funktion des *Layout Client* wurden dabei so umgesetzt, dass schreibende Zugriffe auf das Modell nur an wenigen Stellen im Programm erfolgen und ausschließlich die Parameter für Knotenposition, Knotengröße und Linienpositionen betreffen. Dadurch kann sichergestellt werden, dass sich durch das Erstellen eines Layouts auch bei einem Fehler seitens des Layouters keine *funktionalen* Änderungen am Modell ergeben – eine wichtige Voraussetzung für den späteren Einsatz des Layouters in der Praxis, beispielsweise im Rahmen der Erprobung (siehe Abschnitt 6.1).

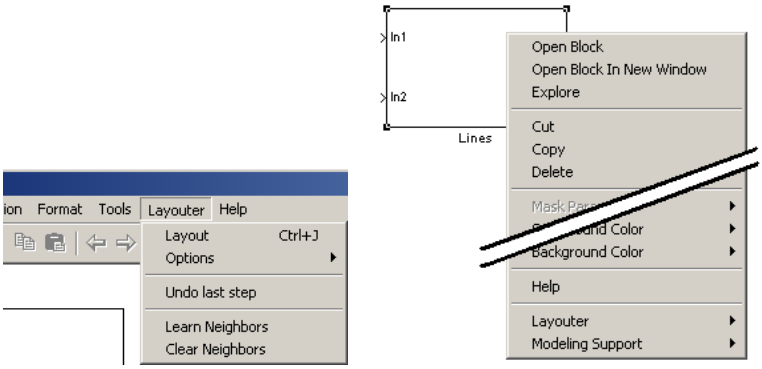
SL stellt Hyperkanten intern als binäre Bäume dar. Dadurch wird indirekt auch das Kantenrouting beeinflusst, da es zum Vertauschen von Abzweigungspunkten bei Hyperkanten mit mehr als zwei Segmenten erforderlich ist, die Kante teilweise zu entfernen und neu zu erstellen. Dies stellt eine strukturelle Änderung am Modell dar und würde damit der Einschränkung der Layout Funktion auf die Layoutparameter des Modells widersprechen. Beim Aktualisieren des Modells werden die Abzweigungspunkte von Hyperkanten daher vom *Layout Client* nicht vertauscht, sondern im Widerspruch zu Punkt 5 von Modellierungsrichtlinie `db_0032` (siehe Abschnitt 2.1.4) auf einer Position zusammengefasst.

```
1 function sl_customization(cm)
2     cm.addCustomMenuFcn('Simulink:MenuBar',
          @getLayoutMenuSchemas);
3     cm.addCustomMenuFcn('Simulink:ContextMenu',
          @getContextMenuSchemas);
4 end
```

Listing 5.5: sl_customization

5.3.2 GUI INTEGRATION

Der *Layout Client* wird an zwei Stellen in die SL GUI eingebunden, als zusätzlicher Eintrag *Layouter* in der Menüleiste des SL Editors und als Eintrag *Layouter* im Kontextmenü eines angewählten Blockes. Zusätzlich wird im Kontextmenü ein Eintrag *Modeling Support* für den Zugriff auf die Funktionen zur Modellierungsunterstützung eingebunden. Abbildung 5.6 zeigt Screenshots dieser zusätzlichen Einträge.



(a) Eintrag *Layouter* in der Menüleiste von SL.

(b) Einträge für Layouter und Modellierungsunterstützung im Kontextmenü eines gewählten Blocks.

Abbildung 5.6: Einträge des *Layout Client* in der SL Menüleiste und im Kontextmenü eines Blocks.

Das Einbinden zusätzlicher Einträge in die Menüleiste und in Kontextmenüs wird in SL mit Hilfe der *sl_customization* Schnittstelle über sogenannte *Schema* Funktionen durchgeführt. Listing 5.5 zeigt die Hauptfunktion der *sl_customization.m* des *Layout Client*, in der die *Schema* Funktionen der obersten Menüebenen registriert werden.

MENÜLEISTE

Das Untermenü *Layouter* in der Menüleiste bietet Zugriff auf die globalen Funktionen des Layouters und der Modellierungsunterstützung:

- *Layout* Das Erstellen eines neuen Layouts; optional wird nur ein Layout für selektierte Blöcke und Kanten erstellt.
- *Options* Globale Layout Optionen, insbesondere das Zurücksetzen aller Layout Optionen im aktuellen Diagramm.
- *Undo last step* Die Undo-Funktion des *Layout Client*.
- *Learn / Clear Neighbors* Das Anlernen und Zurücksetzen des *kontextbasierten Modellierens*. Diese Funktion muss zuvor in der Konfigurationsdatei aktiviert werden, da normale Benutzer sie nicht benötigen.

KONTEXTMENÜ

Das Kontextmenü eines Blocks wird um zwei zusätzliche Einträge ergänzt: *Layouter* und *Modeling Support*. Das *Layouter* Untermenü bietet alle Einträge des *Layouter* Untermenüs in der Menüleiste, wobei das *Options* Untermenü um zusätzliche lokale Optionen ergänzt wird:

- *Fix Block Size* Ist diese Option gesetzt, bleibt die derzeitige Knotengröße des aktuellen Knotens beim automatischen Layout erhalten.
- *Alignment* Sind mehrere Blöcke selektiert, können mit diesem Eintrag Constraints für die Hierarchisierung festgelegt werden.

Das Untermenü *Modeling Support* bietet Zugriff auf die in Abschnitt 4 beschriebenen Funktionen zur Modellierungsunterstützung:

1. *Add Neighbor* ermöglicht die Auswahl des einzufügenden Blocks beim *kontextbasierten Modellieren*.
2. *Insert/Remove Port* ermöglicht das *wahlfreie Einfügen (und Entfernen) von Ports* in *BusCreator* und *Subsystem* Blöcken.
3. *Break Subsystem* bietet Zugriff auf die Funktion zum *Aufbrechen von Subsystemen*.

Es werden jeweils nur diejenigen Menüeinträge angezeigt, die für den gewählten Block anwendbar sind.

6 ANALYSE UND ERPROBUNG

Die Beurteilung von Layoutalgorithmen wird typischerweise anhand quantitativer Vergleiche verschiedener Layouts mit Hilfe von Metriken oder anhand von analytischen Untersuchungen vorgenommen. Die Metriken für quantitative Untersuchungen erfassen dabei Kriterien wie die Anzahl der Kantenkreuzungen, Kantenlängen, Diagrammgröße oder Laufzeit für unterschiedlich erstellte Layouts verschiedener Graphen. Bei Untersuchungen zur Lesbarkeit oder zum Wiedererkennungswert, vor allem bei wiederholten Layouts von Diagrammen mit kleinen Änderungen, stützt man sich zur Datenerhebung auf Fragebögen, sucht jedoch ebenfalls meist nach quantitativen Aussagen. Analytische Untersuchungen weisen im Gegensatz dazu bestimmte Eigenschaften des Algorithmus oder der dadurch erstellten Layouts nach, beispielsweise die Komplexität des Algorithmus oder eine obere Grenze für die benötigte Fläche des Layouts.

Bei der Beurteilung eines erstmalig für eine graphische Modellierungssprache wie SL angepassten Layoutalgorithmus sowie darauf basierender Unterstützungsfunktionen, stellt sich jedoch nicht nur die Frage nach einem quantitativen Vergleich mit bereits bestehenden Algorithmen, sondern auch nach qualitativen Aussagen zur praktischen Anwendbarkeit und zum tatsächlichen Nutzen beim Erstellen und Bearbeiten von Modellen.

6.1 PRAXISERPROBUNG

Zur Ermittlung des praktischen Nutzens des Algorithmus wurde eine Evaluierung im realen Entwicklungsumfeld über eine Dauer von etwa 6 Monaten durchgeführt. Dazu wurde die in Abschnitt 5 vorgestellte Implementierung etwa 30 Probanden aus verschiedenen Bereichen der Vor- und Serienentwicklung des Daimler-Konzerns zum Einsatz während der täglichen Arbeit mit SL Modellen zur Verfügung gestellt. Die Probanden wurden zu Beginn der Erprobung in die Benutzung des Werkzeugs eingewiesen. Zusätzlich stand eine Bedienungsanleitung zur Verfügung. Da für die Erprobung kein festes Zeitbudget zugesichert werden konnte, blieb es den Probanden frei gestellt, ob und in welchem Umfang sie das Werkzeug im Verlauf der Erprobung nutzen. Am Ende der Erprobung

wurden die Probanden zu ihren Erfahrungen mit dem Werkzeug befragt, wobei in einigen Bereichen die Erfahrungen der Probanden aus organisatorischen Gründen bereits vor der eigentlichen Befragung intern gesammelt und zusammengefasst wurden.

In den folgenden Abschnitten wird zunächst der Aufbau der Befragung in Abschnitt 6.1.1 erläutert, nachfolgend die Erfahrungen der Probanden anhand positiver und negativer Kernaussagen in Abschnitt 6.1.2 zusammengefasst und abschließend in Abschnitt 6.1.3 ausgewertet.

6.1.1 STRUKTUR DER BEFRAGUNG

Die Befragung der Probanden wurde in Form von teilstandardisierten Befragungen (semi-structured interviews) [83, Kapitel 2.2] durchgeführt. Die teilstandardisierte Befragung ist bezüglich der Kontrolle über den Verlauf zwischen der strukturierten und der freien Befragung anzuordnen:

Bei einer strukturierten Befragung folgt der Fragende einer vorher festgelegten Vorgehensweise, beispielsweise durch einen festen Fragenkatalog. Dadurch lassen sich zwar einerseits die gewonnenen Antworten gut miteinander vergleichen und bewerten, andererseits ist während der Befragung nur wenig Flexibilität möglich, um beispielsweise detaillierter auf unvorhergesehene Antworten eingehen zu können. Sie eignet sich besonders für quantitative Erhebungen oder in Fällen, bei denen nur mit wenig inhaltlichen Erkenntnissen gerechnet wird. Die freie Befragung weist im Gegensatz dazu, bis auf eventuelle Startfragen, keine feste Fragen auf. Sie gleicht stattdessen eher einem themenbezogenen Gespräch zwischen dem Befragten und dem Fragenden. Dieses wird in hohem Maße aus den Antworten des Befragten heraus während der Befragung entwickelt und ermöglicht es so, auch unvorhergesehene Sachverhalte zu beleuchten und neue inhaltliche Erkenntnisse zu gewinnen.

Eine teilstandardisierte Befragung soll sich im Vergleich dazu wie ein geführtes Gespräch entwickeln. Sie folgt üblicherweise einem vorher entwickelten thematischen Leitfaden mit relevanten Kernfragen zu einzelnen Themenbereichen. Um das Gewinnen neuer inhaltliche Erkenntnisse zu unterstützen, greift der Befragende jedoch aktiv die Antworten des Befragten auf, um aus diesen den weiteren Gesprächsverlauf auch über den Leitfaden hinaus zu entwickeln. Dabei wird dem Befragten ausdrücklich die Freiheit gelassen, auch auf nicht durch den Leitfaden abgedeckte Fragestellungen und Themen einzugehen.

In den durchgeführten Befragungen wurde zunächst der Ablauf erläutert und die Fragen aus dem Themenbereich "Informationen zum Probanden" in strukturierter Form gestellt. Danach wurde die Befragung zu den

anderen Themenbereichen durchgeführt, ohne dabei fest der Reihenfolge der Themenbereiche aus dem Leitfaden zu folgen. Ein Springen zwischen Themenbereichen war daher im zweiten Teil der Befragung jederzeit möglich.

Der Leitfaden zur Befragung deckte die folgenden Themenbereiche ab:

INFORMATIONEN ZUM PROBANDEN

Da die Probanden in unterschiedlichen Positionen der Vorentwicklung und Entwicklung tätig sind und dementsprechend unterschiedliche Aufgaben im Umfeld der modellbasierten Entwicklung mit SL ausüben, soll in diesem Themengebiet zunächst der Kontext und Umfang der vom Probanden gemachten Erfahrungen eingeschätzt werden.

1. Welche Position und welche Aufgabe hat der Proband?
2. In welchem Umfang wurde während der Erprobungszeit aktiv modelliert?
3. Wie häufig wurde das Werkzeug eingesetzt?
4. Das Werkzeug wurde in den folgenden Situationen benutzt:
   a) Beim manuellen Bearbeiten von Diagrammen? (ja/nein)
   b) Beim automatischen Erzeugen von Diagrammen? (ja/nein) (was)
   c) Beim automatischen Ändern von Diagrammen? (ja/nein) (was)
5. Die folgenden Optionen des Werkzeugs wurden genutzt:
   a) Layout ganzer Diagramme? (ja/nein)
   b) Layout von Diagrammteilen? (ja/nein)
   c) Priorisierung von Linien? (ja/nein)
   d) Abschalten der automatischen Größenveränderung von Blöcken? (für alle Blöcke/einzelne Blöcke/nein)
   e) Einstellen der Stärke (COMPACT, BALANCED, STRAIGHT) der Kantenbegradigung? (ja/nein)
   f) Anpassung anderer Parameter? (welche)

BEDIENBARKEIT

In dieses Themengebiet fallen Aussagen zur Bedienbarkeit des Werkzeugs, einschließlich seiner Geschwindigkeit.

1. War eine flüssige Benutzung möglich?
2. Interaktives Layout und Benutzervorgaben?
3. Ist die Parametrierbarkeit angemessen?
4. Ist die Geschwindigkeit ausreichend?
5. Was würden sie an der Bedienung Verbessern?
6. Weitere Punkte?

NUTZEN DES WERKZEUGS

Aussagen zum Nutzen des Werkzeugs durch den Probanden fallen in dieses Themengebiet.

1. Konnte eine Erleichterung der Arbeit festgestellt werden?
2. Konnte eine Beschleunigung der Arbeit festgestellt werden? Wenn ja: Schätzung.
3. Nutzen für automatische Modelltransformationen?
4. Würden sie das Werkzeug anderen Kollegen empfehlen?
5. Welche zusätzlichen Funktionen, Eigenschaften oder Änderungen würden das Werkzeug für den praktischen Einsatz nützlicher machen?
6. Weitere Punkte?

