Universität Koblenz-Landau, Fachbereich 4
Institut für Management

Bachelorarbeit

Ein Layoutalgorithmus für Geschäftsprozessmodelle

Zur Erlangung des Grades eines Bachelor of Science Informationsmanagement

vorgelegt von
Christoph Schneider

Betreuer: Dr. Carlo Simon
Fachbereich 4, Institut für Management

Erstgutachter: Dr. Carlo Simon
Fachbereich 4, Institut für Management

Zweitgutachter: Prof. Dr. Klaus G. Troitzsch
Fachbereich 4, Institut für Wirtschafts- und Verwaltungsinformatik

Koblenz, 19. Juni 2007

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den 19. Juni 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Listings

Kapitel 1

Einleitung

Generally speaking, a business process is a continuous series of enterprise tasks, undertaken for the purpose of creating output. The starting point and final product of the business process is the output requested and utilized by corporate or external “customers”.

August-Wilhelm Scheer

Geschäftsprozesse werden für Menschen erstellt, um ihnen die Grundlage für ihr unternehmerisches Handeln zu liefern. Ein Geschäftsprozessmodell ist eine vereinfachte Darstellung von Geschäftsprozessen, die unwesentliche Details ausblendet und wichtige Bestandteile fokussiert. Dadurch dienen sie besonders Fachleuten, die mit der Grundlage bereits vertraut sind, als Diskussionsgrundlage (Mielke, 2002). Das Verständnis dieser Modelle ist die Grundvoraussetzung dafür und wird erheblich durch eine strukturierte Visualisierung vereinfacht, die die Geschäftsprozesse intuitiv verständlich macht.

Eine Art der Darstellung von Geschäftsprozessen sind die Modulnetze, eine spezielle Form von Petri-Netzen. Sie bieten die Möglichkeit einer dynamischen grafischen statt rein textuellen Veranschaulichung. Die Software *Join* ist in der Lage, Prozessdefinitionen textueller Art entgegenzunehmen und sie in ein Modulnetz zu überführen.
Diese Arbeit beschäftigt sich mit der Entwicklung eines Layout-Moduls für *Join*, das eine grafische Darstellung der Modulnetze generiert und zur Anzeige aufbereitet.

Die Arbeit ist in sieben Kapitel gegliedert. Nach der Einleitung folgt ein Kapitel, das einige Grundlagen der Graphentheorie erläutert. Dabei wird stets genau das bedacht, was für diese Arbeit relevant ist. Es wird auf mathematische Grundlagen, die Abgrenzung zwischen Petri-Netzen und Modulnetzen sowie die Design-Kriterien eingegangen, die dem Layout eines solchen Netzes zu Grunde liegen.

Das darauf folgende Kapitel beschäftigt sich mit der Verteilung aller zum Netz gehörigen Elemente auf einer gedachten Zeichenfläche durch Zuteilung abstrakter Koordinaten mit Hilfe geeigneter Algorithmen. Dies geschieht in drei aufeinanderfolgenden Schritten: Setzen der X-Koordinaten, Erstellen von Dummy-Knoten und Setzen der Y-Koordinaten. Algorithmen, die dafür implementiert und als ungeeignet identifiziert werden konnten, werden hier ebenfalls dokumentiert, um den Entwicklungsprozess zu veranschaulichen.

Kapitel 4 ist der Generierung einer SVG-Datei aus dem abstrakten Layout gewidmet. Eine solche Datei stellt eine Vektorgrafik dar, die mit einem geeigneten Programm am Bildschirm betrachtet werden kann. Bei der Dokumentation der Erstellung der Grafik wird vom Allgemeinen zum Speziellen vorgegangen: Zunächst werden die Grundlagen von SVG erläutert, darauf folgen einfache Methoden zur Generierung des Netzes als SVG-Grafik und zum Schluss wird auf besondere Möglichkeiten von SVG eingegangen, die ein so erzeugtes Netz um ein vielfaches optisch und informativ aufwerten.

Die eigentliche Implementation aller in den beiden vorangegangenen Kapiteln vorgestellten Methoden ist im nächsten Kapitel dokumentiert. Da das entwickelte Modul keine eigenständige Lösung ist, sondern Teil des Gesamtpaketes *Join* ist und auch selbst aus zwei voneinander abgrenzbaren Teilen besteht, wird zunächst die Architektur von *Join* im Allgemeinen und von dem hier entwickelten Layout-Modul im Besonderen erklärt. Es wird eine Übersicht über die in PHP implementierten Klassen, Methoden und Attribute sowie die dem Customizing dienenden globalen Konstanten gegeben.

Kapitel 6 liefert einige Anwendungsbeispiele in Form von Screenshots automatisch generierter Modulnetze mit Kommentaren bezüglich deren Layout.

Das letzte Kapitel fasst rückblickend den Entwicklungsprozess der Layoutkomponente zusammen. Die wichtigsten aufgetretenen Probleme und gewonnenen Erkenntnisse werden hier nochmals erwähnt. Außerdem werden einige Hinweise für die finale Integration der Komponente in *Join* gegeben. Zuletzt werden einige Erweiterungsmöglichkeiten des in dieser Arbeit entwickelten Moduls aufgezählt.

Als Anlage ist dieser Arbeit eine CD-ROM beigelegt, auf der die Folgenden Dateien enthalten sind:

- Diese Arbeit als PDF-Dokument *BA_LayoutalgorithmusGPM.pdf*
- Der lauffähige Programmcode der in dieser Arbeit entwickelten Layoutkomponente im Verzeichnis *\Join*.
- Einige automatisch generierte Modulnetze als SVG-Dateien im Verzeichnis *\SVG-Beispielnetze*.

- XAMPP in der Version 1.5.5 (`http://www.apachefriends.org/de/xampp.html`). Das Paket ermöglicht u.a. die Installation eines Apache-Webservers und PHP zur Ausführung des Programmcodes. Nach der Installation kopieren Sie den gesamten Ordner *\Join* in das Verzeichnis *\htdocs*, das sich in Ihrem XAMPP-Installationsverzeichnis befindet. Nach dem Start des Apache-Servers geben Sie in einem SVG-fähigen Webbrowser die Adresse `http://localhost/Join` ein, um die Testumgebung des Layout-Moduls aufzurufen. Durch Veränderung der Werte der globalen Konstanten in der Datei *config.php* im Verzeichnis *\Join* kann das Netzlayout verändert werden.

- Adobe SVG-Viewer 3.0.3 (`http://www.adobe.com/svg/viewer/install/`) zur Anzeige der SVG-Dateien im MS Internet Explorer.

- Den Webbrowser Mozilla Firefox 2.0.0.4 (`http://www.mozilla-europe.org/de/products/firefox/`) mit nativer SVG-Unterstützung.

Kapitel 2

Graphentheoretische Grundlagen

Ziel der Arbeit ist es, auf Basis eines zugrunde liegenden Modells zur formalen Beschreibung von Geschäftsprozessen, automatisch ein Modulnetz zu erzeugen, also eine Petri-Netz-Beschreibung. Da ein solches Netz mathematisch ein Graph ist, sind die Begrifflichkeiten und Formalismen der Graphentheorie soweit verwendet einzuführen. Dieses Kapitel beginnt in Abschnitt 2.1 mit einer Zusammenstellung der benutzten Fachbegriffe aus den Grundlagen der Graphentheorie. Danach werden in Abschnitt 2.2 die grundlegenden Eigenschaften von Petri-Netzen und Modulnetzen erläutert. Zum Schluss folgt in Abschnitt 2.3 eine Erläuterung der Anforderungen an das Layout von Modul-Netzen, die sich aus dem spezifischen Verwendungszweck dieser Graphen ergeben.

2.1 Begriffe der Graphentheorie

In diesem Abschnitt werden die benutzten Fachbegriffe aus den Grundlagen der Graphentheorie erläutert.

DEFINITION 2.1.1 (GERICHTETER GRAPH)

Ein gerichteter Graph ist nach Krumke und Noltemeier (2005, S. 7) ein Quadrupel $G = (V, E, \alpha, \beta)$, bestehend aus einer Knotenmenge V, einer Kantenmenge E und den Abbildungen $\alpha : E \to V$ und $\beta : E \to V$. Dabei ist $\alpha(e)$ die Anfangsecke der Kante $e \in E$ und $\beta(e)$ die Endecke der Kante $e \in E$. Somit lässt sich jede Kante $e \in E$ als Pfeil von $\alpha(e)$ nach $\beta(e)$ interpretieren. $\Diamond$

DEFINITION 2.1.2 (INZIDENZ UND ADJAZENZ)

Ein Knoten $v \in V$ und eine Kante $e \in E$ heißen inzident, wenn entweder $\alpha(e) = v$ oder $\beta(e) = v$. Zwei Knoten v_1 und v_2 heißen adjazent, wenn sie zur selben Kante $e \in E$ inzident sind. Im Folgenden werden adjazente Knoten auch als *Nachbar-*, *Vorgänger-* und *Nachfolgeknoten* bezeichnet. $\Diamond$

DEFINITION 2.1.3 (INNENGRAD UND AUSSENGRAD)

Der Innengrad eines Knotens $v \in V$, nachfolgend auch $indegree(v)$ genannt, ist die Summe aller eingehenden Kanten in diesen Knoten, d.h. die Summe aller Kanten $e \in E$, für die gilt $\beta(e) = v$.
Der Außengrad eines Knotens $v \in V$, nachfolgend auch $outdegree(v)$ genannt, ist die Summe aller ausgehenden Kanten aus diesem Knoten, d.h. die Summe aller Kanten $e \in E$, für die gilt $\alpha(e) = v$.

$\Diamond$

DEFINITION 2.1.4 (PRESET UND POSTSET)

Das Preset eines Knotens $v \in V$ bilden alle direkten Vorgängerknoten dieses Knotens, d.h. alle Knoten $u_i \in V$, für die gilt: $\beta(u_i) = v$.
Das Postset eines Knotens $v \in V$ bilden alle direkten Nachfolgerknoten dieses Knotens , d.h. alle Knoten $u_i \in V$, für die gilt: $\alpha(u_i) = v$. $\Diamond$

DEFINITION 2.1.5 (QUELLE UND SENKE)

Eine Quelle eines Graphen $G = (V, E, \alpha, \beta)$ ist ein Knoten $v \in V$, der den Innengrad 0 besitzt.
Eine Senke eines Graphen $G = (V, E, \alpha, \beta)$ ist ein Knoten $v \in V$, der den Außengrad 0 besitzt. $\Diamond$

2.2 Petri-Netze und Modulnetze

Ein Petri-Netz besteht aus Stellen und Transitionen, die durch Pfeile miteinander verbunden sind. Stellen sind dabei vergleichbar mit Objekten und Transitionen mit Aktionen, die diese Objekte beeinflussen. Ist eine Stelle *markiert* so bedeutet dies, dass diese Stelle in diesem Moment mit einem (oder mehreren) Objekt(en) besetzt ist. Die Markierung aller Stellen im Preset einer Transition ist Voraussetzung für ihre Aktivierung, d.h. für ihre Fähigkeit die mit ihr verbundene Aktion durchzuführen. Diese Durchführung wird als *Schalten* einer Transition bezeichnet. Die Marken auf den Stellen im Preset der Transition, den *Eingangsstellen*, werden verbraucht. Auf den Stellen im Postset, den *Ausgangsstellen*, werden neue Marken erzeugt. Die Anzahl der erzeugten Marken kann dabei durch Kantengewichte beeinflusst werden. Die Bedingungen zur Aktivierung einer Transition und deren Einfluss auf adjazente Stellen bezeichnet man als *Schaltregel* (Baumgarten, 1990).

Elementare Bestandteile eines Modulnetzes sind die Aktionen oder Elementarprozesse. Innerhalb eines Moduls sind diese durch Operatoren miteinander verbunden. Diese Operatoren können Alternative, Nebenläufigkeit, Aufeinanderfolge, Synchronisation, Wiederholung oder Verneinung von Aktionen bewirken. Kanonische Regeln erstellen aus den Modulen Modulnetze. Diese sind interpretierbar als Petri-Netze mit expliziter Start- und Zieltransition (Simon u. a., 2006, S. 4). Eine formale Erklärung liefert die folgende Definition nach Simon und Dehnert (2004, S. 5):

DEFINITION 2.2.1 (MODULNETZ)

Ein Modulnetz $M = (M, i)$ besteht aus einem Petri-Netz $M = (P, T, F)$ mit der Starttransition s ($indegree(s) = 0$) und der Zieltransition g ($outdegree(g) = 0$) und einer Interpretationsfunktion i

über T mit $i(s) \rightarrow start$ und $i(g) \rightarrow goal$ und für alle anderen Transitionen t $i(t) \rightarrow E_A$, wobei E_A eine (moglicherweise leere) Menge nicht widersprüchlicher Elementarprozesse über eine Menge von Aktionen A ist. ◊

DEFINITION 2.2.2 (SCHALTFOLGE)
Eine Schaltfolge ist die Reihenfolge, in der Transitionen von einer Anfangstransition bis zu einer Endtransition schalten. ◊

Nach Böhm (2000, S. 223) werden solche Schaltfolgen auch *Traces* genannt.

DEFINITION 2.2.3 (PROZESS)
Innerhalb eines Modulnetzes ist ein Prozess jede Schaltfolge, beginnend bei der Starttransition, die durch Erreichen der Zieltransition die leere Startmarkierung reproduziert. ◊

DEFINITION 2.2.4 (KONTROLLFLUSS)
Kontrollflüsse regeln, in Übereinstimmung mit einer sensiblen Prozesslogik, wie Ereignisse angestoßen werden. Sie bestehen im Wesentlichen aus Informationen, die von einem Ereignis an das jeweils nächste weitergeleitet werden und dort die Durchführung einer Aktion anstoßen und beeinflussen. (Scheer, 1999, S. 18) ◊

Nach Jablonski und Bussler (1996) ist ein Kontrollfluss kürzer definiert als die "Ausführungsreihenfolge eines Workflows". Die bereits erwähnte Prozesslogik beinhaltet Konstrukte wie Sequenzen, Parallelen, Alternativen, Bedingungen und Schleifen, womit auch komplexeres Verhalten von Workflows nachgebildet werden kann. Dabei hat jeder Kontrollfluss jedoch nur genau einen definierten Start- und Endzustand.

2.3 Anforderungen an das Layout von Modul-Netzen

Das Layout eines Graphen ist die Anordnung aller seiner Knoten und Kanten auf dem Bildschirm. Jeder Knoten bekommt einen genau definierten Platz zugewiesen. Dieser Platz hängt vor allem von den zu dem Knoten adjazenten Knoten ab. Damit der Gesamteinduck des Graphen gut ist, sind einige Kriterien zu beachten.

Die *ästhetische* Darstellung von Modulnetzen ist ein Ziel dieser Arbeit. Das generierte Layout soll ruhig, strukturiert und sauber wirken, wichtige Informationen hervorheben, dabei jedoch keine Informationen verfälschen oder gar vorenthalten. *Schlechtes* Aussehen des Graphen könnte für den

Betrachter negative Assoziationen zu dessen inhaltlicher Richtigkeit auslösen, was zu vermeiden ist. Für die Ästhetik eines Modulnetzes sind einige grundlegende Kriterien zu berücksichtigen (Ganser u. a., 1993, S. 2)

- Eine generelle Flussrichtung von links nach rechts lässt den Eindruck einer hierarchischen Struktur entstehen.
- Visuelle Anomalien, wie sich überschneidende Kanten oder spitz zulaufende Linienzüge, sind nach Möglichkeit zu vermeiden. Diese lenken den Betrachter ab und liefern keinerlei zusätzliche Information über den Graphen.
- Kurze Kanten lassen den Graph kompakt wirken und vereinfachen das Auffinden von Folge- bzw. Vorgängerknoten.
- Symmetrie und Balance machen den Graph *ruhig*. Die Konzentration des Betrachters wird nicht auf *unförmige* Abschnitte des Graphen gelenkt und er wirkt exakter.

Da es sich bei Modulnetzen um Geschäftsprozessmodelle handelt, gibt es ein weiteres Layoutkriterium, das in dieser Arbeit allen anderen übergeordnet ist:

- Möglichst viele aneinander gereihte und gleichgerichtete Kanten sorgen dafür, dass man zusammengehörige Prozesspfade leicht erkennt.

Dadurch wird eine Hierarchie in den Graph eingebracht, die sich an der Länge der Prozesspfade orientiert. Einige der oben genannten Kriterien können durch diese Art der Modellierung des Graphen verletzt werden. So wird zum Beispiel das Vorkommen langer Kanten und eine geringe Anzahl sich überschneidender Kanten in Kauf genommen. Für den spezifischen Verwendungszweck ist dies jedoch sinnvoll.

Kapitel 3

Algorithmen zum Layout von Modul-Netzen

Dieses Kapitel liefert eine Zusammenstellung der entwickelten Algorithmen, die dem *Layouten* eines Modulnetzes dienen. Dazu wird jedem Knoten eine eindeutige *relative* Position zugewiesen, die von der Lage eines oder mehrerer anderer Knoten des Netzes abhängig ist.

3.1 Voraussetzungen

Um eine Menge von Knoten und Kanten ästhetisch anzuordnen, wird die *Zeichenfläche* bzw. der Bildschirm als Koordinatensystem verstanden. Sein Ursprung $(1, 1)$ liegt in der linken oberen Ecke. Die horizontale Lage eines Knotens wird dabei durch seine X-Koordinate bestimmt, die vertikale Lage durch seine Y-Koordinate. Damit ist die Position jedes Knotens auf der Zeichenfläche eindeutig durch ein Koordinatenpaar (x, y) bestimmt. Die niedrigste mögliche X- und Y-Koordinate ist jeweils 1. Abbildung 3.1 zeigt eine schematische Darstellung des Koordinatensystems.

Das Layout des Netzes wird nach einem *hierarchischen Ansatz* (Battista u. a., 1999) in drei aufeinanderfolgenden Hauptschritten generiert:

1. X-Koordinaten jedes Knotens bestimmen (Abschnitt 3.2)
   - Aufteilen der Knotenmenge in Flüsse
   - Flussweise Zuweisung der X-Koordinaten
2. Dummy-Knoten setzen (Abschnitt 3.3)

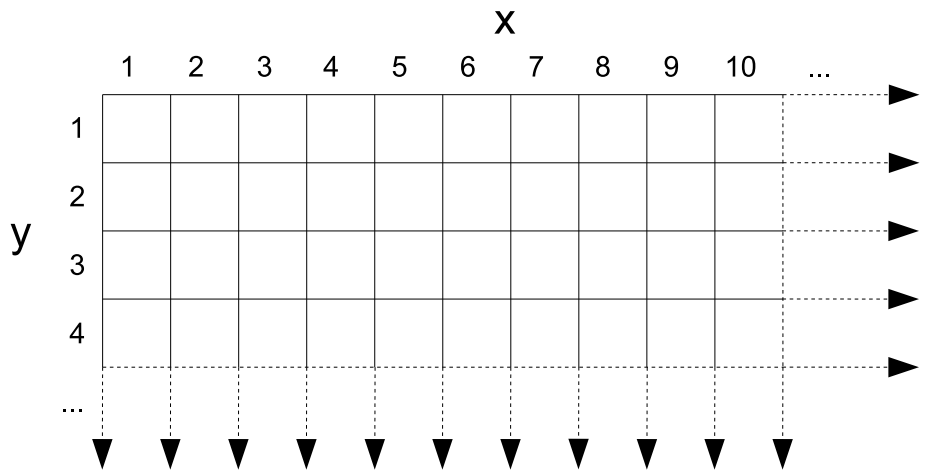

Abbildung 3.1: Koordinatensystem

- Dummy-Knoten erstellen
- Dummy-Knoten in Flüsse integrieren

3. Y-Koordinate jedes Knotens bestimmen (Abschnitt 3.4)
   - Y-Koordinaten flussweise vergeben
   - Kantenüberschneidungen reduzieren
   - Ausbalancieren des Graphen

Im gesamten Kapitel wird auf die Unterscheidung zwischen Stellen und Transitionen in den Abbildungen verzichtet, da diese für das Layouten des Netzes unerheblich ist.

3.2 Berechnung der X-Koordinaten

Die X-Koordinaten der Knoten des Graphen bestimmen dessen horizontale Ausrichtung auf dem Bildschirm. Damit später ein möglichst großer Ausschnitt des Graphen auf dem Bildschirm zu sehen ist, soll er kompakt vom linken Bildschirmrand ausgehend nach rechts angeordnet werden. Trotzdem sollen zusammengehörige Prozessketten durch lange, gleichgerichtete Kantenfolgen hervorgehoben werden.

Dieses Kapitel zeigt die Entwicklung eines Algorithmus zur Kalkulation einer X-Koordinate für jeden Knoten des Netzes. Besondere Aufmerksamkeit wird dem möglichen Vorkommen von Zyklen gewidmet, da dieses den Vorgang erheblich verkompliziert.

3.2.1 Erster Lösungsansatz

In diesem Abschnitt wird eine einfache Möglichkeit zur Kalkulation von X-Koordinaten vorgestellt. Dies ist gleichzeitig die Hinführung zu dem Problem möglicherweise vorkommender Zyklen.

Durch ein rekursives Durchlaufen des Graphen, angefangen bei der Quelle mit der bereits festgelegten X-Koordinate 1, wird jedem Nachfolgeknoten die X-Koordinate seines Vorgängerknotens, erhöht um 1, zugewiesen. Entsprechend wird jedem Vorgängerknoten die X-Koordinate des Nachfolgeknotens, vermindert um 1, zugewiesen. Algorithmus 3.1 verdeutlicht den Vorgang.

Daten : Knotenmenge ohne zugewiesene X-Koordinate
Ergebnis : Knotenmenge mit zugewiesener X-Koordinate
Eingabe : Knoten v, X-Koordinate x
Beginn
  wenn v *bereits besucht* **dann**
    Beende Rekursion;
  sonst
    Weise v die X-Koordinate x zu;
    Markiere v als besucht;
    für jedes $v_j \in postset(v)$ **tue**
      Führe Algorithmus aus mit v_j und $x + 1$;
    für jedes $v_i \in preset(v)$ **tue**
      Führe Algorithmus aus mit v_j und $x - 1$;
Ende

Algorithmus 3.1 : Einfache rekursive Kalkulation von X-Koordinaten

Abbildung 3.2 zeigt eine mit diesem Algorithmus erreichte Verteilung der X-Koordinaten am Beispiel eines einfachen Graphen, bei dem alle Pfeile in eine Richtung zeigen. Befinden sich im

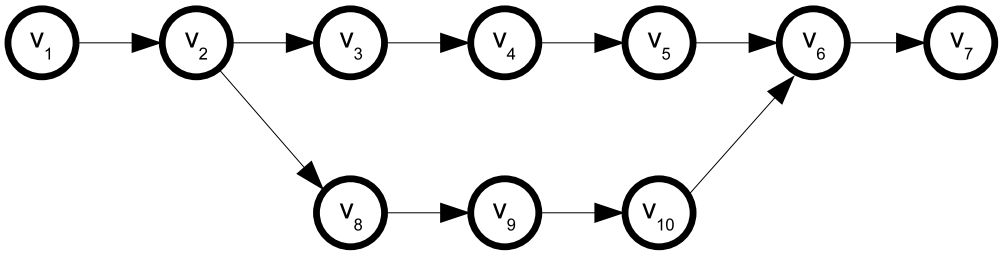

Abbildung 3.2: Gerichteter Graph

Graph auch rückwärts, d.h. in diesem Fall nach links gerichtete Kanten, so erzeugt der Algorithmus eine unbefriedigende Verteilung der X-Koordinaten (Abbildung 3.3). Der Unterschied zum Graph in Abbildung 1 besteht darin, dass die Kantenrichtungen der Kantenmenge $S = \{(v_6, v_{10}), (v_{10}, v_9), (v_9, v_8), (v_8, v_2)\}$ umgedreht wurden. Dadurch ist ein Zyklus entstanden, der auf Abbildung 3.4,

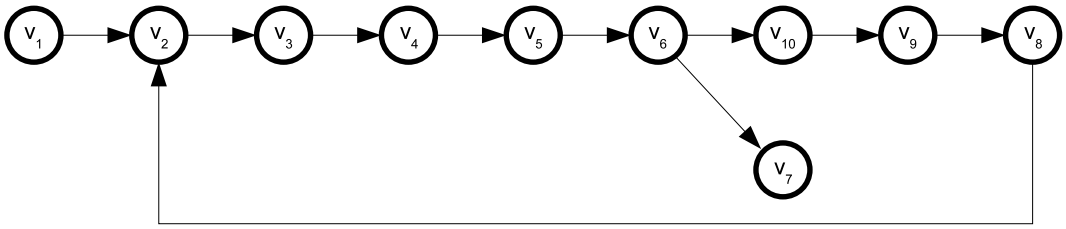

Abbildung 3.3: Gerichteter Graph mit Zyklus

welche die wünschenswerte Verteilung der X-Koordinaten in diesem Fall darstellt, mit gestrichelten Kanten hervorgehoben ist. Ein Lösungsansatz für dieses Problem ist das temporäre Umdrehen aller

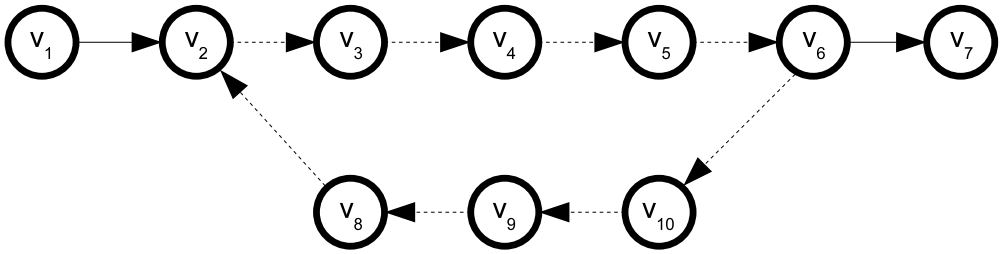

Abbildung 3.4: Gerichteter Graph mit hervorgehobenem Zyklus

Kanten der Kantenmenge S. Damit ist der zuerst vorgestellte zyklenfreie Graph wiederhergestellt, der mit Hilfe des Algorithmus problemlos darzustellen ist. Nach der Berechnung der X-Koordinaten sind lediglich die temporär gedrehten Kanten erneut umzudrehen und der Graph hat die Form aus Abbildung 3.4. Die Hauptquelle dieser Arbeit zum Thema Graphzeichnen (Battista u. a., 1999) stellt einen Algorithmus vor, der Zyklen in einem Graph eliminiert (siehe Kapitel 3.2.4). Ziel dieses Algorithmus ist es, eine möglichst kleine Menge von Kanten zu finden, durch deren Umdrehen man sämtliche Zyklen eines Graphen *bricht*. Diese Kantenmenge entspricht jedoch nicht der oben genannten Kantenmenge S, weshalb dieser Algorithmus für die Lösung des Problems unbrauchbar ist.

Daher wird ein Algorithmus benötigt, der die X-Koordinaten dahingehend optimiert, dass eine Verteilung nach Abbildung 3.4 entsteht.

3.2.2 Behandlung von Zyklen und langen Kanten

In diesem Abschnitt ist die Entwicklung eines Algorithmus beschrieben, der sämtliche zurückgerichtete Kanten, wie die des Graphen in Abbildung 3.4, identifiziert und diese dreht, um anschließend den Knoten X-Koordinaten zuzuweisen. Außerdem wird das Problem langer (das heißt mehrere Knoten überspringender) Kanten behandelt.

Abbildung 3.3 zeigt den zyklischen Graphen, auf den der Algorithmus 3.1 zur initialen Verteilung der X-Koordinaten angewendet wurde. Dabei fällt auf, dass es nur eine nach links gerichtete Kante $e_1 = (v_8, v_2)$ gibt. Dreht man diese Kante und wendet erneut einen Algorithmus zur Verteilung von X-Koordinaten an, so wird der Knoten v_8 durch die nun nach rechts gerichtete Kante e_1 als direkter Nachfolger von Knoten v_2 auf dessen nachfolgende X-Koordinate gesetzt. Dadurch ist die Kante $e_2 = (v_9, v_8)$ nach links gerichtet. Mit dieser verfährt man durch Drehen genau wie mit Kante e_1 zuvor und wendet daraufhin erneut einen Verteilungsalgorithmus an. Die Schritte *nach links gerichtete Kante drehen* und *Verteilungsalgorithmus anwenden* wiederholt man so lange, bis es keine nach links gerichtete Kante mehr gibt. Danach bringt man die gedrehten Kanten wieder in ihre ursprüngliche Richtung. Der Pseudocode eines Algorithmus 3.2 verdeutlicht das Vorgehen.

Eingabe : Knotenmenge V eines Graphen mit links- und rechtsgerichteten Kanten ohne X-Koordinaten

Ausgabe : Knotenmenge V mit zugewiesenen X-Koordinaten

solange *linksgerichtete Kante vorhanden* **tue**
    Drehe linksgerichtete Kanten;
    Berechne X-Koordinaten;
Bringe gedrehte Kanten in ihre ursprüngliche Richtung;

Algorithmus 3.2 : X-Koordinaten berechnen durch Drehen linksgerichteter Kanten

Algorithmus 3.1 ist für die Berechnung der X-Koordinaten nach dem Drehen einer Menge von Kanten nicht geeignet. Denn auch wenn die Kante e_1 gedreht wurde, wird der Knoten v_8 durch Anwendung des Algorithmus seine bisherige X-Koordinate behalten, da er zuerst die X-Koordinate des Knotens v_9 erhöht um 1 erhält, dann als *besucht* gilt und die eigentlich gewünschte Nachfolgekoordinate von v_2 nicht zugewiesen bekommt.

