Visualisierungsinstitut der Universität Stuttgart
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3261

Visualisierung von Anforderungsspezifikationen im Automobilbau

Andreas Paul

Studiengang:	Informatik
Prüfer:	Prof. Dr. Daniel Weiskopf
Betreuer:	Dr. rer. nat. Michael Burch
begonnen am:	03. November 2011
beendet am:	04. Mai 2012
CR-Klassifikation:	H.5.2, I.7.2, D.2.1

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Zusammenfassung

Anforderungsspezifikationen im Automobilbau werden typischerweise als Regeln in textueller Form verstanden. Da die Regelmengen eher groß werden und durch die textuelle Repräsentation eine effektive Analyse für den Nutzer schwierig ist, bietet sich eine visuelle Darstellung der Daten an. Strukturen und Zusammenhänge werden durch die guten perzeptuellen Fähigkeiten des menschlichen visuellen Systems leichter verständlich, als dies in Textform möglich wäre.
In dieser Arbeit wird deshalb ein grafisches Werkzeug vorgestellt, mit dem sich formale Beschreibungen von Anforderungsspezifikationen visualisieren, analysieren und editieren lassen. Der Hauptaspekt der Arbeit liegt in der Visualisierung einer Regelmenge des Faktenkalküls als Graph in Form eines Node-Link-Diagramms. Zusätzlich wird die interaktive Bearbeitung des Graphs und der zugehörigen Regelmenge mittels Linking and Brushing unterstützt. Dies bedeutet, dass eine Änderung der Regelmenge eine gleichzeitige Anpassung der Graphen nach sich zieht und andererseits eine Änderung in der Graphansicht die Regelmenge beeinflusst. Die Implementierung erfolgt in Form eines Eclipseplugins in der Sprache Java, um die Anwendung plattformunabhängig zu gestalten.

1 Einleitung

In diesem Kapitel werden die Motivation und Struktur der Arbeit vorgestellt.

1.1 Motivation

Diese Arbeit beschäftigt sich mit der interaktiven Visualisierung und Bearbeitung von Anforderungsspezifikationen im Automobilbau. Im Anschluss an die Diplomarbeit von Ralf Sommer [Som11] wird nun für den dort entwickelten Faktenkalkül eine grafische Anwendung entwickelt, welche den Benutzer durch eine Visualisierung als Node-Link-Diagramm bei der Erstellung und Bearbeitung von Anforderungsspezifikationen unterstützen soll. Da der Faktenkalkül unter den Gesichtspunkten für den Einsatz in der Entwicklung von Embedded-Systems in der Automobilbranche entwickelt wurde, soll es für diese Anwender eine einfache Möglichkeit geben, Anforderungsspezifikationen zu definieren, ohne sich großartig mit Formalismen zu beschäftigen. Die Regelmenge des Faktenkalküls wird mit Hilfe von Graphen visualisiert und mittels Linking and Brushing mit der textuellen Repräsentation der Regelmenge verknüpft. Dadurch kann die Graphrepräsentation interaktiv bearbeitet werden und den Benutzer bei der Erzeugung und Bearbeitung von Anforderungsspezifikationen unterstützen.

1.2 Beispielszenario

Ein einfaches Anwendungsbeispiel, für eine Anforderungsspezifikation, ist die adaptive Fernlichtsteuerung. Dies mag zwar recht einfach klingen, aber birgt durch den Einfluss verschiedener Komponenten wie Blinkerhebel, Lichtwahlschalter, Lichthupe oder Fernweitesensor schon Fehlerpotenzial, so dass eine verifizierbare formale Definition in Form des Faktengraphen sinnvoll ist um alle einflussnehmenden Faktoren zu berücksichtigen. Um den Zusammenhang der Regeln des Faktengraph zu verdeutlichen, kann dies mit Hilfe der in dieser Arbeit beschriebenen SpecGui-Anwendung visualisiert werden.

1.3 Aufbau der Arbeit

Im Kapitel „2 Related Work“ wird auf die Grundlagen und verwandte Arbeiten sowie verwendete Software eingegangen. Das Kapitel „3 Visualisierung“ beschäftigt sich konkret

mit den Möglichkeiten, Anwendungsspezifikationen zu visualisieren. In Kapitel „4 Implementierung“ werden Implementierungsdetails, wie etwa Datenstrukturen oder Algorithmen, erläutert. Kapitel „5 Performance“ geht auf die Geschwindigkeit der Graphvisualisierung ein. Kapitel „6 Weitere Anwendungsmöglichkeiten“ zeigt weitere Anwendungsmöglichkeiten für das entwickelte Plugin. Kapitel „7 Zusammenfassung und Ausblick“ fasst die Erkenntnisse und Einsichten, die in dieser Arbeit gewonnen wurden, zusammen.

2 Related Work

Da sich diese Arbeit mit Anforderungsspezifikationen im Automobilbau beschäftigt und zur visuellen Darstellung von Graphen die visuelle Metapher der Node-Link-Diagramme verwendet, spaltet sich die Beschreibung verwandter und existierender Arbeiten in vier separat behandelte Teile: Anforderungsspezifikationen, Graphenvisualisierung, Interaktion und existierende Software.

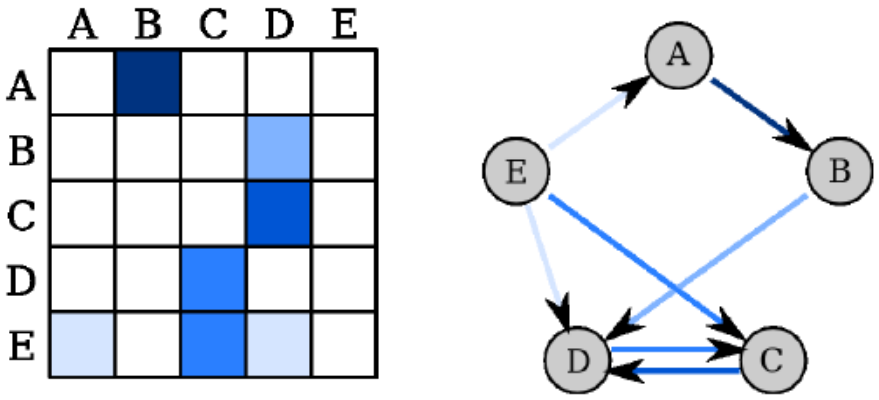
2.1 Anforderungsspezifikationen

Da im Automobilbereich Anforderungen für eingebettete Systeme oft von Personen mit wenig informationstechnischem Hintergrund verfasst werden, sind diese oft nur in Textform vorhanden. Dies erschwert die automatisierte Verarbeitung bei modellgestützter Entwicklung und Testfallerstellung.
In seiner Diplomarbeit „Automatisierte Formalisierung von Anforderungen an eingebettete Systeme im Automobilbau“ [Som11] entwickelte Ralf Sommer einen Faktenkalkül, um Anforderungen in eingebetteten Systemen im Automobilbau formal zu beschreiben. Dieser Faktenkalkül soll die Hürde für die formale Definition von Anforderungen senken, um eine maschinenlesbare Definition zu erhalten, die einfach weiterverarbeitet werden kann.
Für die Beschreibung stehen Systemgrößen, Zustände und Ereignisse zur Verfügung, sowie Prädikate und Funktionen, welche diese ineinander überführen. Jedes dieser Elemente stellt einen Fakt dar. Mehrere Fakten bilden einen Faktengraph.
Als Systemgrößen werden alle möglichen Größen, die kontinuierliche oder diskrete Werte annehmen können, bezeichnet. Dies können auslesbare Eingangsgrößen oder veränderbare Ausgangsgrößen sein. Dazu zählen einerseits interne Variablen aber auch zu messende Umwelteigenschaften wie Luftdruck oder Geschwindigkeit und veränderbare Größen wie Sollgeschwindigkeit. Als eine Variante steht eine kontinuierliche Größe „VAR“ zur Verfügung, die keinen konkreten Datentyp hat, aber mit einem Wertebereich (Minimum und Maximum) versehen ist. Die zweite Variante „ENUM“ ist als diskrete Größe mit endlich vielen Zuständen ausgelegt.
Weiter hat Ralf Sommer Zustände („STATE“) und Ereignisse („EVENT“) definiert, da sich das Verhalten zwar rein über die Systemgrößen beschreiben lässt, dies aber zu abstrakt ist. Zustände erstrecken sich über einen Zeitraum und Ereignisse definieren Zeitpunkte. Für Zustände und Ereignisse gelten die üblichen Operatoren der Mengenlehre. Beispielsweise kann ein Ereignis A auftreten, wenn die Ereignisse B und C gleichzeitig auftreten (Schnitt). Verknüpft werden die Systemgrößen mit Ereignissen und Zuständen über Prädikate. Somit wandeln diese die Größen in Wahrheitswerte um und definieren so, in welchen Belegungen

Zustände und Ereignisse aktiv bzw. inaktiv sind. Prädikate überführen Systemgrößen in Zustände. Die Rückrichtung übernehmen Funktionen, indem sie die Werte von Systemgrößen anhand von Zuständen und Ereignissen verändern.

2.2 Graphvisualisierung

Graphen lassen sich auf verschiedene Arten darstellen. Die gebräuchlichsten Möglichkeiten sind in Form einer Matrix oder als Node-Link Diagramm. In der Matrixdarstellung werden die Knoten horizontal und vertikal als Zeilen und Spalten aufgetragen und an den Kreuzungspunkten vorhandene Kanten als farbkodierte Zellen markiert. Hier ist man in der Darstellung relativ stark festgelegt und es gibt wenig Freiräume zur übersichtlichen Gestaltung. Bei Node-Link-Diagrammen werden Knoten als Kreise dargestellt, die durch die Kanten in Form von Pfeilen verbunden sind. Eine große Herausforderung bei Node-Link-Diagrammen ist



(a) Matrix **(b)** Node-Link-Digramm

Abbildung 2.1: Darstellung als Matrix und Node-Link-Diagramm

die sinnvolle Anordnung der Knoten, die je nach Graph unterschiedlich ist. Es muss also das Layout der Graphen optimiert werden. In der Graphvisualisierung steht dabei stets die Einhaltung ästhetischer Kriterien, die das Layout der Graphen positiv beeinflussen sollen, im Vordergrund. Dies ermöglicht dem Betrachter eines Node-Link-Diagramms, möglichst viele Einsichten aus solchen relationalen Datensätzen zu extrahieren.

2.2.1 Layoutverfahren für Node-Link-Diagramme

Um Node-Link-Diagramme sinnvoll darzustellen, müssen Knoten auf irgendeine Weise auf der Zeichenfläche platziert werden. Für wenige Knoten mit wenigen Kanten reicht es, die Positionen der Knoten zufällig zu bestimmen. Eine relativ einfache Möglichkeit, ein sinnvolles Layout zu berechnen, sind Force-Directed-Algorithmen. Einer der bekanntesten Vertreter

dieser Klasse ist der Algorithmus von Fruchterman und Reingold [FR91]. Das Prinzip funktioniert sowohl im Zwei- als auch im Dreidimensionalen und basiert auf emulierten Kräften zwischen den Knoten. Alle Knoten stoßen sich untereinander ab, je näher sie aneinander sind desto stärker. Knoten, die durch eine Kante verbunden sind, ziehen sich an. Hier kann auch das Kantengewicht in die Stärke der Anziehung einfließen. Diese Art von Algorithmus kann auf alle Graphen angewendet werden, ohne konkretes Wissen über deren Eigenschaften wie beispielsweise Planarität zu besitzen. Weitere Vorteile sind die einfache Implementierbarkeit und guten Ergebnisse, welche wichtige ästhetische Kriterien wie uniforme Kantenlängen, gleichmäßige Knotenverteilung und Symmetrien erfüllen. Durch die meist iterative Implementierung kann es aber auftreten, dass der Algorithmus in ein lokales Minimum läuft und ein nicht-ideales Ergebnis erzeugt. Die resultierenden Diagramme erfüllen jedoch aus perzeptueller Sicht viele der verlangten ästhetischen Kriterien und gelten somit als verbessertes Layout gegenüber einem Randomlayout.
Der größte Nachteil an diesem Verfahren ist die kubische Laufzeit, welche dieses Verfahren für große Graphen unbrauchbar macht. Bei Graphen, die ein Mensch noch überblicken kann (wenige hundert Knoten), lässt sich diese Berechnung jedoch noch in vertretbarer Zeit durchführen.

Bei der Graphvisualisierung in Form von Node-Link-Diagrammen gibt es verschiedene Möglichkeiten, Kanten zu visualisieren. Ungerichtete Kanten werden meist durch einfache Linien dargestellt. Gerichtete Kanten, wie sie in dieser Arbeit ausschließlich vorkommen, brauchen irgendeine Kennzeichnung, welche ihre Richtung verdeutlicht. Am weitesten verbreitet sind klassische Straight-Links mit Pfeilspitze, also gerade Linien mit einer Spitze am vorderen Ende. Je nach Graph kann es sinnvoll sein, ein anderes Kantendesign wie z.B. Tapered Edges (zugespitze, nadelfärmige Kanten) zu verwenden (Abb. 2.2). Eine Vergleichsstudie von Holten et. al [HIF10] hat gezeigt, dass sich bei Tapered und Animated Edges Verbindungen sehr schnell erkennen lassen. Curve-Based-Edges hingegen schneiden schlecht ab. Tapered Edges sind Kanten, die im Prinzip nur aus einer langgezogenen Pfeilspitze bestehen. Animated Edges sind gerichtete Kanten, die durch eine gestrichelte Linie visualisiert werden, welche sich in Richtung des Ziels bewegt. Curve-Based-Edges zeigen ihre Richtung anhand ihrer Krümmung an. Durch Verfolgung im Uhrzeigersinn lässt sich ihre Richtung bestimmen.

2.2.2 Ästhetische Kriterien

Für die Darstellung von Graphen als Node-Link-Digramme gibt es verschiedene ästhetische Kriterien (nach [Puro2]), von denen sich folgende als besonders wichtig herausgestellt haben:

- Minimierung der Kantenkreuzungen
- Minimierung der Kantenbiegung
- Maximierung der Symmetrien
- Maximierung der Winkel zwischen Kanten an einem Knoten
- Maximierung der Orthogonalität

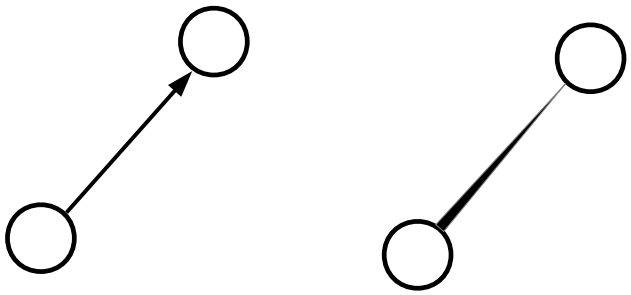


Abbildung 2.2: Vergleich von Straight-Link-Edges (a) mit Tapered Edges (b)

Weitere Kriterien:

- gleichmäßige Knotenverteilung
- gleiche Kantenlängen

2.3 Interaktion

Einer der Hauptaspekte dieser Arbeit ist das Linking and Brushing. Dieses interaktive Feature bezeichnet die Verknüpfung mehrerer verschiedener Ansichten mit einer Datenquelle sowie die Übertragung von Änderungen in einer Ansicht in andere Ansichten. Dies ist insbesondere wichtig für die Multiple-Coordinated-Views [Rob07] und erlaubt dem Betrachter die gleichzeitige Analyse von Daten aus verschiedenen Blickwinkeln. diese Technik wird heutzutage sehr häufig und effektiv im Bereich der Visual Analytics eingesetzt.