GRENZEN

In diesem Themengebiet sollen die Grenzen sowohl des Layoutalgorithmus, als auch der aktuellen Implementierung gesammelt werden.

1. Grenzen bezüglich der Diagrammgröße?
2. Für welche Strukturen eignet sich der Layouter nicht?
3. Weitere Punkte?

6.1.2 Erfahrungen aus der Erprobung

Dieser Abschnitt fasst anhand von Kernaussagen die Erfahrungen der Probanden aus der Erprobung des Layouters zusammen. Einige der dargestellten Aussagen mussten zur besseren Verständlichkeit sprachlich überarbeitet werden.

Die grundsätzliche Einschätzung der Probanden zur Relevanz einer Layoutunterstützung für die grafische modellbasierte Entwicklung deckte sich bei allen Probanden mit der Annahme vom Anfang dieser Arbeit (siehe Abschnitt 1.1). Ein Proband fasst seine Einschätzung dazu wie folgt zusammen:

> Simulink-Programmieren ist hauptsächlich "Schönzeichnen". Der Layouter übernimmt diese Arbeit und der Modellierer kann sich ganz auf die Funktion konzentrieren.
>
> Teamleiter in der System- und Softwareentwicklung

Den Nutzen des Layouters sahen die Probanden hauptsächlich in der Erleichterung beim Modellieren und in der sich dabei ergebenden Zeitersparnis. Von denjenigen Probanden, die damit während der Erprobung beschäftigt waren, wurde zudem hervorgehoben, dass der Layouter beim automatischen Generieren von Modellen besonders hilfreich war. Als weiteren Nutzen nannten die Probanden ein einheitlicheres Aussehen der Modelle bei konsequenter Anwendung des Layouters.

> Es könnte für ein einheitliches Aussehen der Modelle sorgen.
>
> Supporter Toolkette

Erleichterung der Arbeit

Die Erleichterung der Arbeit, insbesondere beim Bearbeiten von kleinen Modellen, wurde von den meisten Probanden als Hauptnutzen des Layouters genannt.

> Bei 08/15-Diagrammen muss man nur noch aufs Knöpfchen drücken – das funktioniert gut.
>
> Softwareverantwortlicher für ein Steuergerät

Als "kleine" oder "08/15" Diagramme bezeichneten die Probanden auf Nachfrage Diagramme mit weniger als 40 Knoten, wobei einige Probanden die Grenze eher bei 20 Knoten sahen und andere damit "alle Diagramme mit Ausnahme ganz großer Diagramme" (mit weit über 100 Knoten) meinten. Diagramme dieser Größe machen, gemessen an den untersuchten Modellen des Fahrzeug-Innenraumbereichs (siehe Abschnitt 2.1.2), etwa 80% aller Diagramme aus.

> Bei kleineren Sachen, oder wenn man schnell mal was ausrichten will, erleichtert es die Arbeit.
> Software-Integrator im Bereich Fahrerassistenzysteme

Viele Probanden merkten während der Befragung an, dass sich der Layouter insbesondere für "Neu erstellte Diagramme" gut eignet:

> Bei neuen Modellteilen ist der Nutzen am Größten.
> Entwickler von Fahrzeugmodellen für Testumgebungen

Diese pauschale Beschränkung auf neue Diagramme war zunächst unerwartet, da sich einerseits neu erstellte Diagramme strukturell nicht von bestehenden Diagrammen unterscheiden und andererseits der Aufwand für Layoutanpassungen beim Ändern von Diagrammen nach Aussage der Probanden ähnlich ist, wie beim neu Erstellen.

> Bei nicht vorher bearbeiteten Subsystemen hat das Werkzeug den größten Nutzen, da ist man deutlich schneller.
> Supporter Toolkette

Im Verlauf der jeweiligen Gespräche zeigte sich, dass diese Unterteilung vor allem auf qualitative Unterschiede zwischen den manuell und automatisch erstellten Layouts zurückzuführen war und nicht auf die eigentliche Zeitersparnis: Bei neu erstellten Diagrammen hat der Proband noch kein bestehendes Layout im Kopf und überlässt dessen Erstellung dem Layouter. Die Akzeptanz des erstellten Layouts ist dann in der Regel gut. Bei Änderungen an bestehenden Diagrammen mit manuell erstelltem Layout ist jedoch bereits das bestehende Layout bekannt, wird in der Regel als gut angesehen und bestimmt das geistige Abbild der jeweiligen Funktion beim Probanden (*Mental Map* [68, 76]) maßgeblich mit. Der Proband hat zudem oft eine klare Vorstellung davon, wie das Layout nach der Änderung aussehen sollte. Wenn das automatisch erstellte Layout dann von dieser Vorstellung abweicht, wird es als weniger gut empfunden, wodurch der Layouter insgesamt in diesen Situationen weniger nützlich erscheint.

> Bei bestehenden Modellen ergibt sich meistens keine Verbesserung des Layouts.
> Entwickler von Fahrzeugmodellen für Testumgebungen

ZEITERSPARNIS

Neben der Erleichterung der Arbeit bestätigten die meisten Probanden, dass sie durch den Einsatz des Layouters Zeit gespart haben. Die Einschätzungen einiger Probanden bezüglich der durch den Layouter eingesparten Zeit gehen dabei teilweise noch über die eigenen Schätzungen aus Abschnitt 4.2.2 hinaus:

> Es gibt Situationen, in denen man für die Änderung einer Funktion 1 Minute, für das Aufräumen des Modells danach aber 30 Minuten braucht. Das ist natürlich der Best-Case für das Werkzeug.
>
> Entwickler von Fahrzeugmodellen für Testumgebungen

Derartige Situationen stellen, wie der Proband ebenfalls anmerkt, jedoch eher die Ausnahme dar. Konkrete Schätzungen bezüglich der durch den Layouter tatsächlich eingesparten Zeit beim Bearbeiten von Modellen wurden nur von wenigen Probanden geäußert:

> Beim manuellen Bearbeiten spart es ca. 20% der Zeit.
>
> Software-Integrator im Bereich Fahrerassistenzysteme

LAYOUT AUTOMATISCH ERZEUGTER DIAGRAMME

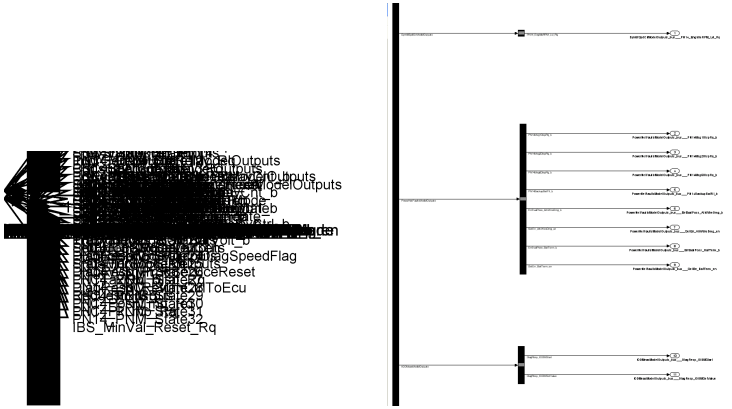
> Wir verwenden das Werkzeug auch, um Layouts für automatisch generierte AUTOSAR Funktionsrahmen zu erstellen.
>
> Softwareverantwortlicher für ein Steuergerät

Aktuelle Softwarearchitekturen wie AUTomotive Open System ARchitecture [3] (AUTOSAR) fordern eine einheitliche Strukturierung von Softwarefunktionen im Fahrzeug, um beispielsweise Kommunikationsbeziehungen zwischen Komponenten beschreiben zu können. Aus diesen Beschreibungen der Kommunikationsbeziehungen lassen sich automatisiert die Schnittstellen von Softwarekomponenten in Form von SL Modellen ableiten. Die dabei erzeugten Diagramme sind jedoch entweder vollkommen unleserlich (siehe beispielsweise Abbildung 6.1a) oder lediglich anhand statisch implementierter Muster, sogenannter *Layoutpatterns*, angeordnet. Die erste Variante erschwert die spätere manuelle Wartung der Diagramme, beispielsweise beim Hinzufügen einzelner Signale. Die zweite Variante erschwert die Entwicklung und Wartung des Werkzeugs zum automatischen Erzeugen der Modelle.

Daher wurde der Layouter, über die Anwendung beim manuellen Bearbeiten von SL Diagrammen hinaus, direkt an ein hauseigenes Werkzeug zum automatischen Erzeugen von Schnittstellenmodellen angebunden.

> Man muss sich damit beim automatischen Generieren von Modellen nicht mehr um das Layout und Layoutpatterns kümmern. Die Modellgeneratoren werden dadurch deutlich einfacher.
>
> Verantwortlicher für Modellierungsmethodik

Abbildung 6.1 zeigt die Zwischenergebnisse des Vorgehens: Zunächst wird das Diagramm unabhängig vom Layout erzeugt und dann im nächsten Schritt vom Layouter automatisch angeordnet.



(a) Automatisch erzeugtes Diagramm. **(b)** Vergrößerter Ausschnitt nach Anwendung des Layouters.

Abbildung 6.1: Diagramme eines automatisch erzeugten Schnittstellenmodells aus dem Fahrzeug-Innenraumbereich vor und nach Anwendung des Layouters.

MODELLIERUNGSUNTERSTÜTZUNG

Die in Abschnitt 4 vorgestellten Funktionen zur Modellierungsunterstützung lagen eigentlich nicht im Fokus der Erprobung. Da jedoch die Funktionen zum *Aufbrechen von Subsystemen* und zum *wahlfreien Einfügen und Entfernen von Ports* zu Beginn der Erprobung bereits im Layouter implementiert waren, standen sie den Probanden zur Nutzung zur Verfügung.

> Dieses Einfügen von Signalen in Busse hat mir schon viel Zeit gespart.
>
> Softwareverantwortlicher für ein Steuergerät

Obwohl sich das *wahlfreie Einfügen und Entfernen von Ports* (siehe Abschnitt 4.2.2) als die einfachste der drei in Abschnitt 4 vorgestellten Funktionen darstellt, wurde sie von den Probanden sehr positiv kommentiert und häufig genutzt. Dabei wurde der in Abschnitt 4.2.2 abgeschätzte Aufwand beim manuellen Einfügen der Ports von den Probanden mehrfach bestätigt.

> Bei Schnittstellenänderungen hat man damit sicherlich 90% Zeit eingespart.
>
> Verantwortlicher für Modellierungsmethodik

Im Vergleich mit dem *wahlfreien Einfügen und Entfernen von Ports* wurde das *Aufbrechen von Subsystemen* (siehe Abschnitt 4.2.1) nur von wenigen Probanden tatsächlich während der Projektarbeit angewandt. Diese stellten die Funktion jedoch als nützliche Erweiterung dar:

> "Break Subsystem" ist eine interessante und sinnvolle Funktion, die ich häufiger genutzt habe.
>
> Teamleiter in der System- und Softwareentwicklung

Die Nützlichkeit der Funktion wurde dabei sowohl auf die erzielte Zeitersparnis, als auch in geringerem Maße auf die Fehleranfälligkeit der manuellen Arbeitsschritte zurückgeführt. Die geringere Gewichtung der reduzierten Fehleranfälligkeit im Vergleich zur Zeitersparnis wurde mit einem noch mangelnden Vertrauen in die prototypische Implementierung begründet: Obwohl keine Fehler festgestellt werden konnten, zogen es die Probanden vor, das Ergebnis der Transformation noch einmal manuell zu überprüfen, um Fehler auszuschließen.

Dieses besonders vorsichtige Vorgehen bei der Anwendung konnte beim *wahlfreie Einfügen und Entfernen von Ports* nicht beobachtet werden. Probanden die beide Funktionen eingesetzt haben begründeten dieses Vorgehen mit der Einschätzung, dass sie das *wahlfreie Einfügen und Entfernen von Ports* als eine einfachere Funktion sahen, die "weniger am Modell kaputt machen könne". Bei beiden Unterstützungsfunktionen wurde zudem angemerkt, dass keine Undo-Funktion für die Funktionen zur Verfügung stand, da die Undo-Funktion des SL Editors bei automatischen Transformationen nicht funktioniert.

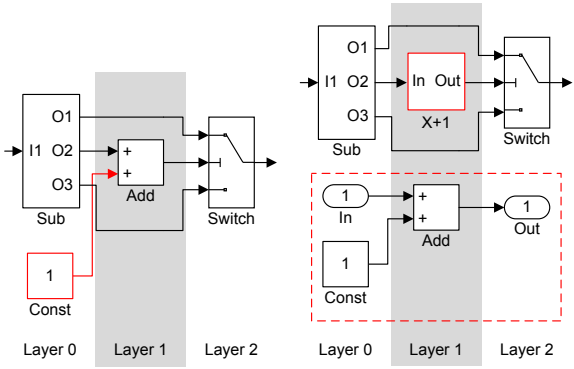
PROBLEMATISCHE HIERARCHISIERUNG

> Die strenge Hierarchisierung ist teilweise sehr einschränkend.
>
> Softwareverantwortlicher für ein Steuergerät

Die frühe Hierarchisierung bei hierarchischen Layoutverfahren kann sich sowohl in Bezug auf die Anzahl der Kreuzungen, als auch bei der Begradigung der Kanten negativ auf das resultierende Layout auswirken. Beispiele hierfür sind in Abschnitt 3.2.2, Abbildung 3.5 und Abbildung 3.6, gezeigt. Dies gilt auch unmittelbar für die in der Erprobung genutzte Version des Layouters, da in dieser die in Abschnitt 3.2.3 vorgestellten Erweiterungen der Hierarchisierung um Benutzervorgaben bezüglich zusätzlicher *Layer* noch nicht umgesetzt war.