Die Lösung dieses Problems liegt in der Anwendung eines Algorithmus zur Breitensuche (siehe zum Beispiel Gumm und Sommer (2002, S. 343 ff.)), der alle Knoten v_i traversiert und *jedem* Knoten aus dem Postset des Knotens v_i die X-Koordinate von v_i erhöht um 1 zuweist und danach nicht mehr versetzt. Algorithmus 3.3 leistet genau dies. Seine Anwendung führt zu einer Verteilung der X-Koordinaten nach Abbildung 3.4.

Zur Veranschaulichung eines weiteren auftretenden Problems dient der neue Beispielgraph aus Abbildung 3.5. Der Zyklus wird hier durch die Knotenmenge $V = \{v_4, v_5, v_6, v_9\}$ beschrieben. Die in diesem Zyklus zu drehenden Kanten sind $e_1 = (v_6, v_9)$ und $e_2 = (v_9, v_4)$.

Die unbefriedigende Verteilung der X-Koordinaten nach Anwendung des Algorithmus 3.2 ist in Abbildung 3.6 zu sehen.

Verantwortlich für diese Anordnung der X-Koordinaten ist die Kante $e_L = (v_8, v_6)$, die in Abbildung 3.5 zwei Knoten *überspringt*. Dadurch wird der Knoten v_6 als direkter Nachfolger des Knotens

Eingabe : Knotenmenge ohne X-Koordinaten

Ausgabe : Knotenmenge mit X-Koordinaten, die nach dem Breitensuchverfahren vergeben wurden

Daten : Knoten v, Schlange s, Pointer p

Beginn

- **wenn** v *ist kein Knoten* **dann**
  - Beende Rekursion;
- **sonst**
  - Markiere v als besucht;
  - **für jedes** $v_j \in postset(v)$ **tue**
    - **wenn** v_j *noch nicht besucht* **dann**
      - Vergebe an v_j die X-Koordinate von v + 1;
      - Füge v_j an s an;
  - Wiederhole den Algorithmus mit $s[p+1]$, s und p;

Ende

Algorithmus 3.3 : Breitensuche im gerichteten Graph

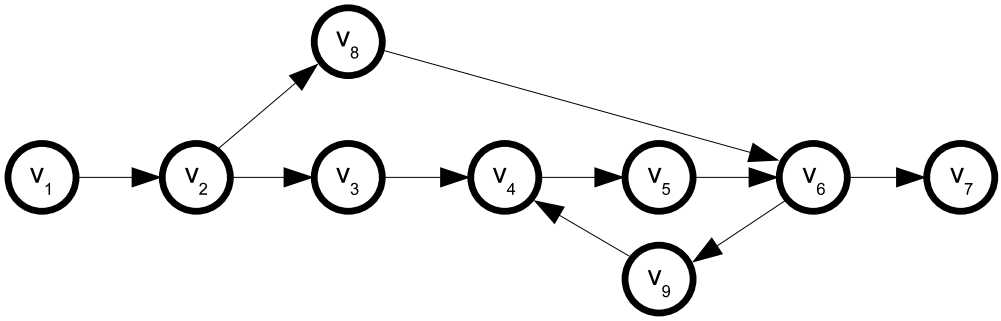

Abbildung 3.5: Neuer Beispielgraph mit Zyklus

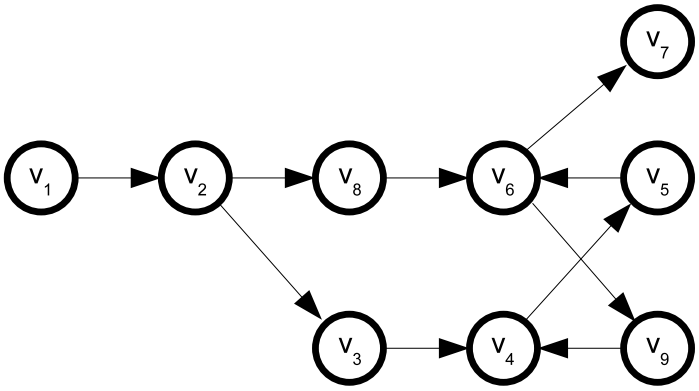

Abbildung 3.6: Schlechte Verteilung der X-Koordinaten durch lange Kante

v_8 auf dessen nachfolgende X-Koordinate gesetzt, obwohl er durch seinen anderen Vorgängerknoten v_5 zwei Schritte weiter nach rechts gehört.

Ein Ansatz zur Lösung dieses Problems ist die Modifizierung des Algorithmus zur Breitensuche 3.3. Setzt man die X-Koordinate der Nachfolgeknoten v_j eines Knotens v, so wird für den Fall, dass ein Knoten v_j bereits besucht wurde, geprüft, ob die aktuell zu setzende X-Koordinate für v_j größer ist als die bereits gesetzte. Der Algorithmus 3.4 verdeutlicht diesen Vergleichsvorgang.

Daten : Knoten v

für alle $v_j \in postset(v)$ **tue**

  wenn *X-Koordinate von v + 1 größer als aktuelle X-Koordinate von v_j* **dann**
    Neue X-Koordinate von v_j ist X-Koordinate von v + 1;

  sonst
    v_j behält X-Koordinate bei;

Algorithmus 3.4 : Größtmögliche X-Koordinate setzen

Für den Fall der langen Kante e_L funktioniert dieses Verfahren. Jedoch funktioniert es nicht, wenn der Graph auch gleichzeitig Zyklen enthält. Die Kanten $e_1 = (v_6, v_9)$ und $e_2 = (v_9, v_4)$ werden als zurückgerichtete Kanten gedreht. Der Knoten v_9 hat damit zwei Vorgänger: v_6 und v_4. Da die X-Koordinate $x(v_6)$ höher ist als $x(v_4)$, folgt $x(v_9) \rightarrow x(v_6) + 1$. Das Ergebnis ist eine Verteilung der X-Koordinaten nach Abbildung 3.7.

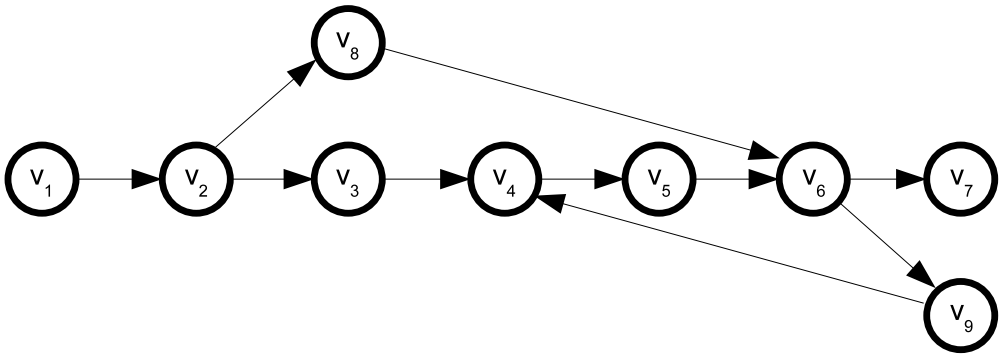

Abbildung 3.7: Schlechte Verteilung der X-Koordinaten durch Zyklus

Allgemein formuliert ist das elementare Problem beim gleichzeitigen Vorhandensein von Zyklen und langen Kanten folgendes: Es existiert kein Unterscheidungskriterium, anhand dessen man beurteilen kann, ob das *erneute* Setzen der X-Koordinate eines Knotens v_j das Resultat einer erkannten Layoutverbesserung ist (weil ein weiter rechts liegender Vorgängerknoten v exisitiert), oder ob der Knoten v_j Teil eines Zyklus ist.

Ein völlig neuer Ansatz bietet die Lösung dieses Problems. Er ist in Abschnitt 3.2.3 beschrieben.

3.2.3 Aufteilung des Graphen in Flüsse

Gegenstand dieses Abschnittes ist die Kalkulation von X-Koordinaten jedes Knotens in zyklischen gerichteten Graphen mit langen Kanten durch die Untergliederung des Graphen in Flüsse.

Die neue Idee ist die Unterteilung des Graphen in *Haupt-* und *Nebenstrukturen*. Bei der Betrachtung von Abbildung 3.5 fällt auf, dass das Layout des Graphen von der Knotenfolge $F = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ und der diese Knoten verbindenden Kantenfolge $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6), (v_6, v_7)\}$ durch die folgenden drei Merkmale dominiert wird:

- Die Gesamtbreite des Graphen entspricht der Anzahl der Knoten dieser Folge,
- innerhalb der Folge existiert keine Kante, deren Länge 1 übersteigt und
- alle Kanten sind nach rechts gerichtet.

Abbildung 3.6 dagegen bietet nicht die Möglichkeit, diese Knotenfolge sofort zu erkennen. Die Kante $e = (v_5, v_6)$ ist Bestandteil von E und nach links gerichtet. Dadurch wird der *Fluss* des Graphen unterbrochen und er wird unübersichtlich. Dies wird durch die bereits erwähnte Kante $e_L = (v_8, v_6)$ verursacht, die $v_6 \in F$ als direkten Nachfolger von $v_8 \notin F$ setzt. Um dies zukünftig zu vermeiden, soll die Position jedes Knotens $v_j \in E$ nur noch durch seinen Vorgängerknoten $v_i \in E$ bestimmt werden. Die Menge $H = E \cup F$ wird ab hier als der *Hauptfluss* des Graphen bezeichnet.

DEFINITION 3.2.1 (HAUPTFLUSS)
Ein Hauptfluss ist die längste zusammenhängende Folge von Knoten innerhalb eines Modulnetzes von der Starttransition bis zur Zieltransition mit den die Knoten verbindenden gleichgerichteten Kanten. ◊

Algorithmus 3.5 identifiziert den längsten gleichgerichteten Pfad eines Netzes zwischen zwei Knoten innerhalb einer vorgegebenen Knotenmenge und speichert diesen in einem globalen Array. Zur Berechnung des Hauptflusses wird er mit der Ziel- und Starttransition aufgerufen. Da der Weg rückwärts vom Ziel- zum Startknoten gesucht wird, muss die gefundene Knotenmenge beim Erreichen des Startknotens umgedreht werden. Damit erhält man die Knotenfolge vom Start- zum Zielknoten.

Für die Lösung eines später auftretenden Problems beinhaltet Algorithmus 3.5 bereits eine Besonderheit: Beim Erreichen des Knotens $goal$ werden alle in der aktuellen Menge befindlichen Knoten zu einer globalen Knotenmenge $sameDirection$ hinzugefügt. Diese beinhaltet nach dem vollständigen Durchlaufen des Algorithmus alle Knoten $v \in vertexSet$, deren inzidente Kanten in dieselbe Richtung wie die des gefundenen längsten Pfades zeigen.

Eingabe : Knoten v, Zielknoten $goal$, Array zur Speicherung der besuchten Knoten A, Knotenmenge innerhalb der der längste Pfad gesucht werden soll $vertexSet$

Ausgabe : Knotenmenge, die den längsten gleichgerichteten Pfad zwischen v und $goal$ darstellt

Daten : Global gespeicherte Knotenmenge A_{glob}

Beginn

- v an A anfügen;
- **wenn** v=$goal$ **dann**
  - Alle Knoten in A_{glob} zu einer globalen Knotenmenge $sameDirection$ hinzufügen;
  - **wenn** *Anzahl der Elemente in A größer als Anzahl der Elemente* A_{glob} **dann**
    - Speichere A in umgekehrter Reihenfolge in A_{glob};
  - **sonst**
    - **wenn** *Anzahl der Elemente in A gleich Anzahl der Elemente in* A_{glob} **dann**
      - **wenn** *Verhältnis von ausgehenden und eingehenden Kanten ist bei A ausgeglichener als bei* A_{glob} **dann**
        - Speichere A in umgekehrter Reihenfolge in A_{glob};
      - **sonst**
        - Behalte A_{glob} bei;
  - Rekursion beenden;
- **sonst**
  - **für jedes** $v_j \in preset(v)$ **tue**
    - **wenn** $v_j \notin A$ **dann**
      - **wenn** $v_j \in vertexSet$ **dann**
        - Wiederhole Algorithmus mit v, v_j, $goal$ und A
  - Lösche v aus A;
  - Beende Rekursion;

Ende

Algorithmus 3.5 : Berechnung des längsten gleichgerichteten Pfades zwischen zwei Knoten

Es kann Fälle geben, in denen ein gefunder Pfad die gleiche Anzahl Knoten besitzt wie die aktuell global gespeicherte Knotenmenge. Dies ist für den Hauptfluss in den Abbildungen 3.8 und 3.9 der Fall. Die vorzuziehende Variante ist hierbei auf Abbildung 3.8 dargestellt. Der Graph in Abbildung 3.9 enthält zum Beispiel die Kante $e = (v_8, v_5)$, die aus dem Unterfluss eines Unterflusses des Hauptflusses auf einen Knoten des Hauptflusses zeigt. Insgesamt verlassen dort 2 Kanten $e_1 = (v_2, v_3)$ und $e_2 = (v_2, v_6)$ den Hauptfluss, während 4 Kanten von außerhalb des Hauptflusses auf Knoten, die Bestandteil des Hauptflusses sind, zeigen. Beim Graph aus Abbildung 3.8 dagegen ist das Verhältnis aus den Hauptfluss verlassenden und den Hauptfluss erreichenden Kanten ausgeglichen. Ebendieses Verhältnis wird in diesem Fall als Entscheidungskriterium des zu wählenden Haupt- (und später auch Ober-)flusses herangezogen. Ist dieses Verhältnis ausgeglichen, so wird der Pfad gewählt, der insgesamt die wenigsten adjazenten Knoten aufweist. Dadurch wird gewährleistet, dass der Fluss eine minimale Anzahl direkter Unterflüsse aufweist, was die Wahrscheinlichkeit erhöht, dass er später ausbalanciert werden kann (siehe Abschnitt 3.4.3).

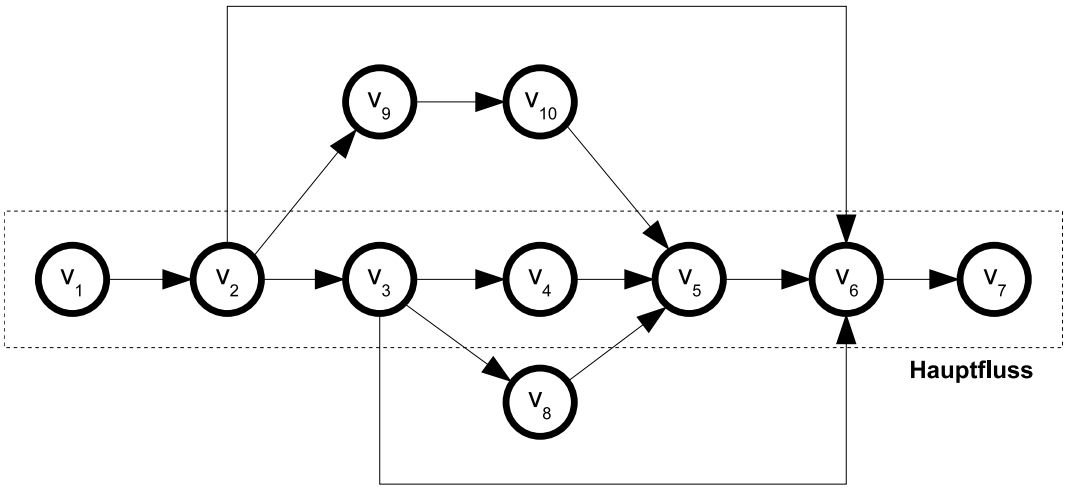

Abbildung 3.8: Graph mit zwei Hauptflussmöglichkeiten (1)

Ist der Hauptfluss des Netzes identifiziert, sind die X-Koordinaten seiner Knoten leicht festzulegen. Die Starttransition erhält die X-Koordinate 1 und die darauf folgenden Knoten erhalten die X-Koordinate des jeweiligen Vorgängerknotens erhöht um 1. Die Zieltransition erhält schließlich die höchste X-Koordinate.

Der neue Programmablauf zur Berechnung der X-Koordinaten ist nun Algorithmus 3.6 zu entnehmen.

Damit wird der Algorithmus 3.2 unverändert ausgeführt. Dadurch befindet sich der Knoten v_9, der Bestandteil des Zyklus ist, an der gewünschten Position. Für alle Knoten aus dem Hauptfluss werden die X-Koordinaten jetzt erneut gesetzt. Das Resultat ist die gewünschte Verteilung der X-Koordinaten nach Abbildung 3.5.

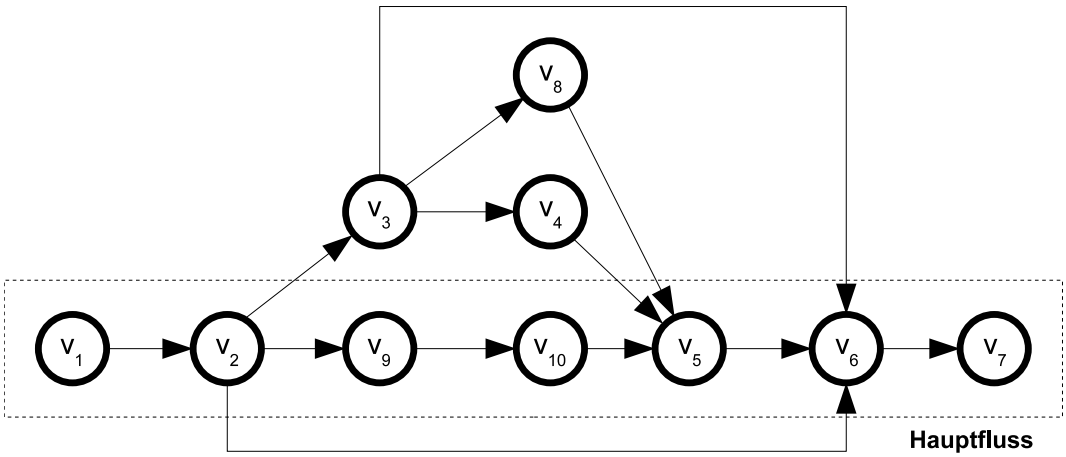

Abbildung 3.9: Graph mit zwei Hauptflussmöglichkeiten (2)

Beginn

solange *linksgerichtete Kante vorhanden* **tue**

Drehe linksgerichtete Kanten;

Berechne X-Koordinaten;

Bringe gedrehte Kanten in ihre ursprüngliche Richtung;

Führe Algorithmus 3.5 zum Setzen der Koordinaten des Hauptflusses aus;

Ende

Algorithmus 3.6 : Layout mit Hauptfluss

Aus der Menge von Knoten und Kanten, die nicht Bestandteil des Hauptflusses sind, lassen sich *Nebenflüsse* bilden.

DEFINITION 3.2.2 (NEBENFLUSS)
Ein Nebenfluss ist jede längstmögliche zusammenhängende Folge von Knoten und gleichgerichteten Kanten innerhalb eines Modulnetzes, wobei kein Knoten und keine Kante Bestandteil des Hauptflusses oder eines anderen Nebenflusses ist. ◊

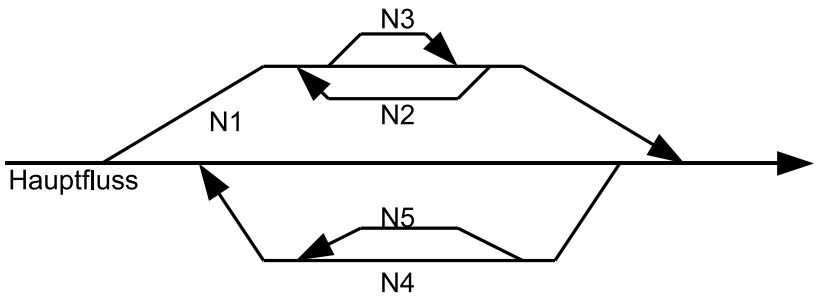

Abbildung 3.10: Graph mit vielen Nebenflüssen

In dem Graph aus Abbildung 3.5 existieren zwei Nebenflüsse N_1 und N_2, wobei $N_1 = \{(v_2, v_8), v_8, (v_8, v_6)\}$ und $N_2 = \{(v_6, v_9), v_9, (v_9, v_4)\}$. Die beiden Nebenflüsse sind dabei völlig unabhängig voneinander, so dass beide am Hauptfluss ausgerichtet werden.

Bei komplizierteren Graphen wie dem aus Abbildung 3.10 existieren jedoch auch Nebenflüsse, die nicht direkt dem Hauptfluss entstammen. N_5 ist Nebenfluss des Nebenflusses N_4; N_2 und N_3 sind Nebenflüsse des Nebenflusses N_1. Da nur der Hauptfluss als *fixe* Knotenfolge festgelegt wurde, existiert das Problem eventuell vorkommender langer Kanten aber immer noch in den Nebenflüssen. Besteht zum Beispiel N_2 nur aus einer langen Kante, so würde für N_1 das oben erwähnte Problem mehrerer Knoten überspringender Kanten auftreten.

Zur Lösung dieses Problems sind einige Begrifssdefinitionen notwendig:

DEFINITION 3.2.3 (UNTERFLUSS)
Ein Unterfluss ist ein Nebenfluss, dessen Anfangs- und Endknoten adjazent zu Knoten eines längeren Nebenflusses oder des Hauptflusses sind. ◊

DEFINITION 3.2.4 (OBERFLUSS)
Ein Oberfluss ist ein Haupt- oder Nebenfluss, zu dem Anfangs- und Endknoten eines kürzeren Nebenflusses adjazent sind. Jeder Unterfluss hat genau einen Oberfluss. ◊

In Abbildung 3.10 ist zum Beispiel N_1 der Oberfluss von N_2 und N_3, wogegen N_5 ein Unterfluss von N_4 ist. Der Hauptfluss ist Oberfluss von N_1 und N_4.

Algorithmus 3.5 findet nicht nur den längsten Fluss vom Start- zum Zielknoten, sondern auch jede andere längste gleichgerichtete Knoten- und Kantenfolge zwischen zwei Knoten. Nach der Identifikation aller zu einem Oberfluss adjazenten Knoten, kann man also mit seiner Hilfe den jeweiligen Unterfluss berechnen. Begonnen wird dabei mit der Berechnung des Hauptflusses, daraufhin werden seine Unterflüsse und deren Unterflüsse berechnet. Diese Vorgehensweise ist in Algorithmus 3.7 implementiert.

Elementare Variablen des Algorithmus 3.7 sind die Menge V, die alle Knoten des Graphen enthält und kontinuierlich um die Elemente jedes gefundenen Flusses reduziert wird. Das zweidimensionale Array $Flows$ enthält alle gefundenen Flüsse, wobei der Hauptfluss an der Stelle [0] steht. Das ebenfalls zweidimensionale Array $sameDir$ beinhaltet die von einem Oberfluss erreichbaren und mit ihm in gleicher Richtung liegenden Knoten. $Connected$ ist ebenfalls ein zweidimensionales Array und beinhaltet alle Knoten, die Bestandteil der Unterflüsse (und deren Unterflüsse) des Oberflusses sind. Der Algorithmus stellt sicher, dass der Flussindex i jedes Flusses in $Flows$ immer auch der Index der zu diesem Fluss gehörenden Knotenmengen in $sameDir$ und $Connected$ ist. Außerdem bekommt jeder Knoten einen *Oberflussindex* zugewiesen, der anzeigt, an welcher Stelle in $Flows$ der zugehörige Oberfluss zu finden ist.

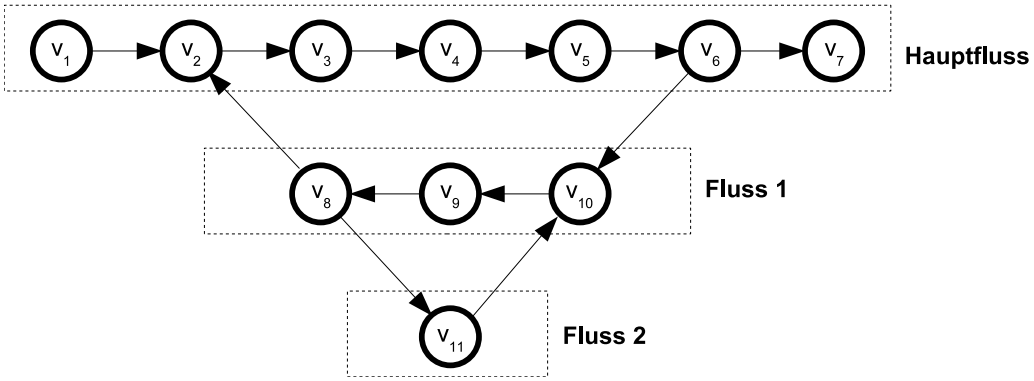

Abbildung 3.11: Aus drei Flüssen bestehender Graph

Wendet man Algorithmus 3.7 auf den Beispielgraph aus Abbildung 3.11 an, so ist die Belegung $Flows$ Tabelle 3.1 zu entnehmen, Tabelle 3.2 enthält die Belegung von $sameDir$ und $Connected$ ist in Tabelle 3.3 dargestellt.

Abbildung 3.12 zeigt einen Graphen, der zwei Knoten v_{13} und v_{14} enthält, die zu Knoten aus jeweils zwei Flüssen H und U adjazent sind. Werden nun die Unterflüsse des Hauptflusses H berechnet, so wird U als erster Unterfluss identifiziert. *In* und *Out* beinhalten jetzt noch die Knoten v_{14} und v_{13}, *maxPath* ist jedoch $\emptyset$, da kein Pfad von v_{14} nach v_{13} gefunden werden kann. Deshalb werden alle Knoten $v \in Out$ als eigenständige Flüsse an *Flows* angefügt mit dem aktuellen i als Oberflussindex. In diesem Fall gilt dies für Knoten v_{13}. Knoten v_{14} wird dann als Unterfluss von U identifiziert.

Eingabe : Ungeordnete Knotenmenge V

Ausgabe : Flussweise geordnete Knotenmenge $Flows$

Beginn

Berechne mit Algorithmus 3.5 Hauptfluss H für die Parameter $Zielknoten$, $Startknoten$, $\emptyset$, V;

Füge H in $Flows[0]$ ein;

Füge die Differenzmenge $V - H$, die alle mit dem Hauptfluss verbundenen Knoten darstellt, in $Connected[0]$ ein;

Füge die globale Knotenmenge $sameDirection$ aus Algorithmus 3.5 in $sameDir[0]$ ein;

$V \rightarrow V - H$;

$i = 0$;

solange *i ist kleiner als die Anzahl der Indizes von $Flows$* **tue**

  für jedes *$v \in Connected[i]$ und $v \notin sameDir[i]$* **tue**

    Setze Knotenrichtung von v entgegen der Knotenrichtung von $Flows[i]$;

  wenn *Anzahl der Elemente von $Flows[i]$ größer als 1* **dann**

    für jedes *v_j, das adjazent zu einem Knoten $v \in Flows[i]ist$* **tue**

      wenn *v_j ist Nachfolger von v* **dann**

        Füge v_j an Out an;

      sonst

        Füge v_j an In an;

  solange *$\exists v_o \in Out$ oder $\exists v_i \in In$* **tue**

    Berechne längste mögliche Knotenfolge von v_o nach v_i in V mit Algorithmus 3.5 und speichere sie in $maxPath$;

    wenn $maxPath = \emptyset$ **dann**

      Speichere jedes $o \in Out$ als eigenständigen Fluss;

    sonst

      $Out \rightarrow Out - v_o$;

      $In \rightarrow In - v_i$;

      Führe Algorithmus 3.5 mit den Parametern $end(maxPath)$, $maxPath[0]$, $\emptyset$ und V aus, um globale Knotenmenge $sameDirection$ zu erzeugen;

      Füge die Menge aller mit $maxPath$ verbundenen Knoten an $Connected$ an;

      Füge die globale Knotenmenge $sameDirection$ an $sameDir$ an;

      für jedes $v \in maxPath$ **tue**

        $V \rightarrow V - v$;

        Setze i als Oberflussindex von v;

      Füge $maxPath$ an $Flows$ an;

  $i \rightarrow i + 1$;

Kehre alle rückwärtigen Kantenfolgen in $Flows$ um;

Ende

Algorithmus 3.7 : Aufteilung der Knotenmenge in Flüsse

Index	0	1	2
Fluss	v_1	v_8	v_{11}
	v_2	v_9	
	v_3	v_{10}	
	v_4		
	v_5		
	v_6		
	v_7		

Tabelle 3.1: Belegung des zweidimensionalen Array $Flows$

Index	0	1	2
Knoten in selber Richtung	$\emptyset$	$\emptyset$	$\emptyset$

Tabelle 3.2: Belegung des zweidimensionalen Array $sameDir$

Index	0	1	2
verbundene Knoten	v_8	v_{11}	$\emptyset$
	v_9		
	v_{10}		
	v_{11}		

Tabelle 3.3: Belegung des zweidimensionalen Array $Connected$

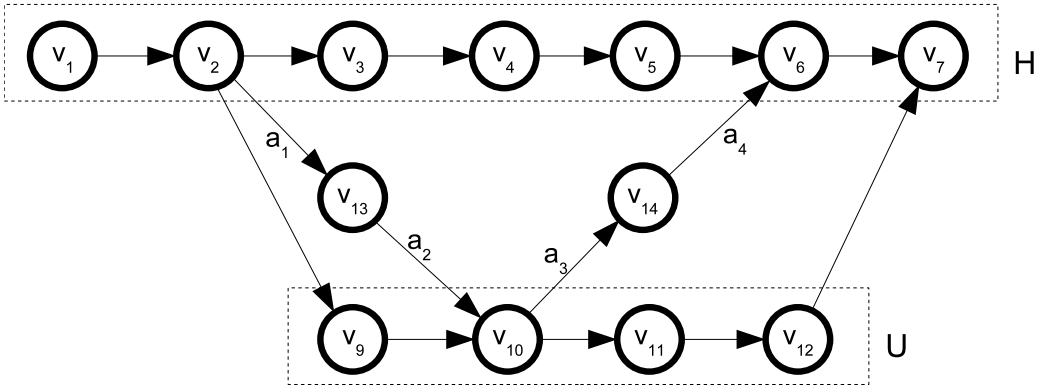

Abbildung 3.12: Graph mit Knoten, die zu zwei verschiedenen Flüssen adjazent sind

Jeder Knoten besitzt ein Attribut, das dessen *Knotenrichtung* anzeigt. Diese entspricht der Richtung der zu diesem Knoten inzidenten Kanten, die Bestandteil des Flusses sind. Im Rahmen von Algorithmus 3.7 wird die Knotenrichtung einiger Knoten manipuliert, damit zum Setzen der X-Koordinaten die zu den entsprechenden *rückwärtigen* Knoten inzidenten Kanten gedreht werden können. Das ist nötig, damit der Graph ausschließlich rechtsgerichtete Kanten und Flüsse enthält, denn nur so können die X-Koordinaten gesetzt werden.