Das „Visual Information Seeking Mantra“ von Ben Shneiderman [Shn96] beinhaltet folgende Richtlinien zur visuellen Darstellung von Informationen:

- Overview first
- Zoom and Filter
- Details-on-Demand

Das Mantra beschreibt den Weg, wie Informationen dargestellt und stückweise analysiert werden können. Zuerst wird dem Betrachter eine grobe Übersicht präsentiert, bei der er dann Teilaspekte auswählen und zu diesen bei Bedarf weitere Details einblenden kann.

2.4 Software

In diesem Abschnitt werden in der Diplomarbeit verwendete Software und Bibliotheken vorgestellt. Dies ist zum einen Software, die zum Entwickeln verwendet wurde und zum anderen Software und Bibliotheken, auf welchen die Anwendung basiert bzw. Bibliotheken, die zur Verwendung eingebunden sind.

2.4.1 Eclipse

Eclipse ist eine weit verbreitete Entwicklungsumgebung, die ursprünglich von IBM entwickelt wurde, aber mittlerweile als Open Source Software unter Leitung der Eclipse Foundation weiterentwickelt wird [Ecl12]. Eclipse bietet in seinen Basisversionen schon Unterstützung für eine Vielzahl an Programmiersprachen und kann durch Erweiterungen auf neue Sprachen angepasst und mit zusätzlichen Funktionen erweitert werden. Eclipse bietet Unterstützung für Perspektiven, das heißt, es lassen sich je nach Anwendungszweck Elemente der grafischen Oberfläche ein- und ausblenden und schnell zwischen verschiedenen Perspektiven wechseln um für verschiedene Aufgaben, verschiedene Steuerelemente zur Verfügung zu haben

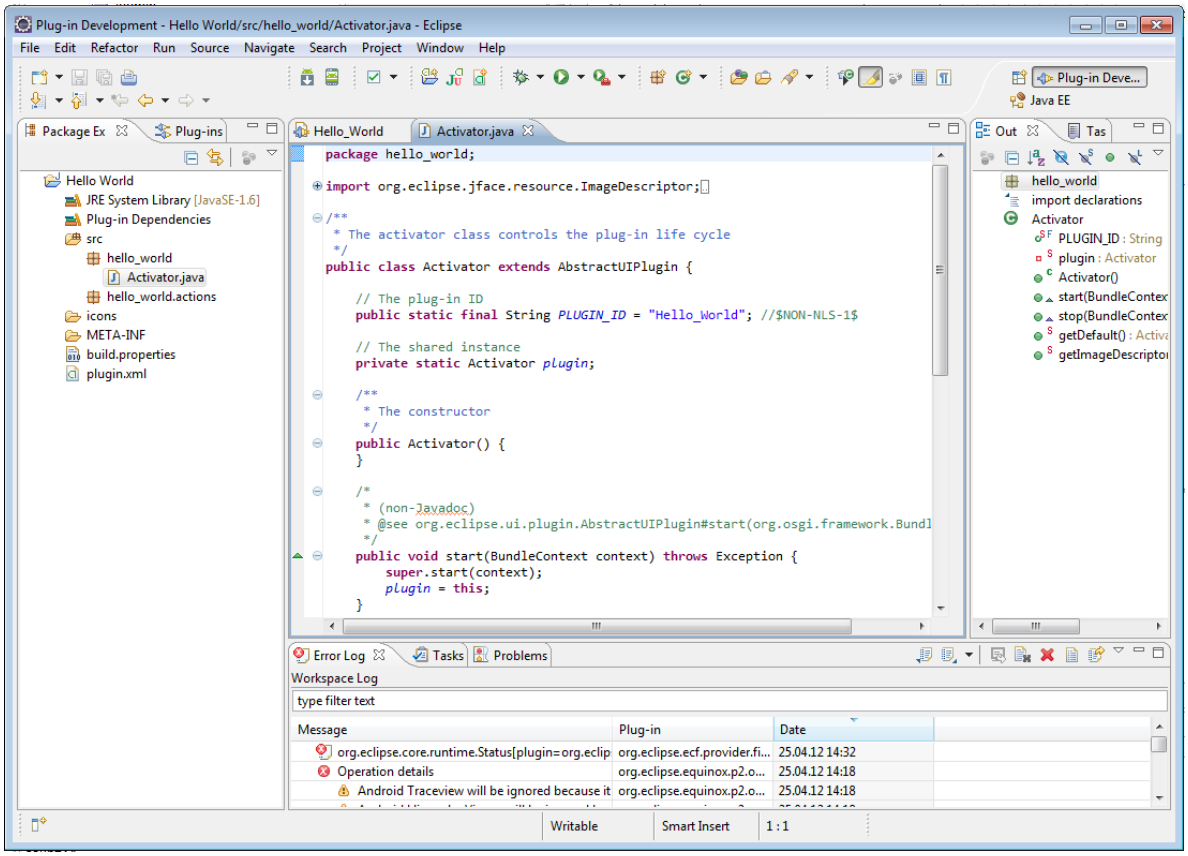


Abbildung 2.3: Entwicklungsumgebung Eclipse

Oberfläche

Die Benutzeroberfläche von Eclipse benutzt hauptsächlich drei Komponenten: Editoren, Views und Perspektiven.
Editoren sind Fenster, die den Inhalt von Dateien anzeigen und eine Bearbeitung dieser Inhalte ermöglichen. Dies können einfache Quelltexteditoren wie z.B. für Java oder C++ sein, aber auch komplexere Editoren, die Formulare enthalten. Zusätzlich gibt es als Spezialfall Multipageeditoren. Dies sind Editoren, die mehrere Tabs enthalten und auf diesen entweder verschiedene Ansichten darstellen oder Teilaspekte der geöffneten Datei präsentieren.
Views sind Fenster, die Daten anzeigen, aber auch zum Editieren verwendet werden können. Der grundlegende Unterschied zu Editoren ist, dass Views der Inhalt explizit gespeichert werden kann. Änderungen innerhalb eines Views werden sofort wirksam oder verändern den Inhalt oder Teile des Inhalts eines Editors. Typische Beispiele für Views sind der Packageexplorer, der Outline-View oder die Console.
Sowohl Editoren als auch Views können frei innerhalb der Workbench (Eclipse Hauptfenster) platziert werden.
In Form von **Perspektiven** können Anordnungen von Views und Editoren gespeichert werden, um diese für verschiedene Anwendungszwecke aufzurufen. Beispielsweise werden zum Editieren von Java-Quelltext die Views, welche Debugginginformationen anzeigen, meistens nicht gebraucht.

Eclipse stellt eine Rich-Client-Platform (RCP) zur Verfügung, die es ermöglicht, Anwendungen auf Basis des Frameworks von Eclipse unabhängig zu erstellen. So lassen sich auch für Eclipse erstellte Plugins mit wenig Aufwand als eigenständige Programme realisieren.

2.4.2 SWT

Das Standard Widget Toolkit (SWT) ist eine Java-Bibliothek zur Erstellung grafischer Oberflächen. Es wurde von IBM für Eclipse entwickelt. Da es die grafische Basis für Eclipse darstellt, wird es auch verwendet, um die Oberflächen von Plugins zu gestalten.

2.4.3 Xtext

Xtext ist ein Framework zur Entwicklung von domänenspezifischen Sprachen (Domain Specific Languages - DSLs), also Sprachen, die für ein spezielles Problem entworfen werden. Durch Xtext lassen sich mit einer einfachen Definition einer Grammatik auf Java basierende Parser generieren. Im Gegensatz zu anderen Parsergeneratoren erstellt Xtext auch die geparsten Dateien als abstrakten Syntaxbaum zur Verfügung, auf den über ein Klasenmodell zugegriffen werden kann. Außerdem erstellt Xtext Editorkomponenten für Eclipse mit Features wie Syntaxhighlighting, Codefolding oder Codecompletion.

2.4.4 Window Builder Pro

Window Builder Pro ist ein Plugin für Eclipse, um grafische Oberflächen für SWT-Anwendungen zu gestalten. Es bietet hierfür eine grafische WYSIWYG-Oberfläche. Window Builder Pro beherrscht auch die Veränderung im grafischen Modus einer Klasse, die „von Hand“ erstellt wurde. Window Builder Pro wurde im erstellten Plugin hauptsächlich für Dialoge verwendet, da dort viele einzelne Widgets platziert werden müssen.

3 Visualisierung

3.1 Visualisierung von Graphen

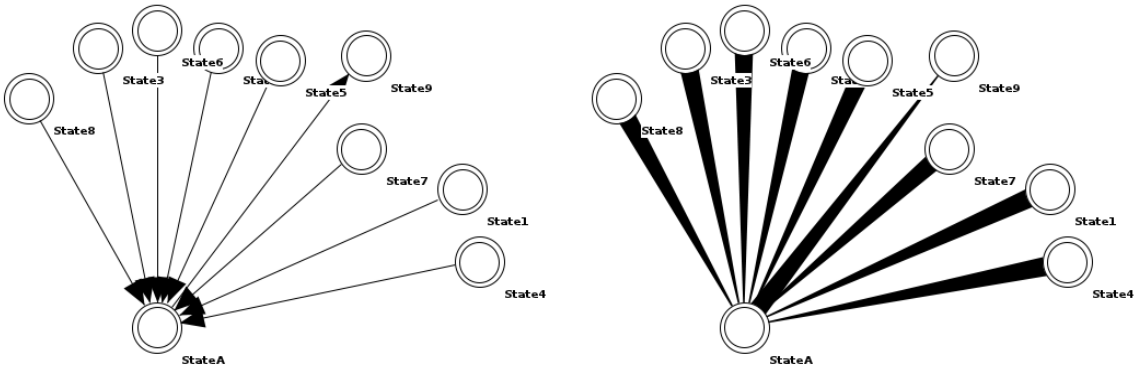
Graphen lassen sich auf verschiedene Arten darstellen. Die wichtigsten Visualisierungsarten sind Matrizen und Node-Link-Diagramme. Matrizen eignen sich gut bei kleineren Graphen um eine Übersicht über den gesamten Graphen zu bekommen, sind in ihren Variationsmöglichkeiten aber relativ stark beschränkt. In Node-Link-Diagrammen kann man sehr gut pfadbezogene Zusammenhänge erkennen und ist durch das Layout der Knoten in der zweidimensionalen Fläche recht frei in der Gestaltung. Da es im Faktengraphen stark auf das Erkennen von Verknüfungen und Pfaden ankommt wird für die Visualisierung in dieser Arbeit auf Node-Link-Diagramme gesetzt.

3.1.1 Kantentypen

Von den möglichen Kantentypen werden in der Standardansicht klassische Straight-Links verwendet. Da diese jedoch aufgrund ihrer Pfeilspitze am Zielknoten viel Platz verdecken, kann bei vielen eingehenden Kanten die Übersicht leiden. Deshalb lässt sich die Visualisierung bei einigen der Kanten durch ein interaktives Feature zu Tapered Edges umschalten.

3.2 Darstellung von Anforderungsspezifikationen als Graph

Zur Darstellung des Faktengraphen eignen sich am besten Node-Link-Diagramme. Hier kann man einfach und schnell die Pfade zwischen einzelnen Elemente erkennen. Diese Aufgabe ist bei einer Darstellung als Matrix nicht so einfach zu lösen. Mohammad Ghoniem und Jean-Daniel Fekete zeigen in [GFC04], dass bei Graphen ab 20 Knoten die Darstellung als Node-Link-Diagramm für pfadbezogene Aufgaben besser abschneidet als die Darstellung als Matrix. Da Node-Link-Diagramme besonders gut geeignet sind für Pfadverfolgung und Graphstrukturanalyse, besteht die hier vorgestellte Graphvisualisierung auf der visuellen Metapher der Knoten und Kanten. Die Darstellung erfolgt durch traditionelle Straight-Links mit Pfeilspitze, die sich auch auf Kanten im Stil von Tapered Edges umschalten lassen.