Diese Problematik wurde auch von den meisten Probanden angemerkt und als unterschiedlich schwerwiegende Einschränkung bewertet. Während die Mehrzahl der Probanden sich in den betroffenen Situationen mit Teillayouts und manuellen Nachbesserungen zu Helfen wusste und diesbezüglich lediglich den Wunsch nach einer *eleganteren Lösung* äußerte,

lag der Anteil an stark betroffenen Diagrammen auf Grund der Strukturierung der bearbeiteten Modelle bei zwei der Probanden so hoch, dass ihnen der Layouter insgesamt zunächst als wenig nützlich erschien.



(a) Ursprüngliche Struktur. **(b)** Ersetzen von *Const* und *Add* durch das Subsystem *X+1*.

Abbildung 6.2: Alternative Strukturierung eines Modells zur Verbesserung des automatischen Layouts.

Die Analyse der problematischen Situationen führte einerseits zu dem in Abschnitt 3.2.3 beschriebenen Vorgehen, insbesondere zur Einführung der benutzerdefinierten Constraints bei der Hierarchisierung, andererseits ergab sie Hinweise auf Vorbesserungspotentiale bei der Strukturierung einiger Modelle. Werden beispielsweise in einem Diagramm viele Zugriffe auf Datenstrukturen zusammen mit Rechenoperationen dargestellt, führt dies sehr häufig zu Problemen mit der Hierarchisierung. Werden diese Rechenoperationen jedoch in Subsysteme zusammengefasst, vereinfacht sich die Struktur des Diagramms deutlich, was häufig auch die Lesbarkeit verbessert. Abbildung 6.2 stellt dies beispielhaft dar: Das Inkrementieren des mittleren Signals wird in ein Subsystem mit adäquater Beschriftung gegliedert, so dass im resultierenden Diagramm kein "gefangener Knoten" mehr enthalten ist.

VERBLEIBENDE LINIENKNICKE

Auch wenn der Algorithmus zur Kantenbegradigung im Layouter durch die Veränderung der Knotengrößen viele Kantenknicke entfernen kann (siehe Abschnitt 6.2.3), werden viele Diagramme noch immer mit Kantenknicken gezeichnet. Die Anzahl und Größe der verbleibenden Kantenkni-

cke variiert dabei einerseits mit der Parametrierung, andererseits mit der Struktur des Diagramms.

> Wenn man das Tool erstmal kennt, erleichtert es einem die Arbeit. Auch wenn man manchmal kleinere Nacharbeiten macht.
>
> Modellintegrator

Viele dieser verbleibenden Linienknicke lassen sich aus Sicht des Anwenders schnell beheben, insbesondere da die Probanden bereit sind, die vom Layouter strikt eingehaltenen Abstände zwischen Knoten und Kanten im Rahmen solch einer Nachbesserung zu verletzen. Daher sind einige Probanden dazu übergegangen, Layouts manuell nachzubessern und zumindest die besonders einfach entfernbaren Kantenknicke zu entfernen. Überraschenderweise stellte sich in der Befragung heraus, dass gerade diese Probanden den Layouter für besonders nützlich hielten und diesem trotz der manuellen Nacharbeiten noch immer eine deutliche Zeitersparnis und Arbeitserleichterung attestierten.

BENUTZERVORGABEN

Der in der Erprobung eingesetzte Layouter ermöglichte es den Probanden, einige der in Abschnitt 3 vorgestellten Benutzervorgaben zum Layout zu machen: Knoten konnten im Modell für die Zyklenerkennung als *flipped* markiert, nachrangig zu begradigende Kanten konnten festgelegt und die Größenveränderung einzelner Knoten konnte deaktiviert werden. Auch das Layout von Teilen eines Diagramms wurde von den Probanden als Benutzervorgabe gesehen. Einzelnen besonders aktiven Probanden wurde zu einem späteren Zeitpunkt während der Erprobung noch eine neuere Version des Layouts zur Verfügung gestellt, in der auch die Angabe benutzerdefinierter Constraints für die Hierarchisierung möglich wurde.

Aus der Befragung geht hervor, dass diejenigen Probanden, die die Funktionen für Benutzervorgaben kannten und verwendet haben, die erstellten Layouts positiver bewertet haben, als Probanden, auf die das nicht zutraf. Die Aussagen einiger Probanden, sowie die eigene Erfahrung des Autors, legen nahe, dass die Benutzervorgaben tatsächlich einen positiven Beitrag zur Qualität der Layouts leisten.

> Das Layout mit Benutzervorgaben war intuitiv, daran habe ich nichts auszusetzen. (...) Erst im Zusammenhang mit der Benutzerinteraktion und den zusätzlichen Constraints kann man die Stärken der Layoutunterstützung voll ausspielen. (...) Ohne die Linienpriorisierung sieht das ja sonst oft nicht schön aus.
>
> Verantwortlicher für Modellierungsmethodik

Dies konnte jedoch durch die Befragungen allein nicht endgültig geklärt werden.

LAUFZEIT

Insgesamt wurde die Laufzeit des Layouters von den Probanden als akzeptabel bewertet, solange sie unterhalb von etwa 10 Sekunden lag. Die Probanden gaben dazu an, dass sie von verschiedenen anderen Werkzeugen Laufzeiten im Bereich "etlicher Minuten", teilweise sogar "Stunden", gewohnt seien und dass ihnen daher "einige Sekunden" für ein neues Layout durchaus angemessen erschienen. Diese Einschätzung teilten selbst besonders kritische Probanden.

> Bei großen Systemen wird es manchmal langsam. Bei normalen Systemen ist die Geschwindigkeit gut.
>
> Software-Integrator im Bereich Fahrerassistenzsysteme

Bei großen Diagrammen überschreitet die Laufzeit des Layouters diese 10 Sekunden Grenze jedoch teilweise deutlich. Die Ursache der als lang empfundenen Laufzeit bei großen Diagrammen war einigen Probanden bereits aus den Statusmeldungen des Layouters bekannt:

> Bei großen Modellen war die Laufzeit, bedingt durch die Schnittstelle, teilweise recht lang.
>
> Verantwortlicher für Modellierungsmethodik

Die vom Benutzer wahrgenommene Laufzeit setzt sich aus der Laufzeit des Layouters selbst (siehe dazu die Messungen in Abschnitt 6.2.3) und der Laufzeit der Anbindung an SL zusammen. Letztere nimmt bei großen Diagrammen den überwiegenden Anteil der Laufzeit in Anspruch. Während der Layouter selbst die genannte Obergrenze von 10 Sekunden auch für große Modelle noch deutlich unterschreitet, benötigt die SL Anbindung (siehe Abschnitt 5) für große Diagramme häufig 30 Sekunden und länger. Der größte Teil entfällt dabei auf das Dekodieren der XML Antwort des Layouters auf SL Seite.

Es ist daher zu erwarten, dass eine verbesserte Anbindung des Layouters an SL, oder eine direkte Einbindung von Layoutfunktionen in den SL Editor, die Laufzeit auf ein akzeptables Niveau senken würde.

WEITERE VERWENDUNG

Die Mehrzahl der Probanden gab in der Befragung an, das Werkzeug auch zukünftig selbst weiter verwenden zu wollen.

> (Der Layouter) ist ein recht nützliches Tool, das werde ich auch weiterhin verwenden. (...) Ich habe das Tool auch an Kollegen weiterempfohlen.
>
> Teamleiter in der System- und Softwareentwicklung

Diese Probanden gaben zudem an, das Werkzeug entweder bereits an Kollegen weiterempfohlen zu haben, oder das auf Nachfrage hin tun zu wollen. Obwohl es sich beim Layouter bisher nur um einen Prototypen handelt, wurde dieser bereits fest in die Standard Werkzeugketten einiger Bereiche aufgenommen.

> Ich habe das Werkzeug in meinem Bereich in die Standard-Toolkette integriert.
>
> Verantwortlicher für Modellierungsmethodik

6.1.3 ZUSAMMENFASSUNG

Die Erfahrungen aus der Praxiserprobung unterstreichen noch einmal die zu Beginn dieser Arbeit dargestellte Bedeutung einer angemessenen Layoutunterstützung in der grafischen modellbasierten Entwicklung mit SL. Sowohl dem Layoutalgorithmus als auch den Funktionen zur Modellierungsunterstützung wird von den Probanden ein großer Nutzen in der Praxis bestätigt. Dieser äußert sich nach Ansicht der Probanden einerseits in einer Erleichterung der Arbeit, andererseits in einer Zeitersparnis von 20% und mehr beim Bearbeiten von Modellen. Auch die Benutzervorgaben wurden von den Probanden positiv aufgenommen, da sie einen gewissen Einfluss auf das entstehende Layout ermöglichen, ohne dabei ein tiefes Verständnis des Algorithmus zu erfordern. Die direkte Einbindung des Layouters in Werkzeuge zum automatischen Erstellen von Modellen ist ein weiteres positives Ergebnis der Erprobung.

Allerdings wurden von den Probanden auch einige negative Punkte angemerkt. Insbesondere zeigte sich bei einigen Diagrammen das die frühe Hierarchisierung bei hierarchischen Layoutverfahren in zusätzlichen Kantenkreuzungen resultieren kann (Priorisierung Ästhetischer Kriterien, siehe z.B. [88] S. 24). Dies führte bei Probanden mit vielen betroffenen Diagrammen zu einer schlechteren Akzeptanz des Layouters an sich. Dieses Problem konnte durch die benutzerdefinierten Constraints bei der Hierarchisierung (siehe Abschnitt 3.2.3) deutlich reduziert werden. Erste Rückmeldungen von Probanden, die mit einer um diese Funktionalität erweiterten Version des Layouters ausgestattet wurden, bestätigen dies.

Als weitere Einschränkung wurde die lange Laufzeit des Layouters bei großen Diagrammen angemerkt. Diese ist, wie zuvor beschrieben, insbesondere der prototypischen Implementierung des Werkzeugs geschuldet,

so dass erwartet wird, durch eine Verbesserung der Anbindung des Layouters an SL eine deutliche Verbesserung in diesem Punkt erzielen zu können.

Insgesamt überwiegen aus Sicht der Probanden die positiven Aspekte des Werkzeugs, so dass dieses sowohl individuell als auch in den Werkzeugketten verschiedener Bereiche auch weiterhin eingesetzt wird.

6.2 QUANTITATIVE ANALYSE

Die Bewertung der in in Abschnitt 3 vorgestellten Kantenbegradigung mit variablen Knotengrößen erfolgt hier anhand verschiedener ästhetischer Kriterien mit Hilfe von darauf basierenden Metriken. Anhand dieser Metriken wird die Kantenbegradigung mit variablen Knotengrößen mit der in Abschnitt 2.2.3 beschriebenen Kantenbegradigung mit festen Knotengrößen, sowie manuell erstellten Referenzlayouts verglichen. Hierfür stehen die bereits in Abschnitt 2.1.2 betrachteten Modelle aus dem Fahrzeug-Innenraumbereich zur Verfügung. Da diese direkt aus der Serienentwicklung von Fahrzeugfunktionen stammen, wiesen die enthaltenen Diagramme bereits manuell erstellte Layouts typischer Qualität auf. Diese manuell erstellten Layouts dienen daher als Referenz für die Bewertung der Kantenbegradigung.

6.2.1 METRIKEN

In der Literatur sind bereits verschiedene ästhetische Kriterien zur Beurteilung von Layoutalgorithmen dokumentiert [16, 71, 73, 76, 85, 86, 90]. Einige dieser Kriterien gehen als Constraints direkt in die Kantenbegradigung ein und werden nicht verletzt, beispielsweise sollen Knoten, Linien und Bezeichner nicht überlappend gezeichnet werden. Andere werden durch die Kantenbegradigung gar nicht erst beeinflusst, wie beispielsweise die Vorgabe Kantenkreuzungen zu minimieren. Die nachfolgend aufgeführten Kriterien wurden zur Beurteilung des in Abschnitt 3.3 ff. vorgestellten Algorithmus zur Kantenbegradigung mit variabler Knotengröße ausgewählt, da sie Eigenschaften betrachten, die vom Algorithmus direkt beeinflusst werden. Aus diesen Kriterien wurden quantitative Metriken zu Beurteilung des Algorithmus abgeleitet:

KANTENKNICKE

Tamassia schlägt in [86] vor, Kantenknicke zu vermeiden, da das Verfolgen abgeknickter Linien für das menschliche Auge aufwändiger sei als

das Verfolgen gerader Linien. Dabei sollten die verbleibenden Kantenknicke möglichst klein bleiben um unnötige Umwege im Kantenrouting zu vermeiden:

- Die *Anzahl der Kantenknicke* soll minimal sein.
- Die *Größe der* verbleibenden *Kantenknicke* soll minimal sein.

Die *Anzahl der Kantenknicke* lässt sich unmittelbar in eine Metrik überführen: Für jede Kante e wird die Anzahl von Kantenknicken als $m_b(e)$ gezählt. Der Mittelwert aller $m_b(e)$ eines Graphen $G = (V, E, P)$ (siehe Def. 1 ff.) entspricht dann dem Wert der Metrik $m_b(E)$:

$$m_b(E) = \frac{\sum_{e \in E} m_b(e)}{|E|} \tag{6.1}$$

Da die Kantenbegradigung bei hierarchischen Layoutverfahren lediglich Kantenknicke in vertikaler Richtung (relativ zur Ausrichtung des Diagramms) beeinflusst, geht in die Metrik *Größe der Kantenknicke* lediglich die Länge der vertikalen Segmente ein: Für jede Kante e wird die Länge ihrer vertikalen Liniensegmente als $m_s(e)$ summiert. Die Metrik $m_s(E)$ bezeichnet dann den Mittelwert von $m_s(e)$ über alle Kanten eines Graphen:

$$m_s(E) = \frac{\sum_{e \in E} m_s(e)}{|E|} \tag{6.2}$$

KNOTENGRÖSSEN

Der in Abschnitt 3 vorgestellte Algorithmus nutzt die Abhängigkeit zwischen den Portpositionen und der Knotengröße bei SL Diagrammen aus, um eine verbesserte Kantenbegradigung zu erreichen. Daher wird die *Knotengröße* als direkt vom Algorithmus beeinflusster Wert ebenfalls als Metrik erfasst, wobei die mittlere vertikale Größe $s_y(V)$ aller Knoten $v \in V$ pro Diagramm $G = (V, U, P)$ als Mittelwert der Größe aller Knoten im Diagramm berechnet wird ($s_y(v)$ sei jeweils die Größe des Knotens v):

$$s_y(V) = \frac{\sum_{v \in V} s_y(v)}{|V|} \tag{6.3}$$

LAUFZEIT

Für die Metrik *Laufzeit* wurde die Laufzeit der verwendeten Implementierung des Layoutalgorithmus zusammen mit der Laufzeit des LP Solvers im Layout Server (siehe Abschnitt 5) erfasst. Der zusätzliche Overhead für die Kommunikation zwischen SL und dem Layout Server, sowie für die anschließende Aktualisierung des Diagramms bleibt unberücksichtigt.