Am Beispiel des Graphen aus Abbildung 3.11 funktioniert die Manipulation des Richtungsattributs folgendermaßen: Initial sind alle Knoten rechtsgerichtet. Ist der Hauptfluss identifiziert, so entspricht die mit ihm verbundene Knotenmenge $C_H = \{v_8, v_9, v_{10}, v_{11}\}$. Die dem Hauptfluss gleichgerichtete Knotenmenge ist $M_H = \emptyset$. Die Menge der entgegen die Hauptrichtung zu drehenden Knoten ist $T_H = C_H - M_H$. Somit sind alle Knoten $v \in T_H = \{v_8, v_9, v_{10}, v_{11}\}$ linksgerichtet. Dann wird $Fluss_1$ als Unterfluss des Hauptflusses identifiziert. Die mit ihm verbundene Knotenmenge ist $C_{Fluss_1} = \{v_{11}\}$. Die gleichgerichtete Knotenmenge ist $M_{Fluss_1} = \emptyset$. Die entgegen der Knotenrichtung von $Fluss_1$ zu drehende Knotenmenge ist also $T_{Fluss_1} = C_{Fluss_1} - M_{Fluss_1} = \{v_{11}\}$. Damit ist v_{11} wieder wie ursprünglich nach rechts gerichtet.

Nun können den einzelnen Knoten ihre X-Koordinaten zugewiesen werden. Algorithmus 3.8 veranschaulicht diesen Vorgang. Da allen Flüssen ihre Koordinaten vom am weitesten links gelegenen bis zum weitest rechts gelegenen Knoten zugewiesen werden, müssen die zu den als rückwärtig gekennzeichneten Knoten inzidenten Kanten dann vorläufig gedreht werden, wenn die Kante zwei rückwärtige Knoten verbindet oder wenn sie einen Knoten eines Oberflusses mit einem rückwärtigen Knoten eines Unterflusses verbindet.

Das Setzen der X-Koordinaten erfolgt jetzt flussweise, angefangen mit dem Hauptfluss. Dessen Anfangsknoten, der Startknoten des Netzes, bekommt die X-Koordinate 1 zugewiesen. Die darauf folgenden Knoten des Hauptflusses erhalten die X-Koordinate ihres Vorgängers erhöht um 1. Die X-Koordinaten der Anfangsknoten aller weiteren Flüsse errechnen sich aus der X-Koordinate ihres Vorgängerknotens, der Bestandteil des Oberflusses ist, erhöht um 1. Mit den restlichen Knoten des Flusses wird analog zum Hauptfluss verfahren.

3.2.4 Exkurs: Eliminierung von Zyklen durch *Greedy Cycle Removal*

Es existieren Algorithmen, die dazu dienen, Zyklen aus Graphen durch Umkehren einer möglichst minimalen Menge von Kanten zu entfernen. Dieses Problem ist bekannt als *Feedback Arc Set Problem* (Battista u. a., 1999, S. 295) und wird als NP-complete charakterisiert (Garey und Johnson, 1979). In (Battista u. a., 1999, S. 296ff.) ist ein Algorithmus namens Greedy Cycle Removal beschrieben, der ein kleines Feedback Arc Set identifiziert. Dazu wird zunächst eine provisorische Vergabe von X-Koordinaten durchgeführt. Die daraus direkt ersichtlichen rückwärtigen Kanten bil-

Eingabe : Flussweise geordnete Knotenmenge V

Ausgabe : Knotenmenge V mit zugewiesenen X-Koordinaten

Daten : Kantenmenge A

Beginn

- **für alle** $a \in A$ **tue**
  - v_a ist Anfangsknoten von a;
  - v_e ist Endknoten von a;
  - **wenn** v_a *ist back und* v_e *ist back* **dann**
    - Drehe a;
  - **sonst**
    - **wenn** v_a *ist back* **dann**
      - **wenn** v_a *in Unterfluss von* v_e **dann**
        - Drehe a;
    - **sonst**
      - **wenn** v_e *ist back* **dann**
        - **wenn** v_e *in Unterfluss von* v_a **dann**
          - Drehe a;
- **für alle** *Flüsse* F **tue**
  - **wenn** F *ist Hauptfluss* **dann**
    - Startknoten von F erhält X-Koordinate 1;
    - Alle Folgeknoten erhalten X-Koordinate des Vorgängerknotens erhöht um 1;
  - **sonst**
    - Startknoten des Flusses erhält X-Koordinate des adjazenten Knotens aus dem Oberfluss erhöht um 1;
    - Alle Folgeknoten erhalten X-Koordinate des Vorgängerknotens erhöht um 1;
- Bringe gedrehte Kanten in ihre ursprüngliche Richtung;

Ende

Algorithmus 3.8 : Flussweises setzen der X-Koordinaten

den das Feedback Arc Set. Algorithmus 3.9 verdeutlicht die Vorgehensweise.

Eingabe : Graph mit eventuell vorkomenden Zyklen

Ausgabe : Azyklischer Graph

Daten : Knotenmenge V, Kantenmenge A

Beginn

  solange *Senke s vorhanden* **tue**
    Füge Senke an den Anfang von S_r ein;
    $V \rightarrow V - s$;

  solange *Quelle q vorhanden* **tue**
    Füge Quelle ans Ende von S_l ein;
    $V \rightarrow V - q$;

  solange $V \neq \emptyset$ **tue**
    Suche das $v \in V$, für das $outdegree(v) - indegree(v)$ maximal ist;
    $V \rightarrow V - v$;

  Füge S_r an S_l an und bilde so S;

  für jedes $a \in A$ **tue**
    wenn *Index des Anfangsknotens von a innerhalb von S ist größer als der Index des Endknotens von a innerhalb von S* **dann**
      Drehe Kante a;

Ende

Algorithmus 3.9 : Greedy Cycle Removal

Durch Suchen des Knotens mit der maximalen Differenz aus Außengrad und Innengrad wird erreicht, dass die Knoten mit den meisten ausgehenden und/oder wenigsten eingehenden Kanten weiter vorne im Array angeordnet sind und die mit den meisten eingehenden und /oder wenigsten ausgehenden Kanten weiter hinten angeordent sind. Da wir später die Kanten zum Feedback Arc Set zählen, die von einem Knoten weiter hinten im Array zu einem Knoten weiter vorne im Array führen, ist diese lokale Konzentration von Knoten mit wenigen ausgehenden Kanten weiter hinten im Array sinnvoll.

Anhand des Beispielgraphen in Abbildung 3.13 wird die Funktionsweise des Algorithmus verdeutlicht.

Abbildung 3.13 zeigt einen Graph mit 12 Knoten, 15 Kanten und insgesamt 5 Zyklen. Die Differenz aus Außengrad und Innengrad ist an jedem Knoten vermerkt. Ausnahmen sind Quelle und Senke, für die diese Kennzahl irrelevant ist.

Knoten	1	6	11	8	9	10	12	3	4	5	2	7
outdegree-indegree	–	2	2	1	0	0	0	−1	−1	−1	−2	–

Tabelle 3.4: Tabelle mit provisorischer Knotenordnung

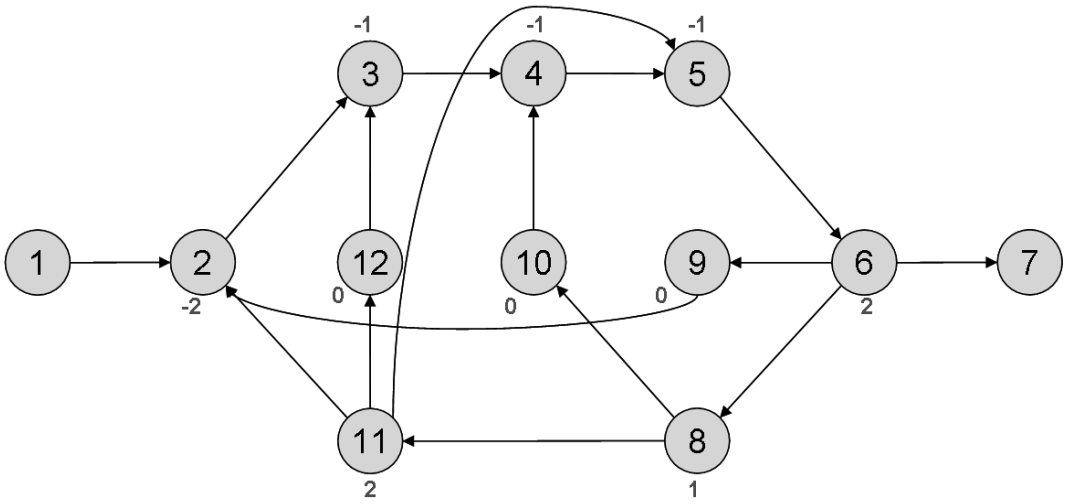

Abbildung 3.13: Graph mit 5 Zyklen

Eine aus diesem Graphen durch den Algorithmus berechenbare Knotenordnung ist in Tabelle 3.4 dargestellt. Die Quelle *1* steht am Anfang der Reihe, die Senke *7* steht am Ende. Dazwischen sind alle Knoten entsprechend ihrer Differenz aus Außengrad und Innengrad geordnet platziert. Ist diese Differenz für mehrere Knoten identisch, so ist deren Anordnung untereinander davon abhängig, in welcher Reihenfolge sie in V gespeichert sind.

Nun werden sämtliche Kanten aus dem Graph auch in der Tabelle zwischen den jeweiligen Knoten gesetzt. Die nach links gerichteten Kanten, die letztendlich unser Feedback Arc Set bilden, sind die Kanten $(2, 3)$, $(8, 11)$ und $(5, 6)$.

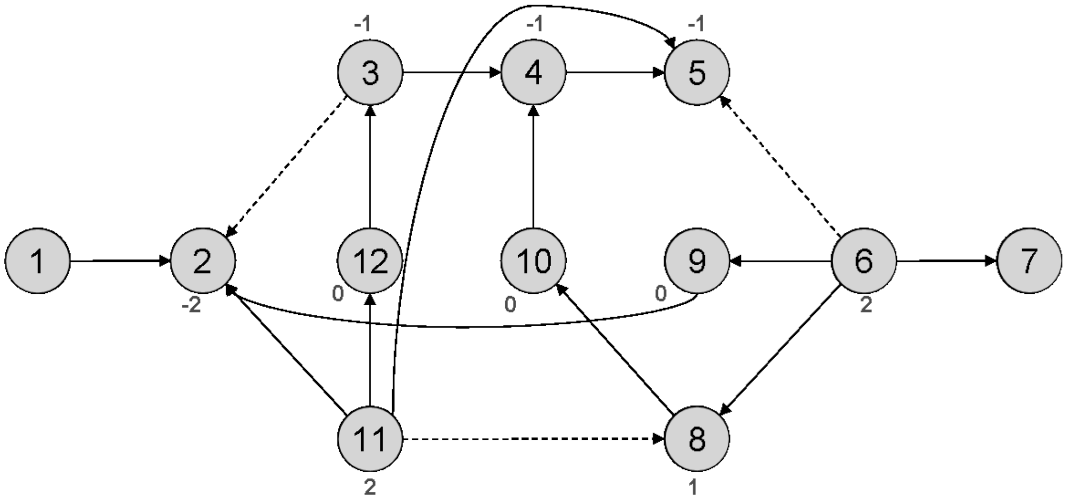

Abbildung 3.14: Graph mit gedrehten Kanten

Der Graph mit den gedrehten Kanten aus dem Feedback Arc Set ist in Abbildung 3.14 skizziert. Die betroffenen Kanten sind hier gestrichelt dargestellt. Sämtliche Zyklen sind eliminiert, jedoch

ist das Feedback Arc Set, wenn auch mit 3 von 15 Kanten relativ klein, keineswegs minimal. Die Kanten $(11, 8)$ und $(2, 3)$ hätten nicht gedreht werden müssen, um den Graph zyklenfrei zu machen.

Abschließend zu diesem Abschnitt sei erwähnt, dass nach Battista u. a. (1999, S. 300) ein Näherungsalgorithmus, der garantiert, dass das gefundene Feedback Arc Set in seiner Größe nah am minimalen Feedback Arc Set ist, wahrscheinlich nicht existieren kann.

3.3 Setzen der Dummy-Knoten

Wenn zwei zueinander adjazente Knoten nicht auf benachbarten X-Koordinaten liegen, dann ist die zu den Knoten inzidente Kante eine *lange Kante*. Eine solche lange Kante ist zum Beispiel die Kante $e = (v_2, v_6)$ in Abbildung 3.15. Sie reicht von X-Koordinate 2 bis 6 und hat damit eine Länge von 4.

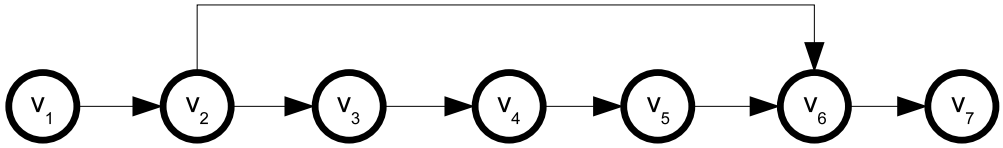

Abbildung 3.15: Graph mit langer Kante

Um später bei der Vergabe der Y-Koordinaten zu vermeiden, dass eine solche lange Kante auf dem Fluss liegt, dem sie entstammt, werden *Dummy-Knoten* erzeugt. Diese sorgen dafür, dass die lange Kante in kurze Kanten der Länge 1 aufgeteilt wird. Dummy-Knoten sind im Graphen nicht sichtbar und werden hier nur zur Veranschaulichung als kleinere Knoten gezeichnet.

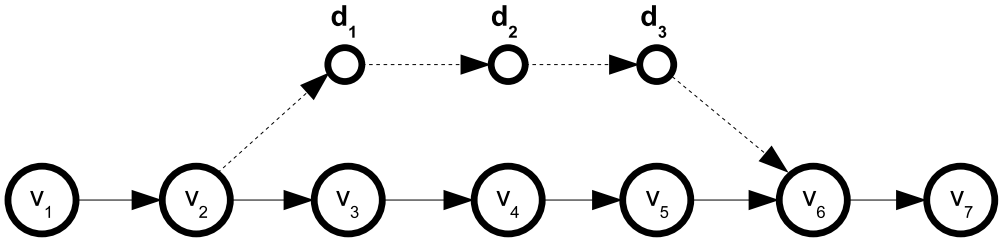

Abbildung 3.16: Lange Kante mit Dummy-Knoten

Abbildung 3.16 zeigt die Aufteilung der langen Kante in 4 kurze Kanten und die drei eingefügten Dummy-Knoten d_1, d_2 und d_3. Algorithmus 3.10 zeigt, wie die Knoten- und Kantenmenge entsprechend erweitert wird. Zuerst wird jeder Kante ein Richtungsattribut vergeben, das anzeigt, ob die Kante nach links oder nach rechts zeigt. Danach wird jede Kante geprüft, ob sie länger als 1 ist. Ist dies der Fall, so wird ein Dummy-Knoten d erzeugt und mit X-Koordinate und Oberflussindex

attributiert. Die lange Kante wird in zwei Kanten aufgeteilt. Diese werden mit d und seinen adjazenten Knoten, dem Vorgänger v_a und dem Nachfolger v_e, verbunden. Die Kante $e_1 = (v_a, d)$ hat hierbei auf jeden Fall die Länge 1. Die Kante $e_2 = (d, v_e)$ kann jedoch immer noch eine lange Kante sein. Am Beispiel von Abbildung 3.16 ist nach dem Setzen des ersten Dummy-Knotens d_1 die Kante $e_{kurz} = (v_2, d_1)$ eine kurze Kante, die Kante $e_{lang} = (d_1, v_6)$ ist aber eine neue lange Kante. Daher muss so lange über die Menge aller Kanten iteriert werden, bis keine lange Kante mehr existiert.

Eingabe : Knotenmenge V, Kantenmenge A mit eventuell vorkommenden langen Kanten
Ausgabe : um Dummy-Knoten erweiterte Knotenmenge V_+, Kantenmenge A_+ mit ausschließlich Kanten der Länge $length(a) = 1$

Beginn

für jedes $a \in A$ **tue**
Vergebe Richtungsattribut $dir(a)$;

solange $\exists a \in A : length(a) > 1$ **tue**
v_a ist Anfangsknoten von a;
v_e ist Endknoten von e;
Erstelle Dummy-Knoten d;
wenn $dir(a)$ *ist "rechts"* **dann**
Weise d die X-Koordinate von v_a erhöht um 1 zu;
sonst
Weise d die X-Koordinate von v_a vermindert um 1 zu;
wenn v_a *ist kein Dummy-Knoten* **dann**
Oberflussindex von d ist Flussindex von v_a;
sonst
Oberflussindex von d ist Oberflussindex von v_a;
Neuer Endknoten von a ist d;
Erstelle neue Kante a_{neu};
Anfangsknoten von a_{neu} ist d;
Endknoten von a_{neu} ist v_e;

Ende

Algorithmus 3.10 : Dummy-Knoten setzen

Um die spätere Verteilung der Y-Koordinaten zu vereinfachen, werden die Dummy-Knoten in die Flüsse der *echten* Knoten integriert. Dafür müssen drei mögliche Fälle berücksichtigt werden.

- Dummy-Knoten sind die Verlängerung eines linksgerichteten Flusses F_{links} (Abbildung 3.17) und werden in $Flows$ an F_{links} angehangen.
- Dummy-Knoten sind die Verlängerung eines rechtsgerichteten Flusses F_{rechts} (Abbildung 3.18) und werden in $Flows$ an F_{rechts} angehangen.
- Eine Folge von Dummy-Knoten kann einen eigenständigen Unterfluss repräsentieren (Abbildung 3.19) und wird als solcher an der nächsten freien Stelle von $Flows$ angefügt.

Diese Unterscheidung ist wichtig, um den korrekten Fluss innerhalb von $Flows$ zu identifizieren, an den die zusammengehörigen Dummy-Knoten angefügt werden, falls es sich bei diesen nicht um einen eigenständigen Unterfluss handelt. Algorithmus 3.11 zeigt den Vorgang schematisch. Dort wird außerdem bei der Fallunterscheidung mitgeprüft, ob der Knoten, an den die Dummy-Knoten *angehangen* werden, der letzte Knoten innerhalb seines Flusses ist. Ist dies nicht der Fall, so ist die Dummy-Knotenfolge keine Verlängerung des Flusses, sondern ein eigenständiger Fluss. Ohne diese Maßnahme kommt es später bei der Verteilung der Y-Koordinaten zu einer Überschneidung der langen Kante mit einem oder mehreren Knoten, wie es in Abbildung 3.20 demonstriert wird. Kante a ist hier fälschlicherweise in den Fluss integriert worden und überlagert einen Knoten.

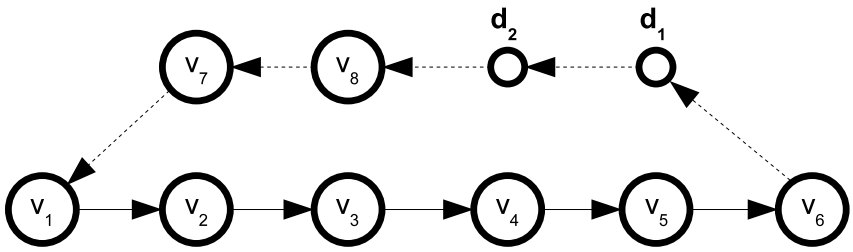

Abbildung 3.17: Nach links gerichteter Fluss aus Dummy-Knoten und echten Knoten

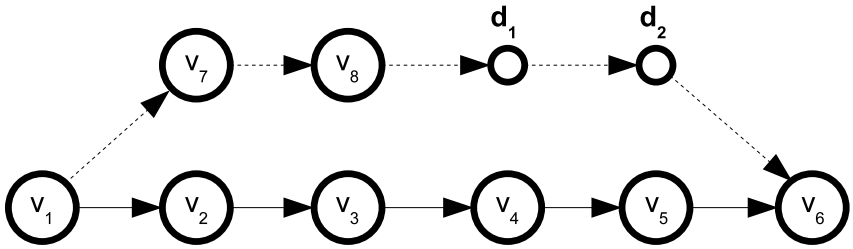

Abbildung 3.18: Nach rechts gerichteter Fluss aus Dummy-Knoten und echten Knoten

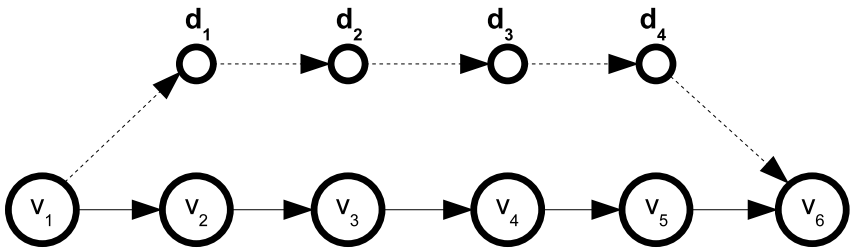

Abbildung 3.19: Aus vier Dummy-Knoten bestehender Fluss

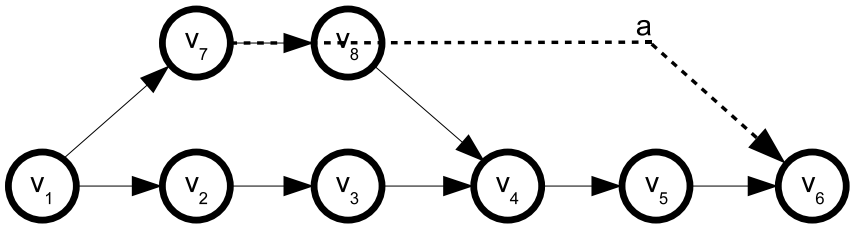

Abbildung 3.20: Schnitt von Kante a durch einen Knoten

Eingabe : Gesamtknotenmenge G mit flussweise geordneter echter Knotenmenge $V \subset G$ und ungeordneter Dummy-Knotenmenge $D \subset G$

Ausgabe : Flussweise geordnete Gesamtknotenmenge G

Beginn

solange *keinem Fluss zugeordneter Dummy-Knoten vorhanden* **tue**

Suche Dummy-Knoten d_a, der echten Vorgängerknoten hat;

Füge d_a in $Fluss[0]$ ein;

solange *Nachfolgender Dummy-Knoten d_{next} vom letzten Knoten in $Fluss$ $end(Fluss)$ vorhanden* **tue**

Füge d_{next} an $Fluss$ an;

$d_e = end(Fluss)$;

wenn *Flussindex f_v des Vorgängerknotens von d_a ist größer als Flussindex f_n des Nachfolgeknotens V_N von d_e UND V_N ist letzter Knoten innerhalb seines Flusses* **dann**

Füge $Fluss$ an $Flows[f_v]$ an;

sonst

wenn *Flussindex f_v des Vorgängerknotens von d_a ist kleiner als Flussindex f_n des Nachfolgeknotens V_N von d_e UND V_N ist letzter Knoten innerhalb seines Flusses* **dann**

Füge $Fluss$ an $Flows[f_n]$ an;

sonst

Füge $Fluss$ an $Flows$ an;

Ende

Algorithmus 3.11 : Dummy-Knoten in Flüsse integrieren

3.4 Berechnung der Y-Koordinaten

Die Y-Koordinaten bestimmen die vertikale Ausrichtung des Graphen. Sie werden flussweise nach den folgenden Kriterien vergeben:

- Knoten dürfen sich nicht *überlagern*.
- Der Mindestabstand zwischen zwei Knoten beträgt 1.
- Unterflüsse sollen möglichst nah bei ihren Oberflüssen liegen.
- Knoten, die zu demselben Fluss gehören, erhalten dieselbe Y-Koordinate (Ausnahme: lange Rückflüsse (siehe Abschnitt 3.4.2) und ausbalancierte Flüsse (siehe Abschnitt 3.4.3)).
- Der Hauptfluss liegt zentral.
- Die Anzahl der Kantenüberschneidungen ist auf ein Minimum reduziert.

In diesem Kapitel wird ein Algorithmus vorgestellt, der zunächst eine initiale flussweise Verteilung der Y-Koordinaten generiert, danach den Sonderfall langer Rückflüsse behandelt und schließlich das Layout durch Ausbalancieren einiger Knoten verbessert.

3.4.1 Initiale Verteilung der Y-Koordinaten

Der rekursive Algorithmus 3.12 berechnet eine anfängliche Verteilung von Y-Koordinaten mit integrierter Reduzierung der Kantenüberschneidungen. Seine Vorgehensweise ist die folgende:

Zuerst wird den Knoten des Hauptflusses die Y-Koordinate $y_{Haupt} = 1$ zugewiesen. Dann werden seinen Unterflüssen abwechselnd (durch die Verwendung einer Variable *Stelle* gesteuert) die Y-Koordinate $y_{Haupt} + 1$ und $y_{Haupt} - 1$ zugewiesen. Da $y_{Haupt} - 1 = 0$ ist, die niedrigst mögliche Y-Koordinate aber 1 ist, werden die Y-Koordinaten sämtlicher bereits gesetzter Knoten um 1 erhöht. Das Resultat ist, dass der Hauptfluss, wenn er mehr als einen Unterfluss hat, *zwischen* diesen Unterflüssen liegt. Dies ist einerseits aus symmetrischen Gründen vorteilhaft und zum anderen werden so von vorne herein die Kantenüberschneidungen minimal gehalten.

Dasselbe Verfahren wird auch für die Unterflüsse des Hauptflusses mit deren Unterflüssen angewendet. Besitzt ein Oberfluss mehrere Unterflüsse, die selbst keine Unterflüsse mehr besitzen, so wird eine Heuristik angewendet, die die Y-Koordinaten dieser Unterflüsse und des Oberflusses untereinander vertauscht und so versucht Kantenüberschneidungen zu minimieren.

Dieses Verfahren zur Reduzierung der Kantenüberschneidungen ist im Vergleich zu Algorithmen wie dem *Layer-by-Layer Sweep* (Battista u. a., 1999, S. 280ff.) deutlich weniger laufzeitintensiv. Die

Knoten werden nicht einzeln miteinander vertauscht, sondern direkt flussweise. Dadurch, dass Unterflüsse aufgrund einer geringeren Knotenanzahl eine geringere horizontale Ausdehnung aufweisen und möglichst nah bei ihren Oberflüssen platziert werden, ist die Ausgangssituation oft schon so gut, dass auch durch Vertauschen keine Reduzierung der Kantenüberschneidungen mehr erreicht werden kann.

Die Funktionsweise von Algorithmus 3.12 wird an dem Graph aus Abbildung 3.21 demonstriert. Dort ist der Graph mit der bereits erfolgten Verteilung der Y-Koordinaten zu sehen, was das Nachvollziehen der einzelnen Schritte vereinfacht.