(a) Straight-Link **(b)** Tapered Edge

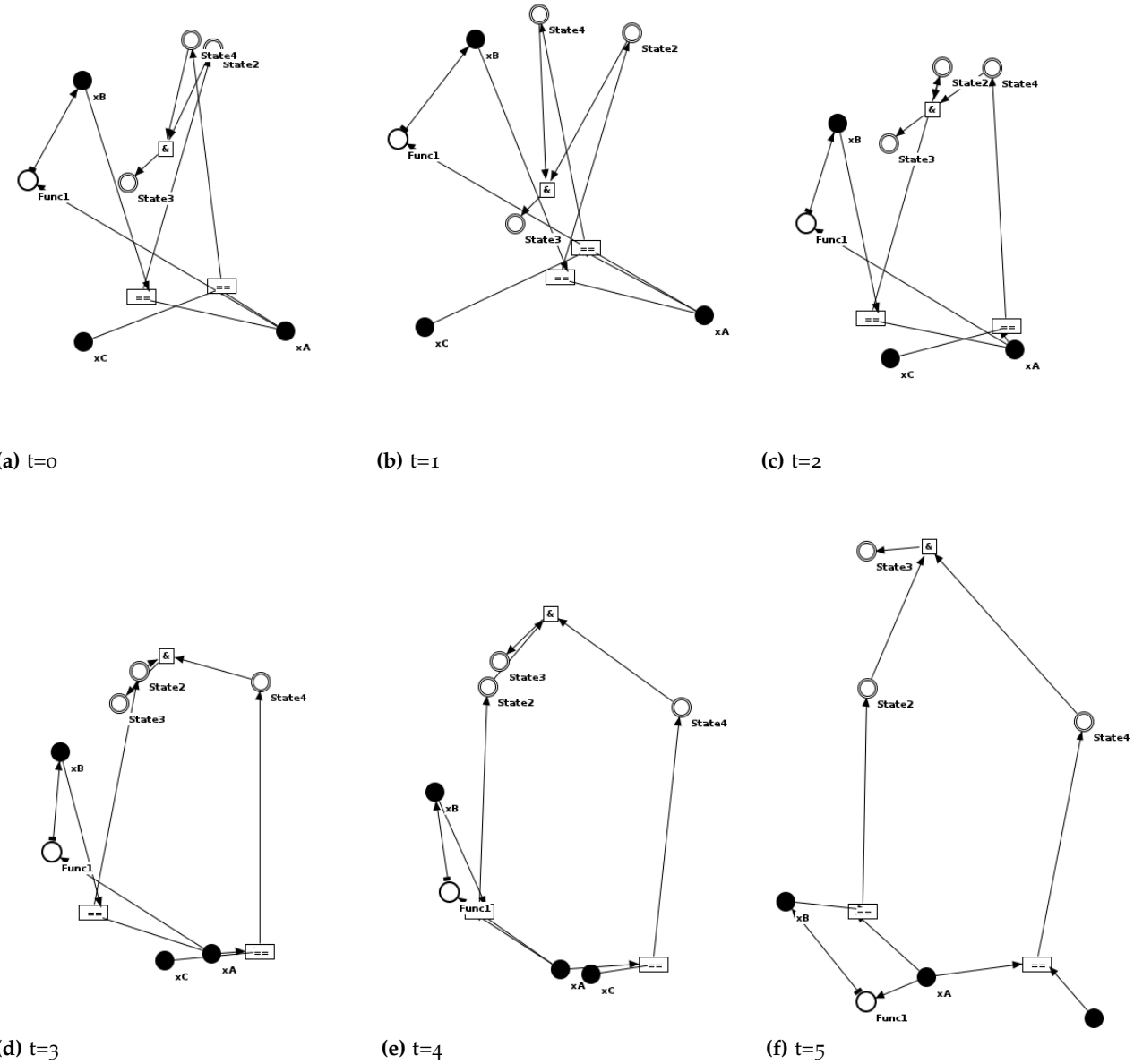
Abbildung 3.1: Vergleich Straight-Link Edge mit Pfeilspize und Tapered Edge. In der Ansicht mit Tapered-Edges ist die eine ausgehende Kante zwischen den vielen eingehenden Kanten besser zu erkennen.

3.2.1 Smooth Animation

Neben einem übersichtlichen statischen Layout ist auch der Weg zu diesem Layout ein wichtiger Aspekt der Visualisierung. Wird von einem Randomlayout ein optimiertes Layout berechnet und dieses direkt angezeigt, so verliert der Betrachter die Übersicht, die er sich vorher erarbeitet hat. Dies entspricht dem Prinzip der MentalMap Erhaltung, wie es unter anderem von Eades et al. [ELMS91] beschrieben wird. Deshalb werden Positionsänderungen von Objekten nicht sofort ausgeführt, sondern über eine Animation zu ihrer Zielposition bewegt. Graphobjekte, welche animiert werden sollen, werden innerhalb des Graphcontainers (siehe 4.2.3) in eine Animationswarteschlange eingefügt. Die Warteschlange wird alle 10oms von einem Animationsthread durchlaufen. Dieser berechnet den nächsten Animationsschritt des Objekts. In der Regel werden nur Knoten explizit animiert, da die Lage von Kanten auf der Leinwand immer von zwei Knoten abhängig ist und somit werden bei Knotenbewegungen die Kanten implizit mitbewegt.

3.2.2 Zuordnung von Fakten zu Graphelementen

Die Zuordnung der Fakten des Faktengraphen von Ralf Sommer [Som11] ergibt sich zum Teil aus Vorarbeiten, aus denen er den Faktengraph entwickelt hat. Recht einfach ersichtlich ist die Darstellung von Systemgrößen (Variablen) als Knoten, da diese keine Vorgänge oder Veränderungen beschreiben, sondern Zustände. Genauso verhält es sich mit Zuständen und Ereignissen. Obwohl Funktionen eine Veränderung beschreiben, wird das eigentliche Funktionsobjekt als Knoten dargestellt, da Funktionen mehrere Eingangswerte auf einen Ausgangswert abbilden. Dies ließe sich mit einer Kante, die nur zwei Objekte miteinander verbindet, nicht darstellen. Zusätzlich gibt es noch im Faktengraph „Requirements“, die



(a) t=0 **(b)** t=1 **(c)** t=2

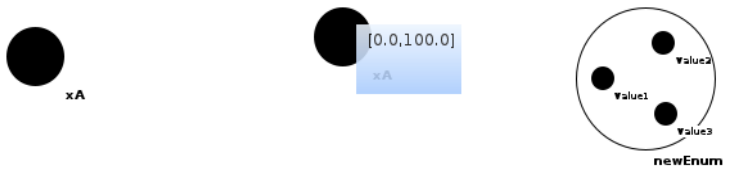
(d) t=3 **(e)** t=4 **(f)** t=5

Abbildung 3.2: Verschiedene Zeitpunkte einer Smooth Animation

definieren, welche Zustände, Events und Funktionen unter welchen Voraussetzungen aktivierbar sind. Dafür muss die Funktion als ein Objekt repräsentiert werden, das von einem Requirement referenziert werden kann. Dies funktioniert mit einem Knoten besser als mit einer Kante. Diese Requirements sind die einzigen Fakten, die durch eine einzelne Kante repräsentiert werden. Alle anderen Fakten (Events, Zustände und Funktionen) bestehen immer aus zwei Teilen. Dem Objekt selber, das als Knoten visualisiert wird, und einer Definition, welche die Abhängigkeiten, die aus Kombinationen von mehreren Knoten und Kanten dargestellt werden, beschreibt.

Variablen

Es gibt zwei Arten von Variablen: Variablen, die Werte repräsentieren (VAR) und diskrete Aufzählungen (ENUM). Der Typ VAR wird als einfacher Kreis dargestellt. ENUM wird als Gruppierung von Knoten dargestellt, welche die einzelnen Belegungen repräsentieren. (Abb. 3.3).



(a) Var **(b)** Var mit Tooltip **(c)** Enum

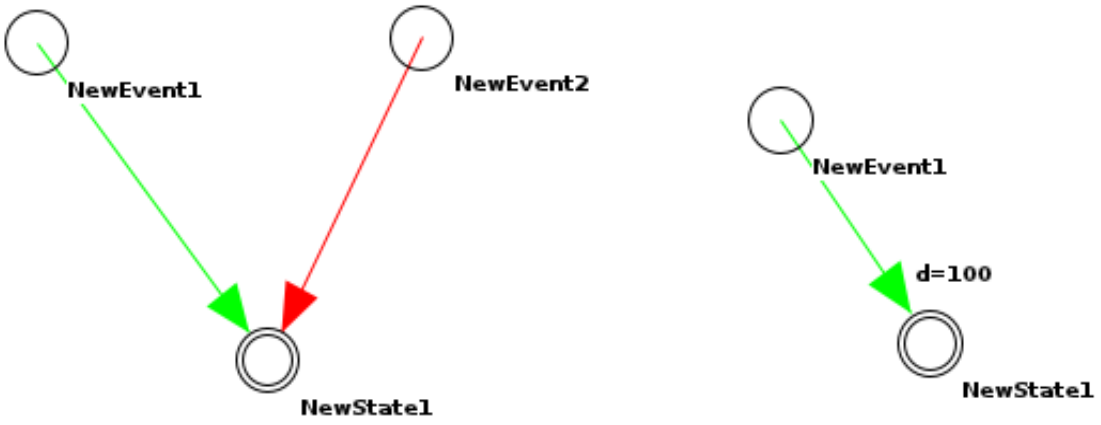
Abbildung 3.3: Visualisierung von Variablen: Abb. (a) und (b) zeigen ein Beispiel für eine Variable mit dem Wertebereich 0 < xVar < 100. In Bild (b) wird der Wertebereich über ein Tooltip beim MouseOver angezeigt (xA := [0.0,100.0]). In (c) wird eine Enumeration mit drei Werten dargestellt (newEnum§ := {"Value1","Value2","Value3"})

Zustände

Der eigentliche Zustandsknoten wird als Doppelkreis dargestellt. Zustände (STATE) sind außer durch den Namen durch unterschiedliche Abhängigkeiten definiert und erstrecken sich über bestimmten Zeiträume. Die Abhängigkeiten, die ihn definieren, beschreiben also die Zeiträume, in denen der Zustand aktiv ist.

Abbildung 3.4: Visualisierung eines Zustands (State)

- **StateDefEv** definiert einen Zustand über ein Start- und ein Endevent oder ein Startevent und eine Dauer. Die Visualisierung erfolgt über eine grün und eine rot gefärbte Kante vom Start- bzw. Endevent (Abb. 3.5a). Bei Definition über ein Startevent und eine Zeitdauer besitzt die Startkante einen Tooltip, der die Dauer angibt (Abb. 3.5b).



(a) Start- und Endevent **(b)** Startevent und Zeitdauer

Abbildung 3.5: Visualisierung StateDefEv: In Abb. (a) wird der Zustand NewState1 durch das Ereignis NewEvent1 gestartet und durch NewEvent2 gestoppt (NewState1# := < NewEvent1 ; NewEvent2 >). In Abb. (b) wird der Zustand durch das Ereignis NewEvent1 gestartet und hält für 100 Zeiteinheiten an (NewState1# := < NewEvent1 , 100 >).

- **StatePreEnm** definiert die Aktivität des Zustands über die Belegung einer Aufzählungsvariable. Visualisiert wird es über eine Kante vom Wert der Aufzählungsvariable zum Zustand. Soll eine „Nichtaktivierung“ ausgedrückt werden, so wird in die Kante am Ende ein kleiner Kreis eingefügt (Abb. 3.6).

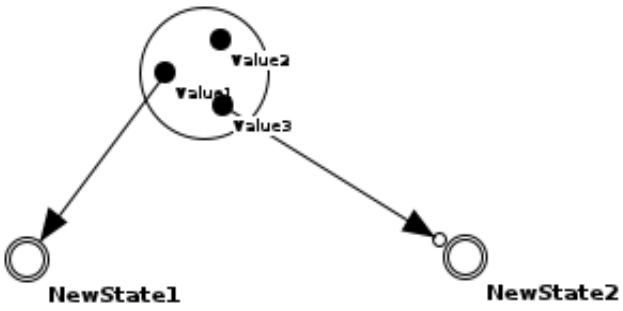
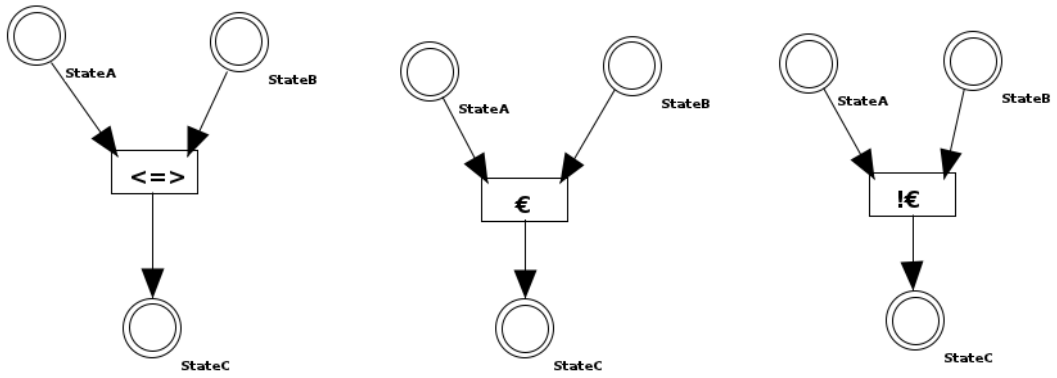


Abbildung 3.6: Visualisierung von StatePreEnm: Zustand „NewState1“ ist aktiv während „NewEnum1“ die Belegung „Value1“ hat. „NewState2“ ist nicht aktiv bei der Belegung „Value2“ (NewState1#:=NewEnum1==Value1; NewState2#:=NewEnum1!=Value3),

- **StatePreSt** beschreibt Relationen von Zuständen zueinander. Hier gibt es die Äquivalenz, die Inklusion und die Nichtinklusion. Dabei ist der Zustand aktiviert, wenn zwei andere Zustände äquivalent (Äquivalenz), ineinander enthalten (Inklusion) bzw. nicht enthalten (Nichtinklusion) sind. Dies wird grafisch über einen zusätzlichen Knoten gelöst, der den jeweiligen Operator darstellt. Bei der Äquivalenz ist dies einfach zu lösen, da diese kommutativ ist. Bei der Inklusion und Nichtinklusion kommt es auf die Reihenfolge der Kanten an. Da dies je nach Positionierung der Knoten nicht eindeutig ist, werden hierfür Operatorknoten eingeführt, bei denen die eingehenden Kanten an bestimmte Stellen zeigen, um so die Zuordnung zu gewährleisten (Abb. 3.7).
- **StateComposition** definiert die Aktivierung des Zustands durch die Kombination anderer Zustände über Mengenoperationen. Es sind die Operationen Schnitt, Vereinigung und Differenz möglich. Die Visualisierung erfolgt in ähnlichem Stil wie bei einem Parsingbaum (Abb. 3.8). Besondere Beachtung benötigt hier die Differenz, da diese nicht kommutativ ist. Auch hier wird wieder der Operatorknoten für nicht-kommutative Operationen zur Visualisierung verwendet.
- **StatePreAri** ist die Aktivierung eines Zustand über einen Vergleich von zwei arithmetischen Ausdrücken. Die Ausdrücke können über folgende Ordnungsrelationen verglichen werden: „<“, „<=“, „==“, „>=“, „>“. Auch hier werden die arithmetischen Ausdrücke über eine baumartige Struktur visualisiert (Abb. 3.9).

Events

Events besitzen Ähnlichkeiten mit Zuständen, beschreiben aber nur einen Zeitpunkt und keinen Zeitraum. Visualisiert wird der Knoten durch einen Kreis. Zur Definition gibt es ähnliche Möglichkeiten wie bei Zuständen.



(a) Äquivalenz **(b)** Inklusion **(c)** Nichtinklusion

Abbildung 3.7: Visualisierung von StatePreSt: In Bild (a) ist Zustand StateC als aktiv definiert, wenn die Zustände StateA und StateB äquivalent sind (StateC# := StateA §StateB). In Bild (b) ist der Zustand StateC aktiviert, wenn der Zustand StateA in StateB enthalten ist (StateC#:=StateA∈StateB). Entsprechend ist in (c) Zustand State C aktiv, wenn StateA nicht in StateB enthalten ist (StateC#:=StateA!∈StateB).