6.2.2 DURCHFÜHRUNG

Der Vergleich wurde anhand der bereits in Abschnitt 2.1.2 untersuchten Diagramme aus den SL Modellen des Fahrzeug-Innenraumbereichs durchgeführt. Da zur Anwendung eines Algorithmus zur Kantenbegradigung wenigstens zwei miteinander verbundene Knoten im Diagramm erforderlich sind, wurden nur die 1774 Diagramme für den Vergleich herangezogen, in denen diese Eigenschaft erfüllt ist. Da Knoten ohne Verbindungen von der Kantenbegradigung nicht verändert werden, wurden sie weder beim Erstellen der Layouts noch bei der Berechnung der Metriken berücksichtigt.

Das Erfassen der Messdaten erfolgte für alle ausgewählten Diagramme nach folgendem Vorgehen:

1. Berechnen der Metriken für das ursprüngliche, manuell erstellte Layout.
2. Anwendung des Layoutalgorithmus mit festen Knotengrößen.
3. Berechnen der Metriken für das automatisch erstellte Layout mit festen Knotengrößen.
4. Anwendung des Layoutalgorithmus mit variablen Knotengrößen.
5. Berechnen der Metriken für das automatisch erstellte Layout mit variablen Knotengrößen.

Die Messdaten wurden auf Grund der großen Anzahl an Diagrammen automatisiert erfasst. Benutzervorgaben zur Verbesserung des Layouts, wie beispielsweise über zusätzliche Constraints bei der Hierarchisierung, wurden daher nicht genutzt.

Die Messungen erfolgten mit der folgenden Hard- und Softwarekonfiguration:

- Hardware: Intel Core 2 T5600 mit 1,83 GHz, 4GB RAM
- Betriebssystem: Windows XP Professional SP3
- Laufzeitumgebung: Matlab R2009b SP1, Java 6 1.6.0_13-b03

- Constraint Solver: SCIP (scip-1.2.0.mingw32_nt-5.1.x86.gnu.opt.spx)
- Version des Layout Servers: 552.8523
- Vom Standard abweichende Konfiguration des Layout Servers:
  - LPNP_SOLVER="SCIP"
  - LPNP_COST_STRAIGHTNESS="STRAIGHT"

6.2.3 MESSDATEN

Die nachfolgende Auswertung der erfassten Metriken *Anzahl der Linienknicke* und *Größe der Kantenknicke* vergleicht die Messwerte des Layoutalgorithmus mit variablen Knotengrößen mit den manuell erstellten Layouts, sowie den Messwerten für den Algorithmus mit festen Knotengrößen. Darauf folgt die Auswertung der Metrik *Knotengröße* anhand eines Vergleichs der Messwerte des neuen Algorithmus mit denen der manuell erstellten Layouts und die Betrachtung der gemessenen *Laufzeiten* für den neuen Algorithmus. Die nachfolgende Auswertung der erfassten Metriken *Anzahl der Linienknicke* und *Größe der Kantenknicke* vergleicht die Messwerte des Layoutalgorithmus mit variablen Knotengrößen mit den manuell erstellten Layouts, sowie den Messwerten für den Algorithmus mit festen Knotengrößen. Darauf folgt die Auswertung der Metrik *Knotengröße* anhand eines Vergleichs der Messwerte des neuen Algorithmus mit denen der manuell erstellten Layouts und die Betrachtung der gemessenen *Laufzeiten* für den neuen Algorithmus.

ANZAHL DER KANTENKNICKE

Die Mittelwerte der Metrik *Anzahl der Kantenknicke* (siehe Abbildung 6.3) zeigen bei der Kantenbegradigung mit variabler Knotengröße im Vergleich zum Algorithmus mit festen Knotengrößen etwa 40% weniger Linienknicke. Zudem nimmt der Anteil an Diagrammen, die im fertigen Layout gar keine Kantenknicke mehr aufweisen, bei der Kantenbegradigung mit variabler Knotengröße von 5% auf 18% zu (siehe Abbildung 6.4). Dabei übertrifft das neue Verfahren selbst die manuell erstellten Layouts (13% der Diagramme ohne Kantenknicke), obwohl diese im Mittel weniger Kantenknicke aufweisen (0,64 zu 0,88 Knicke pro Kante).

GRÖSSE DER KANTENKNICKE

Zusätzlich zur Reduktion der Anzahl an Kantenknicken zeigt die Metrik *Größe der Kantenknicke* für die untersuchten Diagramme beim Algorithmus

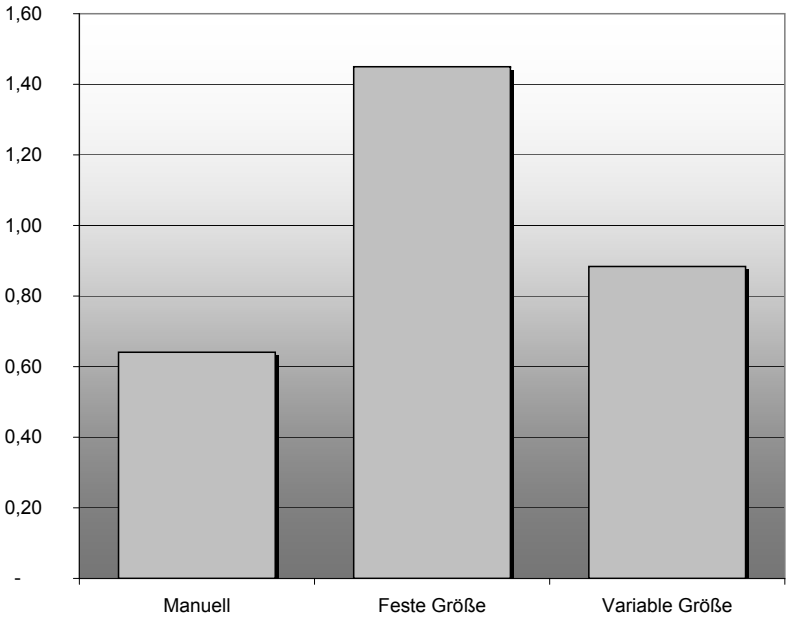


Abbildung 6.3: Mittelwerte der Metrik *Anzahl der Kantenknicke* für manuell erstellte Layouts, automatisch erstelle Layouts mit festen Knotengrößen und automatisch erstellte Layouts mit variablen Knotengrößen.

aus dieser Arbeit eine Reduktion der mittleren Größe von Kantenknicken um etwa 20% (siehe Abbildung 6.5). Auch im Vergleich zu den manuell erstellten Layouts erreicht der Algorithmus im Mittel eine leichte Reduktion in der Größe der Linienknicke von 5%.

KNOTENGRÖSSE

Der Algorithmus zur Kantenbegradigung mit variablen Knotengrößen erreicht die Vorteile bei der Reduktion von Kantenknicken in Anzahl und Größe durch die Veränderung der Knotengröße. Wie erwartet ergibt sich dabei im Mittel eine Vergrößerung der Knoten; für die untersuchten Diagramme beträgt die Vergrößerung der Knoten etwa 40%. Da der Algorithmus mit festen Knotengrößen die Knotengrößen der manuell erstellten Layouts als Eingabe erhält und an diesen keine Veränderungen vornimmt, entfällt die Angabe der mittleren Knotengrößen für den Layoutalgorithmus mit festen Knotengrößen in Abbildung 6.6. Diese entsprechen denen der manuell erstellten Layouts.

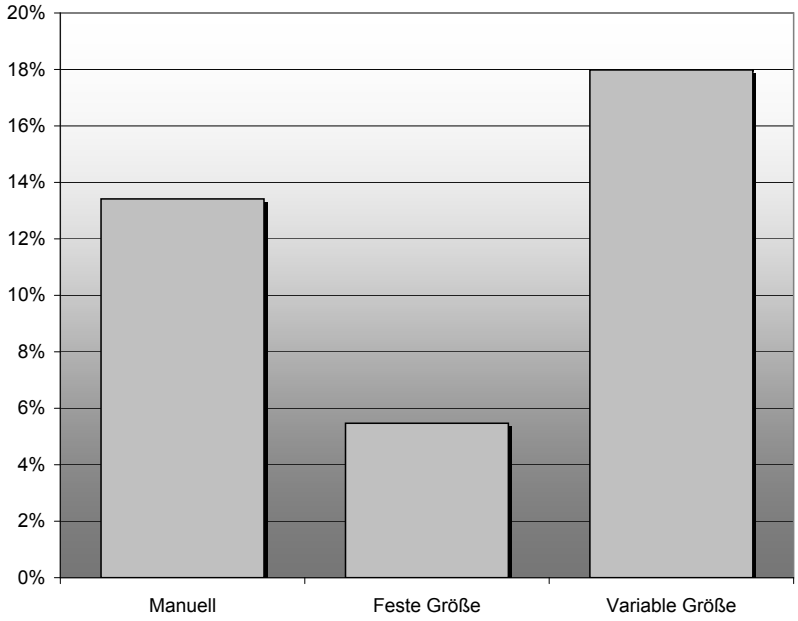


Abbildung 6.4: Anteil der Diagramme, die ohne Kantenknicke gezeichnet wurden.

LAUFZEIT

In Abbildung 6.7 sind die Laufzeiten für die untersuchten Diagramme mit bis zu 100 Kanten (96% der Diagramme) dargestellt. Bei 57 Diagrammen dieser Größenordnung (3%) wurden Laufzeiten über 1000ms gemessen. Diese sind der Lesbarkeit halber in Abbildung 6.7 nicht dargestellt. Die gemessenen Laufzeiten (einschließlich der in Abbildung 6.7 nicht dargestellten Messwerte) zeigen eine näherungsweise exponentielle Abhängigkeit der Laufzeit des Algorithmus von der Anzahl der Verbindungen im Diagramm: Laufzeit $t \approx 74,937e^{0,0218n}$ms für $n = |E|$ Kanten mit einem Bestimmtheitsmaß $R^2 = 0,4481$. Die starke Streuung der Laufzeiten verschiedener Diagramme mit gleich vielen Kanten begründet sich wie folgt: Die Lösung des ganzzahligen linearen Programms bei der Kantenbegradigung mit variabler Knotengröße ist aus komplexitätstheoretischer Sicht zunächst einmal NP-schwer. In der Praxis hängt die tatsächliche Laufzeit des LP Solvers jedoch nicht nur von der Größe des linearen Programms, sondern insbesondere auch vom Aufbau des Constraintsystems ab. Da sich diese direkt aus der Struktur des jeweiligen Diagramms ableitet, können sich bei verschiedenen Diagrammen gleicher Größe stark abweichende Laufzeiten ergeben.

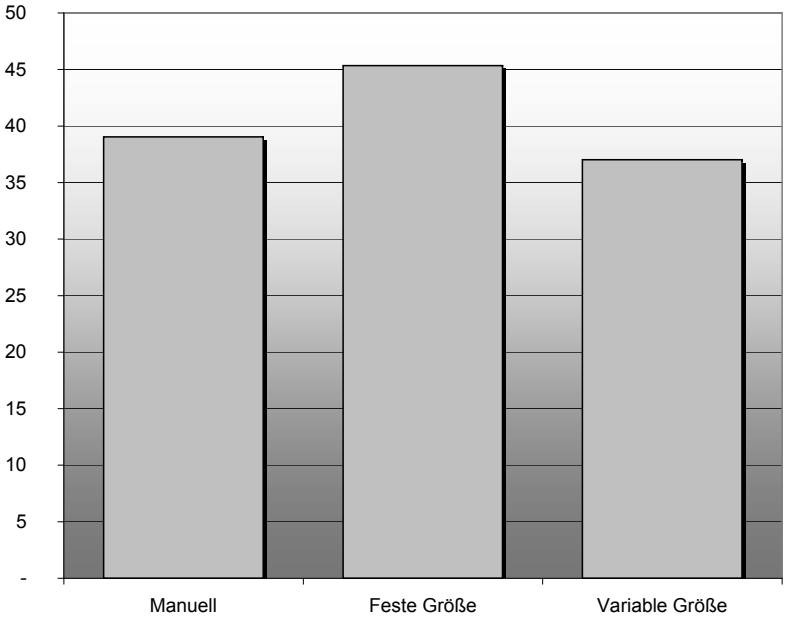


Abbildung 6.5: Mittelwert der Metrik *Größe der Kantenknicke* für manuell erstellte Layouts, sowie automatisch erstelle Layouts mit festen und variablen Knotengrößen.

Trotz dieses eher ungünstigen Zusammenhangs zwischen der Anzahl von Kanten und der Laufzeit bleibt diese für 94% aller untersuchten Diagramme unter 1000ms (für die Häufigkeitsverteilung der Laufzeiten bis 500ms siehe Abbildung 6.8). Bei Beschränkung auf die in der Praxis besonders häufigen Diagramme mit weniger als 100 Knoten (96% der Diagramme) liegt die Laufzeit sogar in 97% der betrachteten Fälle unter 1000ms.