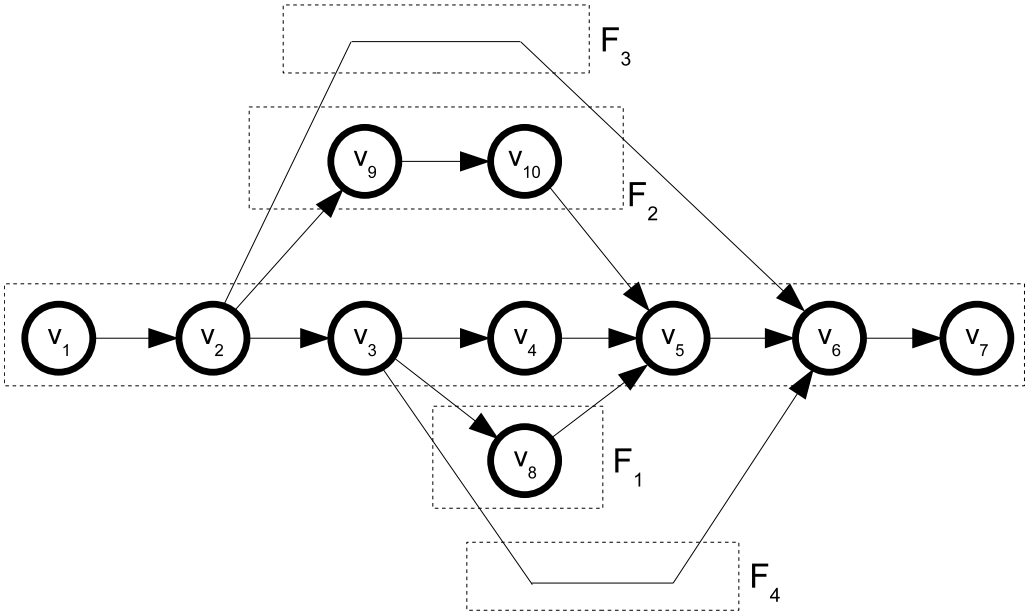

Abbildung 3.21: Graph mit zugewiesenen Y-Koordinaten

Am Anfang wird Algorithmus 3.12 mit dem Hauptfluss als F, $y = 1$ und $s = 0$ aufgerufen. Alle Knoten des Hauptflusses bekommen die Y-Koordinate 1 zugewiesen (Zeile 5). Den aktuellen Zustand zeigt Abbildung 3.22.

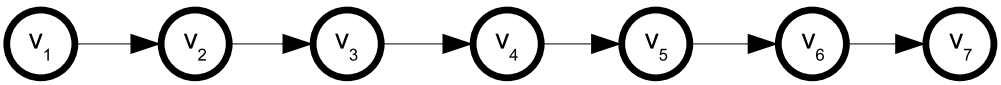

Abbildung 3.22: Dem Hauptfluss H Y-Koordinaten zuweisen

Nun werden alle Unterflüsse von H, die keine Unterflüsse besitzen, U zugewiesen (Zeile 16). Tabelle 3.5 zeigt die Belegung von U.

Der Algorithmus wird wiederholt mit $F = F_1$, $y \rightarrow y_{Oberfluss} + 1 = 2$ und $s = 1$ (Zeile 17). Nachdem F_1 die Y-Koordinate 2 zugewiesen bekommen hat, hat der Graph die Form aus Abbildung 3.23.

Eingabe : Flussweise geordnete Knotenmenge V mit zugewiesenen X-Koordinaten

Ausgabe : Knotenmenge V mit zugewiesenen Y-Koordinaten

Daten : Fluss F, Y-Koordinate y, Stelle s

```
1  Beginn
2      wenn y = 0 dann
3          Erhöhe Y-Koordinate jedes Knotens um 1;
4          Weise allen v ∈ F die Y-Koordinate 1 zu;
5      Weise allen Knoten v ∈ F die Y-Koordinate y zu;
6      wenn Ein oder mehrere Knoten v_i ∈ F überlagern ein oder mehrere andere Knoten dann
7          wenn s = −1 dann
8              für alle v_i ∈ V : y(v_i) > y tue
9                  Erhöhe Y-Koordinate von v_i um 1;
10             Weise allen Knoten v ∈ F die Y-Koordinate y + 1 zu;
11         sonst
12             wenn s = 1 dann
13                 für alle v_i ∈ V : y(v_i) ≥ y tue
14                     Erhöhe Y-Koordinate von v_i um 1;
15                 Weise allen Knoten v ∈ F die Y-Koordinate y zu;
16     Speichere alle direkten Unterflüsse von F, die keine Unterflüsse besitzen, in U;
17     Wiederhole den Algorithmus mit jedem Unterfluss von F, abwechslend y_Oberfluss + 1
       und y_Oberfluss − 1, abwechselnd +1 und −1;
18     Füge F an U an;
19     Berechne Kantenüberschneidungen c_alt;
20     für jedes u_1 ∈ U tue
21         für jedes u_2 ∈ U tue
22             wenn u_1 ≠ u_2 dann
23                 Tausche Y-Koordinaten von u_1 und u_2;
24                 Berechne Kantenüberschneidungen c_neu;
25                 wenn c_neu < c_alt und es existieren keine Knotenüberlagerungen dann
26                     c_alt = c_neu;
27                 sonst
28                     Tausche Y-Koordinaten von u_1 und u_2;
29 Ende
```

Algorithmus 3.12 : Vergabe der Y-Koordinaten mit Reduzierung der Kantenüberschneidungen

Index	0	1	2	3
Direkter Unterfluss	F_1	F_2	F_3	F_4

Tabelle 3.5: Flussmenge U: Direkte Unterflüsse des Hauptflusses

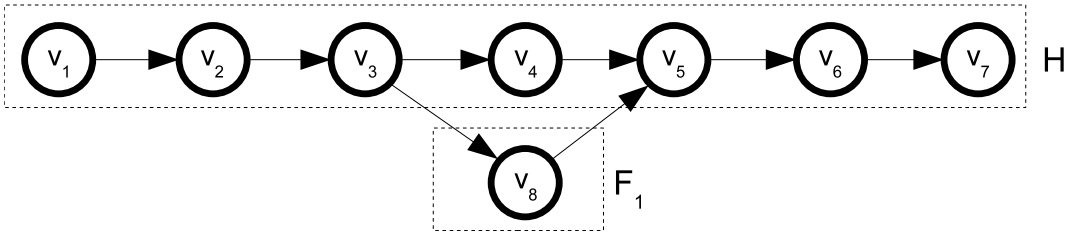

Abbildung 3.23: Dem Fluss F_1 Y-Koordinaten zuweisen

F_1 hat keine Unterflüsse, also wird durch Zeile 17 der Algorithmus mit $F = F_2, y \rightarrow y_{Oberfluss} - 1 = 0$ und $s = -1$ aufgerufen. Die Bedingung in Zeile 2 ergibt *true*, darum werden die Y-Koordinaten von H und F_1 um 1 erhöht und F_2 erhält die Y-Koordinate 1. Daraus resultiert der Graph aus Abbildung 3.24.

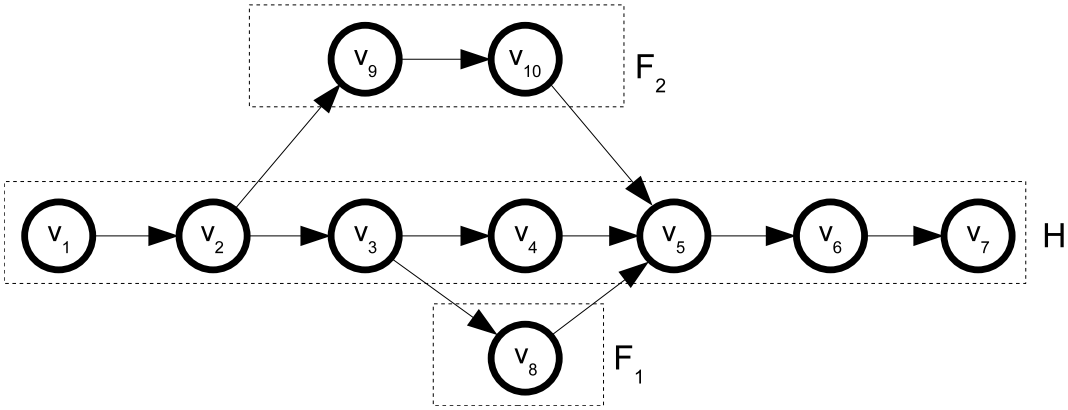

Abbildung 3.24: Dem Fluss F_2 Y-Koordinaten zuweisen

Auch F_2 hat keine Unterflüsse und der Algorithmus wird mit $F = F_3, y \rightarrow y_{Oberfluss} + 1 = 3$ und $s = 1$ aufgerufen. Abbildung 3.25 zeigt, dass sich Knoten aus den Flüssen F_1 und F_3 nun überlagern.

Zeile 12 kommt zum Tragen und alle Knoten mit einer Y-Koordinate, die größer oder gleich 3 ist, bekommen eine um 1 höhere Y-Koordinate. F_1 und F_3 haben somit die Y-Koordinate 4. Danach wird dem Fluss F_3 erneut die Y-Koordinate 3 zugewiesen und er liegt nun nach Abbildung 3.26 zwischen dem Hauptfluss und Fluss F_1.

Zuletzt erhält Fluss F_4 seine Y-Koordinaten durch Aufruf des Algorithmus mit $F = F_4, y \rightarrow$

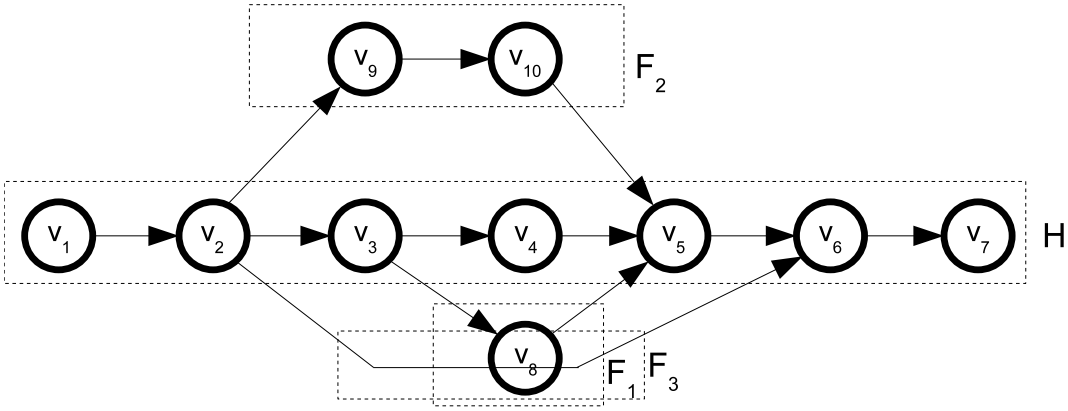

Abbildung 3.25: Dem Fluss F_3 Y-Koordinaten zuweisen. Resultat: Überlagerung von F_1 und F_3

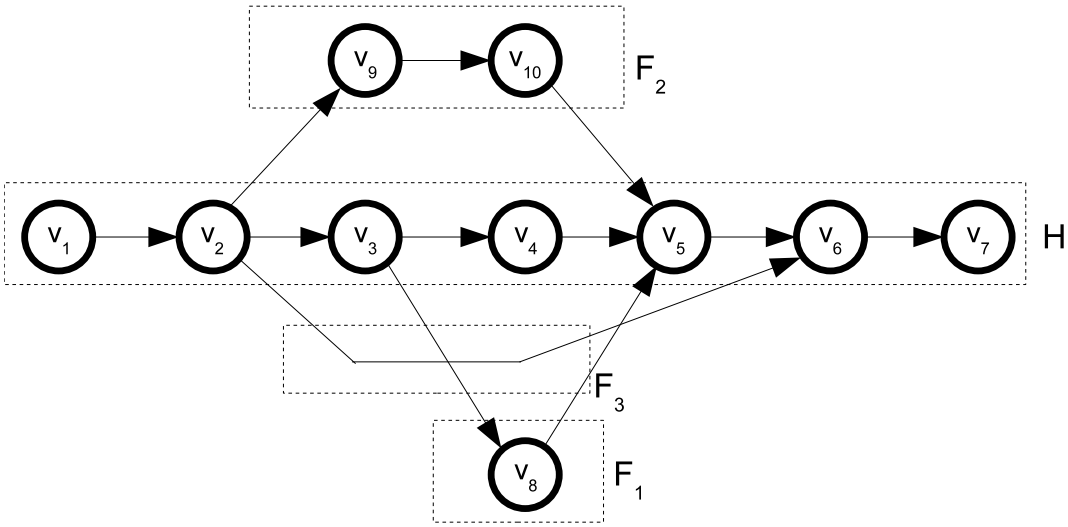

Abbildung 3.26: F_1 und F_3 neu anordnen

$y_{Oberfluss} - 1 = 1$ und $s = -1$. Wie Abbildung 3.27 zeigt, überlagern sich nun die Flüsse F_2 und F_4.

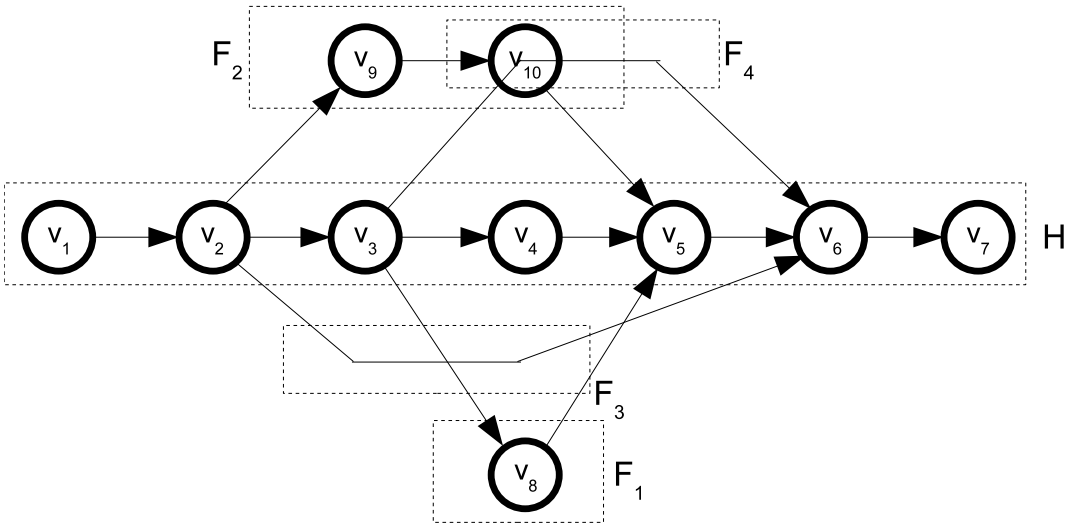

Abbildung 3.27: Dem Fluss F_4 Y-Koordinaten zuweisen. Resultat: Überlagerung von F_2 und F_4

Somit ist Zeile 7 erfüllt und alle Knoten mit einer Y-Koordinate, die größer als 1 ist, bekommen eine um 1 höhere Y-Koordinate. Somit werden H, F_1 und F_3 um 1 nach unten vesetzt. Daraufhin bekommt F_4 die Y-Koordinate $y \rightarrow y + 1 = 2$ zugewiesen und liegt damit zwischen Hauptfluss und F_2 (Abbildung 3.28).

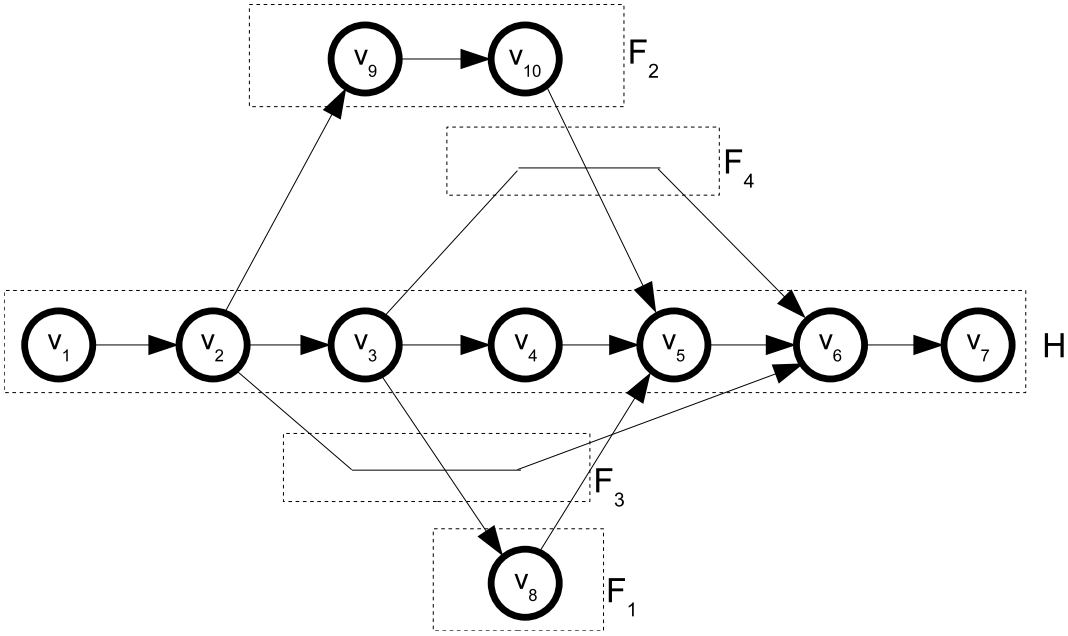

Abbildung 3.28: F_2 und F_4 neu anordnen

Nun sind sämtliche Unterflüsse gesetzt und die Heuristik zur Minimierung der Kantenüberschneidungen beginnt. Zunächst wird U um den Hauptfluss erweitert und beinhaltet nun die Flüsse

gemäß Tabelle 3.6.

Index	0	1	2	3	5
Direkter Unterfluss	F_1	F_2	F_3	F_4	H

Tabelle 3.6: Flussmenge U_{neu}: Direkte Unterflüsse des Hauptflusses inklusive Hauptfluss

Gemäß Abbildung 3.28 gibt es anfangs drei Kantenüberschneidungen. Durch systematisches Tauschen der Y-Koordinaten zweier Flüsse gemäß Zeile 20 bis 28 werden die Kantenüberschneidungen bis auf 0 reduziert und der Graph hat die Form aus Abbildung 3.21.

3.4.2 Lange Rückflüsse

In diesem Abschnitt wird der Sonderfall langer Rückflüsse erläutert. Ein solcher langer Rückfluss ist in Abbildung 3.29 zu sehen. Die Knotenfolge $F = \{v_8, v_9, v_{10}, v_{11}, v_{12}\}$ beinhaltet 5 Knoten, aber die zu ihr adjazenten Knoten des Oberflusses v_2 und v_4 liegen lediglich zwei Schritte auseinander.

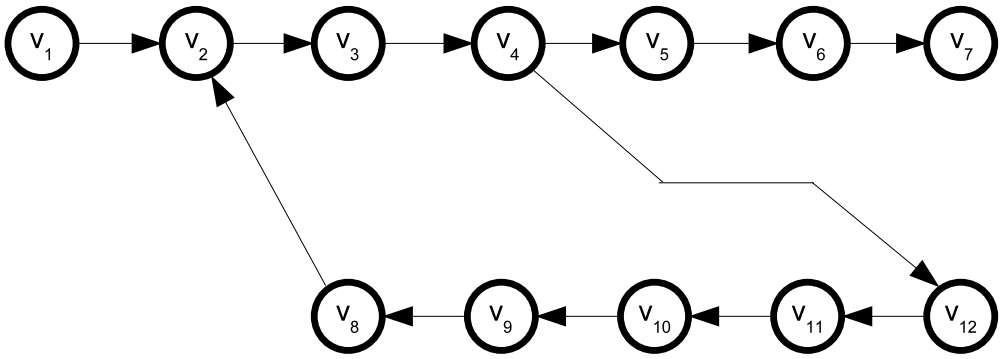

Abbildung 3.29: Graph mit langem Rückfluss

Die Kante $e = (v_4, v_{12})$ beinhaltet zwei Dummy-Knoten, die Bestandteil des Flusses F sind. Diese würden nach dem in Abschnitt 3.4.1 vorgestellten Verfahren dieselbe Y-Koordinate erhalten wie alle anderen Knoten des Flusses F. Damit würden sie auf den Knoten v_{10} und v_{11} liegen. Um dies zu vermeiden, müssen lange Rückflüsse erkannt werden und eine gesonderte Zuteilung der Y-Koordinaten erfahren, so dass sämtliche Knoten zueinander den geforderten Y-Abstand 1 haben.

Algorithmus 3.13 erkennt aufeinanderliegende Knoten innerhalb eines Flusses F und korrigiert ihre Lage. Dazu wird zunächst allen Dummy-Knoten $d \in F$ die Y-Koordinate des Oberflusses zugewiesen. Dadurch haben sie von den echten Knoten $v \in F$ den Abstand 1. Jedoch überlagern sie nun den Oberfluss. Daher wird jetzt der Algorithmus 3.14 ausgeführt, der ausgehend von F rekursiv die Y-Koordinaten sämtlicher Flüsse erhöht oder vermindert, die nicht den Mindestabstand von 1 zu einem anderen Fluss besitzen.

Eingabe : Flussweise geordnete Knotenmenge $Flows$ mit evtl sich überlagernden Knoten

Ausgabe : Knotenmenge $Flows$ ohne sich überlagernde Knoten

Beginn

- **für jedes** $F \in Flows$ **tue**
  - **wenn** *Dummy-Knotenmenge* $D \in F$ *überlagert echte Knotenmenge* $R \in F$ **dann**
    - **wenn** *Oberfluss von* F *liegt über* F **dann**
      - $direction = 1$;
    - **sonst**
      - $direction = -1$;
    - **für alle** $d \in D$ **tue**
      - Weise d die Y-Koordinate des Oberflusses zu;
  - Führe Algorithmus 3.14 aus mit F und $direction$;

Ende

Algorithmus 3.13 : Sich überlagernde Knoten erkennen und ihre Lage korrigieren

Eingabe : Fluss F mit evtl zu geringem Abstand zu einem Nachbarfluss

Ausgabe : Fluss F mit eingehaltenem Mindestabstand 1 zu jedem Nachbarfluss

Daten : d: Zahl, die bei Bedarf auf die Y-Koordinate aufaddiert wird

Beginn

- **wenn** *Es existiert mindestens ein Fluss* F_n*, der einen geringeren Abstand als 1 zu* F *hat* **dann**
  - **für alle** $f \in F$ **tue**
    - $y(f) \rightarrow y(f) + d$;
  - Wiederhole Algorithmus rekursiv mit allen Flüssen, die nun einen zu geringen Abstand zu F haben und d;

Ende

Algorithmus 3.14 : Flüsse versetzen, um Mindestabstand 1 zu gewährleisten

Am Beispiel des Graphen aus Abbildung 3.29 liegt der Oberfluss über dem langen Rückfluss. Damit ist in Algorithmus 3.13 $direction = 1$. Zunächst wird den Dummy-Knoten die Y-Koordinate ihres Oberflusses zugewiesen. Das Resultat ist eine Überlagerung. Danach wird Algorithmus 3.14 ausgeführt, der erkennt, dass der Oberfluss einen geringeren Abstand als 1 zum Unterfluss hat. Daher wird auf die Y-Koordinate aller Knoten des Unterflusses $direction = 1$ aufaddiert und es resultiert der Graph aus Abbildung 3.29.

3.4.3 Ausbalancieren des Graphen

In diesem Abschnitt wird erläutert, wie das Layout des Graphen weiter verbessert werden kann, indem Knoten *ausbalanciert* werden. Um den Begriff zu verdeutlichen, ist in Abbildung 3.30 ein nicht ausbalancierter Graph zu sehen, während Abbildung 3.31 denselben Graph in ausbalancierter Form darstellt.

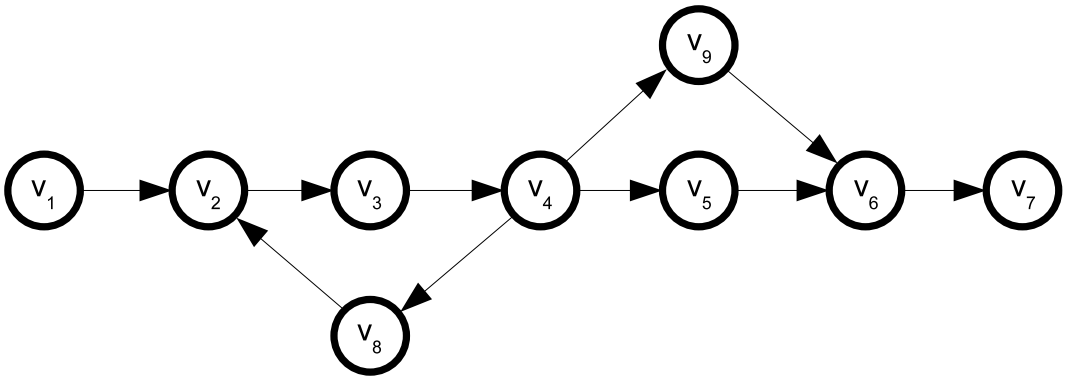

Abbildung 3.30: Nicht ausbalancierter Graph

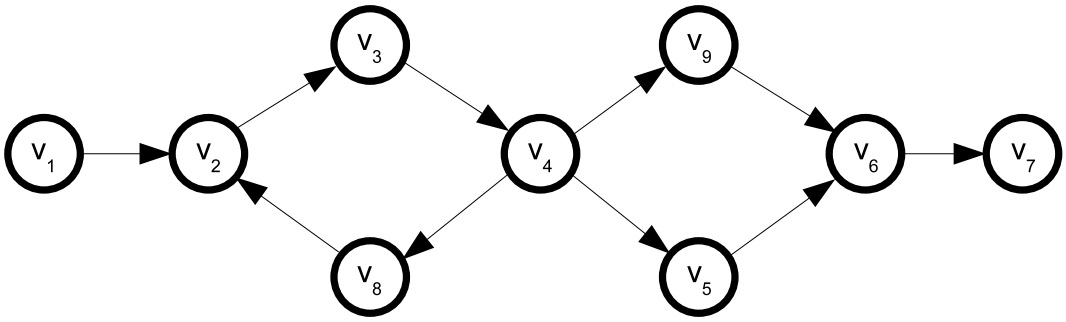

Abbildung 3.31: Ausbalancierter Graph

Die Knoten v_3 und v_8 sind jeweils um $0,5$ nach oben versetzt worden und die Knoten v_9 und v_5 um $0,5$ nach unten.

Algorithmus 3.15 nimmt die Ausbalancierung eines Graphen vor. Dabei wird für jeden Fluss F unterschieden, ob er nur einen oder mehrere *direkte Unterflüsse*, das heißt Flüsse, deren Anfangs-

und Endknoten adjazent zu Knoten aus F sind, besitzt. Im ersten Fall werden sämtliche Knoten des Oberflusses, die eine X-Koordinate besitzen, welche der Unterfluss nicht besitzt, um $0,5$ nach oben oder unten korrigiert. Im zweiten Fall werden die Knoten der balancierbaren Unterflüsse, die selbst keine Unterflüsse mehr besitzen, und die direkt darüber oder darunter liegenden Knoten ihres Oberflusses um $0,5$ nach oben oder nach unten versetzt. Balancierbar ist ein Unterfluss U_1 grundsätzlich dann, wenn kein weiterer Unterfluss U_2 existiert, der Knoten beinhaltet, welche dieselbe X-Koordinate aufweisen wie ein oder mehrere Knoten in U_1. Die Balance ist hier bereits dadurch gegeben, dass sich die beiden Flüsse gegenüberliegen. Zwei nicht ausbalancierbare Unterflüsse sind zur Veranschaulichung in Abbildung 3.32 dargestellt. Danach wird Algorithmus 3.14 ausgeführt, um sicherzustellen, dass der Mindestabstand von 1 überall eingehalten wird. In beiden Fällen wird nur dann ausbalanciert, wenn der Abstand zwischen Oberfluss und Unterfluss höchstens 1 bzw. 2 beträgt. Sonst würde die Ausbalancierung vor allem in großen Graphen das Layout unpräzise erscheinen lassen, da ein Unterfluss, der eine teilweise Verschiebung seines Oberflusses um 0,5 Y-Koordinaten bewirkt, aufgrund seines großen Abstandes zum Oberfluss nicht unmittelbar erkannt wird. Dadurch entsteht leicht der Eindruck, dass der Oberfluss aufgrund seiner teilweise versetzten Knoten falsch gelayoutet wurde.

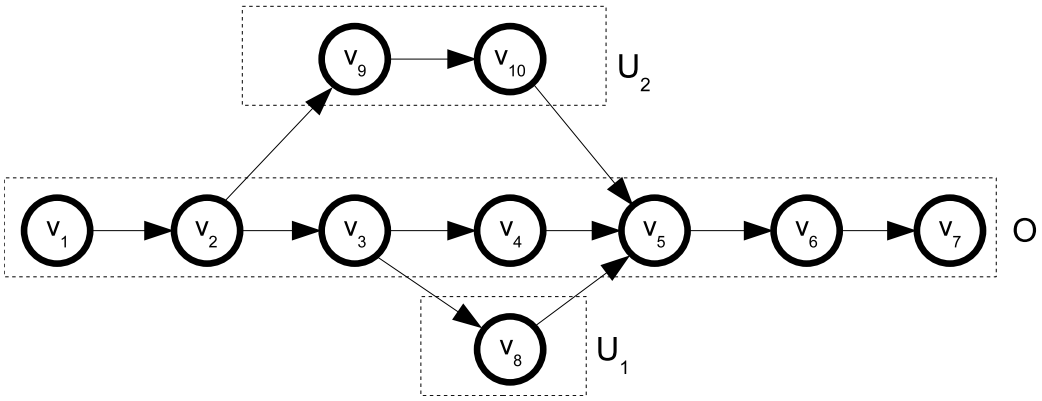

Abbildung 3.32: Zwei nicht ausbalancierbare Unterflüsse U_1 und U_2

Anhand des Graphen in Abbildung 3.33 wird die Funktionsweise von Algorithmus 3.15 verdeutlicht. Die Abbildung zeigt den Graph nach der Berechnung der Y-Koordinaten entsprechend Abschnitt 3.4.1.