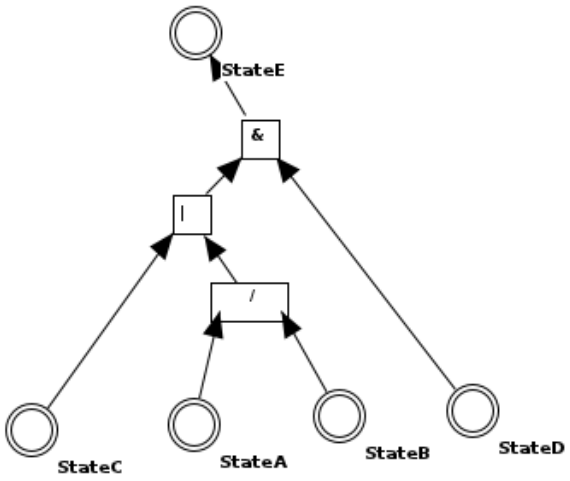


Abbildung 3.8: Visualisierung von StateComposition: Dieses Beispiel zeigt eine Definition des Zustands StateE in Abhängigkeit von StateA, StateB, StateC und StateD nach folgender Regel: StateA# := ()(StateC | (StateA / StateB))& StateD)

- **EventPreEnum** definiert die Auslösung des Events bei Änderung der Belegung einer Aufzählung von einem Wert zu einem anderen. Visualisiert wird dies durch eine Kante von der Enumeration ausgehend (Abb. 3.10). Diese ist mit einem ToolTip versehen, welcher die Belegungen, bei deren Wechsel das Event ausgelöst wird, anzeigt.

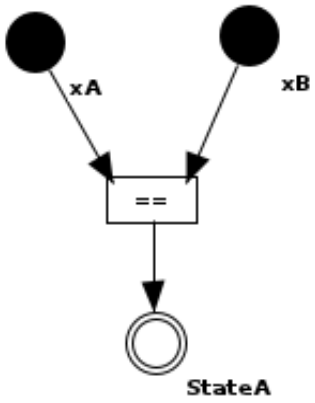
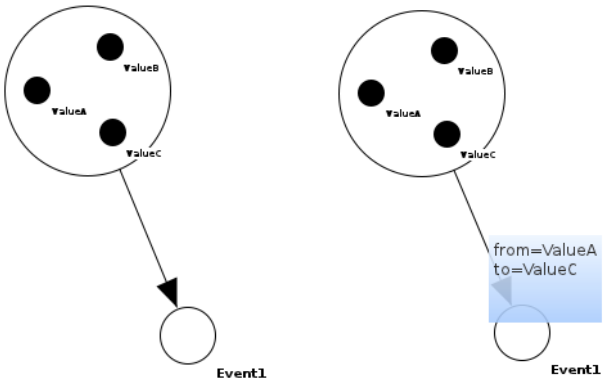


Abbildung 3.9: Visualisierung von StatePreAri: In diesem Beispiel ist der Zustand StateA aktiviert, wenn die Variablen xA und xB den gleichen Wert haben (StateA#:=[xA==Xb]).



(a) ohne ToolTip **(b)** mit ToolTip

Abbildung 3.10: Visualisierung EventPreEnum: In diesem Beispiel wird das Ereignis Event1 durch den Wechsel der Belegung von ValueA zu ValueC definiert (Event1$:= Enum1 $ "Value1"−>"ValueC"). Bild (a) zeigt die normale Ansicht und Bild (b) zeigt die Ansicht mit einem Tooltip beim MouseOver.

- **EventStateRef** definiert die Auslösung des Events bei Beginn oder Ende eines Zeitraums. Die Beziehung wird durch eine grüne bzw. rote Kante visualisiert. Eine grüne Kante visualisiert die Auslösung zu Beginn des Zeitraums, die rote eine Auslösung zum Ende. Beispiel siehe Abb. 3.11.

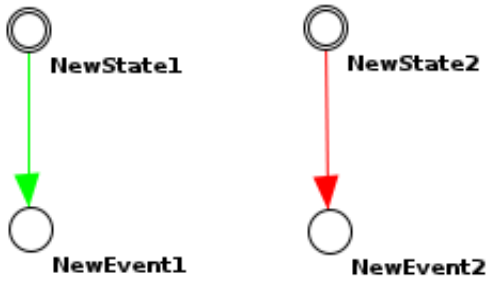


Abbildung 3.11: Visualisierung EventStateRef: NewEvent1 wird getriggert, wenn der Zeitraum von NewState1 beginnt (NewEvent1$:= NewState1 $START). NewEvent2 wird getriggert, wenn der Zeitraum von NewState2 endet (NewEvent2$:= NewState2 $END).

- **EventComposition** definiert ähnlich wie StateComposition ein Event über die Zusammensetzung aus anderen Events. Da Events Zeitpunkte beschreiben, werden hier zur Kombination die booleschen Operatoren AND ('&&') sowie OR ('||') verwendet. Abb. 3.12 zeigt ein Beispiel für eine Zusammensetzung (EventD := ((EventA && EventB) || EventC))

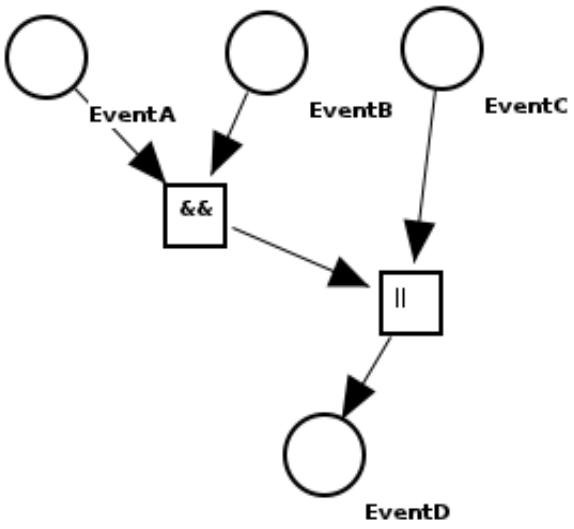


Abbildung 3.12: Visualisierungsbeispiel für EventComposition: EventD ist abhängig von EventA, EventB und EventC. Die Aktivierung ergibt sich nach der booleschen Formel: $EventD := (EventA \wedge EventB) \vee EventC$

Funktionen

Eine Funktion setzt, in Abhängigkeit von Eingangswerten, den Wert einer Variable. Obwohl Funktionen eine Veränderung beschreiben, wird das eigentliche Funktionsobjekt als Knoten dargestellt, da Funktionen mehrere Eingangswerte auf einen Ausgangswert abbilden. Dies lässt sich mit einer Kante, die nur 2 Objekte miteinander verbindet, nicht darstellen. Zusätzlich braucht es ein Objekt, das durch ein Requirement referenziert werden kann. Unterschieden wird zwischen FuncVar und FuncEnum.

- **FuncVar**
  FuncVar setzt Werte von Var-Objekten. Im Gegensatz zu Enum-Objekten gibt es hier auch eine eingehende Kante, welche ähnlich wie bei StateComposition einen Berechnungsbaum aus anderen Variablen und Werten bildet (Abb. 3.14).
- **FuncEnum**
  FuncEnum setzt Werte von Enum-Objekten. Dabei kann die Funktion genau eine Belegung eines Enum-Objektes setzen. Deshalb gibt es von einem FuncEnum-Objekt nur eine ausgehende Kante zu einer Belegung eines Enum-Objekts (Abb. 3.13).

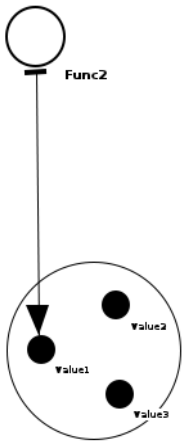


Abbildung 3.13: Visualisierung von FuncEnum: Die Funktion Func2 setzt die Enumeration auf die Belegung „Value1“ (Func2 := Enum1 #= Value1).

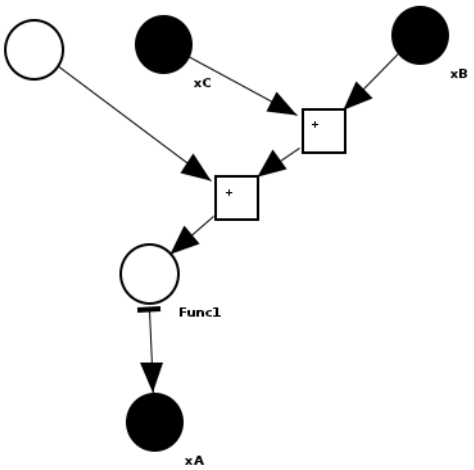
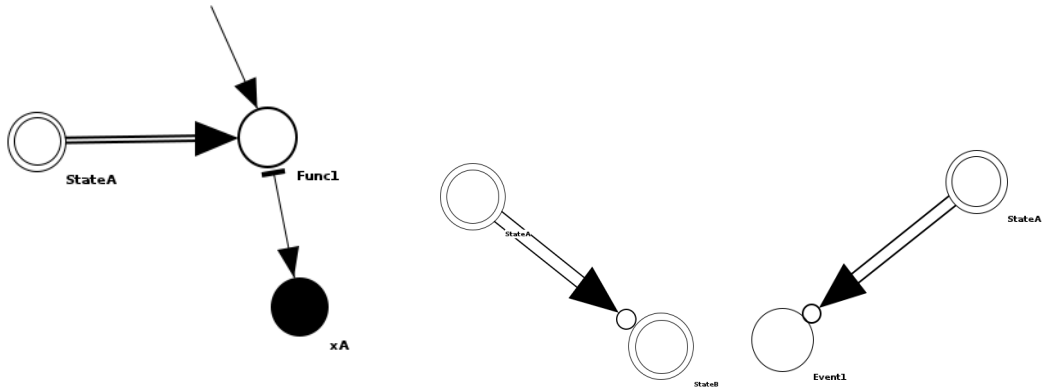


Abbildung 3.14: Visualisierung von FuncVar: (Func1 := xA = (5.0 + (xC + xB))) Die Funktion Func2 setzt der Variable xA den Wert der arithmetischen Formel: $xA := 5.0 + (xC + xB)$

Requirements

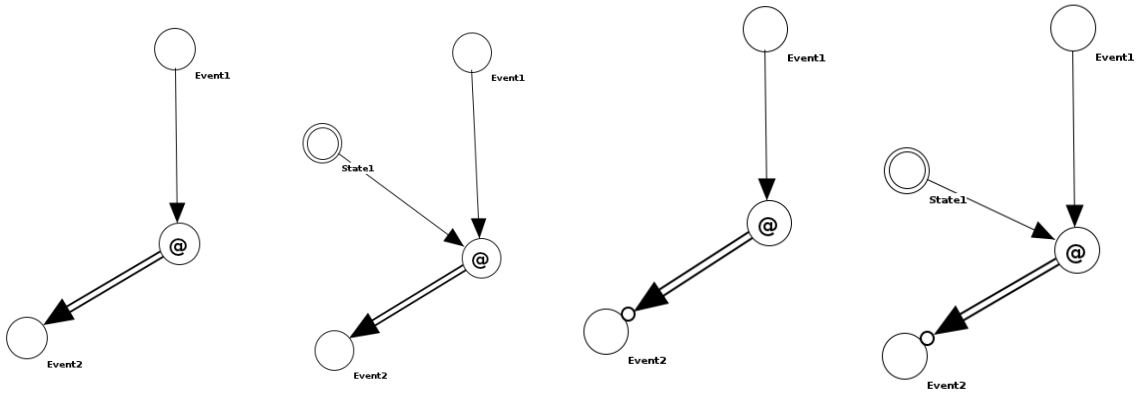
Anforderungsmuster (Requirements) lassen sich in SituationPattern und ReactionPattern unterteilen. SituationPattern beziehen sich auf Zeiträume (zustandsbasiert), ReactionPattern auf Zeitpunkte (ereignisbasiert).

- **SituationPattern**
  Unter den SituationPattern gibt es die Aktivität (SituActivity), das Aktivitätsverbot (SituActProh) und das Ereignisverbot (SituEVProh). Die Aktivität legt unter einer Vorbedingung (Zustand) eine Funktion fest, die ausgeführt wird, solange die Vorbedingung gilt (Abb. 3.15(a)). Das Aktivitätsverbot verbietet das Auftreten eines Zustandes, solange die Vorbedingung gilt (Abb. 3.15(b)). Entsprechend verbietet das Ereignisverbot das Auftreten eines Ereignisses (Abb. 3.15(c)). Visualisiert werden alle SituationPattern mit einem Pfeil mit Doppellinie. Die Verbote erhalten an der Pfeilspitze zusätzlich einen kleinen Kreis als Symbol für eine Negation.
- **ReactionPattern**
  Unter den ReactionPattern (Abb. 3.16) gibt es die Ereignisforderung (ReactionEvent) und das Ereignisverbot (ReactionProhibition). Die Ereignisforderung aktiviert ein Ereignis, das Ereignisverbot untersagt das Auftreten eines Ereignisses. Beide werden durch ein Event unter einem aktivierten Zustand als Vorbedingung getriggert. Da hier insgesamt drei Objekte mitspielen (Vorbedingung, gefordertes bzw. verbotenes Ereignis und triggerndes Ereignis), muss hier wieder auf einen Knoten als Hilfsmittel zurückgegriffen werden um die drei Objekte zu verknüpfen.



(a) SituActivity **(b)** SituActProh **(c)** SituEVProh

Abbildung 3.15: Visualisierung von SituationPattern: In Beispiel (a) wird die Funktion Func1, die den Wert von xA setzt, ausgeführt, solange der Zustand StateA aktiv ist (∼(StateA):∼Func1). In (b) wird das Auftreten des Zustands StateB verboten, solange Zustand StateA aktiv ist (∼(StateA)!∼StateB). Bild (c) zeigt das ein Verbot von Event1, solange Zustand StateA aktiv ist (∼(StateA)!@Event1).