6.2.4 ZUSAMMENFASSUNG

Die Messdaten belegen, dass die Veränderung der Knotengrößen einen positiven Beitrag zur Kantenbegradigung leistet, auch wenn dieser größtenteils mit einer Vergrößerung der Knoten erkauft wird. Im Vergleich zum herkömmlichen Algorithmus mit festen Knotengrößen zeigt der Ansatz bezogen auf die betrachteten Metriken *Anzahl der Kantenknicke* und *Größe der Kantenknicke* deutlich bessere Resultate. Auch im Vergleich zu den manuell erstellten Layouts ist der neue Ansatz durchaus konkur-

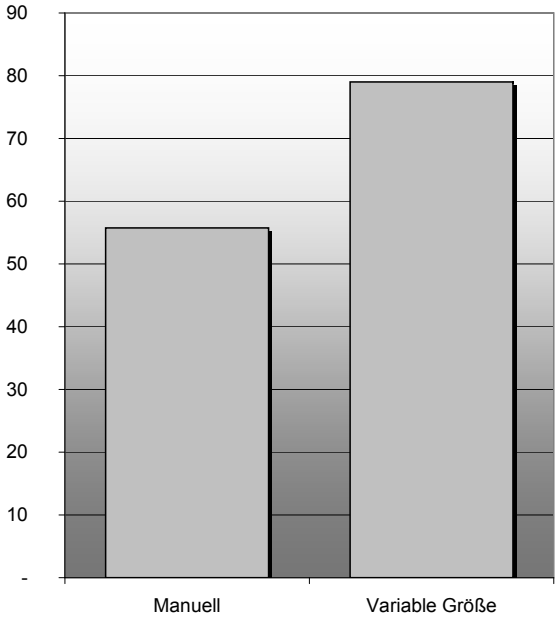


Abbildung 6.6: Mittelwerte der Metrik *Knotengröße* für manuell erstellte Layouts und automatisch erstellte Layouts mit variablen Knotengrößen.

renzfähig: Der Anteil der Diagramme, die vom neuen Ansatz ganz ohne Kantenknicke gezeichnet werden konnten, liegt höher als bei den manuell erstellten Layouts; auch die mittlere *Größe der Kantenknicke* nimmt im Vergleich ab.

Das Erstellen von Layouts erfolgte beim Algorithmus mit variablen Knotengrößen in der vorliegenden prototypischen Implementierung für typische Diagramme und auf einer üblichen Hardware bereits überwiegend in praxistauglicher Zeit.

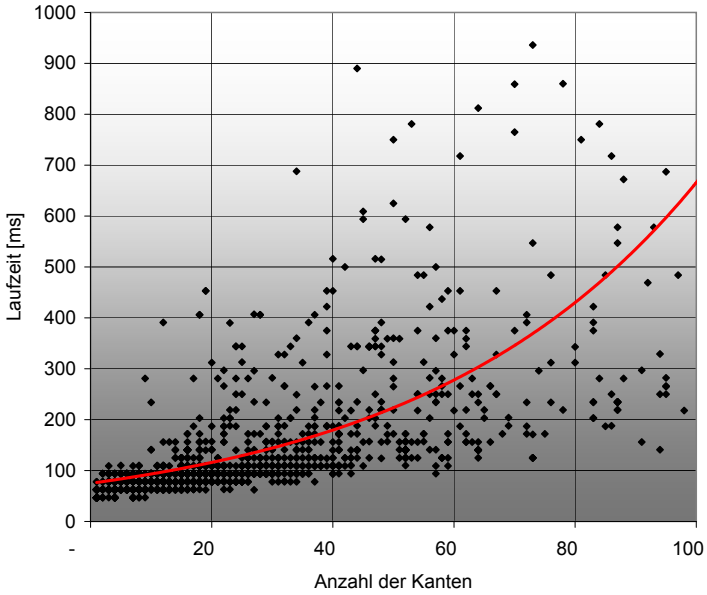


Abbildung 6.7: Laufzeiten für Diagramme mit bis zu 100 Kanten.

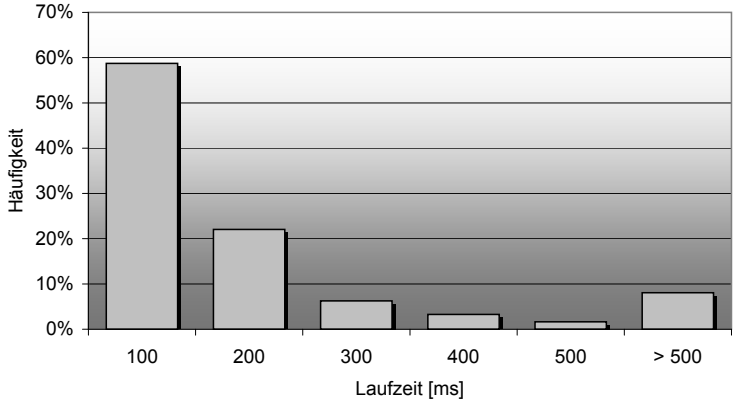


Abbildung 6.8: Häufigkeit verschiedener Laufzeiten des Layoutalgorithmus mit variablen Knotengrößen bei Anwendung auf die ausgewählten Diagramme.

7 ZUSAMMENFASSUNG UND AUSBLICK

Die modellbasierte Softwareentwicklung mit Werkzeugen wie SL ist im Automobilbereich weit verbreitet. Verfügbare grafische Editoren weisen jedoch insbesondere in der Benutzerfreundlichkeit deutliche Schwächen auf und bieten beim Bearbeiten der Modelle kaum Unterstützung an. Allein durch die fehlende Layoutunterstützung entsteht beim Bearbeiten von SL Modellen ein Mehraufwand von etwa 20%. Fortschrittliche Funktionen zum strukturellen Bearbeiten der Modelle, beispielsweise beim Einfügen neuer Knoten oder beim Ändern von Datenstrukturen, fehlen.

7.1 ZUSAMMENFASSUNG

Die in dieser Arbeit vorgestellten Verfahren tragen zur Weiterentwicklung der modellbasierten Softwareentwicklung mit Modellierungssprachen wie SL bei, indem sie das Betrachten und Bearbeiten von Diagrammen erleichtern und effizienter machen. Der vorgestellte Layoutalgorithmus ermöglicht es, SL Diagramme unter Berücksichtigung von Benutzervorgaben automatisch anzuordnen. Er erleichtert dadurch nicht nur das manuelle Bearbeiten der Diagramme, sondern leistet auch einen wichtigen Beitrag zur Entwicklung von Werkzeugen zum automatischen Erzeugen und Modifizieren von Modellen. Aufbauend auf diesem Algorithmus wurden in dieser Arbeit Funktionen zur Modellierungsunterstützung entwickelt, die typische strukturelle Änderungen an SL Diagrammen als Modelltransformationen automatisieren.

LAYOUT VON SIMULINK DIAGRAMMEN

Der in Abschnitt 3 beschriebene Algorithmus zum Layout von SL Diagrammen basiert auf dem Algorithmus zum hierarchischen Zeichnen gerichteter Graphen von Sugiyama *et al.* [85], sowie den darauf aufbauenden Ansätzen von Spönemann *et al.* [84] und Gansner *et al.* [42] und erweitert diesen zur Berücksichtigung der in Abschnitt 2.1.1 beschriebenen grafischen Eigenschaften der Diagramme sowie typischer Modellierungsrichtlinien:

Zunächst wurde die Zyklenerkennung erweitert, um Hinweise bezüglich der semantischen Bedeutung von Kanten durch den Benutzer zu berücksichtigen. Für die nachfolgende Zyklenbehandlung wurde ein auf *Inverter* Knoten basierender Ansatz vorgestellt, der es ermöglicht, die Kantenknicke in Zykluskanten im Vergleich zu bisherigen Ansätzen zu reduzieren. Benutzerdefinierte Constraints während der Hierarchisierung erlauben es, die typischen, durch die frühe Hierarchisierung entstehenden Probleme hierarchischer Layoutalgorithmen bei der Kreuzungsreduktion und Kantenbegradigung zu vermeiden. Weitere Constraints ermöglichen die Beachtung von Modellierungsrichtlinien bezüglich der horizontalen Anordnung von Knoten. Die auf Grundlage der Arbeit von Gansner *et al.* [42] entwickelte Kantenbegradigung für Hyperkanten in Diagrammen mit verschiedenen Port Constraints und variablen Knotengrößen stellt einen wesentlichen Beitrag der Arbeit dar. Sie ermöglicht es, mit Hilfe der aus einer Analyse der Portpositionen in SL abgeleiteten Linearisierung in Abschnitt 3.5.1, die Knotengrößen in SL Diagrammen automatisch anzupassen, um Kantenknicke zu vermeiden. Ein weiterer Beitrag dieser Arbeit zur Vermeidung von Kantenknicken stellt der auf dem Einfügen zusätzlicher *Dummy* Knoten basierende Ansatz zum Routing von Kanten an oben und unten liegenden Ports dar.

MODELLIERUNGSUNTERSTÜTZUNG

In Abschnitt 4 wurden verschiedene Ansätze vorgestellt, mit denen bisher manuell durchgeführte Schritte beim Erstellen und Bearbeiten von SL Modellen automatisiert werden können:

Das kontextbasierte Modellieren unterstützt das Auswählen, Einfügen und Verbinden neuer Knoten mit einer Heuristik. Diese bietet im Kontext eines gewählten Knoten und Ports diejenigen Knoten zur Auswahl an, die in anderen Modellen besonders häufig mit Knoten des gewählten Typs verbunden waren. Wählt der Benutzer einen dieser Knoten aus, wird er automatisch eingefügt und mit dem zuvor gewählten Port verbunden. Die Funktionen *Aufbrechen von Subsystemen* und *Wahlfreies Einfügen von Ports* fassen typische Arbeitsschritte beim Bearbeiten von SL Modellen zusammen und automatisieren sie als Modelltransformationen.

Da Funktionen zur Modellierungsunterstützung auch strukturelle Änderungen am Modell durchführen, wird ihr Einsatz erst durch die Verfügbarkeit einer automatischen Layoutunterstützung, beispielsweise durch den in dieser Arbeit vorgestellten Layoutalgorithmus, sinnvoll möglich.

VALIDIERUNG

Sowohl der Layoutalgorithmus als auch die Funktionen zur Modellierungsunterstützung wurden prototypisch in einem Werkzeug, dem Layouter, implementiert (Abschnitt 5).

Dieser Layouter wurde bezüglich der Kantenbegradigung mit Hilfe von Metriken untersucht (Abschnitt 6.2) und in verschiedenen Bereichen der Entwicklung eines Automobilherstellers erprobt (Abschnitt 6.1). Dabei konnte unter Anderem gezeigt werden, dass die Kantenbegradigung mit variabler Knotengröße die Anzahl und Größe von Kantenknicken im Vergleich zum ursprünglichen Ansatz mit festen Knotengrößen für die betrachteten Diagramme aus der Praxis deutlich reduzieren konnte. In der Praxiserprobung konnte die anfängliche Annahme bestätigt werden, dass eine angemessene Layoutunterstützung im Allgemeinen und der Algorithmus dieser Arbeit im Speziellen die Arbeit an SL Diagrammen deutlich erleichtert und beschleunigt und insbesondere für das automatische Erstellen von Modellteilen von zentraler Bedeutung ist. Dem Layouter wurde von den Probanden bezüglich dieser Punkte ein positiver Beitrag bestätigt, er hat in der vorliegenden prototypischen Implementierung bereits Eingang in die Werkzeugketten einiger an der Erprobung beteiligter Entwicklungsbereiche finden können. Darüber hinaus haben die bereits während der Erprobung gewonnenen Erkenntnisse die Entwicklung der benutzerdefinierten Constraints motiviert und somit direkt zur Verbesserung des Layoutalgorithmus beigetragen.

Obwohl die Funktionen zur Modellierungsunterstützung nicht im Fokus der Erprobung lagen, standen die Funktionen zum *Aufbrechen von Subsystemen* und zum *wahlfreien Einfügen von Ports* einigen Probanden bereits zur Verfügung. Beide Funktionen wurden von diesen Probanden als sehr nützlich bewertet, wobei von einer Zeitersparnis von bis zu 90% bei bestimmten Arbeitsschritten berichtet wurde.

7.2 AUSBLICK

Ausgehend von den Ergebnissen dieser Arbeit erscheint es sinnvoll, die folgenden Themenbereiche im Bereich der Layoutalgorithmen und der Modellierungsunterstützung künftig zu bearbeiten. Die in dieser Arbeit vorgestellten Ansätze zum Kantenrouting von Zykluskanten und Kanten an oben oder unten liegenden Ports wurden bereits im Rahmen der Zusammenarbeit des Autors mit der KIELER Arbeitsgruppe durch Schulze aufgegriffen und erweitert [81]. Schulze fügt die zusätzlichen *Dummy* Knoten bei oben und unten liegenden Ports bereits vor der Kreuzungsreduktion ein, um die Anzahl von Kreuzungen mit diesen

Kanten zu verbessern. Dazu wird die Kreuzungsreduktion erweitert, um zusätzliche Constraints für diese Knoten und bei Zykluskanten zu berücksichtigen.

LAYOUTALGORITHMUS

Obwohl sich der vorgestellte Layoutalgorithmus bereits gut zum Anordnen von SL Diagrammen eignet und die strukturelle Schwäche der frühen Hierarchisierung in hierarchischen Layoutalgorithmen mittels benutzerdefinierten Constraints behoben werden kann, erscheint es wünschenswert, solche Situationen künftig entweder automatisiert erkennen und die notwendigen Constraints automatisch einfügen zu können, oder eine effiziente Umsetzung eines auf Planarisierung basierenden Ansatzes zu finden, so dass auf die frühe Hierarchisierung ganz verzichtet werden kann.

In Abschnitt 3.5.1 wurde darauf hingewiesen, dass die Linearisierung der Portpositionen bei vielen Ports und ungünstigen Kombinationen der Port Anzahl gegenüberliegender Seiten viele der in SL möglichen Knotengrößen unberücksichtigt lassen muss. Der hier vorgestellte Ansatz sieht vor, in solchen Fällen schrittweise auf eine weniger genaue Linearisierung zurückzufallen, was zu zusätzlichen Kantenknicken führen kann. Diese zusätzlichen Kantenknicke gilt es künftig zu vermeiden, beispielsweise durch alternative Linearisierungen für bestimmte Grenzfälle.