In Zeile 2 wird F zunächst der Hauptfluss zugewiesen. Zeile 4 greift, da der Hauptfluss nur einen direkten Unterfluss hat. Nun kommt Zeile 6 zum Tragen , denn der Oberfluss liegt über dem Unterfluss. Die Y-Koordinaten der Knoten v_1 und v_9 werden um $0,5$ erhöht und damit herabgesetzt. Es resultiert ein Layout nach Abbildung 3.34.

Nun wird der aus den Knoten v_{10}, v_{11}, v_{12}, v_{13} und v_{14} bestehende direkte Unterfluss U_H des Hauptflusses behandelt. Dieser hat zwei Unterflüsse $F_1 = \{v_{15}\}$ und $F_2 = \{v_{16}\}$ und Zeile 14 des

Eingabe : Flussweise geordnete unbalancierte Knotenmenge $Flows$

Ausgabe : Balancierte Knotenmenge $Flows$

1 **Beginn**
2 **für alle** $F \in Flows$ **tue**
3 Speichere alle direkten Unterflüsse von F in U;
4 **wenn** F *hat nur einen Unterfluss* F_1 **dann**
5 **wenn** $|y(F) - y(F_i)| \leq 2$ **dann**
6 **wenn** *Oberfluss liegt über Unterfluss* **dann**
7 **für alle** $v_F \in F : \nexists v_{Fi} \in F_i : x(v_F) = x(v_{Fi})$ **tue**
8 $y(v_F) \rightarrow y(v_F) + 0,5$;
9 Führe Algorithmus 3.14 mit F_i und 0,5 aus;
10 **sonst**
11 **für alle** $v_F \in F : \nexists v_{Fi} \in F_i : x(v_F) = x(v_{Fi})$ **tue**
12 $y(v_F) \rightarrow y(v_F) - 0,5$;
13 Führe Algorithmus 3.14 mit F_i und -0,5 aus;
14 **sonst**
15 Speichere alle balancierbaren Unterflüsse von F in U;
16 **für alle** $F_i \in U$ **tue**
17 **wenn** F_i *hat Unterflüsse ODER* F_i *ist langer Rückfluss ODER* $|y(F) - y(F_i)| > 1$ **dann**
18 $U \rightarrow U - F_i$;
19 **für jedes** $F_i \in U$ **tue**
20 **wenn** *Oberfluss liegt über Unterfluss* **dann**
21 $y(F_i) \rightarrow y(F) + 0,5$;
22 **für alle** $v_F \in F : \exists v_{Fi} \in F_i : x(v_F) = x(v_{Fi})$ **tue**
23 $y(v_F) \rightarrow y(v_F) - 0,5$;
24 Führe Algorithmus 3.14 mit F_i und -0,5 aus;
25 **sonst**
26 $y(F_i) \rightarrow y(F) - 0,5$;
27 **für alle** $v_F \in F : \exists v_{Fi} \in F_i : x(v_F) = x(v_{Fi})$ **tue**
28 $y(v_F) \rightarrow y(v_F) + 0,5$;
29 Führe Algorithmus 3.14 mit F_i und 0,5 aus;
30 **Ende**

Algorithmus 3.15 : Knoten ausbalancieren

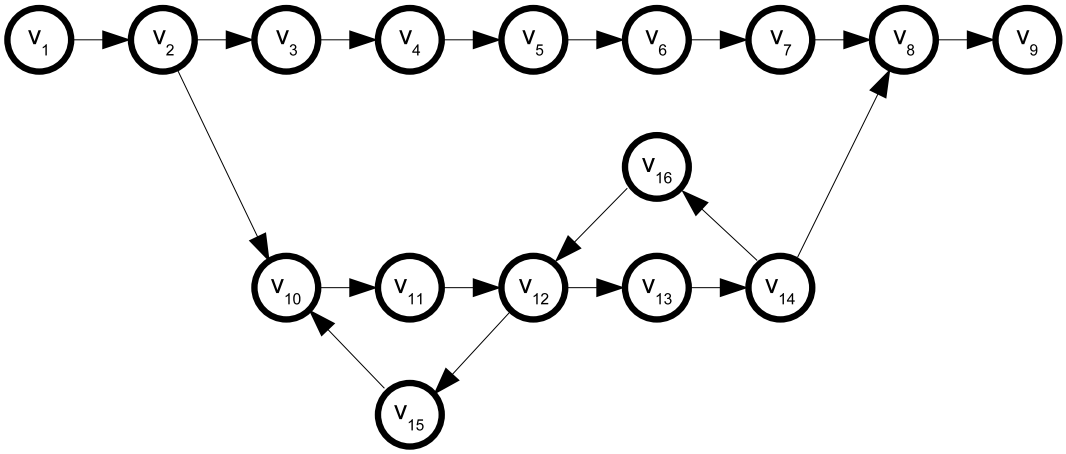

Abbildung 3.33: Unbalancierter Graph mit mehreren Unterflüssen

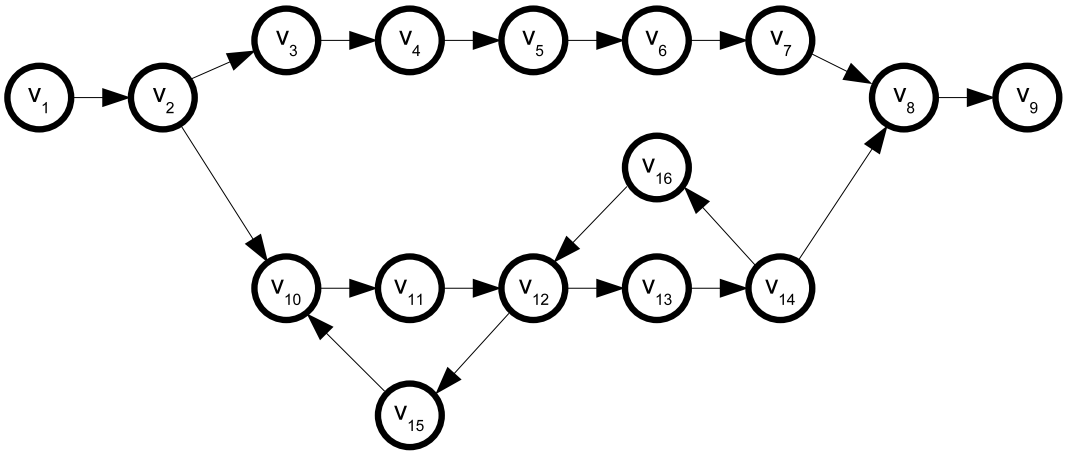

Abbildung 3.34: Herabsetzen von vier Knoten des Hauptflusses

Algorithmus greift. Zeile 16 weist F_i zunächst F_1 zu. Damit ist Zeile 20 gültig, denn U_H liegt über F_1. Allen Knoten $v \in F_1$, in diesem Fall also nur der Knoten v_{15}, wird nun die Y-Koordinate von U_H erhöht um $0,5$ zugewiesen. Die Y-Koordinate des Knotens v_{11} wird um $0,5$ vermindert. Da es keine zu geringen Abstände zwischen Flüssen gibt, bringt die Ausführung von Algorithmus 3.14 in Zeile 24 keine Änderung mit sich. Der Graph hat nun die Form aus Abbildung 3.35.

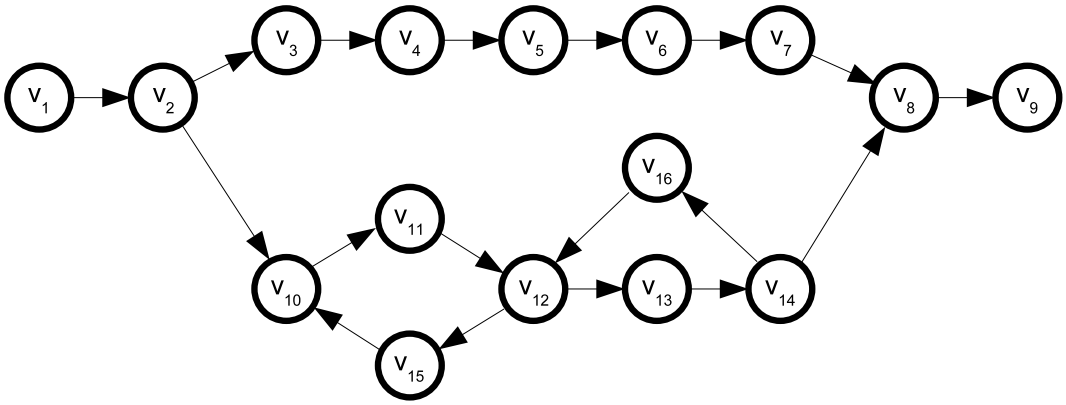

Abbildung 3.35: Ausbalancierung des Unterflusses (1)

Nun wird in Zeile 16 F_2 auf F_i zugewiesen. Zeile 25 gilt nun, da F_2 über U_H liegt. Knoten v_16 wird nun die Y-Koordinate von U_H vermindert um 1 zugewiesen (Zeile 23). Die Zeilen 27 und 28 sorgen dafür, dass die Y-Koordinate von v_{13} um $0,5$ erhöht wird. Abbildung 3.36 zeigt das hierdurch erzeugte endgültige Layout.

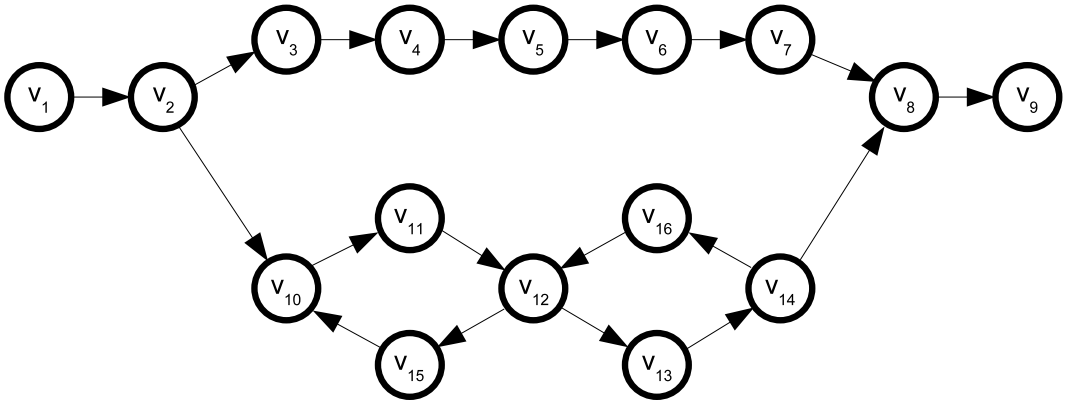

Abbildung 3.36: Ausbalancierung des Unterflusses (2)

Durch Ausführung von Algorithmus 3.15 und 3.14 kann es vorkommen, dass Knoten so verschoben werden, dass die niedrigste Y-Koordinate nicht mehr 1 ist. Deshalb folgt als abschließender Schritt zur Generierung des Layouts Algorithmus 3.16, der sicherstellt, dass die niedrigste Y-Koordinate 1 beträgt, indem das gesamte Netz bei Bedarf nach oben oder unten verschoben wird.

Eingabe : Graph mit Kotenmenge V und unbekannter niedrigster Y-Koordinate

Ausgabe : Graph mit niedrigster Y-Koordinate 1

```
1 Beginn
2     minY ist aktuell niedrigste Y-Koordinate;
3     wenn minY ≠ 1 dann
4         d → 1 − minY;
5         für alle v ∈ V tue
6             y(v) → y(v) + d;
7 Ende
```

Algorithmus 3.16 : Niedrigste Y-Koordinate auf 1 setzen

Kapitel 4

SVG-Generierung

In diesem Kapitel wird beschrieben, wie aus dem bisher nur schematisch erstellten Layout ein Bild entsteht. Dieses Bild wird in Form einer SVG-Grafik erzeugt. Die Überführung der abstrakten Beschreibung des Modulnetzes in Form von Knoten mit relativen X- und Y-Koordinaten und inzidenten Kanten in eine visuelle Darstellung wird im Folgenden als *Rendern* bezeichnet. Der SVG-Code wird dabei vom Programm automatisch erzeugt und in einer SVG-Datei abgespeichert. Diese kann dann von einem Web-Browser mit SVG-Unterstützung oder von einem SVG-Viewer betrachtet werden.

4.1 Allgemeine Grundlagen zu SVG

SVG steht für *Scalable Vector Graphics* und ist eine XML-Grammatik zur Beschreibung von Grafiken, die aus drei Elementarten zusammengesetzt sein können: Vektorgrafiken, Bilder und Text (World Wide Web Consortium, 2003a).

Die hier vorgestellten Grundlagen decken die Bereiche ab, die im Rahmen der Implementation benutzt wurden und in Folge dessen zum Verständnis dieses Kapitels notwendig sind.

4.1.1 Koordinatensystem

Positionsangaben beziehen sich in SVG auf ein Koordinatensystem, das seinen Ursprung in der linken oberen Ecke hat. Die Angabe von X-Koordinaten bezieht sich auf die horizontale Entfernung eines Punktes vom Ursprung, Y-Koordinaten beschreiben die vertikale Entfernung. Somit ist der grundsätzliche Aufbau der Zeichenfläche derselbe wie in Abbildung 3.1 in Kapitel 3.1.

4.1.2 Aufbau einer SVG-Datei

Eine SVG-Datei ist in ihrer Grundform immer nach demselben Muster aufgebaut. Dieses ist SVG-Code 4.1 zu entnehmen.

```
1  <?xml version="1.0" standalone="no"?>
2  <!DOCTYPE svg PUBLIC "-//W3C//DTD␣SVG␣1.0//EN" "http://www.w3.org/
       TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
3  <svg width="1000" height="800" version="1.1" xmlns="http://www.w3.
       org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
4  <script type="text/ecmascript" >
5  <![CDATA[
6  <!-- Hier kann Script-Code definiert werden -->
7  ]]>
8  </script>
9  <style type="text/css">
10 <![CDATA[
11 <!-- Hier können CSS-Definitionen vorgenommen werden -->
12 ]]>
13 </style>
14 <defs>
15 <!-- Hier können Attribute definiert werden -->
16 </defs>
17 <g id="einmaligeID">
18 <!-- Hier werden alle Objekte definiert, die später in der SVG-
       Grafik zu sehen sein sollen -->
19 </g>
20 </svg>
```

Listing 4.1: Aufbau einer SVG-Datei

Erklärung zu Listing 4.1:

Zeile 1: Hier wird festgelegt, dass es sich bei dem Dokument um ein XML-Dokument handelt.

Zeile 2: Hier wird das Dokument genauer definiert als SVG-Grafik.

Zeile 3: Mit *width* und *height* wird die Größe der Grafik (ohne Angabe einer Einheit standardmäßig in Pixeln) definiert. Außerdem wird die SVG-Version angegeben.

Zeilen 4 bis 8: Falls JavaScript-Funktionen benötigt werden, so werden diese hier eingestellt.

Zeilen 9 bis 13: Wenn CSS-Definitionen verwendet werden, so werden diese hier deklariert.

Zeilen 14 bis 16: In diesem Bereich werden Definitionen vorgenommen, die mittels einer ID später wiederverwendet werden können.

Zeilen 17 bis 19: Hier werden alle Objekte, die später in der SVG-Grafik angezeigt werden sollen, definiert. Mit dem optionalen *g*-Attribut können beliebig viele Objekte gruppiert werden. Dadurch ist es möglich, allen Objekten der Gruppe durch einmaliges definieren eines Attributes dieses Attribut zuzuweisen. Hierbei ist jedoch Vorsicht geboten, falls *text*-Objekte zur Gruppe gehören, denn die Textdarstellung kann durch Attribute wie *stroke-width* negativ beeinflusst werden. Die Vergabe einer ID an die Gruppe sorgt dafür, dass diese referenziert werden kann. Die Gruppe wird durch *</g>* geschlossen.

Zeile 20: Das Ende des SVG-Dokuments wird durch *</svg>* angezeigt.

4.1.3 Einheiten und das viewBox-Attribut

Die Angabe von Koordinaten kann mit und ohne Angabe einer Einheit erfolgen. Erlaubte Einheiten sind nach World Wide Web Consortium (2003c) *em, ex, px, pt, pc, cm, mm, in* und *percentages*. Erfolgt keine Angabe einer Einheit, wird automatisch die *user-Einheit* benutzt. Diese entspricht im Grundzustand einem Pixel auf dem Ausgabemedium und verändert sich bei Benutzung von Attributen wie *viewBox*. Durch die Angabe einer *viewBox* kann bestimmt werden, welcher Ausschnitt einer Grafik auf der Zeichenfläche angezeigt wird. Durch die Vergrößerung des Bildausschnittes entspricht die Länge und Dicke sämtlicher Formen auf dem Ausgabemedium nicht mehr der vorherigen Anzahl an Pixeln, wird aber von SVG als dieselbe Anzahl interpretiert. Somit bewirkt eine Änderung von einem Pixel in der viewBox eine deutlich größere Änderung auf dem Ausgabemedium als dieselbe Veränderung ohne viewBox. Eine solche viewBox wird beispielsweise in Abschnitt 4.2 bei der Erzeugung von Pfeilspitzen benutzt. Ein Beispiel für die Auswirkung der Anwendung des Attributes *viewBox="200 150 200 150"* ist in Abbildung 4.1 dargestellt.

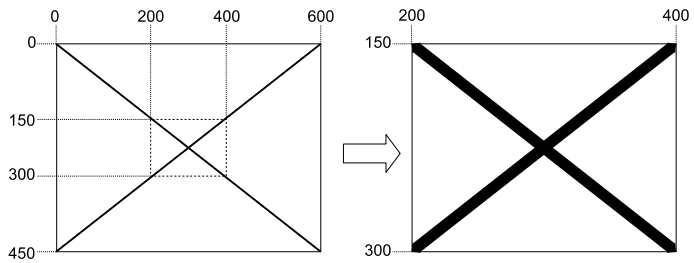

Abbildung 4.1: Beispiel für die Anwendung des viewBox-Attributes

4.1.4 Grundformen

Grundformen in SVG sind Rechtecke, Kreise, Ellipsen, Linien, Polylinien und Polygone. Zur Darstellung von Knoten und Kanten für ein Modulnetz werden Rechtecke, Kreise und Linien benötigt. Deren Erzeugung ist in Abschnitt 4.2 erklärt und wird deshalb hier nicht weiter erläutert. Eine detailierte Beschreibung aller Grundformen ist in World Wide Web Consortium (2003b) zu finden.

4.2 Grundlagen zum Rendern von Knoten und Kanten

Die *relativen* Koordinaten x_R und y_R, die den Knoten unseres Netzes in Kapitel 3 schematisch zugewiesen wurden, müssen zunächst in *SVG-taugliche*, also *absolute*, Koordinaten x_A und y_A umgewandelt werden. Dazu reicht es aus, x_R und y_R mit einer gewünschten Pixel-Distanz der Knoten zueinander zu multiplizieren. Sollen die Mittelpunkte der Knoten beispielsweise eine Distanz von 100 Pixeln aufweisen, so würde ein Knoten v mit $x_R(v) = 1$ und $y_R(v) = 1$ die absoluten Koordinaten $x_A(v) = 100$ und $y_A(v) = 100$ zugewiesen bekommen. Ab hier bezeichnen Koordinaten, die nicht explizit als relativ oder absolut gekennzeichnet sind, immer absolute Koordinaten.

Nun wird auch die in Kapitel 3 vernachlässigte Unterscheidung der Knoten zwischen Stellen und Transitionen wieder eingeführt. Diese werden zunächst als einfache Kreise und Rechtecke dargestellt, da dies die Veranschaulichung des Problems der exakten *Anstoßpunkte* der Kanten vereinfacht. Später wird auch das Einbinden kleiner Bilddateien bzw. *Icons* zur Visualisierung von Knoten ermöglicht.

4.2.1 Darstellung von Stellen als Kreise

Ein Kreis bzw. eine Stelle mit den Mittelpunktkoordinaten $x = 400$ und $y = 100$ sowie dem Radius 20 wird durch den SVG-Code 4.2 erzeugt. Die Attribute *fill=white* und *stroke=black* sorgen dafür, dass der Kreis weiß ausgefüllt wird und die Linienfarbe schwarz ist.

```
<circle cx="400" cy="100" r="20" fill="white" stroke="black" />
```

Listing 4.2: SVG-Code: Erzeugung eines Kreises

4.2.2 Darstellung von Transitionen als Rechtecke

Die Position eines Rechtecks bzw. einer Transition wird durch Angabe von dessen linker oberer Ecke bestimmt. Als weitere Attribute werden die Seitenlängen benötigt. Da bisher nur die Koordinaten des

Mittelpunktes des Rechtecks bekannt sind, wird davon jeweils die halbe Seitenlänge abgezogen, um die Position der linken oberen Ecke zu erhalten. Der SVG-Code 4.3 erzeugt ein Rechteck mit den Seitenlängen 40 und 40 sowie mit den Eckpunktkoordinaten 480 und 80. Damit läge das Rechteck im Abstand 100 rechts neben dem eben erzeugten Kreis auf gleicher Höhe.

```
<rect x="480" y="80" width="40" height="40" fill="white"
stroke="black" />
```

Listing 4.3: SVG-Code: Erzeugung eines Rechtecks

4.2.3 Darstellung von Kanten als Linien

Kanten werden in SVG als gerade Linien mit dem *line*-Attribut gesetzt. Dieses erwartet die Koordinaten des Anfangs- und Endpunktes und zieht eine Linie zwischen diesen Koordinaten. SVG-Code 4.4 erzeugt eine schwarze Linie der Stärke 2 Pixel vom Punkt $(50, 50)$ bis $(500, 350)$.

```
<line x1="50" y1="50" x2="500" y2="350" stroke="black" stroke-
    width="2" />
```

Listing 4.4: SVG-Code: Erzeugung einer Linie

4.2.4 Berechnung der Anstoßpunkte

Anfangs- und Endpunkt, die zur Erzeugung einer Linie benötigt werden, sind die Anstoßpunkte der Kante an die inzidenten Knoten. Die idealen Anstoßpunkte sind auf Abbildung 4.2 rechts zu sehen. Sie sind der Schnittpunkt einer gedachten Linie zwischen den Mittelpunkten beider Knoten mit deren Rand. Links auf der Abbildung ist eine unbefriedigende Verbindung der Knoten zu sehen. Die X-Koordinate des Anstoßpunktes ist hier jeweils die X-Koordinate des Mittelpunktes des Knotens. So sind die Punkte zwar leicht zu berechnen, das erzielte Ergebnis ist jedoch suboptimal.

Die Anstoßpunkte entsprechend der rechten Kante in Abbildung 4.2 sind mit Hilfe mathematischer Methoden aus der Analysis zu berechnen.

Die allgemeine Form einer Geradengleichung lautet

$$y = m * x + b \tag{4.1}$$

Da die Mittelpunkte beider Knoten bekannt sind, kann die Geradengleichung einer Kante mit der

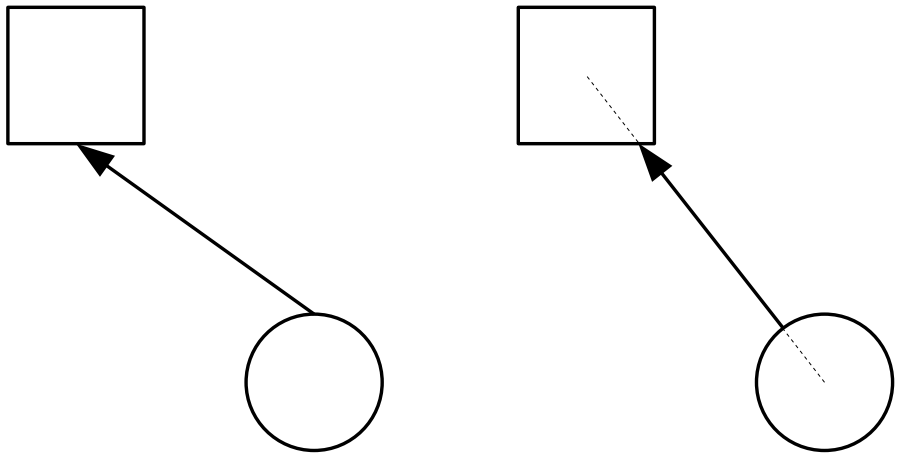

Abbildung 4.2: Anstoßpunkte von Kanten

Punktsteigungsform aufgestellt werden. Die Koordinatenpaare (x_1, y_1) und (x_2, y_2) seien die Mittelpunkte der beiden Knoten. m bezeichne die Steigung der Geraden und b den Y-Achsenabschnitt. Dann folgt:

$$y_1 = m * x_1 + b \tag{4.2}$$

$$y_2 = m * x_2 + b \tag{4.3}$$

Zieht man die erste von der zweiten Gleichung ab, so folgt

$$y_2 - y_1 = m * (x_2 - x_1) \tag{4.4}$$

Für die Steigung m ergibt sich somit

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{4.5}$$

Zur Berechnung von b setzt man Formel 4.5 in 4.2 ein und stellt nach b um, womit

$$b = y_1 - \frac{y_2 - y_1}{x_2 - x_1} * x_1 \tag{4.6}$$

Damit ist eine Geradengleichung nach 4.1 aufstellbar und die Schnittpunkte zwischen der Geraden und den Knoten, d.h. entweder einem Kreis oder einem Rechteck, können berechnet werden.

Um die Schnittpunkte zwischen einer beliebigen Geraden und einem Kreis zu berechnen, wird die Kreisgleichung benötigt. Diese ist

$$(x - x_m)^2 + (y - y_m)^2 = r^2 \tag{4.7}$$

wobei (x_m, y_m) der Mittelpunkt und r der Radius des Kreises ist. Nun wird eine Geradengleichung der Form 4.1 in die Kreisgleichung 4.7 eingesetzt und es folgt

$$(x - x_m)^2 + (m * x + b - y_m)^2 = r^2 \tag{4.8}$$

Diese Gleichung wird jetzt umgeformt, bis sich die pq-Formel-taugliche Normalform ergibt:

$$x^2 + \frac{-2x_m + 2bm - 2my_m}{1 + m^2} * x + \frac{x_m^2 + b^2 - 2by_m + y_m^2 - r^2}{1 + m^2} = 0 \tag{4.9}$$

Damit gilt

$$x_{1;2} = -\frac{-2x_m + 2bm - 2my_m}{2 * (1 + m^2)} \pm \sqrt{\left(\frac{-2x_m + 2bm - 2my_m}{2 * (1 + m^2)}\right)^2 - \frac{x_m^2 + b^2 - 2by_m + y_m^2 - r^2}{1 + m^2}} \tag{4.10}$$

Von den beiden resultierenden X-Koordinaten der Schnittpunkte wird nun der gewählt, der den geringsten Abstand zum adjazenten Knoten aufweist. Die Y-Koordinate des Schnittpunktes ergibt sich nun durch Einsetzen des gefundenen x in 4.1.

Ein Rechteck wird als Gruppierung von vier Geraden betrachtet, wobei jede Seite eine Gerade darstellt. Mit einem Rechteck hat die Gerade, wenn sie nicht zwei Punkte mit identischen Y-Koordinaten verbindet, vier Schnittpunkte. Ein solcher Schnitt einer Geraden durch ein Rechteck ist in Abbildung 4.3 illustriert.

Die vier Schnittpunkte sind S_a, S_b, S_c und S_d mit den Geraden a, b, c und d. Von jeder der vier Geraden wird die Geradengleichung aufgestellt. Für die linke und rechte Seite des Rechtecks, also die Geraden d und b, lauten diese

$$x = x_d \tag{4.11}$$

$$x = x_b \tag{4.12}$$

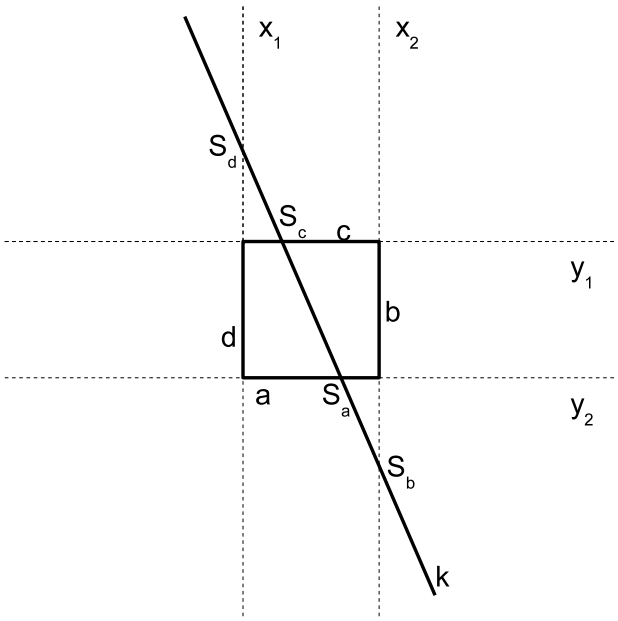

Abbildung 4.3: Schnitt einer Geraden durch ein Rechteck

und für die Geraden a und c

$$y = y_a \tag{4.13}$$

$$y = y_c \tag{4.14}$$

wobei y_a, x_b, y_c und x_d bekannt sind. Somit müssen diese Werte in die Gleichung 4.1 der Geraden k eingesetzt werden und man erhält die entsprechende fehlende Koordinate des Schnittpunktes. Von den vier Punkten kommen nun genau die beiden in Frage, deren X-Koordinate zwischen x_1 und x_2 und deren Y-Koordinate zwischen y_1 und y_2 liegt. Von diesen beiden Punkten wird nun derjenige ausgewählt, der näher am adjazenten Knoten liegt.