(a) ReactionEvent ohne Vorbedingung **(b)** ReactionEvent mit Vorbedingung **(c)** ReactionProhibition ohne Vorbedingung **(d)** ReactionProhibition mit Vorbedingung

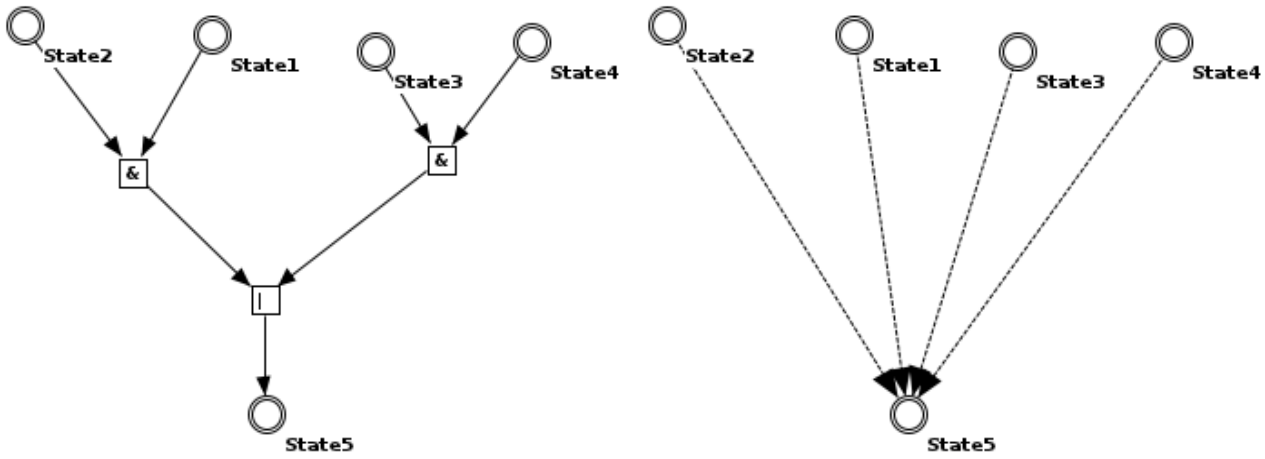
Abbildung 3.16: Visualisierung von ReactionPattern: In (a) wird eine Ereignisforderung für Event2 dargestellt, die von Event1 getriggert wird (@(Event1):Event2). In (b) muss zusätzlich State1 als Vorbedingung gelten (@(Event1 in State1):Event2). In (c) wird ein Ereignisverbot für Event2 dargestellt, das von Event1 getriggert wird (@(Event1)!:Event2). In (d) muss zusätzlich die Vorbedingung State1 gelten (@(Event1 in State1)!:Event2).

3.2.3 Optimierte Darstellung

Die bisherige Visualisierung zeigt zwar einen Faktengraphen effektiv an, bietet aber noch Möglichkeiten das Layout weiter zu optimieren.

3.2.4 Shrink

Es gibt mehrere Elemente im Graphen, die einen unübersichtlichen Baum erzeugen können. Hierzu zählen alle Elemente mit jeglicher Art von Formel. Dies sind die Komposition von Zuständen und Ereignissen sowie arithmetische Formeln.
Um hier die Übersicht zu erhöhen gibt es die Möglichkeit, alle Knoten und Kanten, die zu einer Formel gehören, auszublenden und nur Kanten anzeigen zu lassen, die zeigen, von welchen Objekten das definierte Objekt abhängig ist.

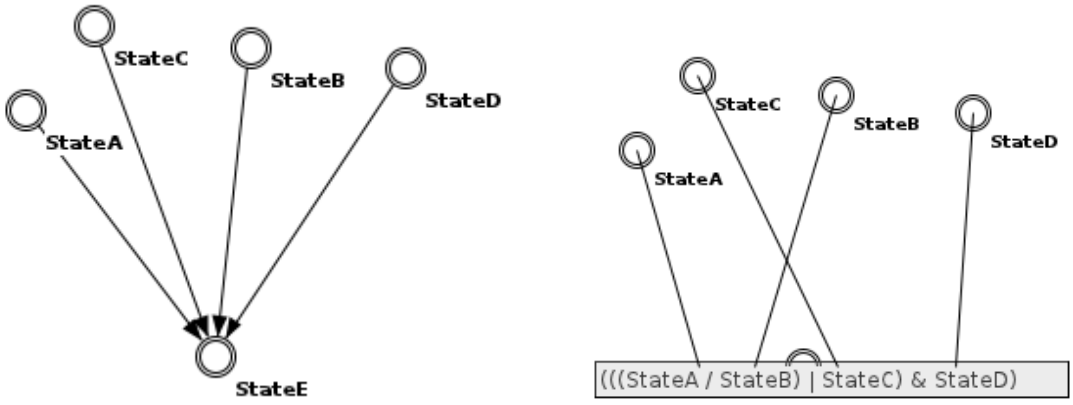


(a) (b)

Abbildung 3.17: Visualisierung von ausgeblendeten Bäumen

3.2.5 Overlay

Um bei ausgeblendeten Strukturen ohne umzuschalten trotzdem die dahinter liegende Struktur zu sehen, wird bei den Zielknoten, welche die jeweilige Definition beinhalten, bei einem MouseOver ein Overlay eingeblendet. Dieses Overlay besteht aus der Formel, die in der Definition enthalten ist, welche ähnlich einem Tooltip über den Knoten gelegt wird. Die jeweiligen Elemente in der Formel werden mit Kanten ausgehend von den Grundbausteinen verbunden. Die vorhandenen Kanten werden dabei ausgeblendet, um die Übersicht zu erhöhen (Abb. 3.18b).



(a) **(b)**

Abbildung 3.18: Visualisierung Overlay: (a) StateE ist abhängig von den anderen Zuständen, der Baum der die Zusammensetzung repräsentiert ist ausgeblendet. (b) Bei einem MouseOver erscheint die Formel als Overlay mit Kanten zu den jeweiligen Elementen

3.3 Anwendungsbeispiel

Mit den dargestellten Elementen lassen sich beliebige Instanzen des Faktengraphen visualisieren. Als Beispiel wird hier eine Modellierung für ein adaptives Fernlicht von Ralf Sommer [Som11] verwendet.
Dieses Beispiel besteht aus verschiedenen Ein- und Ausgangsgrößen, Funktionen und Aktivitäten.

```
~(zLichthupe ) :~ fFernlicht
~(zAdaptivmodus ) :~ fEinstelllicht
~(zNormalfernmodus ) :~ fFernlicht
xLichtkegel
Blinkerhebel§:= {"gezogen","neutral","gedrueckt"}
Lichtschalter§ := {"normal","adaptiv"}
xSensorwert
zLichthupe# := Blinkerhebel == "gezogen"
zAdaptivwunsch# := Lichtschalter == "adaptiv"
zAdaptivmodus# := (zAdaptivwunsch & zFernlichtanforderung)
zNormalfernmodus# := (zNormalbetrieb & zFernlichtanforderung)
zNormalbetrieb# := Lichtschalter == "normal"
zFernlichtanforderung# := Blinkerhebel == "gedrueckt"
fFernlicht := xLichtkegel = xFernweite
xFernweite := [50.0,50.0]
fEinstelllicht := xLichtkegel = xSensorwert
```

Listing 3.1: Anwendungsbeispiel: adaptives Fernlicht

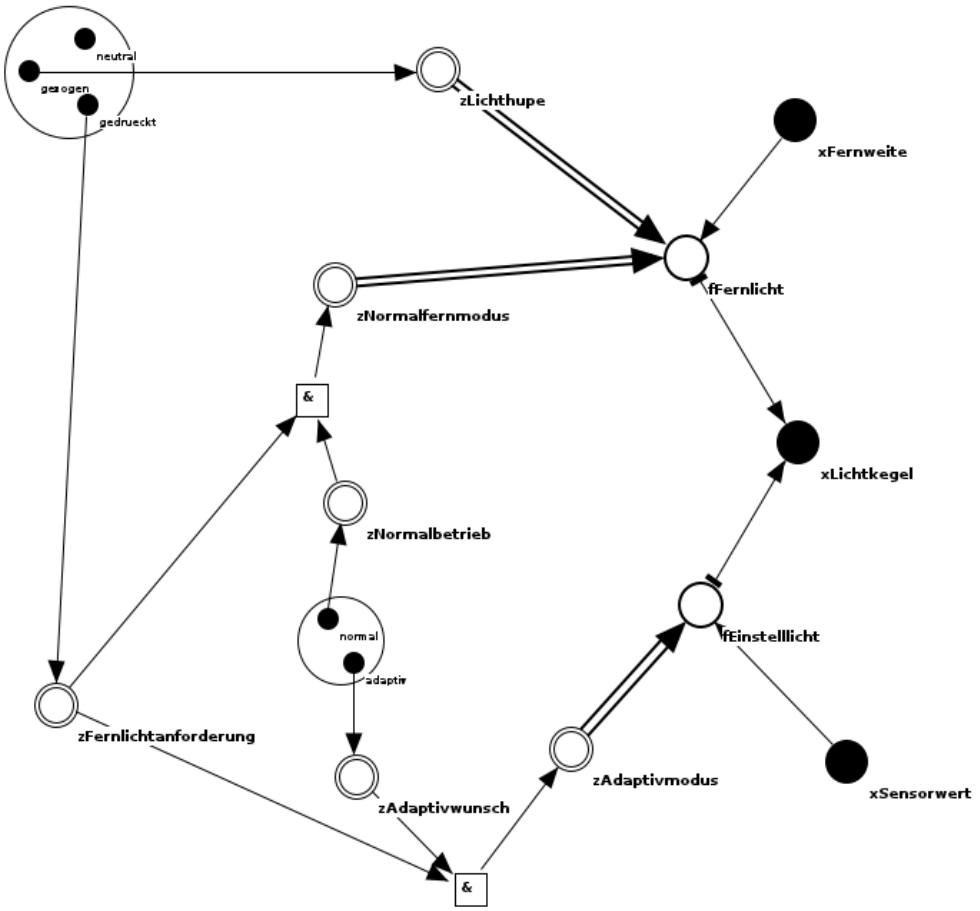


Abbildung 3.19: Visualisierung Faktengraph für adaptives Fernlicht aus Listing 3.1

4 Implementierung

Die im Rahmen dieser Arbeit entwickelte Visualisierung ist nicht als eigenständige Anwendung, sondern als Plugin namens „SpecGui“ für Eclipse entwickelt worden. Da Eclipse weit verbreitet ist, bleibt der Aufwand zur Einarbeitung für den Anwender meistens gering. Dieses Kapitel beschreibt, wie dieses Plugin realisiert wurde.
Im Abschnitt „4.1 GUI“ wird auf die Darstellung eingegangen. Der Abschnitt „4.2 Datenmodell“ beschreibt das darunterliegende Datenmodell und die verwendeten und erstellten Klassen. In „4.4.1 Layoutverfahren“ werden die verwendeten Layoutalgorithmen vorgestellt.

4.1 GUI

Die grafische Oberfläche innerhalb von Eclipse besteht aus zwei Hauptkomponenten: Einem Multipageeditor und einer Viewkomponente. Dabei lässt es Eclipse zu, diese prinzipiell frei zu platzieren. Jedoch hat sich die Übersicht wie in Abb. 4.1 als praktisch erwiesen. Links befindet sich relativ schmal der Projektexplorer von Eclipse. Den Hauptteil der Fläche teilen sich der Editor und die View. Diese Standardansicht ist in einer Perspektive, welche in dem Plugin Anwendung integriert ist, vorgegeben. Diese lässt sich aber auch beliebig ändern.

4.1.1 Multipageeditor

Der Multipageeditor besteht aus mehreren Seiten, die in Tabs angeordnet sind. Die erste Seite enthält einen normalen Texteditor, mit der sich die geöffnete SpecWorks-Datei in Textform bearbeiten lässt (Abb. 4.3). Auf einer zweiten Seite befindet sich eine tabellarische Ansicht (Abb. 4.3b) der einzelnen Fakten. Für jedes Element lässt sich ein Kontextmenü aufrufen, mit dem die Fakten über Dialoge manipuliert werden können.
Die dritte Seite enthält eine Ansicht der Fakten in Form von Sätzen in Prosaform (Abb. 4.3c).

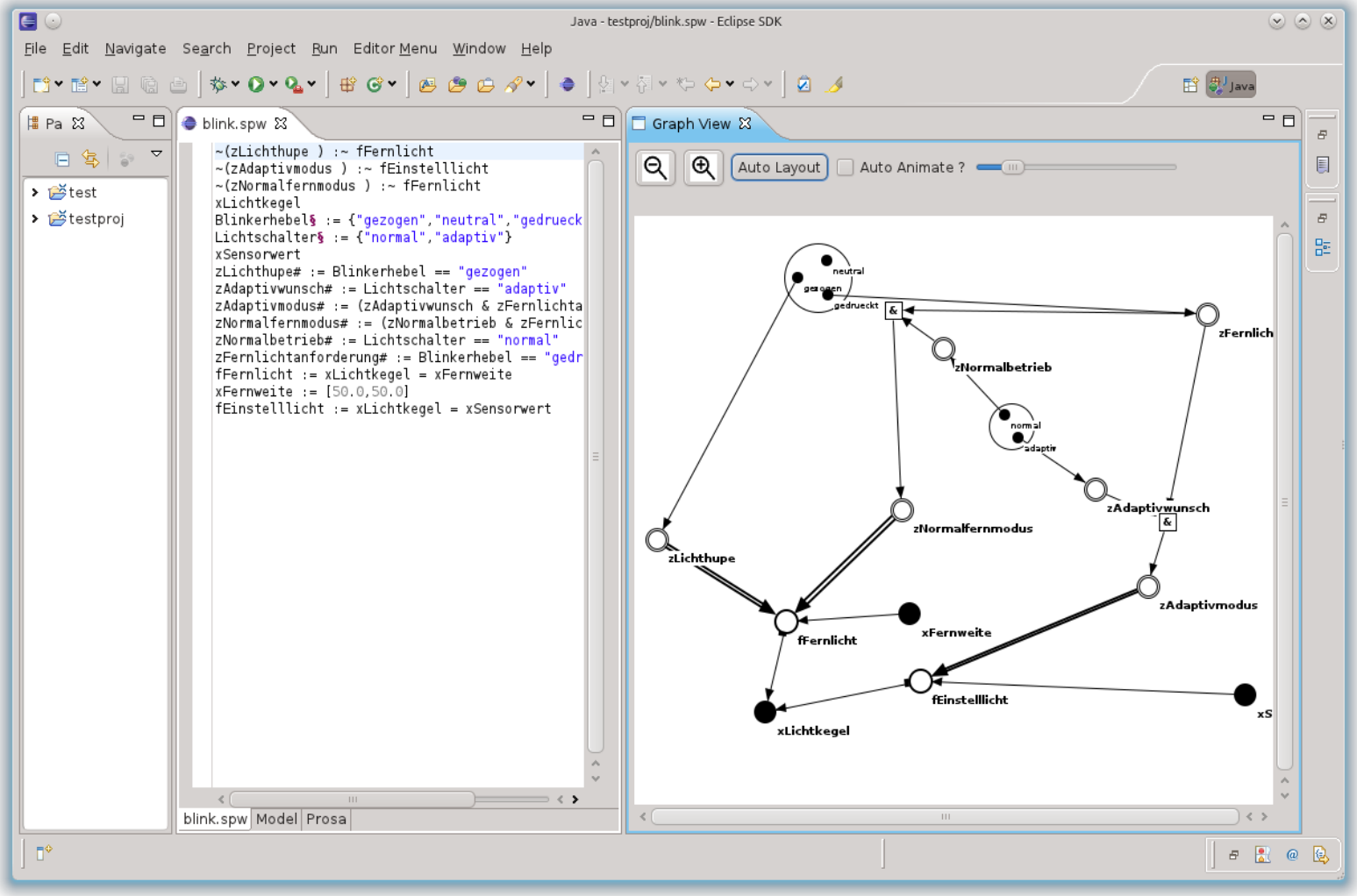


Abbildung 4.1: Übersicht über die Anwendungsoberfläche

4.1.2 View

In der Viewkomponente wird der Graph als Node-Link-Diagramm dargestellt. Hier lässt sich der Graph interaktiv manipulieren. Die Viewkomponente besteht aus mehreren Schaltflächen und einem Canvas, auf dem der Graph dargestellt wird (Abb. 4.2).