MODELLIERUNGSUNTERSTÜTZUNG

Im Rahmen dieser Arbeit konnten die Funktionen zur Unterstützung des Benutzers beim Erstellen und Bearbeiten von SL Diagrammen nur beispielhaft behandelt und erst teilweise in der Praxis erprobt werden. Eine umfassendere Untersuchung dieser Funktionen, insbesondere des *kontextbasierten Modellierens*, im Hinblick auf die hierdurch bei der Modellierung eingesparte Zeit wäre wünschenswert.

In Anlehnung an *Refactorings* aus der textuellen Softwareentwicklung wurden mit dem *Aufbrechen von Subsystemen* und dem *wahlfreien Einfügen von Ports* zwei typische Arbeitsschritte als Modelltransformationen automatisiert. Die Erfahrungen aus der Praxiserprobung zeigen, dass das Thema Modelltransformationen aus Sicht der Praxis von großer Relevanz für die *Pragmatik* der modellbasierten Entwicklung mit Werkzeugen wie SL ist. Obwohl es bereits theoretische Ansätze zur automatischen Transformation von Modellen gibt, beispielsweise das auf der Fujaba Tool Suite [39] (FUJABA) basierende Model Analysis and Transformation Environment for MATLAB Simulink [61] (MATE), fehlen Lösungen für die Praxis noch fast völlig. Es erscheint daher ratsam, sich bei der Weiterentwicklung von Methoden und Werkzeugen der grafischen modellbasierten

Entwicklungen stärker als bisher an den tatsächlichen Bedürfnissen der Praxis zu orientieren und dabei auch die Umsetzbarkeit der entwickelten Verfahren zu berücksichtigen.

Zudem stellt sich beim *kontextbasierten Modellieren* die Frage, inwieweit es möglich und sinnvoll wäre, über die statische Analyse der Struktur anderer Modelle hinaus auch andere Verfahren des maschinellen Lernens für die Auswahl möglicher Knoten zu nutzen.

BENUTZERSCHNITTSTELLE

In der vorliegenden Arbeit findet sowohl beim Layoutalgorithmus als auch bei der Modellierungsunterstützung eine Interaktion mit dem Benutzer statt. Der Layoutalgorithmus verwendet benutzerdefinierte Constraints und Gewichte bei Hierarchisierung und Kantenbegradigung und nutzt Benutzerhinweise in der Zyklenerkennung. Das *kontextbasierte Modellieren* erfordert die zweimalige Auswahl von Knoten und Ports: Zunächst für den aktuellen Knoten und danach für den neuen Knoten. Obwohl die Praxiserprobung der prototypischen Umsetzung dieser Funktionen bereits eine gute Nutzbarkeit bestätigt, erscheinen Effizienz und Ergonomie dieser Benutzerinteraktionen angesichts der bisher verwendeten Kontextmenüs (siehe Abbildung 4.1) noch verbesserungswürdig.

Teil IV

ANHANG

A BEISPIEL LAYOUTS

Die in diesem Abschnitt dargestellten Layouts verschiedener SL Diagramme wurden direkt mit dem in Abschnitt 5 beschriebenen Layouter, der Implementierung des Layoutalgorithmus aus Abschnitt 3, erstellt.

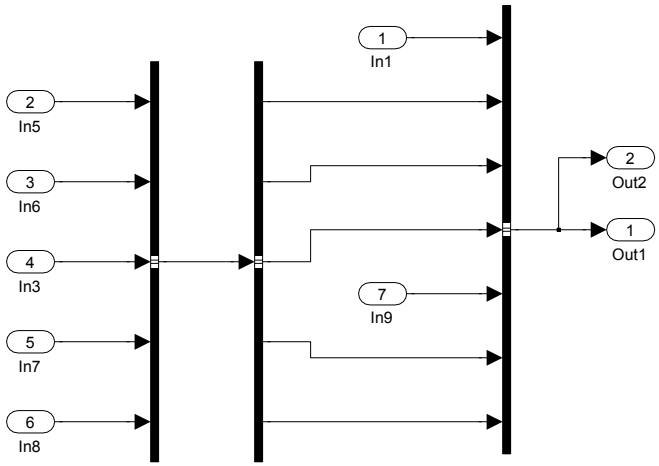


Abbildung A.1: Layout eines Diagramms mit innen liegendem Knoten. Benutzerdefiniertes Constraint *In9 rechts vom BusSelector*.

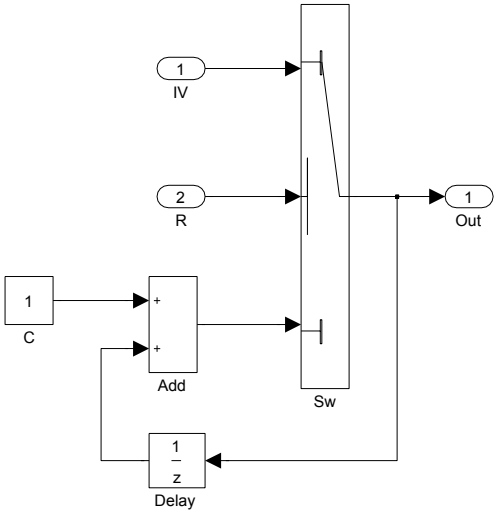


Abbildung A.2: Layout eines Zähler Modells; nur der *Delay* Knoten ist *flipped*.

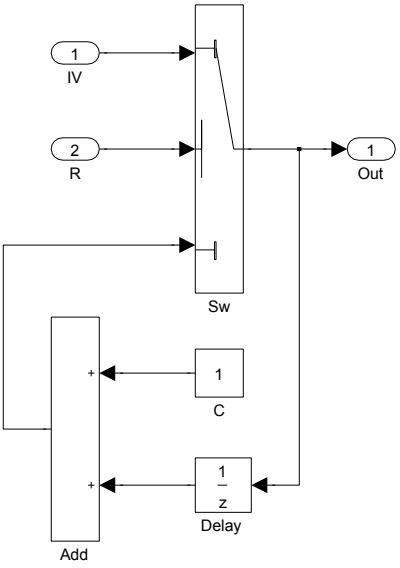


Abbildung A.3: In der Praxis unübliches Layout eines Zähler Modells; zusätzliche *flipped* Knoten.

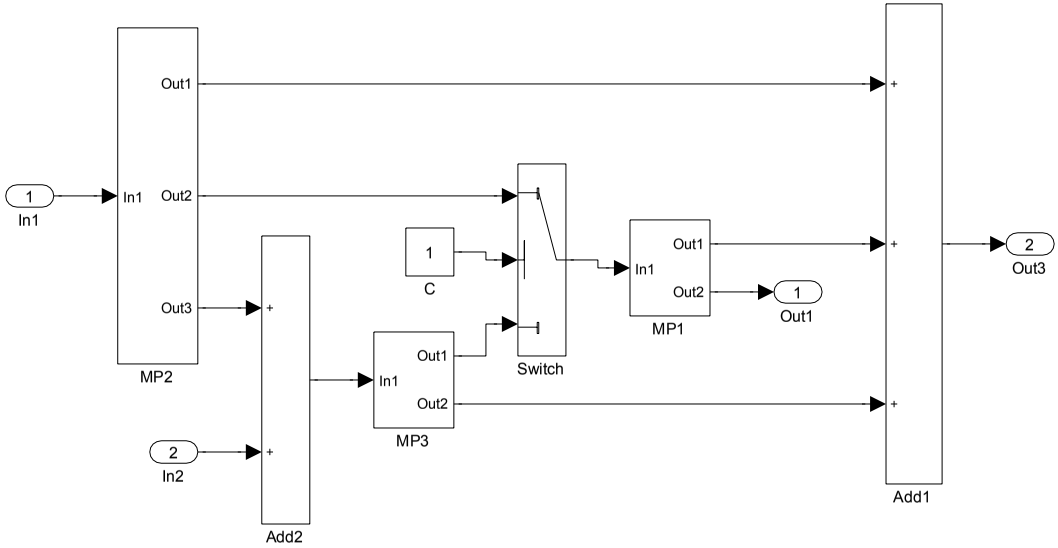


Abbildung A.4: Layout eines Diagramms ohne Zykluskante mit benutzerdefiniertem Constraint *Out1 links von Add1*.

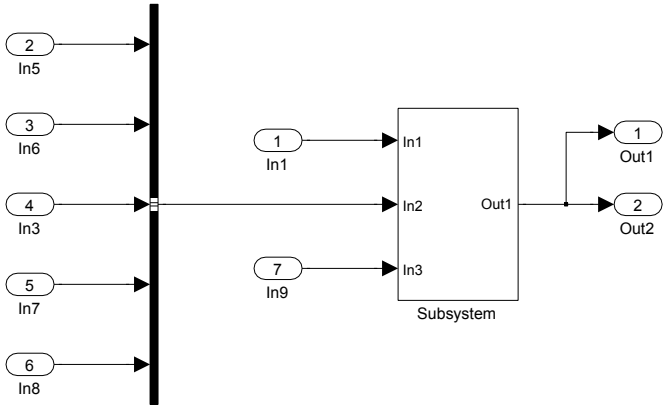


Abbildung A.5: Layout eines Diagramms mit benutzerdefiniertem Constraint *In1 rechts von BusCreator*.

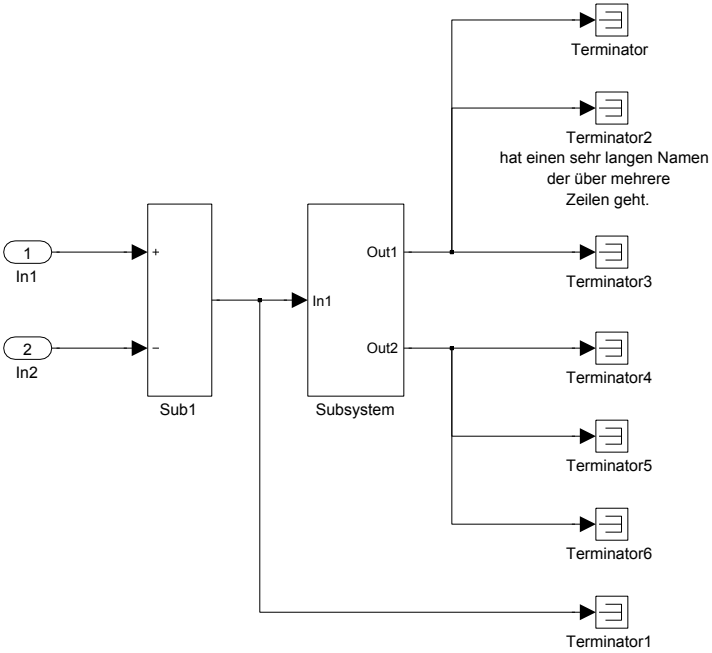


Abbildung A.6: Layout eines Diagramms mit einem hohen Anteil von Hyperkanten und langen Bezeichnern. Benutzerdefiniertes Constraint *Terminator1 aligned mit Terminator*.

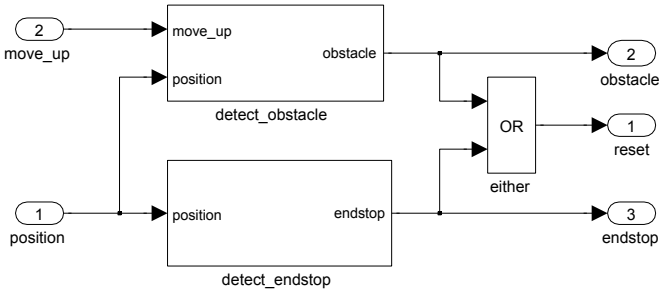


Abbildung A.7: Layout des Simulink Beispieldiagramms: *DetectObstacle/Endstop*.

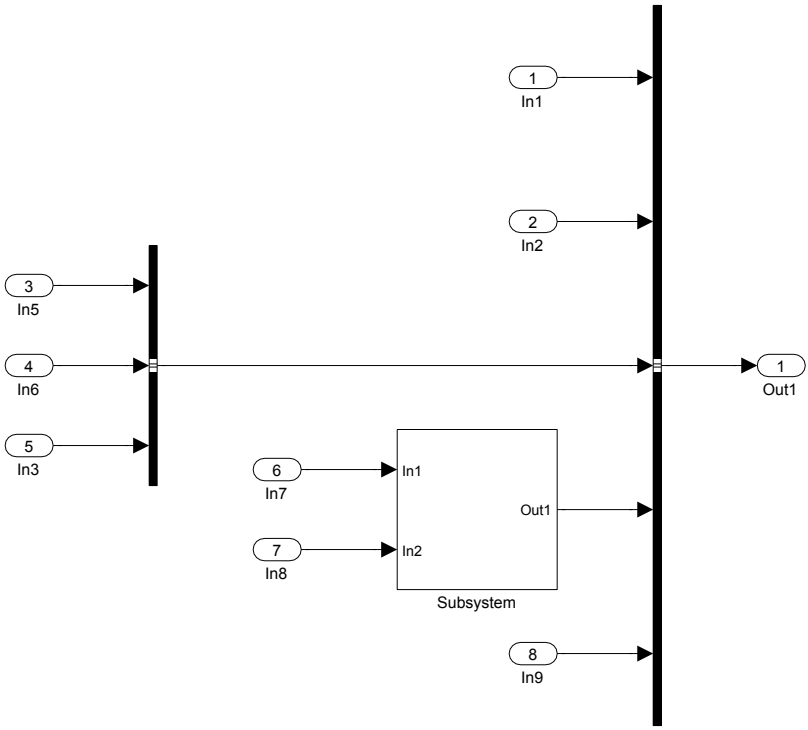


Abbildung A.8: Layout eines kleinen beispielhaften Schnittstellenmodells mit einem benutzerdefinierten Constraint zur Anordnung des linken *BusCreator* Knotens.