4.2.5 Erzeugung der Pfeilspitzen

Da die Knoten des Modulnetzes nicht durch Linien, sondern durch Pfeile verbunden sind, fehlt nun am Ende jeder Linie eine Pfeilspitze. Diese wird durch das *marker*-Attribut erzeugt. Dazu wird zunächst eine entsprechende Pfeilspitze definiert und mit einer ID versehen (SVG-Code 4.5).

Die in Listing 4.5 verwendeten Optionen sind im Einzelnen:

viewBox Legt den sichtbaren Bereich des Markers in Form eines Rechtecks fest. Die vier erwarteten Angaben sind X-und Y-Koordinate der linken oberen Ecke sowie Breite und Höhe des

```
<marker id="dreieck" viewBox="0␣0␣10␣10" refX="10" refY="5"
    markerUnits="markerUnits" markerWidth="6" markerHeight="6"
    orient="auto" stroke="black"  fill="black">
<path d="M␣0␣0␣L␣10␣5␣L␣0␣10␣Z" />
</marker>
```

Listing 4.5: SVG-Code: Definition einer Pfeilspitze

Rechtecks.

refX, refY Bestimmt die Lage des Markers in Bezug zum Pfadende. Bei $refX = refY = 0$ stößt der Marker mit der linken oberen Ecke genau an das Ende der Linie. Hier wird der Marker durch entsprechende Werte so verschoben, dass die 'Spitze' des Dreiecks mit dem Ende der Linie zusammenfällt.

markerUnits Bestimmt die verwendete Maßeinheit. Zulässige Werte sind *strokeWidth* und *userSpaceOnUse*. Im ersten Fall passt sich die Größe der Pfeilspitze an die Dicke der zugehörigen Linie an. Die zweite Möglichkeit sorgt dafür, dass die Pfeilspitze immer dieselbe definierte Größe behält.

markerWidth, markerHeight Ermöglicht die Veränderung der dargestelleten Größe des Markers in der in *markerUnits* eingestellten Maßeinheit.

orient Mit dem Wert *auto* wird die Pfeilspitze automatisch so rotiert, dass sie in die Richtung der zugehörigen Linie zeigt. Alternativ kann auch ein Winkel angegeben werden.

stroke Hier kann die Farbe der Umrandung der Pfeilspitze angegeben werden.

fill Lässt die Angabe einer Füllfarbe der Pfeilspitze zu.

path Dieses Attribut legt die Form der Pfeilspitze fest. Dabei steht M für *moveTo*, L für *lineTo* und Z für *closePath*. Die darauf folgenden Zahlen sind Koordinaten der entsprechenden Punkte, zwischen denen eine Linie gezogen wird. Z sorgt dafür, dass wieder zum Ausgangspunkt zurückgekehrt wird.

Eine Kante wird, sofern sie nicht zwei Dummy-Knoten verbindet, fortan mit dem SVG-Code 4.6 erzeugt.

```
<line x1="50" y1="50" x2="500" y2="350" stroke="black" stroke-
    width="2" marker-end="url(#dreieck)␣/>
```

Listing 4.6: SVG-Code: Erzeugung eines Pfeils

4.2.6 Beispiel eines automatisch gerenderten Graphen

Ein Screenshot eines nach den in diesem Abschnitt vorgestellten Methoden gerenderten SVG-Graphen ist in Abbildung 4.4 zu sehen. Der zugehörige automatisch generierte SVG-Code findet sich in Listing 4.7.

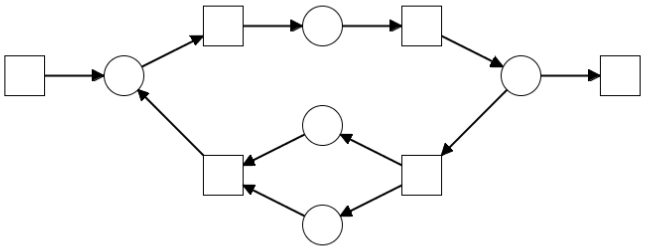

Abbildung 4.4: Screenshot eines SVG-Graphen

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD␣SVG␣1.0//EN" "http://www.w3.org/
    TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="800" height="400" version="1.1" xmlns="http://www.w3.
    org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
<!-- Define arrowhead of color black -->
<defs>
<marker id="blackTriangle" viewBox="0␣0␣10␣10" refX="10" refY="5"
    markerUnits="strokeWidth" markerWidth="6" markerHeight="6"
    orient="auto" stroke="black"  fill="black">
<path d="M␣0␣0␣L␣10␣5␣L␣0␣10␣z" />
</marker>
</defs>
<g id="moduleNet">
<rect x="80" y="130" width="40" height="40" fill="white" stroke="
    black" />
<rect x="280" y="80" width="40" height="40" fill="white" stroke="
    black" />
<rect x="280" y="230" width="40" height="40" fill="white" stroke="
    black" />
<rect x="480" y="80" width="40" height="40" fill="white" stroke="
    black" />
```

```
<rect x="480" y="230" width="40" height="40" fill="white" stroke="
    black" />
<rect x="680" y="130" width="40" height="40" fill="white" stroke="
    black" />
<circle cx="200" cy="150" r="20" fill="white" stroke="black" />
<circle cx="400" cy="100" r="20" fill="white" stroke="black" />
<circle cx="400" cy="200" r="20" fill="white" stroke="black" />
<circle cx="600" cy="150" r="20" fill="white" stroke="black" />
<circle cx="400" cy="300" r="20" fill="white" stroke="black" />
<line stroke-width="2" x1="218" y1="141" x2="280" y2="110" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="382" y1="209" x2="320" y2="240" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="382" y1="291" x2="320" y2="260" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="420" y1="100" x2="480" y2="100" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="586" y1="164" x2="520" y2="230" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="620" y1="150" x2="680" y2="150" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="120" y1="150" x2="180" y2="150" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="280" y1="230" x2="214" y2="164" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="320" y1="100" x2="380" y2="100" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="480" y1="240" x2="418" y2="209" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="520" y1="110" x2="582" y2="141" stroke=
    "black" marker-end="url(#blackTriangle)" />
<line stroke-width="2" x1="480" y1="260" x2="418" y2="291" stroke=
    "black" marker-end="url(#blackTriangle)" />
</g>
</svg>
```

Listing 4.7: SVG-Code: Beispiel eines automatisch generierten Graphen

4.3 Ästhetische, informative und interaktive Gestaltung des Modulnetzes

Die genaue Position, Größe und Form der Knoten und Kanten auf der Zeichenfläche steht fest. Damit ist eine Zeichnung nach Abbildung 4.4 zu realisieren. Um ein solches Netz visuell ansprechender zu gestalten, wird hier beschrieben, wie die Knoten eingefärbt werden können. Da das Netz dann zwar ästhetisch aussieht, aber keinerlei Informationen beinhaltet, wird das Einfügen von Text zur Beschriftung der Knoten erläutert. Ein weiterer Abschnitt wird dem Erzeugen von Tooltips und dem Erstellen von Skript-Code zur Ereignisbehandlung gewidmet.

4.3.1 Einfärben von Knoten

Eine einfache Möglichkeit zur Farbvergabe an Knoten ist die Benutzung des *fill*-Attributes gefolgt von einer Farbe wie in Listing 4.8 gezeigt.

```
<circle cx="200" cy="150" r="20" fill="red" stroke="black" />
```

Listing 4.8: SVG-Code: Erzeugung eines roten Kreises

Das Ergebnis ist eine schwarz umrandete Stelle mit roter Füllung. Optisch anregender ist die Verwendung eines Farbverlaufes zur Füllung der Knoten. Dieser lässt zwei vorher definierte Farben innerhalb der referenzierenden Form auf eine selbst zu definierende Art und Weise ineinander verlaufen.

Es werden zwei Arten von Farbverläufen voneinander unterschieden: *Linearer* und *radialer* Farbverlauf. Diese werden in den Definitionsbereich der SVG-Datei geschrieben und mit einer ID zur späteren Referenzierung versehen. Die Listings 4.9 und 4.10 definieren einen linearen und einen radialen Farbverlauf.

```
1 <linearGradient id="verlauf_lin" x1="0%" x2="80%" y1="0%" y2="80%"
      spreadMethod="reflect">
2 <stop offset="0%" stop-color="white" />
3 <stop offset="100%" stop-color="black" />
4 </linearGradient>
```

Listing 4.9: SVG-Code: Definition eines linearen Farbverlaufes

Erklärung zu Listing 4.9:

Zeile 1: Der lineare Farbverlauf wird eingeleitet durch *linearGradient* gefolgt von der ID, die den

Verlauf eindeutig kennzeichnet. Die vier Positionsangaben bestimmen, wie die referenzierende Form mit dem Farbverlauf gefüllt wird. In diesem Beispiel findet der Farbverlauf von 0% bis 80% der Breite und Höhe der Form statt und verläuft somit diagonal. Da der Farbverlauf die Form nicht komplett ausfüllt, wird das Attribut *spreadMethod="reflect"* verwendet. Dieses sorgt dafür, dass der Verlauf im nicht ausgefüllten Teil der Form umgekehrt fortgesetzt wird.

Zeilen 2 und 3: Hier werden die Farben des Verlaufs definiert. Die *stop offset*-Werte beziehen sich auf den in Zeile 1 definierten Bereich und geben an, wo der Farbverlauf dort beginnen und enden soll. In diesem Beispiel findet der Farbverlauf zwischen 0% und 100%, also im gesamten definierten Bereich, statt.

Zeile 4: Hier wird die Definition geschlossen.

```
1 <radialGradient id="verlauf_rad" cx="70%" cy="70%" fx="20%" fy="
      20%" r="80%">
2 <stop offset="0%" stop-color="white" />
3 <stop offset="100%" stop-color="black" />
4 </radialGradient>
```

Listing 4.10: SVG-Code: Definition eines radialen Farbverlaufes

Erklärung zu Listing 4.10:

Zeile 1: Die Definition des radialen Farbverlaufs wird durch *radialGradient* eingeleitet. Darauf folgt wieder die Festlegung einer ID. Die Attribute *cx* und *cy* beschreiben den Kreismittelpunkt relativ zur referenzierenden Form. *r* ist der Radius des Kreises, innerhalb dem der Farbverlauf dargestellt wird. *fx* und *fy* bezeichnen die Stelle innerhalb des Kreises, an der die zugewiesene Farbe die höchste Intensität besitzt. Werden diese beiden Attribute nicht angegeben, so ist ihr Wert automatisch 50% bzw 0,5. Damit befindet sich die genannte Stelle genau in der Kreismitte.

Zeilen 2 und 3: Die hier definierten Attribute werden analog zu den Zeilen 2 und 3 aus Listing 4.9 benutzt.

Zeile 4: Hier wird die Definition geschlossen.

Nach der Definition der Farbverläufe können Formen mit Hilfe des *fill*-Attributes mit ihnen gefüllt werden. Listing 4.11 erzeugt ein Rechteck und füllt es mit dem in Listing 4.9 definierten Farbverlauf. Listing 4.12 weist einem Kreis den in Listing 4.10 erstellten Farbverlauf zu. Die resultierenden Konstrukte sind in vergrößerter Form in Abbildung 4.5 zu sehen.

```
<rect x="100" y="100" width="100" height="100" fill="url(#
    verlauf_lin)" stroke="black" />
```

Listing 4.11: SVG-Code: Füllen eines Rechtecks mit einem selbst definierten Farbverlauf

```
<circle cx="300" cy="150" r="50" fill="url(#verlauf_rad)" stroke="
    black" />
```

Listing 4.12: SVG-Code: Füllen eines Kreises mit einem selbst definierten Farbverlauf

4.3.2 Beschriftung der Knoten

Da ein Netz ohne Beschriftung seiner Knoten keinerlei verwertbare Informationen beinhaltet, werden die Knoten beschriftet. Die Schrift wird zentral zu seinem Mittelpunkt unterhalb des jeweiligen Knotens platziert. Dazu dient das *text*-Attribut, mit dessen Hilfe man Text in der gewünschten Formatierung an der gewünschten Stelle auf der Zeichenfläche platzieren kann. Listing 4.13 zeigt die Erzeugung einer Stelle mit dem dazugehörigen Text.

Das *text*-Attribut erwartet eine Positionsangabe in Form einer X- und einer Y-Koordinate. Die Y-Koordinate bezieht sich dabei auf die Grundlinie des Textes. Der Bezug der X-Koordinate ist abhängig vom Attribut *text-anchor*. Dieses kann die Werte *start*, *middle* und *end* annehmen. Bei *start* entspricht der linke Rand des Textes der X-Koordinate, bei *end* der rechte Rand und bei *middle* befindet sich genau die Mitte des Textes auf der X-Koordinate. Abbildung 4.6 zeigt dreimal denselben Text mit der X-Koordinate 100. Text $a)$ hat dabei *text-anchor="start"*, Text $b)$ *text-anchor="end"* und Text $c)$ *text-anchor="middle"* erhalten.

Die X-Position des Textes entspricht daher im Modulnetz genau der X-Koordinate des Mittelpunktes des Knotens unter Verwendung von *text-anchor="middle"*. Die Y-Koordinate des Textes ist die Y-Koordinate des Mittelpunktes des Knotens zuzüglich des Radius (bei Stellen) oder der halben Knotenhöhe (bei Transitionen) zuzüglich der Schriftgröße.

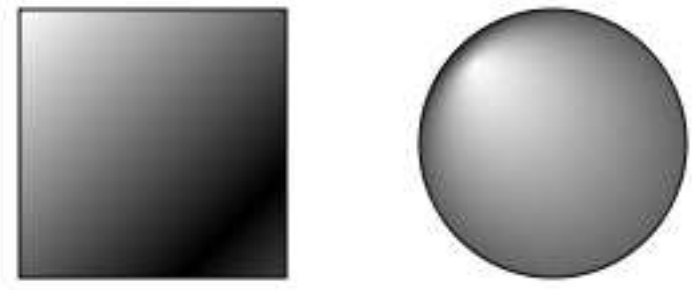

Abbildung 4.5: Rechteck mit linearem und Kreis mit radialem Farbverlauf

```
<rect x="280" y="230" width="40" height="40" stroke="black" />
<text x="300" y="282" font-family="Arial␣Unicode␣MS" font-size="12
    px" text-anchor="middle">Beschreibung</text>
```

Listing 4.13: SVG-Code: Beschriftung einer Stelle

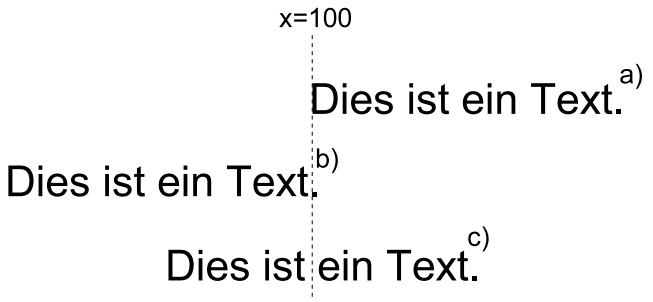

Abbildung 4.6: Textpositionierung mit dem *textAnchor*-Attribut

Zur Variation der Schrift stehen zahlreiche Attribute zur Verfügung, so z.B. die Schriftart (*font-family*), die Schriftgröße (*font-size*), die Schriftdicke (*font-weight*), die Schriftfarbe (*fill*) und weitere, die in World Wide Web Consortium (2003e) detailiert beschrieben sind.

4.3.3 Tooltips

Die Beschriftung der Knoten liefert bereits wichtige Informationen über deren Funktion, jedoch ist es wünschenswert, dass auch Detailinformationen, falls vorhanden, zu den Knoten des Netzes eingesehen werden können. Da solche Informationen aber eine deutlich umfangreichere Textmenge darstellen, würde deren permanente Darstellung im Netz störend wirken. Die Lösung sind *Tooltips*, also Textboxen, die sich in der Nähe eines Knotens öffnen, sobald man mit dem Mauszeiger darüber fährt. Verlässt der Mauszeiger den Knoten, so verschwindet die Textbox automatisch. SVG stellt keine vorgefertigten Werkzeuge zur Anzeige von Tooltips zur Verfügung, daher ist diese Funktionalität mit Hilfe von JavaScript zu implementieren.

Zunächst ist zu definieren, an welcher Stelle der Tooltip eingeblendet werden soll. Dazu wird die Zeichenfläche in vier Quadranten nach Abbildung 4.7 unterteilt.

Für Knoten aus dem ersten Quadrant werden die Tooltips rechts unter dem Knoten angezeigt, im zweiten Quadrant werden sie links unter dem Knoten platziert, der dritte Quadrant positioniert sie rechts über dem zugehörigen Knoten und der vierte links über dem jeweiligen Knoten. Dies hat den Vorteil, dass die Tooltips nicht aus der Zeichenfläche herausragen und somit nur zum Teil angezeigt werden können. Knoten, die eventuell durch die erscheinende Textbox verdeckt werden,

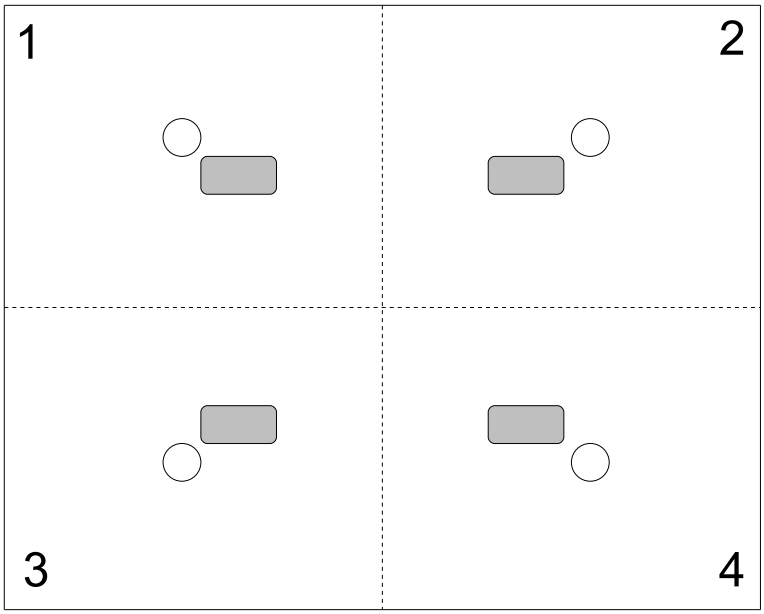

Abbildung 4.7: Aufteilung der Zeichenfläche in Quadranten zur Positionierung der Tooltips

sind trotzdem weiterhin zu erkennen, denn der Hintergrund der Textbox ist in unterschiedlichen Graden transparent einstellbar.

Eine Textbox wird als Rechteck mit abgerundeten Ecken dargestellt und hat einen hellgelben, zu 90% transparenten Hintergrund und eine schwarze, 1,5px dicke und ebenfalls zu 90% transparente Umrandung. Eine solche Textbox wird in Listing 4.14 erstellt. Ihre Breite ergibt sich zunächst aus einer vorher zu bestimmenden maximalen Zeichenanzahl n für eine Zeile Text in der Textbox. Da nicht alle Zeichen dieselbe Breite besitzen, muss die Breite der Textbox geschätzt werden und somit wird sie mit $\frac{1}{2} * f * n$ initialisiert, wobei f der Schriftgröße entspricht. Ist die Zeichenanzahl des Textes kleiner als n, so ergibt sich dir Breite der Box aus $0,6 * f * n$, da der Rahmen bei einer geringen Anzahl Zeichen sonst möglicherweise nicht breit genug ist. Untersteigt die Zeichenanzahl 15, so ist die Rahmenbreite $0,8 * f * n$. Durch mehrfaches Ausprobieren wurden so die besten Ergebnisse erzielt. Ist die Box zu kurz, so ragt der Text rechts aus ihr heraus, was unbedingt zu vermeiden ist. Eine zu lange Textbox, bei der mehr Platz als nötig zwischen der längsten Zeile und ihrem rechten Rand ist, ist akzeptabel, solange der nicht ausgefüllte Bereich nicht unverhältnismäßig groß ist. Eine Methode zur genauen Bestimmung der optimalen Breite der Textbox entsprechend der längsten darin enthaltenen Zeile wurde mit Hilfe von JavaScript implementiert. Dabei hängt es allerdings von dem Programm ab, mit dem die SVG-Datei geöffnet wird, ob der ScriptCode wunschgemäß interpretiert wird. Diese Methode wird später an passenderer Stelle erläutert.

Die Höhe der Textbox ergibt sich aus der benötigten Zeilananzahl l multipliziert mit der Schriftgröße. Dazu werden 6 Pixel addiert, damit der Text einen Abstand zum Rand der Box einhalten

kann.

```
<rect x="430" y="180" rx="5" ry="5" width="250" height="56" style=
    "fill: #FFC; fill-opacity: 0.9; stroke: #000; stroke-width: 1.5
    px; stroke-opacity: 0.9" />
```

Listing 4.14: SVG-Code: Textbox eines Tooltips

Nun muss der Text erstellt werden. Da es keine Funktion zum automatischen Umbrechen der Zeilen gibt, wird hierfür Algorithmus 4.1 implementiert. Er fügt solange Wörter aus dem übergebenen Text zu einer Zeile hinzu, bis ihre maximale Zeichenanzahl überschritten wird. Dann wird mit dem letzten eingefügten Wort eine neue Zeile begonnen.

Daten : Zusammenhängender Text T

Ergebnis : Zeilenweise umgebrochener Text $Lines$

Eingabe : Maximale Zeichenanzahl pro Zeile z

Beginn

  i=0;

  für jedes *Wort* $w \in T$ **tue**

    wenn *Zeichenanzahl* $Lines[i] < z$ **dann**

      $Lines[i] \cup w$;

    sonst

      $i \rightarrow i + 1$;

      $Lines[i] \cup w$;

    $T \rightarrow T - w$;

Ende

Algorithmus 4.1 : Text in Zeilen umberechen

Der Text wird nun in dem Rahmen positioniert. Hierfür muss für jede Zeile ein eigenes *text*-Attribut erstellt werden. Die *text-anchor*-Eigenschaft wird jeweils auf *start* gesetzt und jede Zeile bekommt die X-Koordinate der linken Rahmenseite mit 2px Abstand zugewiesen. Die Y-Position ergibt sich aus $t + n_{Line} * f + d$, wobei t die Y-Koordinate der Oberseite des Rahmens, n_{Line} die Zeilennummer, f die Schriftgröße und d ein Abstand zum Rahmen von 2px ist.

Nachdem der SVG-Code der Textbox und der darin enthaltenen Schriftzeilen erstellt wurde, werden diese Elemente gruppiert. Die Gruppe erhält eine ID, die sich aus den Buchstaben *tt* gefolgt von der X- und Y-Koordinate des referenzierenden Knotens zusammensetzt. Ist kein Text für den Tooltip vorhanden, so ist diese ID 0. Desweiteren wird für die gesamte Gruppe durch die Anwendung von *style="visibility: hidden"* die Sichtbarkeit ausgeschaltet. Ein erzeugter SVG-Code einer solchen Gruppe ist Listing 4.15 zu entnehmen.

Um die Tooltip-Gruppe bei Bewegung des Mauszeigers über den referenzierenden Knoten ein-

```
<g id="tt300250" style="visibility: hidden" >
<rect x="320" y="194" rx="5" ry="5" width="250" height="36" style=
    "fill: #FFC; fill-opacity: 0.9; stroke: #000; stroke-width: 1.5
    px; stroke-opacity: 0.9" />
<text x="322" y="206" font-family="serif" font-weight="normal"
    font-size="10px" text-anchor="start">Zeile 1</text>
<text x="322" y="216" font-family="serif" font-weight="normal"
    font-size="10px" text-anchor="start">Zeile 2</text>
<text x="322" y="226" font-family="serif" font-weight="normal"
    font-size="10px" text-anchor="start">Zeile 3</text>
</g>
```

Listing 4.15: SVG-Code: Tooltip-Gruppe

und auszublenden, werden dem Knoten zunächst zwei *event-handler* zugewiesen. Event-Handler sind Attribute, die bei bestimmten Ereignissen in der Lage sind, Funktionen aufzurufen. Eine Liste aller unterstützten Events ist in World Wide Web Consortium (2003d) zu finden. Die hier benötigten Event-Handler sind *onmouseover* und *onmouseout*. Diese treten in Kraft, sobald der Mauszeiger über den Knoten fährt bzw. sobald der Mauszeiger den Knoten wieder verlässt. Diesen Event-Handlern wird dann die entsprechende Funktion zugeordnet, die beim Eintreten des Events ausgeführt werden soll. Listing 4.16 zeigt die Einbindung der Event-Handler in die Beschreibung einer Transition.