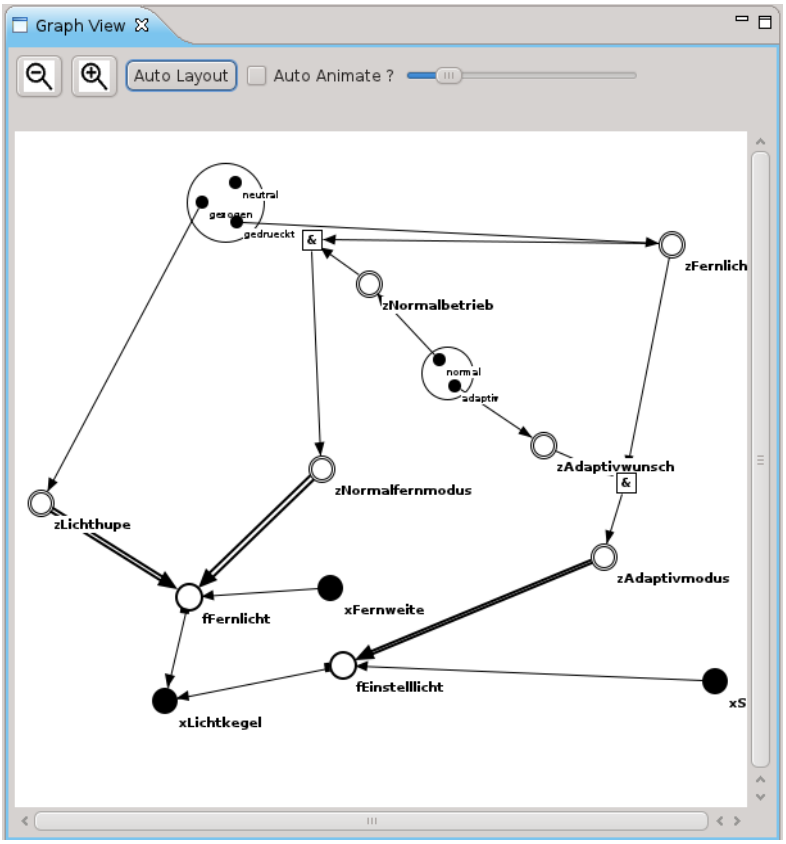
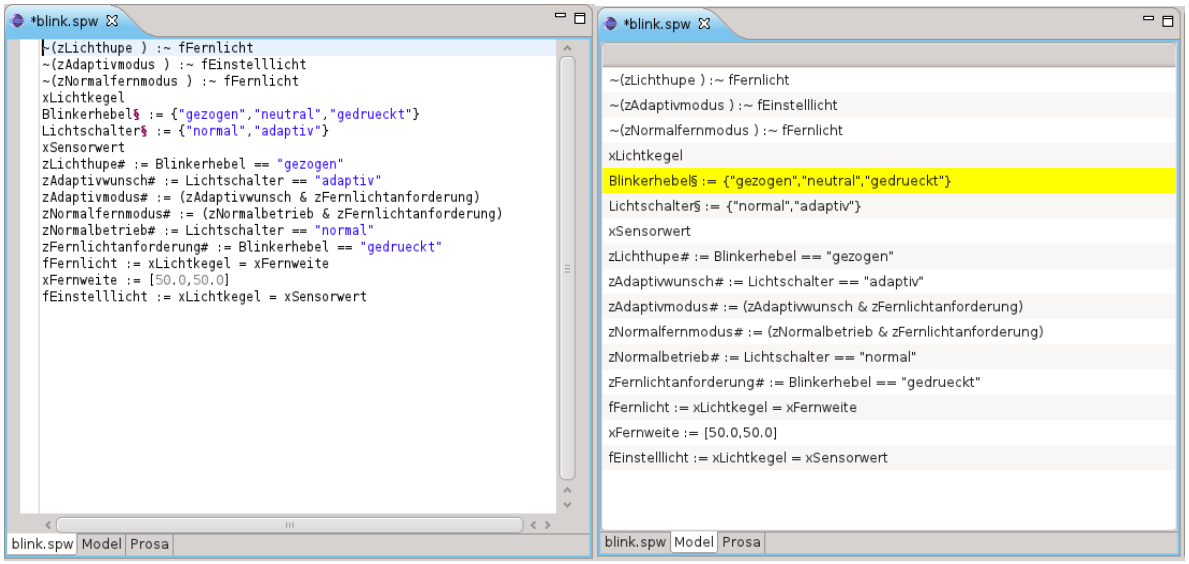
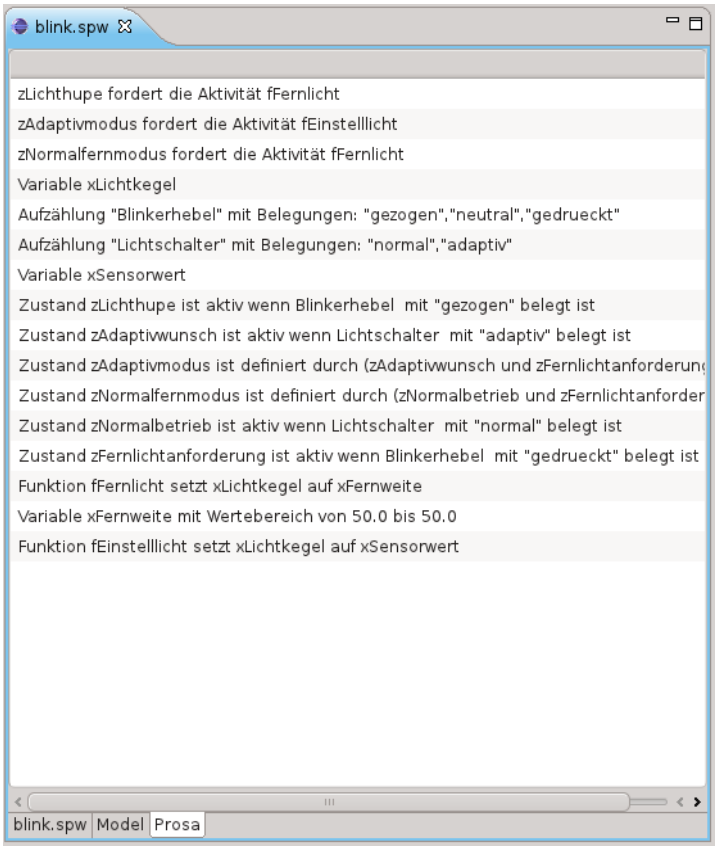


Abbildung 4.2: Graph-View Ausschnitt: Die Graph-View-Komponente enthält das Canvas und die entsprechenen Steuerelemente für Zoom, Auführen des Layoutalgorithmus und Animationsgeschwindigkeit



(a) Dateiansicht **(b)** tabellarische Ansicht



(c) Prosa Ansicht

Abbildung 4.3: Beispielansichten Multipageeditor

4.2 Datenmodell

Die Editorkomponente und die Viewkomponente nutzen das gleiche Datenmodell, das in einer Containerklasse gespeichert wird. Eine Instanz dieser Containerklasse wird in der entsprechenden Editorinstanz gespeichert. Da die Viewkomponente immer nur Daten einer Datei anzeigen kann, bezieht sie ihre Daten immer aus der Editorinstanz, die gerade im Fokus ist. Das Datenmodell basiert auf zwei unterschiedlichen Basisklassen, die jeweils erweitert werden: Zum einen **GraphObject**, welche die Objekte des Graphen (Knoten und Kanten) repräsentiert und entsprechende Zeichenfunktionen beinhalten, zum anderen **SpecFact**, deren Instanzen je einen Fakt des Faktengraphen repräsentieren. Diese Instanzen enthalten eine oder mehrere Instanzen der GraphObject-Klasse.

4.2.1 Klassenbeschreibung

SpecEditor

Diese Klasse erweitert die Eclipseklasse „MultiEditorPart“, welche einen mehrseitigen Editor ermöglicht. Diese enthält einerseits eine Instanz des XtextEditors, die das Xtext-Datenmodell der geöffneten Datei enthält, zum anderen enthält sie eine Instanz der „GraphContainer“-Klasse.

4.2.2 GraphView

Diese Klasse stellt eine Eclipse-View Komponente zur Verfügung, die, wenn ein GraphEditor geöffnet ist, den dort geöffneten Graph als Node-Link-Diagramm darstellt. Zum Visualisieren des Graphen wird eine GraphCanvas-Instanz verwendet, deren Interaktionsmethoden mit Schaltflächen des GraphView verknüpft werden. (Abb. 4.2)

4.2.3 GraphCanvas

GraphCanvas stellt ein SWT-Canvas zur Verfügung, auf dem ein Graph aus einer Graphcontainer-Klasse in Abhängigkeit vom gewählten Bildausschnitt und Zoomfaktor dargestellt werden kann. Neben der reinen Darstellung bietet diese Klasse auch interaktive Features, um beispielsweise Objekte an der Stelle des Mauszeigers zu bestimmen. Weiterhin stellt diese Klasse einen Animationsthread zur Verfügung, der die Objekte im GraphContainer animiert, indem er den nächsten Animationsschritt aufruft.

Graphcontainer

In dieser Klasse wird der gesamte Faktengraph gespeichert und verarbeitet.

Gespeicherte Objekte:

- ArrayList<SpecFact> **facts**: Liste mit den einzelnen Fakten.
- ArrayList<GraphNode> **NodeList**: Adjanzenzliste mit Knoten des darzustellenden Graphen.
- ArrayList<GraphEdge> **EdgeList**: Liste mit Kanten des darzustellenden Graphen.
- ArrayList<GraphObject> **animQueue**: Liste der Graphobjekte, die momentan animiert werden.
- ArrayList<GraphObject> **selectedObjects**: Liste der ausgewählten Objekte die hervorgehoben werden.
- Rectangle **bounds**: Speichert die aktuelle Fläche der von Objekten belegten Leinwand (repräsentiert durch ein Rechteck).

Wichtige Methoden:

- public boolean **runAnimationStep**(double speed): Wird vom Animationsthread aufgerufen und berechnet einen Animationsschritt der Objekte in der animQueue.
- public void **updateFromFile**(Model m, PropertiesFile propfile): Aktualisiert den Faktengraph aus dem Datenmodell eines XtextEditors.
- public String **getFileContent**(): Erzeugt aus dem Faktengraph den Inhalt der SpecWorks-Datei.
- public String **getPropertiesFileContents**(): Erzeugt den Dateiinhalt für die *.prop Datei.
- public float **overallnodedistance()**: Berechnet die Distanz aller Knoten auf der Leinwand zueinander.

SpecFact

Diese abstrakte Klasse repräsentiert je einen Fakt des Faktengraphen. Dafür wird sie für die verschiedenen Fakten jeweils durch eine Kindklasse erweitert.

Wichtige Methoden:

- public String **getFactString**(): Liefert die formale Regel des Fakts in Textform
- public String **getProsaString**(): Liefert eine textuelle Beschreibung des Fakts in Prosaform.
- public ArrayList<GraphObject> **getGraphObjectObjects**(): Gibt alle Graphobjekte zurück, die diesen Fakt repräsentieren.

- public ArrayList<GraphNode> **getGraphNodeObjects**(): Gibt alle Knotenobjekte zurück, die diesen Fakt repräsentieren.
- public ArrayList<GraphEdge> **getGraphEdgeObjects**(): Gibt alle Kantenobjekte zurück, die diesen Fakt repräsentieren.
- public void **parse**(EObject o): Extrahiert aus dem übergebenen Objekt die Daten. „o“ ist eine Instanz aus der Xtext Repräsentation.
- public void **secondParse**(): Verknüpft die Graphelemente, die in diesem Fakt enthalten sind, mit Graphelementen aus anderen Fakten.

GraphObject

Diese Basisklasse repräsentiert ein Objekt (Knoten oder Kante) eines Graphen und bietet entsprechende Methoden, um diese zu zeichnen. Sie wird durch Kindklassen (**GraphNode**, **GraphMultiNode**, **GraphEdge**) erweitert, welche die objektspezifischen Methoden hinzufügen. Von diesen werden wiederum die konkreten Objekte des Faktengraphen abgleitet.
Wichtige Methoden:

- abstract public void **draw**(GC gc, Point canvasPos, float zoom): Zeichnet das Objekt auf dem grafischen Kontext gc, der die Leinwand repräsentiert. Der Ausschnitt der Leinwand befindet sich an Position „canvasPos“ und hat den den Zoomfaktor „zoom“.
- abstract public void **drawClick**(GC gc, Point canvasPos, float zoom, Color color): Entspricht der Methode **draw**, zeichnet aber die Repräsentation für die Klickerkennung (siehe: 4.5).
- public abstract Rectangle **getBounds**(): Gibt den Bereich zurück, in dem das Objekt auf der Leinwand liegt.
- public abstract String **getToolTipText**(): gibt einen Text für einen Tooltip bei MouseOver zurück. Gibt es keinen Text, so wird null zurückgegeben.
- public abstract boolean **animationStep**(double speed): Führt einen Animationsschritt für das Objekt durch. Der Parameter „speed“ gibt an, wie schnell die Animation abläuft. Der Rückgabewert gibt an, ob es einen weiteren Animationsschritt gibt.
- public Menu **getContextMenu**(Control parent): Gibt ein Kontextmenü zurück, das erscheinen soll, wenn auf das Objekt ein Rechtsklick ausgeführt wird.

GraphNode

Diese Klasse ist abgeleitet von **GraphObject** und repräsentiert einen Knoten eines Graphen.
Zusätzliche wichtige Methoden:

- public void **addIncomingEdge**(GraphEdge e): Teilt dem Knoten mit, welche Kante auf ihn zeigt.