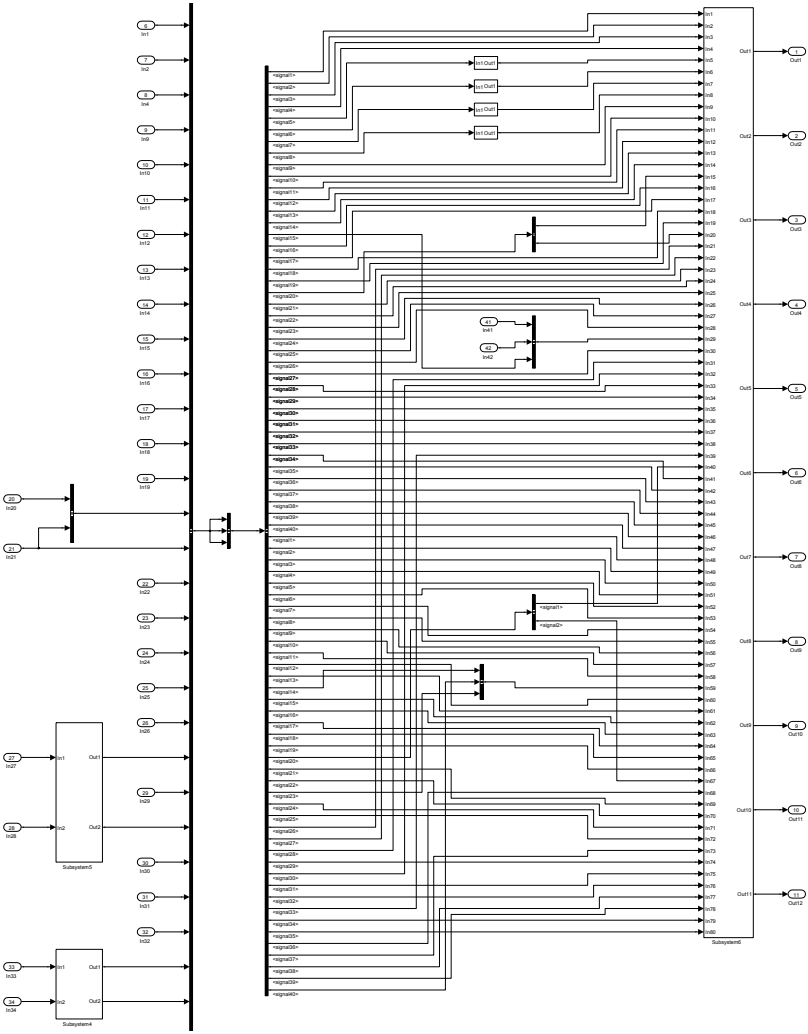


Abbildung A.9: Layout einer größeren Beispielschnittstelle. Mehrere benutzerdefinierte Constraints.

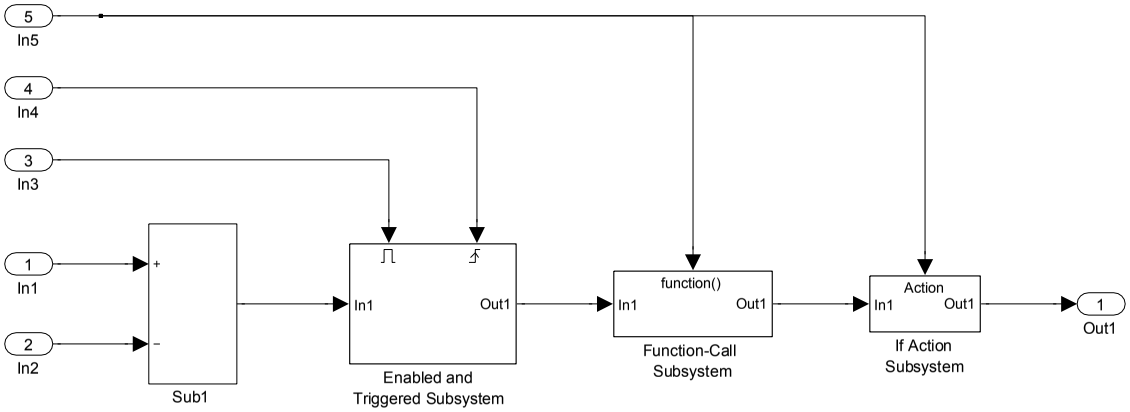


Abbildung A.10: Layout eines Diagramms mit mehreren oben liegenden Ports. Benutzerdefinierte Constraints ordnen die *Inport* Knoten links an.

LITERATURVERZEICHNIS

[1] T. Achterberg (2007). *Constraint Integer Programming*. Dissertation, Technische Universität Berlin. http://opus.kobv.de/zib/volltexte/2009/1153/. (Zitiert auf Seite 36.)

[2] T. Achterberg (2009). *SCIP: Solving constraint integer programs*. Mathematical Programming Computation, **1**(1), 1–41. URL http://mpc.zib.de/index.php/MPC/article/view/4. (Zitiert auf den Seiten 36, 90 und 144.)

[3] AUTOSAR (2010). *Automotive Open System Architecture 4.0*. Online. URL http://autosar.org. (Zitiert auf den Seiten 101 und 143.)

[4] W. Barth, M. Jünger und P. Mutzel (2002). Simple and efficient bilayer cross counting. Technischer Bericht, Institut für Computergraphik und Algorithmen, TU Wien, Favoritenstraße 9-11, A-1040 Wien, Austria Institut für Informatik, Universität zu Köln, Pohligstraße 1, D-50969 Köln, Germany. (Zitiert auf Seite 50.)

[5] B. Becker, E. Nardelli und R. Tamassia (1986). *A Layout Algorithm for Data-Flow Diagrams*. IEEE Trans. Softw. Eng., **12**(4), 538–546. (Zitiert auf Seite 30.)

[6] U. Brandes und B. Köpf (2002). *Fast and Simple Horizontal Coordinate Assignment*. In Graph Drawing 2001, Band 2265 von *LNCS*, Seiten 31–44. Springer. (Zitiert auf Seite 29.)

[7] M. Broy (2006). *Challenges in Automotive Software Engineering*. In ICSE 06: Proceedings of the 28th international conference on Software engineering, Seiten 33–42. (Zitiert auf Seite 3.)

[8] C. Buchheim, M. Jünger und S. Leipert (1999). A fast layout algorithm for k-level graphs. Technischer Bericht, Institut für Angewandte Mathematik und Informatik, Universität zu Köln. (Zitiert auf Seite 29.)

[9] M. Chimani, C. Gutwenger, P. Mutzel und H.-M. Wong (2008). *Layer-Free Upward Crossing Minimization*. In Lecture Notes in Computer Science, Band 5038/2008, Seiten 55–68. Department of Computer Science, Technical University of Dortmund, Germany, Springer Berlin / Heidelberg. (Zitiert auf Seite 30.)

[10] M. Chimani, C. Gutwenger und P. Mutzel (2009). *Upward planarization layout*. In Proceetings of Graph Drawing, Band 5849 von *LNCS*, Seiten 94–106. Springer. (Zitiert auf Seite 30.)

[11] M. Chimani, C. Gutwenger, P. Mutzel, M. Spönemann und H.-M. Wong (2011). *Crossing Minimization and Layouts of Directed Hypergraphs with Port Constraints*. In Proceedings of the 18th International Symposium on Graph Drawing (GD'10), Band 6502 von *LNCS*, Seiten 141–152. Springer. (Zitiert auf Seite 30.)

[12] J.-H. Chuang, C.-C. Lin und H.-C. Yen (2004). *Drawing Graphs with Nonuniform Nodes Using Potential Fields*. In G. Liotta, Herausgeber, Graph Drawing 2003, Band 2912 von *Lecture Notes in Computer Science*, Seiten 460–465. Dept. of Computer and Information Science, National Chiao-Tung University, Taiwan and Dept. of Electrical Engineering, National Taiwan University, Taiwan, Springer-Verlag Berlin Heidelberg. (Zitiert auf Seite 25.)

[13] G. B. Dantzig (1966). *Lineare Programmierung und Erweiterungen*. Springer-Verlag. (Zitiert auf Seite 33.)

[14] G. B. Dantzig und M. N. Thapa (2003). *Linear Programming 2: Theory and Extensions*. Springer. (Zitiert auf Seite 36.)

[15] R. Davidson und D. Harel (1989). Drawing graphs nicely using simulated annealing. Technischer Bericht CS 89-13, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot. (Zitiert auf Seite 25.)

[16] R. Davidson und D. Harel (1996). *Drawing Graphics Nicely Using Simulated Annealing*. ACM Trans. Graph., **15**(4), 301–331. (Zitiert auf den Seiten 25 und 108.)

[17] G. Di Battista, P. Eades, R. Tamassia und I. G. Tollis (1994). *Algorithms for Drawing Graphs: an Annotated Bibliography*. Computational Geometry: Theory and Applications, **4**, 235–282. (Zitiert auf Seite 23.)

[18] dSpace GmbH (2010). *dSPACE Modeling Guidelines for TargetLink*. dSpace GmbH. URL `http://www.dspace.de/de/gmb/home/support/kb/kbtl/tlmodguide/tlapp_modelguide.cfm`. (Zitiert auf Seite 21.)

[19] T. Dwyer, K. Marriott und M. Wybrow (2007). *Integrating Edge Routing into Force-Directed Layout*. In Graph Drawing, Band 4372 von *Lecture Notes in Computer Science*, Seiten 8–19. Springer. (Zitiert auf Seite 25.)

[20] P. Eades (1984). *A Heuristic for Graph Drawing*. Congressus Numerantium, **42**, 149–160. (Zitiert auf Seite 24.)

[21] P. Eades (1994). *Drawing graphs in two layers*. Theoretical Computer Science, **131**, 361–374. (Zitiert auf Seite 50.)

[22] P. Eades und N. C. Wormald (1994). *Edge Crossings in Drawings of Bipartite Graphs*. Algorithmica, **11**, 379–403. (Zitiert auf Seite 28.)

[23] P. Eades, X. Lin und W. F. Smyth (1993). *A fast and effective Heuristic for the feedback Arc Problem*. Information Processing Letters, **47**, 319–323. (Zitiert auf Seite 40.)

[24] Eclipse Foundation (2011). *Eclipse*. Online. URL `http://eclipse.org/`. (Zitiert auf Seite 31.)

[25] M. Eiglsperger, U. Fößmeier und M. Kaufmann (2000). *Orthogonal Graph Drawing With Constraints*. In Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, Seiten 3–11. (Zitiert auf Seite 30.)

[26] M. Eiglsperger, M. Siebenhaller und M. Kaufmann (2004). *An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing*. In Graph Drawing 2004, Band 3383 von *Lecture Notes in Computer Science*, Seiten 155–166. Springer-Verlag Berlin Heidelberg. URL `http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=3383{&}spage=155`. (Zitiert auf Seite 25.)

[27] K. Eikland, P. Notebaert und J. Ebert (2010). *Introduction to lp_solve 5.5.2.0*. Online. URL `http://lpsolve.sourceforge.net/5.5/`. (Zitiert auf den Seiten 36 und 90.)

[28] J. Eker, J. W. Janneck, E. A. Lee, A. Jiu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs und Y. Xiong (2003). *Taming Heterogeneity - the Ptolemy Approach*. In Proceedings of the IEEE, Band 91/2. (Zitiert auf Seite 3.)

[29] Esterel Technologies, Inc (2011). *SCADE Suite :: Products*. Online. URL `http://www.esterel-technologies.com/products/scade-suite/`. (Zitiert auf Seite 3.)

[30] ETAS GmbH, Stuttgart, Deutschland (2011). *ASCET Software-Produkte*. Online. URL `http://www.etas.com/de/products/ascet_software_products.php`. (Zitiert auf den Seiten 3 und 143.)

[31] E. Foundation (2011). *Eclipse Public License - v 1.0*. Online. URL `http://www.eclipse.org/legal/epl-v10.html`. (Zitiert auf den Seiten 31 und 143.)

[32] M. Fowler, K. Beck, J. Brant, W. Opdyke und D. Robert (2002). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. (Zitiert auf den Seiten 6 und 74.)

[33] Free Software Foundation Inc. (1999). *GNU Lesser General Public License*, 2. Auflage. URL `http://lpsolve.sourceforge.net/5.5/LGPL.htm`. (Zitiert auf den Seiten 36 und 143.)

[34] H. Fuhrmann (2011). *KIVi: KIELER View Management*. Online. URL `http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KiVi`. (Zitiert auf den Seiten 31 und 143.)

[35] H. Fuhrmann und R. von Hanxleden (2010). *On the Pragmatics of Model-Based Design*. In Monterey'08 Proceedings of the 15th Monterey conference on Foundations of Computer Software: future Trends and Techniques for Development. Springer. (Zitiert auf Seite 31.)

[36] H. Fuhrmann und R. von Hanxleden (2010). *Taming Graphical Modeling*. In Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). LNCS, Springer. (Zitiert auf den Seiten 4 und 32.)

[37] H. Fuhrmann, M. Spönemann und R. von Hanxleden (2010). *KLAY Subproject of KIELER Framework*. Online. URL `https://rtsys.informatik.uni-kiel.de/svn/kieler/trunk/plugins/de.cau.cs.kieler.klay.layered`. (Zitiert auf den Seiten 31 und 143.)

[38] H. Fuhrmann, M. Spönemann und R. von Hanxleden (2010). *Kiel Integrated Environment for Layout Eclipse Rich Client*. URL `http://www.informatik.uni-kiel.de/rtsys/kieler/`. (Zitiert auf den Seiten 9 und 143.)

[39] Fujaba Tool Suite Developer Team (2011). *FUJABA Tool Suite*. Online. URL `http://www.fujaba.de/`. (Zitiert auf den Seiten 120 und 143.)

[40] D. Gale (2007). *Linear Programming and the Simplex Method*. Notices of the AMS, **54**(3), 364–369. (Zitiert auf Seite 33.)

[41] E. R. Gansner, E. Koutsofios, S. C. North und K.-P. Vo (1993). A technique for drawing directed graphs. Technischer Bericht, AT&T Bell Laboratories. (Zitiert auf Seite 25.)

[42] E. R. Gansner, E. Koutsofios, S. C. North und K.-P. Vo (1993). *A Technique for Drawing Directed Graphs*. IEEE Transactions on Software Engineering, **19**, 214–230. (Zitiert auf den Seiten 27, 29, 39, 42, 47, 48, 51, 52, 117 und 118.)