```
<rect x="680" y="130" width="40" height="40" fill="url(#verlauf)"
    stroke="black" onmouseover="showTooltip('tt700150');"
    onmouseout="hideTooltip('tt700150');" />
```

Listing 4.16: SVG-Code: Transition mit Event-Handler

Die referenzierten Funktionen sind hier *showTooltip* und *hideTooltip*, welche als Parameter die ID der zur Transition gehörigen Tooltip-Gruppe erhalten. Die beiden Scripte sind in den Listings 4.17 und 4.18 dargestellt. Zunächst wird eine Referenz auf die Tooltip-Gruppe erstellt durch *document.getElementById(ttID)*. Danach wird mit *setAttribute* dem Attribut, welches durch den ersten Parameter angegeben wird, der Wert des zweiten Parameters zugewiesen. In diesem Fall wird dem Attribut *style* entweder der Wert *visibility: visible* oder *visibility: hidden* zugewiesen, je nachdem ob die Tooltip-Gruppe ein- oder ausgeblendet werden soll.

Beide JavaScript-Funktionen werden in den in Abschnitt 4.1.2 erläuterten Bereich der SVG-Datei geschrieben.

Wie bereits erwähnt, ist es mit JavaScript auch möglich, die Breite der Textbox genau an die längste Zeile des Tooltip-Textes anzupassen, sodass kein unangemessen großer Abstand zwischen

```
function showTooltip(ttID){
        if(ttID != 0){
                var tooltip = document.getElementById(ttID);
                tooltip.setAttribute("style", "visibility: visible
                    ");
        }
}
```

Listing 4.17: JavaScript-Code: Einblenden eines Tooltips

```
function hideTooltip(ttID){
        if(ttID != 0){
                var tooltip = document.getElementById(ttID);
                tooltip.setAttribute("style", "visibility: hidden"
                    );
        }
}
```

Listing 4.18: JavaScript-Code: Ausblenden eines Tooltips

dem Text und dem rechten Rand der Box entsteht. Zu diesem Zweck wird zunächst der Textrahmen sowie jede Zeile des Textes mit einer ID ausgestattet. Der Rahmen erhält dabei die ID der Tooltip-Gruppe, zu der er gehört, mit vorangestelltem r. Die Zeilen erhalten die ID ihrer Tooltip-Gruppe gefolgt von der Zeilennummer innerhalb dieser Gruppe. Die Zeilennummer geht dabei von 0 bis $z-1$, wobei z die Anzahl der Zeilen ist. Der Tooltip-Gruppe wird der Event-Handler *onload* zugeordnet, der in Kraft tritt, sobald alle Elemente der Gruppe fertig geladen sind. Von ihm wird die Script-Funktion *getTextLength(id, lines)* aufgerufen. *id* wird dabei durch die ID der Tooltip-Gruppe ersetzt und *lines* gibt die Anzahl der Zeilen innerhalb der Gruppe an.

Erklärung zu Listing 4.19:

Zeile 4: Die for-Schleife iteriert über alle Zeilen des Tooltips.

Zeile 5: Die ID der aktuellen Zeile wird erzeugt durch Anhängen des Schleifenzählers i an die ID der Tooltip-Gruppe.

Zeile 6: Der Variablen *line* wird eine Instanz der aktuellen Zeile zugewiesen.

Zeile 7: Der Variablen *length* wird mit der Funktion *getComputedTextLength()* die Länge der aktuellen Zeile in Pixeln zugewiesen.

Zeilen 8 bis 10: Falls die berechnete Zeilenlänge größer ist als die bisher maximale Zeilenlänge, so