- public void **addOutgoingEdge**(GraphEdge e): Teilt dem Knoten mit, welche Kante von ihm ausgeht.
- public void **delIncomingEdge**(GraphEdge e): Teilt dem Knoten mit, welche eingehende Kante entfernt wurde.
- public void **delOutgoingEdge**(GraphEdge e): Teilt dem Knoten mit, welche ausgehende Kante entfernt wurde.

GraphMultiNode

Diese Klasse ist von GraphNode abgeleitet und repräsentiert einen Multiknoten. Ein Multiknoten ist ein Knoten, der mehrere Kindknoten besitzt. Dies wurde z.B. verwendet, um Enumerationen (siehe Abb. 3.3c) zu visualisieren

Zusätzliche wichtige Methoden:

- public void **setChildCount**(int n): Setzt die Anzahl der Kindknoten.
- public GraphNode **getChild**(int index): Gibt den n-ten Kindknoten zurück.

GraphEdge

Diese Klasse ist abgeleitet von **GraphObject** und repräsentiert eine Kante eines Graphen.

Zusätzliche wichtige Methoden:

- public boolean **setStart**(GraphNode s): Setzt den Startknoten der Kante. Der Rückgabewert gibt an, ob bei der Kante vorher schon ein Startknoten gesetzt war.
- public boolean **setEnd**(GraphNode s): Setzt den Zielknoten der Kante. Der Rückgabewert gibt an, ob bei der Kante vorher schon ein Zielknoten gesetzt war.
- public GraphNode **getStart**(): Gibt den Startknoten zurück.
- public GraphNode **getEnd**(): Gibt den Zielknoten zurück.
- public float **getWeightFactor**(): gibt einen Faktor zurück, der bestimmt ob der Layoutalgorithmus diese Kante länger oder kürzer macht.

CanvasObject

Diese abstrakte Klasse dient als Grundlage für grafische Objekte, die auf einer Leinwand gezeichnet werden. Dazu zählen einfache geometrische Formen wie Kreise, Quadrate, Linien und komplexere Objekte wie beispielsweise Pfeile. Diese Klasse dient hauptsächlich dazu mehrere Zeichenaufrufe in einer Funktion zu bündeln, damit komplexere Objekte wie Pfeile nicht in jeder Klasse neu konstruiert werden müssen.

4.2.4 PropertiesFile

Diese Klasse verwaltet eine zusätzliche Datei (Endung „.prop“), die Eigenschaften des Faktengraphen speichern kann, welche nicht in den SpecWorks Dateien gespeichert werden.

4.2.5 FruchtermanLayout

Diese Klasse kann auf einen gegebenen Graphen das Layoutverfahren nach Fruchterman/Reingold anwenden [FR91]. Dies wurde für diesen konkreten Anwendungsfall etwas modifiziert. Die konkrete Funktionsweise ist in 4.4.1 erklärt.

wichtige Methoden:

- public void **iterate**(): Führt eine Iteration durch.
- public void **start**(double threshold): Startet den Layoutalgorithmus. Der Parameter „threshold“ gibt an, wie groß die Verbesserung pro Iterationsschritt minimal sein muss, bevor der Algorithmus terminiert. Die Verbesserung ist bestimmt durch die Differenz der Knotendistanzen zwischen zwei Interationsschritten.

4.3 Xtext-Klassen

Xtext bietet eine Vielzahl von Klassen, von denen hier die wichtigsten verwendeten vorgestellt werden sollen. Dabei gibt es zwei Arten von Klassen. Einerseits die Klassen, welche direkt von Xtext bereitgestellt werden und andererseits Klassen, die Xtext abhängig von der verwendeten DSL generiert.

- **XtextEditor** ist eine Editorkomponente, die genutzt wird um erzeugten DSLs einen Editor zur Verfügung zu stellen.
- **EObject** ist die Basisklasse auf der Objekte, die aus einer geparsten Datei stammen, bestehen

4.3.1 Parser

Unter den vielen Parsergeneratoren, die es für Java gibt, wurde Xtext [xte12] ausgewählt, um die Dateien mit der Regelmenge zu parsen. Einerseits war aus [Som11] schon die Grammatik für den Faktengraphen für Xtext gegeben und andererseits bietet Xtext eine gute Integration in Eclipse. So erzeugt Xtext eine Editorkomponente für Eclipse, die schon einige übliche Features wie Syntaxhighlighting oder Codefolding beherrscht.

4.4 Darstellung

4.4.1 Layoutverfahren

Wird eine Datei geladen, so werden die Knoten zufällig auf der Zeichenfläche platziert. Neu angelegte Knoten des Benutzers werden, je nachdem in welcher Ansicht sie erzeugt wurden, zufällig oder an einer benutzerdefinierten Position erzeugt. Um die Knoten automatisch sinnvoll anzuordnen, wurde eine modifizierte Version des Algorithmus von Fruchterman/Reingold [FR91] verwendet (Alg. 4.1).

Algorithmus 4.1 Force-directed placement nach Fruchterman/Reingold [FR91]

```
    area := w * L
    G := V, E
    function f_a(z)
        return x^2/k;
 5: end function
    function f_r(z)
        return k^2/z;
    end function
    for i := 1 to iterations do                          // abstoßende Kräfte
10:     for v in V do
            v.disp := 0;
            for u in V do
                if v ≠ u then
                    Δ := v.pos - u.pos
15:                 v.disp := v.disp + ( Δ/|Δ|)*f_r(|Delta|)
                end if
            end for
        end for
        for e in E do                                    // anziehende Kräfte
20:         Delta := e.v.pos - e.u.pos;
            e.v.disp := e.v.disp - ( Delta/|Delta|)*f_a(|Delta|);
            e.u.disp := e.u.disp + ( Delta/|Delta|)*f_a(|Delta|);
        end for
        for v in V do
25:         v.pos := ( v.disp/|v.disp|)*min(v.disp,t);
            v.pos.x := min( W/2, max(−W/2, v.pos.x));
            v.pos.y := min( L/2, max(−L/2, v.pos.y));
        end for
        t := cool(t);
30: end for
```

Es wurden folgende Modifikationen vorgenommen:

- Änderung der Begrenzung der Zeichenfläche: Der Algorithmus von Fruchterman/Reingold geht von einer rechteckigen Zeichenfläche aus, deren linke obere Ecke (0,0) ist und nach rechts unten begrenzt ist. Im SpecGui-Tool ist die Zeichenfläche in alle Richtungen unbegrenzt (außer durch die Größe von Integer) und setzt sich nach links und oben mit negativen Koordinatenwerten fort.
- Die Größe der Zeichenfläche ändert sich dynamisch: Da die Zeichenfläche unbegrenzt ist, würden die Knoten entweder immer weiter auseinander driften, da der Originalalgorithmus versucht, die Knoten gleichmäßig zu verteilen, oder im Falle einer für den Algorithmus begrenzten Fläche zu nah aneinander hängen. Deshalb wird eine künstliche Begrenzung der Zeichenfläche dynamisch anhand des Verhältnisses zwischen Knoten und der zur Verfügung stehenden Fläche eingeführt.
- Um eine kontinuierliche Animation der Knoten zu ermöglichen, operiert der Algorithmus nicht auf den echten Koordinatenwerten, sondern auf einer Kopie, welche nach Berechnungsende für eine Animation als Zielposition verwendet wird.
- Unterschiedlicher Knotenabstand: Einige Knoten gehören thematisch stark zueinander. Der Abstand dieser Knoten soll deshalb kleiner sein.

```
public class Fruchterman{

	GraphContainer container;

	public Fruchterman(GraphContainer container){
		this.container = container;
	}
	double t = 10;
	double k;
	@Override
	public void iterate(){
		Rectangle bounds = container.getBounds();
		for(GraphNode v : container.getAllVisNodes()){
			v.setDisp(0,0);
		}
		for(GraphNode v : container.getAllNodes()){
				for(GraphNode u : container.getAllNodes()){
				if	(!u.equals(v)){
					int dx = v.getPos().x - u.getAPos().x;
					int dy = v.getPos().y - u.getPos().y;
					double l = Math.sqrt(dx*dx+dy*dy);
					if(l<1){
						v.addDisp(10,10);
						u.addDisp(-10,-10);
					}else{
						v.addDisp((dx/l)*fr(l),(dy/l)*fr(l));
					}
				}
			}
		}
```

```
                for(GraphEdge e : container.getAllEdges()){
                        int dx = e.getEnd().getPos().x - e.getStart().getPos().x;
                        int dy = e.getEnd().getPos().y - e.getStart().getPos().y;
                        float w = e.getWeightFactor();
                        double l = Math.sqrt(dx*dx+dy*dy);
                        if(l<0.001)l=0.0000001;
                        e.getEnd().addDisp(- (dx/l)*fa(l)*w,- (dy/l)*fa(l)*w);
                        e.getStart().addDisp( (dx/l)*fa(l)*w, (dy/l)*fa(l)*w);
                }

                for(GraphNode v : container.getAllNodes()){
                        double l = Math.sqrt(v.getDispX()*v.getDispX()
                            +v.getDispY()*v.getDispY());
                        double x = v.getPos().x + ((
                            v.getDispX()/l)*Math.min(Math.abs(v.getDispX()), t));
                        double y = v.getPos().y + ((
                            v.getDispY()/l)*Math.min(Math.abs(v.getDispY()), t));

                        x = Math.min(bounds.x+bounds.width*1.1,
                            Math.max(bounds.x-bounds.width*0.1, x));
                        y = Math.min(bounds.y+bounds.height*1.1,
                            Math.max(bounds.y-bounds.height*0.1, y));
                        v.setPos((int)x, (int)y);
                }
                t=t/1.5;
        }

        private double fa(double z){
                return z*z/k;
        }

        private double fr(double z){
                return k*k/z;
        }
        @Override
        public void start() {
                Rectangle bounds = container.getBounds();
                k= Math.sqrt(bounds.height*bounds.width/container.getAllVisNodes().size());
                float last = -10000;
                float current= 0;
                current = container.overallnodedistance();
                int count = 0;
                while(Math.abs(last-current)>10){
                        iterate();
                        count++;
                        last =current;
                        current = container.overallnodedistance();
                }
        }
}
```

Listing 4.1: modifizierter Fruchterman/Reingold-Algorithmus, vereinfacht

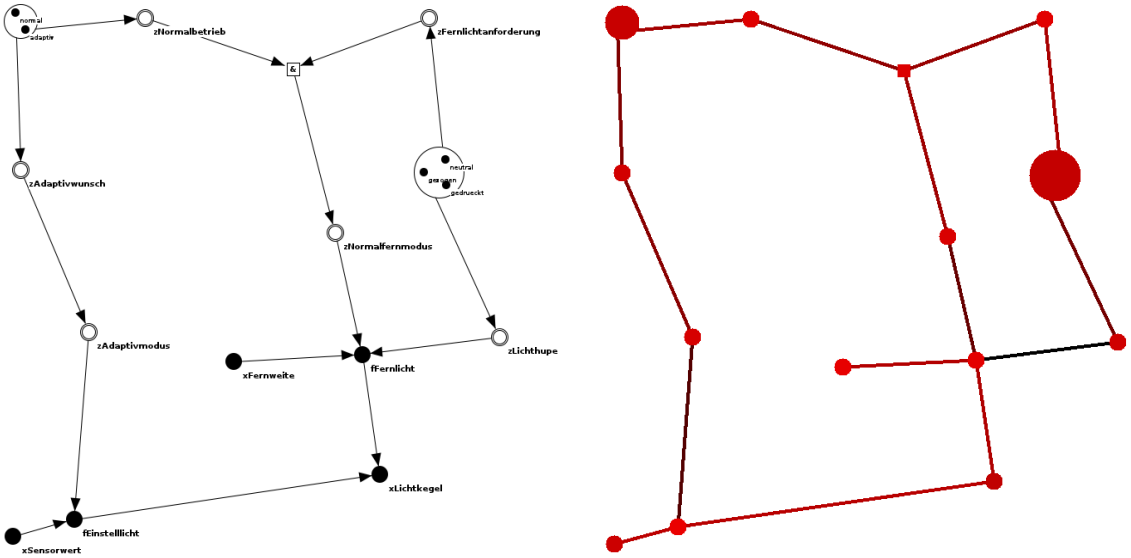
4.5 Klickerkennung

Um die grafische Ansicht des Faktengraph interaktiv bearbeiten zu können, müssen Elemente auf der Zeichenfläche mit dem Mauszeiger anklickbar sein. Mit Boardmitteln des SWT-Toolkits ist dies nur auf Basis von Widgets möglich. Da hier die Visualisierung aber nur auf einem Canvas erfolgt, kann nicht zwischen einzelnen gezeichneten Elementen unterschieden werden, da SWT das Canvas nur als ein ganzes Objekt erkennt. Um zu erkennen, welches Objekt angeklickt wurde bzw. über welchem Objekt der Mauszeiger schwebt, muss eine Klickerkennung implementiert werden. Hierfür gibt es zwei Möglichkeiten:

- Vergleich jedes Objekts mit den Mauskoordinaten, ob diese mit dem Objekt übereinstimmen: Dies ist allerdings bei vielen Objekten recht rechenintensiv, da alle Objekte verglichen werden müssen. Zudem muss für jedes Objekt eine berechenbare Vergleichsfläche gefunden werden, was für Quadrate und Kreise einfach ist, aber z.B. für Pfeile schon schwieriger wird. Es ist zwar möglich durch geschickte Datenstrukturen wie Quadtrees die Anzahl der Vergleiche zu verringern, dies bringt jedoch wieder neue Nachteile mit sich, z.B. wie Objekte behandelt werden sollen, die sich über mehrere Quadranten erstrecken. Zudem muss ein Quadtree bei jeder Positionsänderung eines Objekts aktualisiert werden.
- Erzeugung eines internen Bildes mit Farbkodierung der Objekte: Die Objekte werden zusätzlich zur normalen Darstellung in ein zweites Bild gezeichnet, wobei jedem Objekt eine eindeutige Farbe zugeordnet wird, in der es gezeichnet wird. Auf diese Weise kann durch einfaches Nachschlagen der Pixelfarbe überprüft werden, welches Objekt sich an dieser Stelle befindet. Für das Mapping von Farbe zu Objekt wird der Farbwert direkt als Index für ein Array verwendet, in dem die Objekte gespeichert sind. Für die Zeichnung kann die normale Zeichenfunktion verwendet werden. Es muss nur die Zeichenfarbe festgelegt und Weichzeichner oder Antialiasingfunktionen deaktiviert werden, um die Farbwerte nicht zu verfälschen. Lediglich Objekte, die z.B. durch einen Kreisring repräsentiert werden, müssen durch einen gefüllten Kreis ersetzt werden, da auch das Kreisinnere dem Objekt zuzuordnen ist. Bei dieser Methode muss bei einer Änderung eines Objekts der ganze aktuelle Bildausschnitt neu gezeichnet werden, jedoch nur die Objekte, die momentan sichtbar sind. Genauso muss bei Veränderung des Bildausschnitts oder der Zoomstufe das interne Bild neu berechnet werden. Dafür können in konstanter Zeit Objekte anhand von Koordinaten gefunden werden. Abbildung 4.4 zeigt die normale Ansicht eines Graphen, die ein Benutzter sieht neben dem internen Bild zur Objekterkennung. Dies wird zur Verdeutlichung in Falschfarben dargestellt, da die Farbwerte sich eigentlich nur um wenige Farbstufen unterscheiden.