[43] M. R. Garey und D. S. Johnson (1983). *Crossing Number is NP-Complete*. SIAM J. Algebraic Discrete Methods, **4**(3), 312–316. (Zitiert auf Seite 28.)

[44] Gurobi Optimization Inc. (2011). *Gurobi Optimizer Homepage*. URL http://www.gurobi.com/. (Zitiert auf den Seiten 35 und 90.)

[45] C. Gutwenger, K. Klein und P. Mutzel (2008). *Planarity Testing and Optimal Edge Insertion with Embedding Constraints*. Journal of Graph Algorithms and Applications, **12**(1), 73–95. URL http://jgaa.info/. (Zitiert auf Seite 30.)

[46] D. Harel und Y. Koren (2000). Drawing graphs with non-uniform verticles. Technischer Bericht, Dept. of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel. (Zitiert auf Seite 25.)

[47] T. W. Hill Jr. und A. Ravindran (1975). *On Programming with Absolute-Value Functions*. Journal of Optimization Theory and Applications, **17**(1/2), 181–183. (Zitiert auf Seite 35.)

[48] A. Hunter, P. Aurelien und A. Boyko (2011). *Graphical Modeling Framework (GMF) Runtime*. Online. URL http://www.eclipse.org/modeling/gmp/?project=gmf-runtime. (Zitiert auf den Seiten 31 und 143.)

[49] A. Hunter, A. Nyssen, F. Steeg und I. Bull (2011). *Graphical Editing Framework (GEF)*. Online. URL http://www.eclipse.org/gef/. (Zitiert auf den Seiten 31 und 143.)

[50] K. Hussey, E. Merks und E. Stepper (2011). *Eclipse Modeling Framework*. Online. URL http://www.eclipse.org/modeling/emf/. (Zitiert auf den Seiten 31 und 143.)

[51] M. Jünger und P. Mutzel (1997). *2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms*. Journal of Graph Algorithms and Applications, **1**(1), 1–25. (Zitiert auf Seite 28.)

[52] R. E. Kahn und V. G. Cerf (1981). Rfc 693 transmission control protocol. Technischer Bericht, Information Sciences Institute, University of Southern California. (Zitiert auf den Seiten 83 und 144.)

[53] L. W. Kantorowitsch (1960). *Mathematical Methods of Organizing and Planning Production*. Management Schience, **6**(4), 366–422. (Zitiert auf Seite 33.)

[54] R. M. Karp (1972). *Reducability among Combinatorial Problems*. In Complexity of Computer Computations. (Zitiert auf Seite 47.)

[55] M. Kaufmann und D. Wagner (2001). *Drawing Graphs: Methods and Models*, Band 2025 von *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg. (Zitiert auf Seite 23.)

[56] L. K. Klauske und C. Dziobek (2010). *Improving Modeling Usability: Automated Layout Generation for Simulink*. In Mathworks Automotive Conference. The Mathworks, Stuttgart. (Zitiert auf Seite ix.)

[57] L. K. Klauske und C. Dziobek (2011). *Effizientes Erstellen von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus*. In H. Giese, M. Huhn, J.Philipps, und B. Schätz, Herausgeber, Tagungsband zum Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, Seiten 127–138. Nummer 7 in Model-Based Development of Embedded Systems, fortiss GmbH, Guerickestr. 25, 80805 Muenchen. (Zitiert auf Seite ix.)

[58] L. K. Klauske, C. D. Schulze, M. Spönemann und R. von Hanxleden (2012). *Improved Layout for Data Flow Diagrams with Port Constraints*. In Diagrams 2012: The 7th International Conference on the Theory and Application of Diagrams, Accepted Paper. (Zitiert auf den Seiten ix, 31 und 32.)

[59] V. Klee, G. J. Kinty und O. E. Shisha (1972). *How good is the simplex algorithm?* Inequalities, **3**, 159–175. (Zitiert auf Seite 33.)

[60] Y.-Y. Lee, C.-C. Lin und H.-C. Yen (2006). *Mental Map Preserving Graph Drawing Using Simulated Annealing*. In K. Misue, K. Sugiyama, und J. Tanaka, Herausgeber, Asia Pacific Symposium on Information Visualisation (APVIS2006), Band 60 von *CRPIT*, Seiten 179–188. ACS, Tokyo, Japan. URL `http://crpit.com/confpapers/CRPITV60Lee.pdf`. (Zitiert auf Seite 25.)

[61] E. Legros, W. Schäfer, A. Schürr und I. Stürmer (2010). *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*, Band 6100 von *Lecture Notes in Computer Science (LNCS)*, Kapitel MATE – A Model Analysis and Transformation Environment for MATLAB Simulink, Seiten 323–328. Springer Verlag. (Zitiert auf den Seiten 120 und 143.)

[62] Mathworks (2011). *Simulink R2011a Documentation*. The Mathworks. URL `http://www.mathworks.com/help/toolbox/simulink/`. (Zitiert auf den Seiten 9 und 83.)

[63] M. MathWorks Automotive Advisory Board (2007). *Control Algorithm Modeling Guidelines Using Matlab, Simulink and Stateflow*. The Mathworks. (Zitiert auf den Seiten 21 und 143.)

[64] M. Matzen (2010). *A Generic Framework for Structure-Based Editing of Graphical Models in Eclipse*. Diploma thesis, Christian-Albrechts-Universitaet zu Kiel. URL `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf`. (Zitiert auf den Seiten 4 und 32.)

[65] M. Matzen (2011). *KSBASE – Kieler Structure Based Editing*. Online. URL http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KSBasE. (Zitiert auf den Seiten 31 und 143.)

[66] MISRA Consortium (2009). *MISRA AC SLSF: Modelling design and style guidelines for the application of Simulink and Stateflow*. MISRA Consortium. URL http://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-ac-slsf. (Zitiert auf den Seiten 21 und 143.)

[67] H. Mittelmann (2011). *Mixed Integer Linear Programming Benchmark*. Online. URL http://plato.asu.edu/ftp/milpf.html. (Zitiert auf den Seiten 35 und 36.)

[68] K. Muisue, P. Eades, W. Lai und K. Sugiyama (1995). *Layout Adjustment and the Mental Map*. Journal of Visual Languages and Computing, **6**, 183–210. (Zitiert auf den Seiten 25 und 100.)

[69] A. Noack (2007). *Energy Models for Graph Clustering*. Journal of Graph Algorithms and Applications, **11**(2), 453–480. (Zitiert auf Seite 25.)

[70] Object Management Group, Inc. (2009). *Documents associated with UML Version 2.2*. Online. URL http://www.omg.org/spec/UML/2.2/. (Zitiert auf den Seiten 84 und 144.)

[71] A. Papakostas, J. M. Six und I. G. Tollis (1997). *Experimental and Theoretical Results in Interactive Orthogonal Graph Drawing*. In Proceedings of the Symposium for Graph Drawing, Seiten 371–386. Springer-Verlag. (Zitiert auf Seite 108.)

[72] Paragon Decision Technology B.V., Herausgeber (2010). *AIMMS Modeling Guide*, Kapitel VI: Linear Programming Tricks, Seiten 63–75. Paragon Decision Technology B.V., 3. Auflage. URL http://www.aimms.com. (Zitiert auf Seite 34.)

[73] M. Petre (1995). *Why looking isn't always seeing: readership skills and graphical programming*. Communications of the ACM, **38**, 33–44. ISSN 0001-0782. URL http://doi.acm.org/10.1145/203241.203251. (Zitiert auf den Seiten 3 und 108.)

[74] S. Prochnow (2008). *Efficient development of complex statecharts*. Dissertation, Technische Fakultät, Christian-Albrechts-Universitaet zu Kiel. (Zitiert auf Seite 32.)

[75] S. Prochnow (2009). *Kiel Integrated Environment for Layout*. Online. URL http://www.informatik.uni-kiel.de/rtsys/kiel/. (Zitiert auf den Seiten 31 und 143.)

[76] H. C. Purchase, E. Hoggan und C. Görg (2007). *How Important is the Mental Map? – An Empirical Investigation of a Dynamic Graph Layout Algorithm*. In Graph Drawing 2006, Band 4372 von *LNCS*, Seiten 184–195. Springer. (Zitiert auf den Seiten 25, 100 und 108.)

[77] G. Sander (1994). Graph layout through the vcg tool. Technischer Bericht A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken. (Zitiert auf Seite 29.)

[78] G. Sander (1995). *A fast heuristic for hierarchical Manhattan layout*. In Proceedings of the Symposium on Graph Drawing, Band 1027 von *LNCS*, Seiten 447–458. Springer. (Zitiert auf Seite 30.)

[79] G. Sander (2004). *Layout of Directed Hypergraphs with Orthogonal Hyperedges*. In Graph Drawing 2003, Seiten 381–386. LNCS, Springer. (Zitiert auf Seite 25.)

[80] G. Sander und A. Vasiliu (2001). *The ILOG JViews graph layout module*. In Proceedings of the 9th International Symposium on Graph Drawing, Band 2265 von *LNCS*, Seiten 469–475. Springer-Verlag. (Zitiert auf Seite 30.)

[81] C. D. Schulze (2011). *Optimizing Automatic Layout for Data Flow Diagrams*. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. URL `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cds-dt.pdf`. (Zitiert auf den Seiten 30, 31, 32, 55 und 119.)

[82] D. Shanno und R. Weil (1971). *"Linear" Programming with Absolute Value Functionals*. Operations Research, **19**(1), 0. (Zitiert auf Seite 35.)

[83] F. Shull, J. Singe und D. I. Sjøberg, Herausgeber (2008). *Guide to Advanced Empirical Software Engineering*. Springer-Verlag. (Zitiert auf Seite 96.)

[84] M. Spönemann, H. Fuhrmann, R. von Hanxleden und Mutzel (2009). *Port Constraints in Hierarchical Layout of Data Flow Diagrams*. In Proceedings of the 17th International Symposium on Graph Drawing (GD'09). Springer. (Zitiert auf den Seiten 28, 30, 31, 54, 89 und 117.)

[85] K. Sugiyama, S. Tagawa und M. Toda (1980). *Methods for Visual Understanding of Hierarchical System Structures*. In IEEE Transactions on Systems, Man and Cybernetics, Band 11/2 von *IEEE Transactions*, Seiten 109–125. (Zitiert auf den Seiten 25, 27, 29, 108 und 117.)

[86] R. Tamassia (1987). *On Embedding a Graph in the Grid with the Minimum Number of Bends*. SIAM J. Comput., **16**(3), 421–444. (Zitiert auf den Seiten 5, 30 und 108.)

[87] R. Tamassia, G. Di Battista und C. Batini (1988). *Automatic Graph Drawing and Readability of Diagrams*. In IEEE Transactions on Systems, Man and Cybernetics, Band 18 von *IEEE Transactions*, Seiten 61–79. IEEE Systems, Man, and Cybernetics Society. ISSN 0018-9472. (Zitiert auf Seite 30.)

[88] I. G. Tollis, G. Di Battista, P. Eades und R. Tamassia (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Alan Apt. ISBN 0133016153. (Zitiert auf den Seiten 23, 26 und 107.)

[89] Tom Sawyer Software (2011). *Homepage*. Online. URL `http://www.tomsawyer.com/home/index.php`. (Zitiert auf Seite 30.)

[90] J. Voelcker (2008). *A quanititative analysis of Statechart aesthetics and Statechart development methods*. Diploma thesis, Christian-Albrechts-Universitaet zu Kiel. URL `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jovo-dt.pdf`. (Zitiert auf Seite 108.)

[91] J. N. Warfield (1977). *Crossing Theory and Hierarchy Mapping*. IEEE Trans. Syst. Man Cybern., 7(7), 505–523. (Zitiert auf den Seiten 25 und 28.)

[92] I. Wegener (2003). *Komplexitätstheorie – Grenzen der Effizienz von Algorithmen*. Springer-Verlag. (Zitiert auf Seite 33.)

[93] M. Wernicke (2008). *AUTOSAR auf dem Weg in die Serie*. Elektronik Praxis, **02**. URL `http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/analyseentwurf/articles/105576/`. (Zitiert auf Seite 3.)

[94] F. Wohlgemuth, C. Dziobek und T. Ringler (2008). *Erfahrungen bei der Einführung der modellbasierten AUTOSAR-Funktionsentwicklung*. In Tagungsband des Modellierungs-Workshops Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Seiten 6–15. (Zitiert auf Seite 3.)

[95] R. Wunderling (1996). *Paralleler und Objektorientierter Simplex-Algorithmus*. Dissertation, TU Berlin. (Zitiert auf Seite 36.)

ABKÜRZUNGSVERZEICHNIS

ASCET	Advanced Simulation and Control Engineering Tool ® [30]
AUTOSAR	AUTomotive Open System ARchitecture [3]
DTP	Desktop-Publishing
EMF	Eclipse Modeling Framework [50]
EPL	Eclipse Public License [31]
FUJABA	Fujaba Tool Suite [39]
GEF	Graphical Editing Framework [49]
GMF	Graphical Modeling Framework [48]
GUI	Graphical User Interface (Grafische Benutzerschnittstelle)
KIEL	Kiel Integrated Environment for Layout [75]
KIELER	Kiel Integrated Environment for Layout – Eclipse Rich Client [38]
KIVi	KIELER View Management [34]
KLay Layered	KIELER Layouters for Layered Graphs [37]
KSBasE	KIELER Structure Based Editing [65]
LGPL	GNU Lesser General Public License [33]
MAAB	Mathworks Automotive Advisory Board [63]
MATE	Model Analysis and Transformation Environment for MATLAB Simulink [61]
MISRA	Motor Industry Software Reliability Association [66]
ML	MATLAB®

RTW	RealTime Workshop®
SCIP	Solving Contraint Integer Programming [2]
SL	Simulink®
TCP	Transmission Control Protocol [52]
TL	Targetlink®
UML	Unified Modeling Language [70]
WYSIWYG	*What you see is what you get*