```
function getTextLength(ID, lines){
        if(ID != 0){
                var maxLine = 0;
                for(i=0; i<lines; i++){
                        var IDi = ID + i.toString();
                        var line = document.getElementById(IDi);
                        var length = line.getComputedTextLength();
                        if(length > maxLine){
                                maxLine=length;
                        }
                }
                if(maxLine > 0){
                        var IDrect = "r"+ID;
                        var box = document.getElementById(IDrect);
                        box.setAttribute("width", maxLine+5);
                }
        }
}
```

Listing 4.19: JavaScript-Code: Setzen der optimalen Tooltip-Breite

wird diese mit dem neuen Wert aktualisiert.

Zeile 13: Die ID des Tooltip-Rahmens wird erzeugt.

Zeile 14: Der Variablen *box* wird eine Instanz des Tooltip-Rahmens zugewiesen.

Zeile 15: Dem Rahmen wird die Breite der längsten Zeile zugewiesen. Der Abstand der längsten Zeile zu den Seitenrändern des Rahmens beträgt 5px.

Das Script aus Listing 4.19 wird vom Adobe SVG-Viewer problemlos interpretiert. Mozilla Firefox (in der zum Zeitpunkt der Implementation aktuellsten Version 2.0.0.4) dagegen liefert bei Ausführung der Funktion *getComputedTextLength()* als Reaktion auf ein *onload*-Event 0 zurück und somit beträgt die Länge der längsten Zeile ebenfalls 0. Daher wird in Zeile 12 überprüft, ob die längste Zeile einen Wert größer als 0 besitzt und nur im positiven Fall wird die Breite des Tooltip-Rahmens entsprechend angepasst. Im negativen Fall behält der Rahmen die bereits zugewiesene Breite $\frac{1}{2} * f * n$ mit der Schriftgröße f und der maximal möglichen Zeichenanzahl pro Zeile n bei. Der Unterschied in der Darstellung eines Tooltips ist den beiden Screenshots in den Abbildungen 4.8 und 4.9 zu entnehmen.

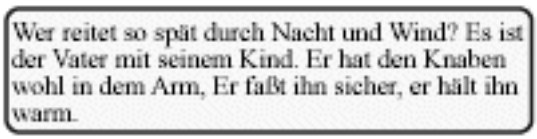

Abbildung 4.8: Darstellung eines Tooltips mit dem Adobe SVG-Viewer

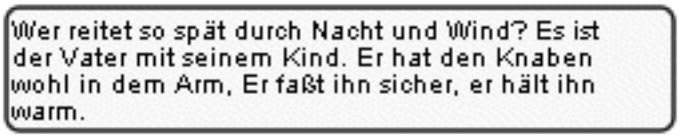

Abbildung 4.9: Darstellung eines Tooltips mit Mozilla Firefox

Kapitel 5

Dokumentation der Implementation

Das in dieser Arbeit entwickelte Layout-Modul besteht aus 13 Klassen mit zahlreichen Attributen und Methoden. In diesem Abschnitt soll die Struktur der Implementation verdeutlicht werden. Dazu wird zunächst auf die Architektur des Layout-Moduls als Teil der Gesamtarchitektur eingegangen. Der darauf folgende Abschnitt ist der Nennung und Erläuterung globaler Konstanten, deren Deklaration für das Rendern des Modul-Netzes sinnvoll ist, gewidmet. Darauf folgt die Dokumentation der Klassen durch ein Klassendiagramm.

5.1 Architektur

Diese Arbeit beschäftigt sich mit der Entwicklung eines Layout-Moduls für die Software *Join*, die die Beschreibung, Manipulation und Visualisierung von Geschäftsprozessmodellen ermöglicht. Um die Integration des Moduls in Join zu erläutern, wird zunächst das für *Join* verwendete Architekturschema *Model-View-Controller* allgemein vorgestellt und danach auf die konkrete Implementation von *Join* nach diesem Schema eingegangen.

5.1.1 Model-View-Controller

Model-View-Controller (MVC) ist ein Designschema zum Entwurf von Software. Nach Gamma u. a. (1995) besteht es aus drei *Objekttypen*:

- *Model* ist das Anwendungsobjekt,
- *View* ist seine Bildschirmpräsentation und

- *Controller* definiert die Art und Weise wie die Benutzerschnittstelle auf Eingaben reagiert.

Die *View*-Komponente repräsentiert dabei zu jeder Zeit den Zustand des *Model*. Ändert sich dieses, so wird *View* benachrichtigt und aktualisiert. Auf diese Weise ist es möglich, verschiedene *View*-Komponenten zu implementieren, die das *Model* auf unterschiedliche Art und Weise sogar gleichzeitig repräsentieren. Gamma u. a. (1995) bezeichnet diese Interaktion, in der die Änderung einer Komponente K automatisch alle betroffenen Komponenten K_i beeinflusst, ohne dass K detaillierte Informationen über ein K_i besitzt, als *Observer*-Designschema.

Desweiteren können verschiedene *View*-Komponenten so kombiniert werden, dass sie für verschiedene spezifische Verwendungszwecke geeignet sind. Die Kombinierbarkeit mehrerer Einheiten zu einem Gesamtmodul, das genau so behandelt und genau dort eingesetzt werden kann wie jedes seiner Teilmodule, heißt nach Gamma u. a. (1995) *Composite*-Designschema.

Durch die Entkopplung der *Controller*-Komponente wird erreicht, dass die Reaktionsstrategie auf Benutzereingaben verändert werden kann, ohne Eingriffe in die beiden anderen Komponenten vornehmen zu müssen. Es exisitieren also für dasselbe Ereignis unterschiedliche Algorithmen um dieses zu behandeln. Diese Algorithmen sind je nach Verwendungszweck beliebig austauschbar. Gamma u. a. (1995) nennt dieses Designschema *Strategy*.

Zusammenfassend kann gesagt werden, dass MVC eine Kombination aus den drei Hauptdesignschemata *Observer*, *Composite* und *Strategy* darstellt.

5.1.2 Implementation von *Join* nach der MVC-Architektur

Das verwendete Architekturschema MVC eignet sich nach Schlossnagle (2004, S. 135) besonders für objektorientierte Webseiten-Programmierung. *Model* ist dabei der “funktionale” Teil des Programms. In ihm werden die Eingabedaten verarbeitet. *View* beschreibt die Ausgabeeinheit des Programms. Die im *Model* aufbereiteten Daten werden hier in der gewünschten Form zurückgegeben. *Controller* sorgt dafür, dass Eingabedaten zur Verarbeitung entgegengenommen und für das *Model* zur Verfügung gestellt werden. Durch diese Trennung erlangt das Programm eine erhöhte Flexibilität, da die drei Komponenten unabhängig voneinander manipuliert werden können. Desweiteren muss genau definiert werden, welche in das Programm einzubringende Funktionalität zu welchem der drei Programmteile gehört. Dadurch wird der Code strukturierter, was nicht zuletzt auch der Wiederverwendbarkeit zu Gute kommt.

Da das Layout-Modul in zwei weitestgehend voneinander unabhängigen Schritten funktioniert, dem *Layouten* und dem *Rendern*, ist es nicht komplett auf der *View*-Ebene implementiert, sondern es wird eine vierte Komponente *ModelLayout* auf der Hierarchieebene des *Model* hinzugefügt. Dort

ist die Logik zum Layouten des Modulnetzes untergebracht, während das Rendern in der *View*-Komponente durchgeführt wird.

Die Idee hinter dieser Hierarchie ist, dass Zugriffe zwischen verschiedenen Hierarchiestufen nur von oben nach unten ermöglicht werden sollen. Die Steuerung des Ablaufes geht von der *Control*-Komponente aus, die damit in der Hierarchie ganz oben angeordnet ist und Zugriff auf alle Komponenten hat. Der Ablauf der Erzeugung eines Modulnetzes aus Geschäftsprozessdefinitionen ist schematisch in Abbildung 5.1 zu sehen. *Control* weist zunächst die Komponenten *Model* (1) und *ModelLayout* (2) an, ihren Dienst zu verrichten. Dabei greift *ModelLayout* auf die in *Model* erzeugten Daten zu (3). Dieser Zugriff ist allerdings eher symbolisch zu verstehen, da die in *Model* erzeugten Daten in Form einer Netzbeschreibung von *Control* an *ModelLayout* übergeben wird. Danach wird *View* zur Ausführung angestoßen (4) und greift auf die in *ModelLayout* erzeugten Objekte zu (5).

Die hier beschriebenen Zugriffsbeziehungen äußern sich im Quellcode objektiv in den *include*-Definitionen im Kopf der PHP-Klasse. Diese ermöglichen es, Methoden aus anderen Klassen zu nutzen und dadurch auf deren Attribute zuzugreifen. Nach Balzert (2000, S. 1050) sind diese *Benutzbarkeitsbeziehungen* oder *Importbeziehungen* fester Bestandteil einer modularen Software-Architektur.

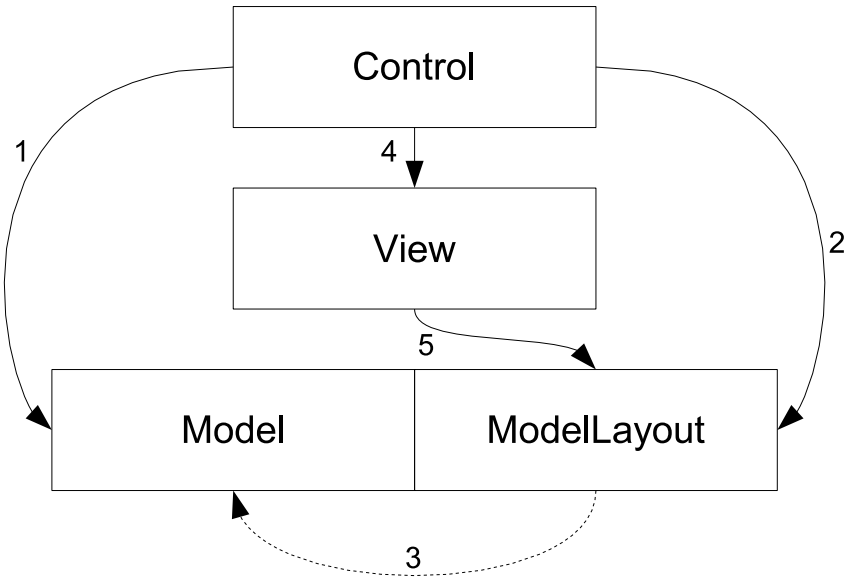

Abbildung 5.1: Programmarchitektur mit Zugriffen

Insgesamt beinhalten *View* und *ModelLayout* 13 Klassen, die in Abschnitt 5.2 genannt und beschrieben werden.

5.2 Klassen, Attribute und Funktionen

In diesem Abschnitt sind die innerhalb des Layout-Moduls implementierten Klassen beschrieben. Da das Gesamtmodul aus zwei Einheiten besteht, *ModelLayout* und *View*, wurde für jedes Modul ein Klassendiagramm erstellt. Die beiden Diagramme (Abbildungen 5.2 und 5.3) bieten einen Überblick über die einzelnen Klassen mit deren Interdependenzen, Attributen und Funktionen. Da insgesamt etwa 120 Methoden und 50 Attribute implementiert wurden, werden nur die wichtigsten mit einer kurzen Erklärung bedacht. Wo immer möglich, wird zu einer Methode auf den entsprechenden in dieser Arbeit erklärten Algorithmus oder das entsprechende Listing verwiesen.

5.2.1 Das ModelLayout-Modul

Das zu diesem Modul gehörende Klassendiagramm ist in Abbildung 5.2 auf Seite 73 zu sehen. Die wichtigste Klasse ist hier LGraph, die den gesamten Layoutgraph repräsentiert. Diese ist eine Spezialisierung der abstrakten Klasse LGraphExtended. Dies dient lediglich der Übersichtlichkeit des Codes, denn würden sämtliche Methoden und Attribute dieser beiden Klassen in einer Klasse implementiert, so wäre diese deutlich zu groß, um effizient damit arbeiten zu können. Deshalb wurden sämtliche Attribute und einige Methoden in die Klasse LGraphExtended ausgelagert, was den Strukturierungsgrad deutlich erhöht. LGraph enthält dabei die “Hauptmethoden” zum Berechnen der X-und Y-Koordinaten und zum Setzen der Dummy-Knoten wie es in Kapitel 3 ab Seite 8 dokumentiert ist. Außerdem sind alle öffentlichen Methoden, die den Zugriff von außen auf die Instanz eines Layout-Netzes erlauben, ebenfalls in LGraph implementiert.

LGraph

Diese Klasse repräsentiert den gesamten Layoutgraphen. Sie übernimmt die Beschreibung eines Modulnetzes aus dem Modul *Model*, erzeugt Instanzen vom Typ *LArc* und *LVertex* und weist den Knoten Koordinaten zu.

__construct(net: Net) Der Konstruktor übernimmt die Modulnetzbeschreibung aus dem Modul *Model* und erzeugt zunächst Instanzen vom Typ *LArc* und *LVertex*, die in den Attributen *arrArcs* bzw. *arrVertices* gespeichert werden. Dann werden nacheinander die Methoden *calcX()*, *setDummies()* und *calcY()* aufgerufen.

calcX() Diese Methode steuert die Berechnung der X-Koordinaten, indem zunächst *calcFlows()* und dann *setFlowsXCor()* aufgerufen wird.

calcFlows() Hier wird das Netz in Flüsse aufgeteilt. Diese werden in dem Attribut *arrFlows* gespeichert. Algorithmus 3.7 auf Seite 21 zeigt diesen Vorgang.

setFlowsXcor() Die X-Koordinaten werden an die Flüsse vergeben (siehe Algorithmus 3.8 auf Seite 24).

setDummies() Dummy-Knoten werden nach Algorithmus 3.10 auf Seite 28 erzeugt.

calcY() Diese Methode steuert die Berechnung und Vergabe der Y-Koordinaten. Dazu werden nacheinander die Methoden *setFlowsYcor()*, *eliminateOverlappingGlobal()*, *balanceY()*, *normalizeY()* und *calcMaxYcor()* ausgeführt.

setFlowsYcor(flow: Array, ycor: Int, place: Int) Hier werden die Y-Koordinaten mit gleichzeitiger Reduzierung der Kantenüberschneidungen an die Flüsse vergeben (siehe Algorithmus 3.12 auf Seite 33).

eliminateOverlappingGlobal() Falls lange Rückflüsse existieren (siehe Abschnitt 3.4.2), so werden diese hier erkannt und deren Knotenüberschneidungen durch Anwendung von Algorithmus 3.13 auf Seite 38 eliminiert.

balanceY() Diese Methode balanciert den Graph aus. Dafür ist Algorithmus 3.15 auf Seite 41 zuständig.

normalizeY() Hier wird durch Algorithmus 3.16 auf Seite 44 sichergestellt, dass die niedrigste Y-Koordinate 1 ist.

calcMaxYcor() Wenn alle Knoten Y-Koordinaten erhalten haben, wird hier die höchste vergebene Y-Koordinate ermittelt und an das Attribut *maxYcor* übergeben.

LGraphExtended

Diese Klasse dient der Code-Strukturierung und stellt eine Ergänzung zu *LGraph* mit allen benötigten Attributen und Hilfsmethoden dar.

net: Net Beinhaltet die Modulnetzbeschreibung vom Typ Net aus dem *Model*-Modul.

arrVertices: Array und arrArcs: Array Beinhalten alle Instanzen vom Typ *LVertex* bzw. *LArc*, die die Knoten und Kanten des Netzes repräsentieren.

startVertex: LVertex und endVertex: LVertex Stellen Start- und Zieltransition des Netzes dar.

maxXcor: Int und maxYcor: Int Höchste X- bzw. Y-Koordinate des Netzes. Diese werden benötigt, damit beim Rendern die benötigte Breite und Höhe der Zeichenfläche ermittelt werden kann.

arrFlows: Array In diesem Array werden alle Flüsse des Netzes gespeichert. Eine Beispielhafte Belegung dieser zweidimensionalen Datenstruktur ist in Tabelle 3.1 auf Seite 22 zu finden.

recursiveCalcLongestPath(mainFlow: Array, v: LVertex, goal: LVertex, vs: Array) Diese Methode dient der Berechnung eines längsten Pfades. Sie ist in Algorithmus 3.5 auf Seite 16 erklärt.

setDummiesInFlows() Die Dummy-Knoten werden hier in das Attribut *arrFlows* integriert (siehe Algorithmus 3.11 auf Seite 30).

globalCrossings(): Int Sämtliche Kantenüberschneidungen des Netzes werden hiermit gezählt.

distance(flow: Array, d: Int) Diese rekursive Methode sorgt dafür, dass alle Flüsse, ausgehend vom übergebenen Fluss, den korrekten Y-Abstand zueinander einhalten (siehe Algorithmus 3.14 auf Seite 38).

getFlowIndex(v: LVertex): Int Zu einem Knoten v wird der Index innerhalb *arrFlows* zurückgeliefert.

getAdjacenceMatrix(): Array Aus dem Attribut net wird die Adjazenzmatrix des Netzes berechnet und in Form eines zweidimensionalen Arrays zurückgegeben.

LArc

Diese Klasse repräsentiert eine LAyout-Kante des Gesamtnetzes. Da ein Modulnetz aus mindestens zwei Knoten bestehen muss, muss auch mindestens eine Kante existieren.

from: LVertex und to: LVertex Anfangs- und Endknoten der Kante.

direction: Int Ist der Wert diese Attributes 1, so zeigt die Kante nach rechts. Bei -1 zeigt sie nach links.

turned: Boolean Dieses Attribut zeigt an, ob die Kante zum Layouten gedreht wurde, damit sie vor dem Rendern wieder in ihre ursprüngliche Richtung gebracht werden kann.

turn() Die Methode verändert den Wert von *turned*.

LVertex

Diese Klasse repräsentiert einen Layout-Knoten des Gesamtnetzes. Ein Modulnetz muss aus mindestens zwei Knoten besetehen, der Start- und der Zieltransition.

oid: Int Jeder Knoten bekommt diese Identifikationsnummer zugewiesen, die ihn eindeutig identifiziert.

label: String Hier wird eine Kurzbeschreibung des Knotens gespeichert. Diese wird beim Rendern unter den Knoten geschrieben.

back:Boolean Dieses Attribut zeigt an, ob der Knoten Teil eines Rückflusses ist.

MainFlowIndex: Int Der Index des Oberflusses dieses Knotens.

nextDummy Falls der Knoten ein Dummy-Knoten ist, wird hier der auf ihn folgende Dummy-Knoten instanziiert, falls er existiert. Bei echten Knoten oder Dummy-Knoten ohne nachfolgenden Dummy-Knoten ist der Wert dieses Attributes *null*.

LCoordinate

Diese Klasse repräsentiert eine Layout-Koordinate. Da jeder Knoten eine Koordinate besitzt, bildet die Klasse eine Generalisierung von *LVertex*, um komfortabler auf die Koordinaten eines Knotens zugreifen zu können.

x: Int, y: Int X- und Y-Koordinate.

setXY(x: Int, y: Int) Diese Methode weist beiden Attributen der Klasse einen Wert zu.

setX(x: Int) und setY(y: Int) Hier können den Attributen der Klasse einzeln Werte zugewiesen werden.

getX(): Int, getY(): Int Rückgabe der X- bzw. Y-Koordinate.

5.2.2 Das View-Modul

In diesem Modul sind alle Klassen enthalten, die die SVG-Generierung, also das Rendern des Graphen wie in Kapitel 4 ab Seite 45 beschrieben, übernehmen. Die Struktur ähnelt dabei derjenigen des *ModelLayout*-Moduls, denn es existiert auch hier eine Hauptklasse *RNet*, die den gesamten Render-Graphen repräsentiert. Knoten werden hier jedoch in Stellen, Transitionen und Dummy-Knoten unterteilt. Die abstrakte Klasse *RenderClass* bildet die Schnittstelle des *View*-Moduls nach außen. Sie initiiert die Erzeugung einer Instanz von *RNet* aus einer Instanz der Klasse *LGraph* und übernimmt das eigentliche Schreiben der SVG-Datei. Sämtliche hier beschriebenen Längen-, Entfernungs- oder Positionsangaben verstehen sich in Pixel-Einheiten.

Abbildung 5.2: Klassendiagramm des ModelLayout-Moduls

RenderClass

RenderClass ist eine abstrakte Klasse, die statische Methoden zum Schreiben einer SVG-Datei zur Darstellung eines Graphen bereitstellt.

_svgFP: Filepointer Diese globale Variable bildet einen internen Zeiger auf die Datei, die mit SVG-Code gefüllt wird.

_svgS(String): String Die Methode wandelt Umlaute und einige Sonderzeichen des übergebenen Strings in Unicode um.

_svg(t: String): String, svgS(t: String): String und svg(t:String): String Alle drei Methoden schreiben den übergebenen Text in die durch *_svgFP* referenzierte Datei.

svgHeader(graph: RNet) Hier wird der Kopf der SVG-Datei erzeugt (siehe Abschnitt 4.1.2 ab Seite 46). Dabei wird die Größe der Zeichenfläche aus graph ausgelesen.

def_svgArrowHead(String) Definition der Pfeilspitze in der übergebenen Farbe nach Listing 4.5 auf Seite 53.

def_[...]Farbverlauf[...](fromColor: String, toColor: String) Hier werden lineare und radiale Farbverlaufsdefinitionen zum Füllen einer Transition oder einer Stelle erzeugt.

skript_ttEinblenden() Der JavaScript-Code zum Einblenden eines Tooltips wird erzeugt (siehe Seite 4.17 auf Seite 63).

skript_ttAusblenden() Der JavaScript-Code zum Ausblenden eines Tooltips wird erzeugt (siehe Seite 4.18 auf Seite 63).

skript_setTTBoxWidth() Diese Methode erzeugt den JavaScript-Code zur Berechnung der optimalen Tooltip-Breite nach Listing 4.19 auf Seite 64.

renderSVGFile(RGraph: RNet) Die korrekte Gliederung der SVG-Datei wird durch diese Methode sichergestellt, indem die Erstellung der einzelnen Bereiche der Datei vom Header bis zum abschließenden Tag von hier aus initiiert wird.

showRenderedSVGFile(graph: LGraph) Diese Methode wird von außen aufgerufen mit einer Instanz des Typs *LGraph* als Parameter. Daraufhin erzeugt die Methode eine Instanz des Typs *RNet* aus diesem Parameter und stößt die Erzeugung der SVG-Datei daraus durch Aufruf der Funktion *renderSVGFile(RGraph: RNet)* an.

RNet

Die Klasse repräsentiert das gesamte gerenderte Modul-Netz und stellt somit Methoden bereit, die den SVG-Code desselben liefern.

width:Int und height: Int Stellen Breite und Höhe der benötigten Zeichenfläche zur kompletten Darstellung des Modul-Netzes dar.

arrRVertices: Array und arrRArcs: Array In diesen Attributen werden Referenzen auf alle Instanzen von *RVertex* bzw. *RArc* gespeichert.

getSVGCode(): String Diese Methode fragt den SVG-Code aller Knoten und Kanten ab und liefert diesen zurück.

getSVGTooltips(): String Die Methode liefert den SVG-Code sämtlicher Tooltip-Gruppen zurück.

RArc

RArc ist eine Klasse zur Repräsentation einer Render-Kante des Netzes. Ein Render-Netz beinhaltet mindestens eine Render-Kante.

layoutArc: LArc Layout-Kante, die durch diese Render-Kante repräsentiert wird.

startVertex: RVertex und endVertex: RVertex Sind Start- und Endknoten der Kante.

startPoint: RCoordinate und endPoint: RCoordinate Stellen Koordinaten des Start- und Endpunktes der Kante auf der Zeichenfläche dar.

getSVGCode(): String Liefert den SVG-Code der Kante nach Art von Listing 4.6 auf Seite 53 zurück.

RVertex

RVertex ist eine abstrakte Klasse zur Repräsentation eines Render-Knotens des Netzes. Von ihr wird keine Instanz erzeugt, da sie die Klassen *RTransition*, *RPlace* und *RDummy* generalisiert. Ein Render-Netz besteht aus mindestens zwei Render-Knoten.

layoutVertex: LVertex Layout-Knoten, der durch diesen Render-Knoten repräsentiert wird.

tooltipText: String Text, der im Tooltip angezeigt werden soll.

ttTextLines: Array Array, das den in Zeilen umgebrochenen Text des Tooltips enthält.

ttLines: Int Anzahl der Textzeilen des Tooltips.

tooltipID: String Einzigartige Zeichenkette zur Referenzierung des Tooltips in der SVG-Datei.

calcTooltipLines(): Array Der Text aus tooltipText wird nach Algorithmus 4.1 auf Seite 61 in Zeilen umgebrochen und das Ergebnis in *ttTextLines* gespeichert.

getSVGTooltipText(left: Int, top: Int): String Diese Methode liefert den SVG-Code des "Tooltip-gerechten" Textes zurück. Sie wird nur von den generalisierten Klassen aufgerufen, um den gruppierten SVG-Code des gesamten Tooltips zu erzeugen. Daher ist sie *protected*.

getConnectionPoint(fromX: Int, fromY: Int): RCoordinate Hier findet die Berechnung des Schnittpunktes der Geraden von der übergebenen Koordinate zum Mittelpunkt dieses Knotens mit der Methode aus Abschnitt 4.2.4 ab Seite 49 statt. Im Falle eines Dummy-Knotens werden seine Koordinaten zurückgegeben.

RCoordinate

Diese Klasse repräsentiert eine Render-Koordinate. Da jeder Knoten eine Koordinate besitzt, bildet die Klasse eine Generalisierung von *RVertex*, um komfortabler auf die Koordinaten eines Knotens zugreifen zu können. Methoden zum Setzen der Koordinaten implementiert die Klasse nicht, da der Konstruktor der generalisierten Klasse die Koordinaten festlegt und diese danach nicht mehr geändert werden.

x: Int und y: Int X- und Y-Koordinate in Pixeln.

getX(): Int und getY(): Int Rückgabe der Koordinaten.

RTransition

RTransition repräsentiert eine Transition des Modul-Netzes und ist eine Spezialisierung von *RVertex*.

sideXlength: Int und sideYlength:Int Breite und Höhe des Rechtecks, das die Transition darstellen soll.

left, right, top, bottom: Int X-Koordinaten der linken und rechten Seite sowie Y-Koordinaten der Unter- und Oberseite des Rechtecks.

getSVGCode: String Liefert den SVG-Code für das Rechteck der Transition und den darunter zu platzierenden Text. Die entsprechenden Vorgehensweisen finden sich in den Abschnitten 4.2.2, 4.3.1 und 4.3.2.

createTooltip(width: Int, height: Int): String Diese Methode liefert den SVG-Code der gesamten Tooltip-Gruppe zurück, dessen Erstellung in Abschnitt 4.3.3 ab Seite 59 beschrieben ist.

RPlace

Diese Klasse repräsentiert eine Stelle des Modul-Netzes und ist eine Spezialisierung von *RVertex*.

radius: Int Radius der Stelle.

getSVGCode(): String Liefert den SVG-Code für den Kreis der Transition und den darunter zu platzierenden Text. Die entsprechenden Vorgehensweisen finden sich in den Abschnitten 4.2.1, 4.3.1 und 4.3.2.

createTooltip(width: Int, height: Int): String Diese Methode liefert den SVG-Code der gesamten Tooltip-Gruppe zurück, dessen Erstellung in Abschnitt 4.3.3 ab Seite 59 beschrieben ist.

RDummy

RDummy ist eine Klasse zur Repräsentation eines Render-Dummy-Knotens. Sie ist eine Spezialisierung von *RVertex*. Sie beinhaltet lediglich einen Konstruktor, der ihre Koordinaten initialisiert.

5.3 Globale Konstanten

Das Aussehen der SVG-Grafik kann durch zahlreiche Attribute variiert werden. Farben, Abstände, Schrifteinstellungen und Größen sollen vom Betrachter so verändert werden können, dass ein seinem persönlichen Geschmack entsprechendes Gesamtbild entsteht. Ein solches Attribut taucht meist an mehreren Stellen im Quelltext auf. Damit es bei seiner Veränderung nicht an jeder Stelle einzeln manipuliert werden muss, existiert eine Konfigurationsdatei, in der man Konstanten für die Werte solcher Attribute zentral definieren kann. Diese Attribute werden fortan nicht mehr mit Werten belegt, sondern mit der entsprechenden Konstanten. Verändert man nun den Wert der Konstanten in der Konfigurationsdatei, so wird der Wert des zugehörigen Attributes automatisch im gesamten Quelltext, also global, verändert.

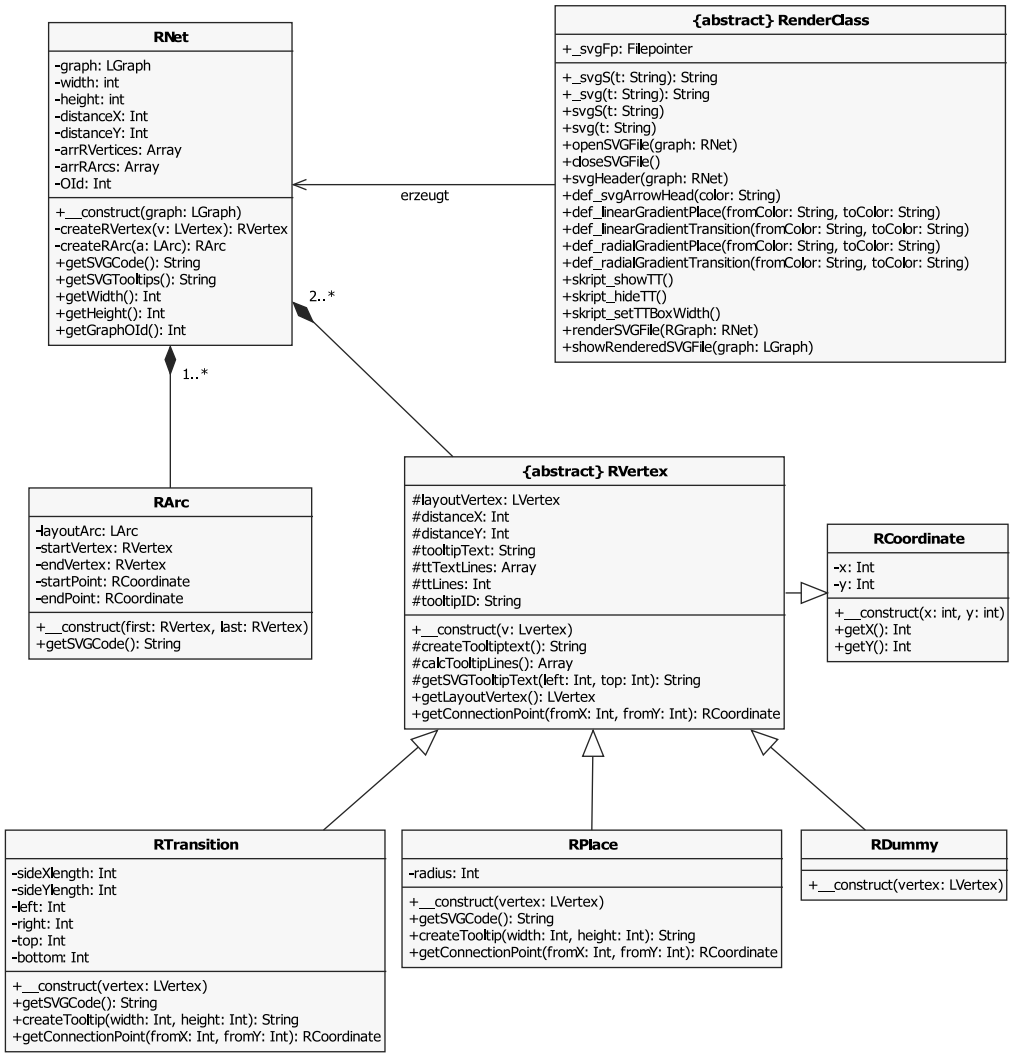

Abbildung 5.3: Klassendiagramm des View-Moduls

Es folgt eine Tabelle von Konstanten, die im Zuge der Erstellung dieser Arbeit in der Konfigurationsdatei definiert wurden. Diese ist nach Einsatzgebieten gegliedert und zu dem Namen der Konstante wird jeweils eine Kurzbeschreibung geliefert. Größen- oder Längenangaben können stets mit oder ohne Einheit angegeben werden. Ohne Einheit wird standardmäßig die Einheit *px* (Pixel) oder die gültige *user-Einheit* (siehe Abschnitt 4.1.3) gesetzt.

Name	Kurzbeschreibung
Knotenabstände	
_VERTEXDISTANCEX	Horizontaler Abstand der Knotenmittelpunkte zueinander
_VERTEXDISTANCEX	Vertikaler Abstand der Knotenmittelpunkte zueinander
Knotengröße	
_TRANSITIONLENGTHX	Breite einer Transition
_TRANSITIONLENGTHX	Höhe einer Transition
_RADIUS	Radius einer Stelle
Farbkonfiguration der Knoten	
_COLOR1TRANSITION	Erste Verlaufsfarbe einer Transition
_COLOR2TRANSITION	Zweite Verlaufsfarbe einer Transition
_COLOR1PLACE	Erste Verlaufsfarbe einer Stelle
_COLOR2PLACE	Zweite Verlaufsfarbe einer Stelle
_GRADIENTTRANSITION	Farbverlaufstyp einer Transition: [*rad*, *lin*]
_GRADIENTPLACE	Farbverlaufstyp einer Stelle: [*rad*, *lin*]
Farbkonfiguration der Kanten	
_ARCCOLOR	Farbe der Kanten
_ARROWHEADCOLOR	Farbe der Pfeilspitzen
Schrifteinstellungen	
_FONTSIZE	Schriftgrad
_FONTFAMILY	Schriftart
_FONTWEIGHT	Schriftdicke
Tooltipkonfiguration	
_TOOLTIPWIDTH	Maximale Zeichenanzahl einer Tooltipzeile
_FILLTRANSPARENCY	Transparenz des Tooltiphintergrundes: [0..1]
_STROKETRANSPARENCY	Transparenz des Tooltiprahmens: [0..1]
_TTFONTSIZE	Schriftgrad der Tooltipschrift
_TTFONTFAMILY	Schriftart der Tooltipschrift
_TTFONTWEIGHT	Schriftdicke der Tooltipschrift

Tabelle 5.1: Globale Konstanten

Kapitel 6

Anwendungsbeispiele

In diesem Kapitel werden Screenshots von einigen generierten Modulnetzen präsentiert. Die Definitionen der ersten beiden Netze in den Abbildungen 6.1 und 6.2 sind dabei Simon und Dehnert (2004) und Simon und Olbrich (2005) entnommen worden. Zwei weitere Netze wurden von mir erstellt, um einige layouttechnische Varianten präsentieren zu können. Wo es möglich ist, wird auf relevante Stellen innerhalb der Dokumentation verwiesen.

Das Layout des Netzes in Abbildung 6.1 bietet eine Vielzahl an Besonderheiten. Der Hauptfluss von der Starttransition bis zur am weitesten rechts liegenden (unbeschrifteten) Zieltransition ist leicht zu erkennen. Der längste Nebenfluss ist direkt von der Starttransition aus erreichbar und beinhaltet die Transitionen *register fee paid* und *synchronize*, sowie eine integrierte lange Kante bestehend aus zwei Dummy-Knoten (siehe Algorithmus 3.11 auf Seite 30). Diesem Nebenfluss entspringt an der ersten Stelle eine weitere lange Kante, die aber als eigenständiger Fluss behandelt wird, wie es am Ende von Abschnitt 3.3 auf Seite 29 beschrieben ist. Desweiteren beinhaltet der Nebenfluss einen kleinen Rückfluss bestehend aus der Transition *rollback fee*, die zu zwei linksgerichteten Kanten inzident ist. Die Transition wurde mit ihrem Oberfluss ausbalanciert (siehe Abschnitt 3.4.3 ab Seite 39).

Die Transition *formal reg. OK* bildet einen Unterfluss des Hauptflusses, die darauf folgende Stelle ist ein Unterfluss dieser Transition. Das Vorgehen für den Fall, dass ein Knoten zu Knoten aus zwei verschiedenen Flüssen adjazent ist, ist auf Seite 20 beschrieben.

Der Haupfluss des Netzes in Abbildung 6.2 wird durch die Knotenfolge von *check credit* bis *archive* gebildet. Er besitzt zwei Unterflüsse. In beiden ist eine lange Kante integriert (siehe Seite 29). Der Unterfluss unterhalb des Hauptflusses besitzt selbst einen Unterfluss mit zwei Stellen und der Transition *notify cancel* sowie einer langen Kante. Er ist mit seinem Oberfluss ausbalanciert (siehe Abschnitt 3.4.3 ab Seite 39).

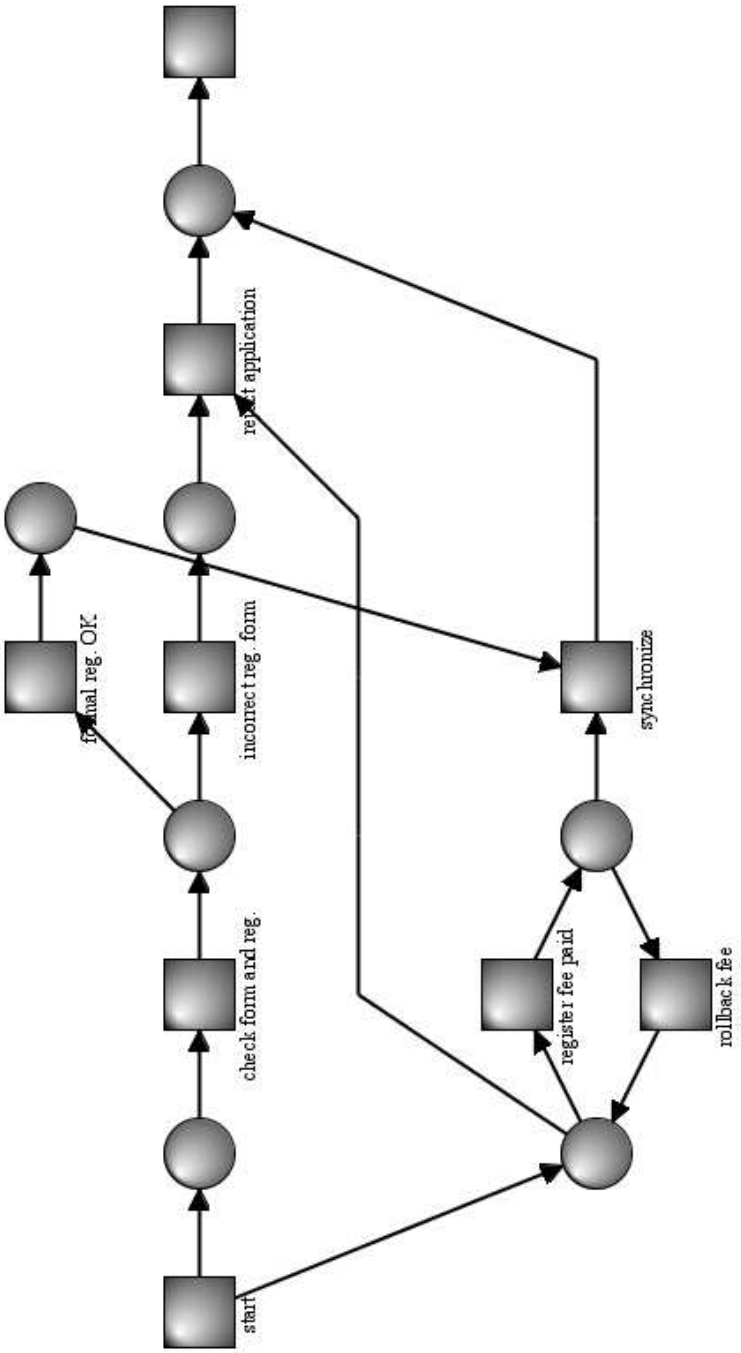

Abbildung 6.1: "Rollback on fees" (Simon und Olbrich, 2005)

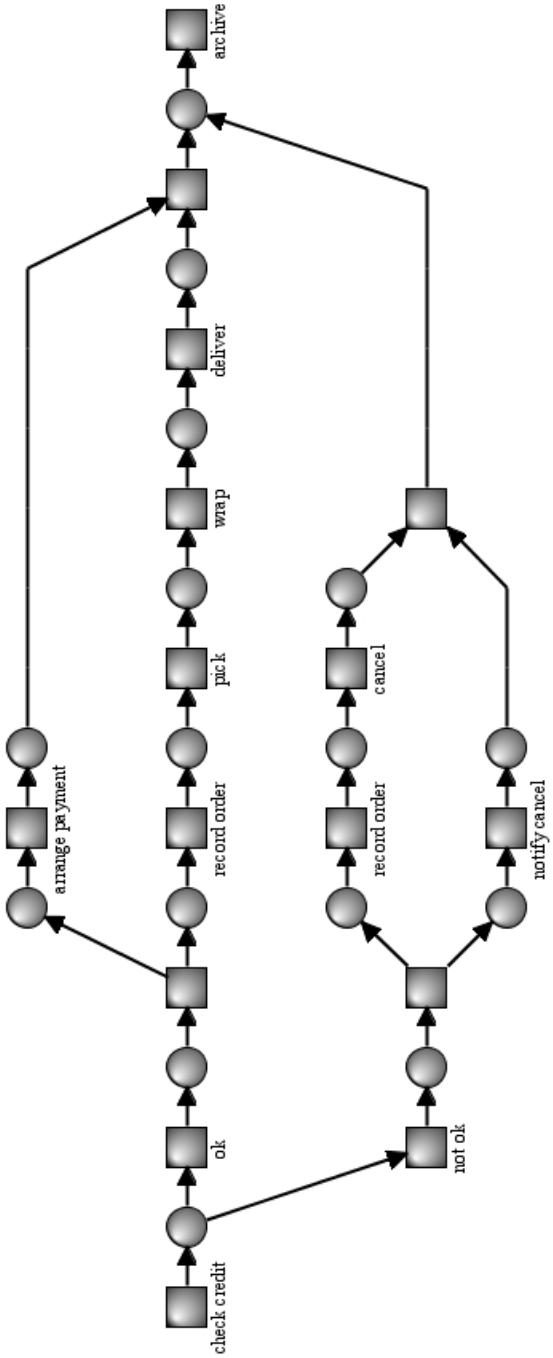

Abbildung 6.2: "Sound WF-net *Handling an incoming order*" (Simon und Dehnert, 2004)

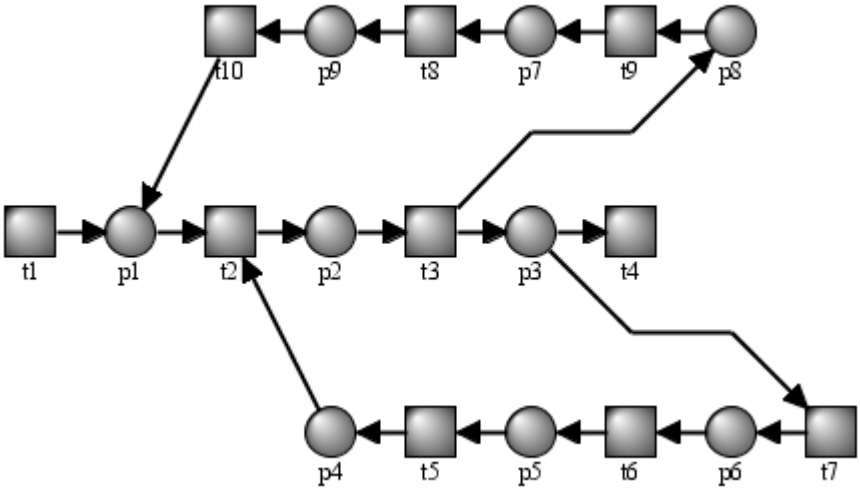

Abbildung 6.3: Hauptfluss mit zwei langen Rückflüssen

Das Netz in Abbildung 6.3 beinhaltet neben dem mittig platzierten Hauptfluss von *t1* bis *t4* zwei lange Rückflüsse als Nebenflüsse (siehe Abschnitt 3.4.2 ab Seite 37). Hier ist insbesondere zu sehen, dass der Mindestabstand von 1 auch zwischen Dummy-Knoten und "echten" Knoten eingehalten wird. Außerdem ist es in diesem Sonderfall möglich, dass die Zieltransition nicht mehr die höchste X-Koordinate des Netzes besitzt.

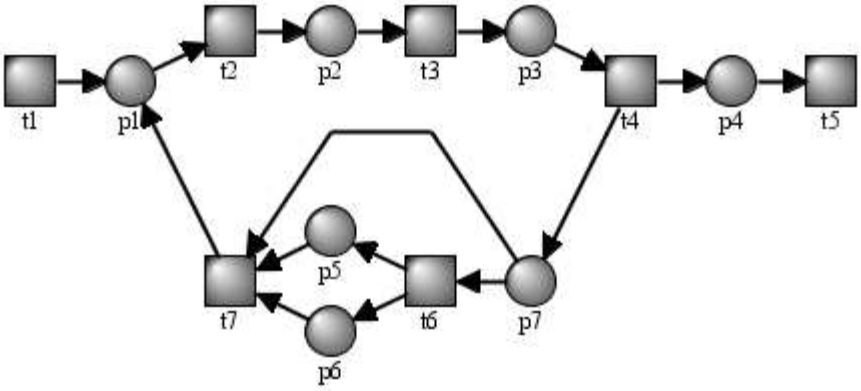

Abbildung 6.4: Netz mit vier Flüssen

In Abbildung 6.4 ist ein Netz mit seinem Hauptfluss von *t1* bis *t5* zu sehen. Der Hauptfluss hat hier einen direkten linksgerichteten Unterfluss. Durch die Ausbalancierung (siehe Abschnitt 3.4.3) der Stellen *p5* und *p6* ist ohne weitere Informationen nicht nachzuvollziehen, welche der beiden

Stellen Teil dieses Unterflusses ist und welche für sich einen Unterfluss desselben bildet. Ein zweiter Unterfluss wird durch eine lange linksgerichtete Kante repräsentiert.

Weitere automatisch erzeugte Netze sind auf der beigelegten CD in Form von SVG-Dateien im Verzeichnis *\SVG-Beispielnetze* zu finden. Diese können durch Ausführung der Sourcen in Verzeichnis *\Join* auch neu generiert werden. Eine kleine Eingabemaske bietet die Möglichkeit, ein Netz aus einer Liste auszuwählen und optional textuelle Informationen aus dem Model- und/oder dem ModelLayout-Modul zu dem Netz anzeigen zu lassen. Durch Veränderung der globalen Konstanten der Datei *config.php* im Verzeichnis *\Join* kann Einfluss auf das Netzlayout genommen werden.

Kapitel 7

Zusammenfassung und Ausblick

Geschäftsprozessmodelle sind von zentraler Bedeutung, wenn es um das Verständnis von Unternehmen geht. Dieses Verständnis wird erleichtert durch die Modellierung mittels Modulnetzen, einer speziellen Form von Petri-Netzen. Petri-Netze werden charakterisiert durch eine schlichte grafische Notation und eine formale Basis als Grundlage für deren Simulation und Analyse (Philippi, 1999).

Join ist eine Software zur Geschäftsprozessmodellierung mittels Petri-Netzen, für die im Rahmen dieser Arbeit ein Layout-Modul entwickelt wurde. Dieses Modul ist in der Lage, die formale Beschreibung eines Modulnetzes, bestehend aus Knoten, Kanten, deren Inzidenzbeziehungen sowie textuellen Informationen, in ein grafisches Layout umzuwandeln.

Dazu muss zunächst die Position jedes Knotens auf der Zeichenfläche bestimmt werden, die sich aus der Position der jeweils adjazenten Knoten ergibt. Dies hat sich zu einem ungeahnt großen Problem entwickelt, denn die Methoden zum Zeichnen von Grafen wie sie Battista u. a. (1999) vorstellt, sind zur Lösung dieses Problems nur bedingt geeignet. Das hierarchische Layouten eines Graphen konzentriert sich dort auf Designkriterien wie möglichst wenige Kantenüberschneidungen und möglichst kurze Kanten. Geschäftsprozessmodelle bestehen aus *Prozessen* und diese sind zusammengesetzt aus *Elementarprozessen*, welche durch die Knoten des Netzes repräsentiert werden. Daher ist es wünschenswert, dass das Layout diese Prozesse in Form aneinandergereihter Knoten wiederspiegelt. Dies hat jedoch zur Folge, dass lange Kanten und einige Kantenüberschneidungen akzeptiert werden müssen.

Daher wurde ein völlig neuer Ansatz entwickelt, der den Graph in *Flüsse*, also zusammenhängende Knotenfolgen, aufteilt und das gesamte Layout an diesen Flüssen ausrichtet. X- und Y-Koordinaten werden nicht knotenweise, sondern flussweise, vergeben, was auch einen geringeren Rechenaufwand mit sich bringt. Insgesamt erfolgt das Layouten in drei Schritten:

- Berechnung der X-Koordinaten und Unterteilung der Knotenmenge in Flüsse
- Setzen der Dummy-Knoten und Integration der Dummy-Knoten in die Flüsse
- Setzen der Y-Koordinaten und Ausbalancierung des Graphen

Nach dem Layouten folgt das Rendern, in dem der Graph in Form einer SVG-Grafik optisch aufbereitet generiert wird. Stellen, Transitionen, Kanten, Farben und Text werden so auf der Zeichenfläche platziert, dass ein ästhetischer Gesamteindruck entsteht. Da die Ästhetik jedoch immer im Auge des Betrachters liegt, wurden einige globale Konstanten definiert, die die Möglichkeit bieten, das Netz hinsichtlich der individuellen Vorstellungen zu gestalten. Beim Einbringen von Interaktivität in die Grafik durch Anzeige von Tooltips wurde deutlich, dass nicht jede Software zur Darstellung einer SVG-Grafik diese auch auf dieselbe Art und Weise interpretiert. Daher wurden die Tooltips auf zwei verschiedene Weisen programmiert: Eine exakte Lösung, die nicht von allen Darstellungsprogrammen interpretiert wird, und eine Backup-Lösung, die immer funktioniert und ein passables aber dennoch nicht optimales Ergebnis liefert.

Die Integration des hier entwickelten Layoutmoduls in die Software *Join* konnte noch nicht komplett abgeschlossen werden, daher sind später einige Anpassungen des Layout-Moduls nötig. Modulnetzbeschreibungen werden später automatisch von der Software entgegengenommen und bearbeitet. Knoten, Kanten und *Labels*, also die Kurzbeschreibungen der Knoten, werden momentan manuell generiert. Der Tooltip-Text jedes Knotens kann dabei noch nicht definiert werden. Zur Manipulation des Tooltip-Textes existiert in der Klasse *RVertex* die Funktion *createTooltiptext()*, die das Setzen des jeweiligen Tooltip-Textes ermöglicht. Die aktuelle Version des Programms erzeugt dort als Tooltip-Text das Label des zugehörigen Knotens und muss später angepasst werden. Die definierten globalen Konstanten können in die noch zu entwerfende grafische Benutzeroberfläche integriert werden. Somit sind sie komfortabel per Mausklick bedienbar. Schriftarten und -größen können dann beispielsweise per Dropdown-Menü eingestellt werden, wie es in vielen Programmen zur Textverarbeitung praktiziert wird, was der intuitiven Bedienbarkeit zu Gute kommt.

Mit Hilfe des Entwickelten Layoutmoduls ist es durch einige Modifikationen auch realisierbar, *Ereignisgesteuerte Prozessketten (EPKs)*, wie sie beispielsweise in Scheer (2000) verwendet werden, zu visualisieren. Die Prozessmodellierung mit Hilfe von EPKs benutzt drei Hauptkonstrukte: *Informationsobjekt*, *Funktion* und *Ereignis* (Keller u. a., 1992). Desweiteren gibt es die logischen Operatoren *und* ($\wedge$), *oder* ($\vee$) und *exklusives oder* (XOR), die Beziehungen zwischen den Konstrukten darstellen. Das Layouten dieser Knoten kann mit den hier vorgestellten Algorithmen nur bedingt vollzogen werden, da ein EPK-Modell mehrere Quellen und Senken enthalten kann, wogegen ein Modulnetz genau eine Quelle und eine Senke in Form von Start- und Zieltransition besitzt. Beim Rendern sind ebenfalls einige Anpassungen vorzunehmen. Die Klasse *RVertex* kann wieder als Generalisierung der verschiedenen Knotentypen dienen. Sie spezialisiert dann vier Unterklassen für

die Knotentypen *Informationsobjekt*, *Ereignis*, *Funktion*, *logischer Operator* und *Dummy*. Informationsobjekte stellen nach Keller u. a. (1992, S. 9) Mengen realer oder abstrakter Dinge dar. Daher könnte der Knotentyp *Informationsobjekt* eine Generalisierung weiterer Untertypen darstellen.

Das in dieser Arbeit entwickelte Render-Modul ermöglicht die Darstellung des Gesamtnetzes in seinem initialen Zustand. Petri-Netze bieten die Möglichkeit, einen Prozessablauf dynamisch durch Belegung von Stellen mit *Marken* oder *Tokens* und Aktivierung von Transitionen, wie in Kapitel 2.2 ab Seite 5 beschrieben, nachzubilden. Um diese Art der Visualisierung auch mit *Join* zu ermöglichen, kann eine weitere Netz-Sicht in Form einer zweiten Render-Klasse integriert werden. Diese stößt ebenfalls die Erzeugung einer Instanz von *RNet* an und kann damit auf alle darin enthaltenen Attribute und Methoden zurückgreifen. Sie unterscheidet sich von der ersten Render-Klasse darin, dass sie Transitionen aktiviert oder nicht aktiviert darstellen und Stellen mit einer beliebigen Anzahl an Tokens besetzen kann. Die Integration einer dritten Netz-Sicht, die bestimmte Prozesspfade hervorheben kann, ist ebenfalls realisierbar.

Literaturverzeichnis

[Balzert 2000] BALZERT, H.: *Lehrbuch der Software-Technik: Software-Entwicklung*. Heidelberg, Berlin : Spektrum, Akad., Verl., 2000

[Battista u. a. 1999] BATTISTA, G. D. ; EADES, P. ; TAMASSIA, R. ; TOLLIS, I. G.: *Graph Drawing - Algorithms for the Visualization of Graphs*. New Jersey : Prentice-Hall, Inc., 1999

[Baumgarten 1990] BAUMGARTEN, B.: *Petri-Netze: Grundlagen und Anwendungen*. Mannheim, Wien, Zürich : BI - Wissenschaftsverlag, 1990

[Böhm 2000] BÖHM, M.: *Entwicklung von Workflow-Typen*. Berlin, Heidelberg : Springer-Verlag, 2000 (Informationstechnologien für die Praxis)

[Gamma u. a. 1995] GAMMA, E. ; HELM, R. ; JOHNSON, R. ; VLISSIDES, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston : Addison-Wesley, 1995

[Ganser u. a. 1993] GANSER, E. R. ; KOUTSOFIOS, E. ; NORTH, S .. ; VO, K. P.: A Technique for Drawing Directed Graphs. In: *IEEE Transactions On Software Engineering* 19 (1993), S. 214–230

[Garey und Johnson 1979] GAREY, M. R. ; JOHNSON, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York : W. H. Freeman, 1979

[Gumm und Sommer 2002] GUMM, H. P. ; SOMMER, M.: *Einführung in die Informatik*. München, Wien : Oldenbourg, 2002

[Jablonski und Bussler 1996] JABLONSKI, S. ; BUSSLER, C.: *Workflow Management - Modeling Concepts, Architecture and Implementation*. London : International Thomson Computer Press, 1996

[Keller u. a. 1992] KELLER, G. ; NÜTTGENS, M. ; SCHEER, A.-W.: Semantische Prozeßmodellierung auf der Basis Ereignisgesteuerter Prozeßketten (EPK) / Universität des Saarlandes. Institut für Wirtschaftsinformatik, Saarbrücken, 1992 (89). – Forschungsbericht

[Krumke und Noltemeier 2005] KRUMKE, S. O. ; NOLTEMEIER, H.: *Graphentheoretische Konzepte und Algorithmen*. Wiesbaden : Teubner Verlag, 2005

[Mielke 2002] MIELKE, C.: *Geschäftsprozesse - UML-Modellierung und Anwendungs-Generierung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2002

[Philippi 1999] PHILIPPI, S.: *Synthese von Petri-Netzen und objektorientierten Konzepten*. Koblenz : Fölbach, 1999 (Koblenzer Schriften zur Informatik)

[Scheer 1999] SCHEER, A.-W.: *ARIS - Business Process Frameworks*. 3rd. Berlin : Springer, 1999

[Scheer 2000] SCHEER, A.-W.: *ARIS - Business Process Modeling, 3rd ed.* Berlin : Springer, 2000

[Schlossnagle 2004] SCHLOSSNAGLE, G.: *Professionelle PHP 5-Programmierung - Entwicklerleitfaden für große Webprojekte*. München : Addison-Wesley, 2004

[Simon und Dehnert 2004] SIMON, C. ; DEHNERT, J.: From Business Process Fragments to Workflow Definitions. In: FELTZ, F. (Hrsg.) ; OBERWEIS, A. (Hrsg.) ; OTJACQUES, B. (Hrsg.): *EMISA 2004 - Informationssysteme im E-Business und E-Government*. Luxemburg, 2004 (Gesellschaft für Informatik, Lecture Notes in Informatics P-56), S. 95–106

[Simon u. a. 2006] SIMON, C. ; FREIHEIT, J. ; OLBRICHT, S.: Using BPEL Processes defined by Event-driven Process Chains. In: *5. GI-Workshop - EPK 2006 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*. Wien, 2006, S. 121–135

[Simon und Olbrich 2005] SIMON, C. ; OLBRICH, S.: The Influence of Legal Constraints on Business Process Modeling. In: IRANI, Z. (Hrsg.) ; ELLIMAN, T. (Hrsg.) ; SARIKAS, O. D. (Hrsg.): *Proceedings of the eGovernment Workshop '05 (eGOV05)*. London, UK, 2005. – 14 pages

[World Wide Web Consortium 2003a] WORLD WIDE WEB CONSORTIUM: *About SVG*. `http://www.w3.org/TR/SVG/intro.html#AboutSVG` (zuletzt besucht am 19. Mai 2007). 2003. – letze Änderung: 14. Januar 2003

[World Wide Web Consortium 2003b] WORLD WIDE WEB CONSORTIUM: *Basic Shapes*. `http://www.w3.org/TR/SVG/shapes.html` (zuletzt besucht am 25. Mai 2007). 2003. – letze Änderung: 14. Januar 2003

[World Wide Web Consortium 2003c] WORLD WIDE WEB CONSORTIUM: *Coordinate Systems, Transformations and Units*. `http://www.w3.org/TR/SVG/coords.html` (zuletzt besucht am 19. Mai 2007). 2003. – letze Änderung: 14. Januar 2003

[World Wide Web Consortium 2003d] WORLD WIDE WEB CONSORTIUM: *Interactivity*. `http://www.w3.org/TR/SVG/interact.html#SVGEvents` (zuletzt besucht am 25. Mai 2007). 2003. – letze Änderung: 14. Januar 2003

[World Wide Web Consortium 2003e] WORLD WIDE WEB CONSORTIUM: *Text*. http://www.w3.org/TR/SVG/text.html (zuletzt besucht am 25. Mai 2007). 2003. – letze Änderung: 14. Januar 2003