4.6 Bufferimage

Bei vielen Objekten kann es aufgrund des Zeichenaufwands zu Ruckeln kommen. Solange der Bildausschnitt nicht bewegt wird, ist meistens trotzdem ein flüssiges Arbeiten möglich,



(a) Normale Ansicht **(b)** Internes Klickimage

Abbildung 4.4: Internes Bild zur Klickerkennung

da interaktive Features wie Mauserkennung, unabhängig von der Objektanzahl sind. Wird der Bildausschnitt aber verändert, machen sich viele Objekte bemerkbar. Um ein flüssiges Scrollen und Zoomen zu ermöglichen, wird, sobald vom Benutzer keine Eingabe kommt, intern in ein Bufferimage der gesamten Leinwand gezeichnet. Beim Scrollen kann so direkt auf dieses zugegriffen werden und die Bewegung erscheint flüssig. Das Bild muss nur neu berechnet werden, wenn sich Objekte verändern. Meistens fällt dies dem Benutzer gar nicht auf, da die Berechnung selbst bei 1000 Objekten in weniger als einer Sekunde erfolgt. (siehe auch Kapitel „5 Performance“)

5 Performance

Bei einer interaktiven Anwendung ist die Reaktionsgeschwindigkeit der Anwendung ein entscheidendes Kriterium für die Akzeptanz des Benutzers.
Überprüft wurde hier wie lange es braucht um die Leinwand zu zeichnen, da dies die Zeit bestimmt, die der Nutzer auf seine Eingaben warten muss bis diese eine Aktion auslösen. Außerdem bestimmt sie wie flüssig Animationen ausgeführt werden können. Der Test wurde auf folgendem System durchgeführt:

- CPU: Core2Duo E6750
- RAM: 8GB
- Grafik: Nvidia GeForce 8600 GT
- Betriebssystem: Ubuntu Linux
- Eclipse: 3.7 Indigo
- Java: OpenJDK 6

Überprüft wurde, wie lange es dauert um einen Graphen einmal komplett zu zeichnen. Dies wurde mit Graphen unterschiedlicher Größe getestet. Alle Graphen bestehen aus gleich vielen Knoten mit einer Beschriftung und Kanten. Die angegebene Objektzahl ist die Anzahl der Kanten und Knoten.

Ab ca 200 Objekten fällt auf, dass die Zeichenfläche nicht mehr flüssig gezeichnet wird. Bis zu 400 Objekten ist aber noch ein flüssiges Arbeiten möglich. Ab etwa 1500 Knoten ist die Grenze erreicht in der Arbeiten an einzelnen Objekten nur noch mit großer Anstrengung möglich ist. Das verschieben des Bildauschnitts hingegen läuft aufgrund des Bufferimage **??** flüssig.

Bei den Performancetests fiel auf, dass nur sichtbare Objekte zu einer Verlangsamung beitragen. Offensichtlich kann SWT hier optimieren und ignoriert Zeichenanweisungen die außerhalb des sichtbaren Canvas liegen. Da die Zeit, die benötigt wird um durch alle Objekte zu iterieren, fast keinen Einfluss auf die Renderinggeschwindigkeit hat, da das Rendern eines Objekts viel mehr Zeit benötigt als ein Iterationsschritt durch die Renderingqueue, wurde auf Optimierungen wie beispielsweise Quadtrees in den Datenstrukturen verzichtet.

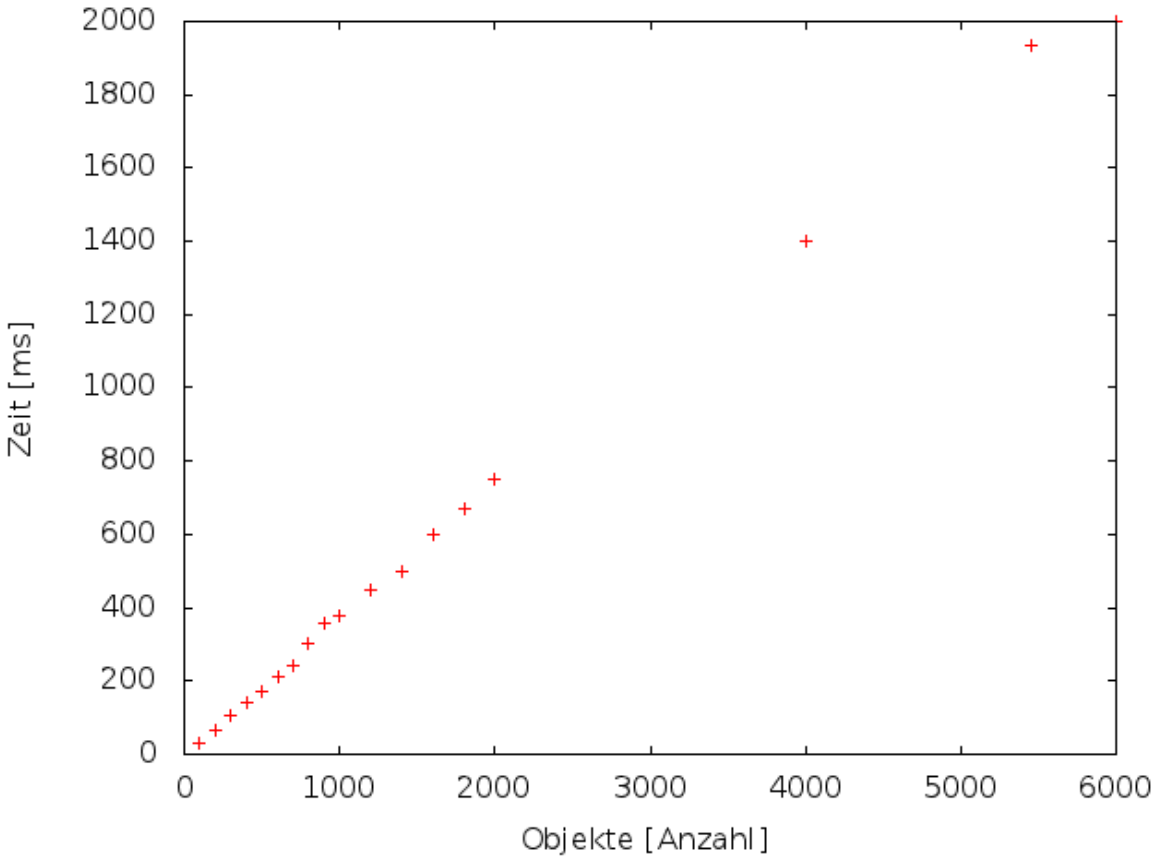


Abbildung 5.1: Renderingbenchmark: benötigte Zeit um Objekte zu zeichnen

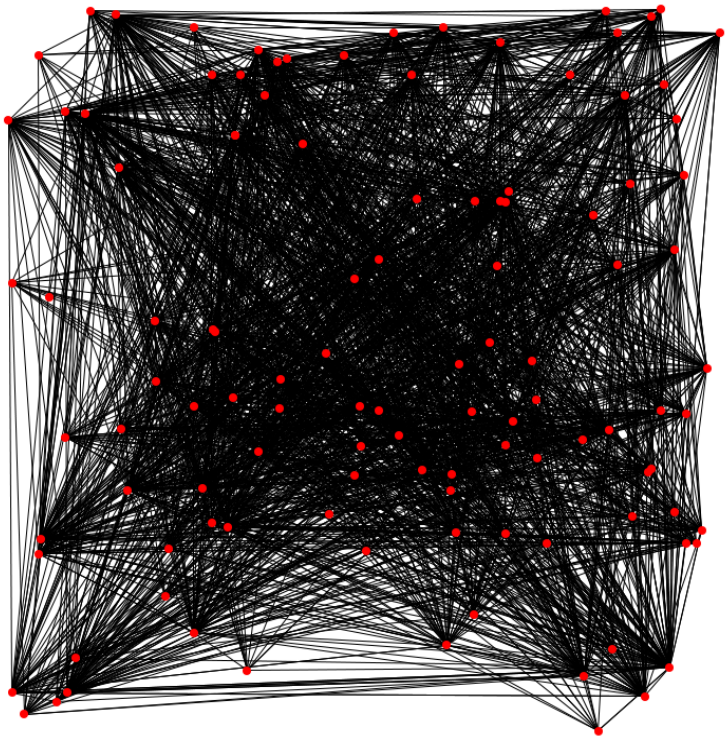
6 Weitere Anwendungsmöglichkeiten

Das hier vorgestellte SpecGui-Plugin eignet sich nicht nur für die Bearbeitung von SpecWorks-Dateien. Mit wenigen Änderungen lassen sich auch Plugins mit Multiple-Coordinated-View für andere textuelle Graphenbeschreibungen erstellen.
Um ein Plugin für ein anderes Anwendungsgebiet zu erstellen, ist es nur nötig die Beschreibung der Daten für den Xtextparsergenerator zu erstellen und mit Hilfe von Kindklassen der Klassen GraphNode und GraphEdge eine Zuordnung der Elemente in der Beschreibungsdatei zu Graphelementen zu erstellen.

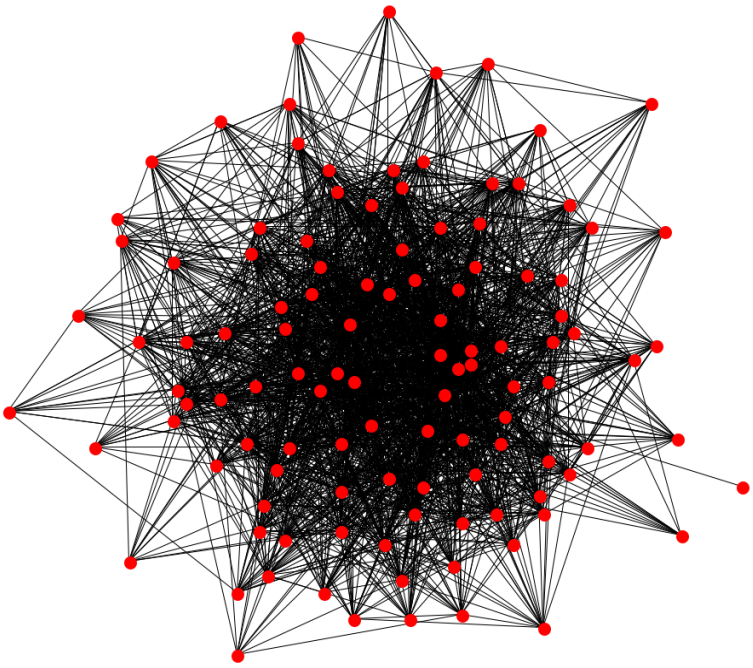
Eine Anwendungsmöglichkeit ist beispielsweise das Darstellen von sozialen Netzwerken. Die Beschreibungsdatei stellt in jeder Zeile eine Interaktion zwischen zwei Personen dar. Jede Zeile ist nach dem Muster „<StartKnoten-ID> <ZielKnoten-ID> <Gewicht>“ aufgebaut. Dies lässt sich mit Xtext mit den Reglen in Listing 6.1 beschreiben. Das daraus resultierende Xtext-Modell besteht aus den Klassen „Model“, „Dialog“ und „Value“ aus denen einfach die entsprechenden Knoten und Kanten erzeugt werden können. Der so erstellte Graph lässt sich dann mit Hilfe des Fruchterman/Reingold-Algorithmus (Abb. 6.1), aber auch von Hand (Abb. 6.2) layouten.

```
grammar de.kb256.xtext.social.Social with org.eclipse.xtext.common.Terminals
generate social "http://www.kb256.de/xtext/social/Social"
Model:
       dialogs+=Dialog*;
Dialog:
       from=INT ' ' to=INT ' ' weight=Value;
Value:
       pre=INT '.' post=INT;
```

Listing 6.1: Xtext-Beschreibung für soziale Graphen



(a) Randomlayout



(b) berechnetes Layout

Abbildung 6.1: Vergleich Randomlayout und berechnetes Layout: (a) zeigt den Graph eines sozialen Netzwerks in Form eines Random-Layouts. (b) zeigt den gleichen Graphen, der mit dem Fruchterman/Reingold-Alogrithmus gelayoutet wurde.

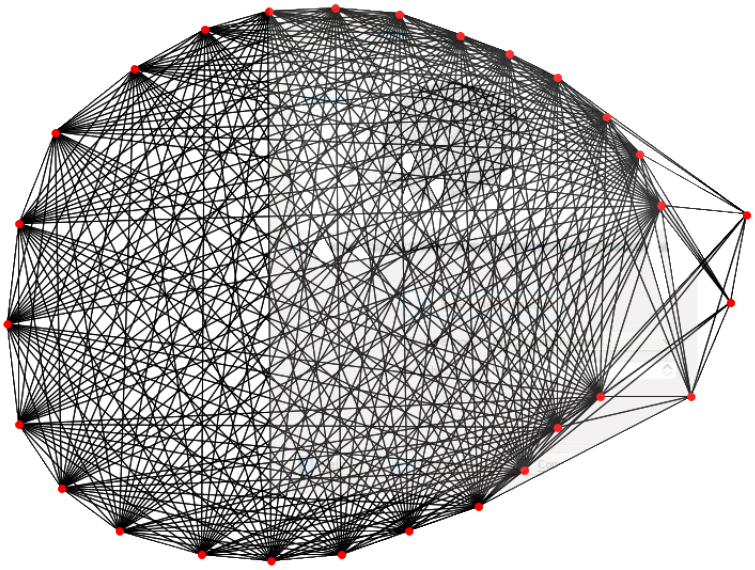


Abbildung 6.2: Visualisierung eines von Hand gelayouteten sozialen Netzwerks.

7 Zusammenfassung und Ausblick

Dieses Kapitel fasst Erkenntnisse dieser Arbeit zusammen und zeigt Möglichkeiten zur Weiterentwicklung.

Zusammenfassung

In dieser Arbeit wurde das Plugin „SpecGui“ für Eclipse vorgestellt. Mit diesem Plugin lassen sich textuelle Regelmengen des Faktengraphen grafisch und interaktiv bearbeiten. Es bietet eine Multiple-Coordinated-View mit der sich der Faktengraph gleichzeitig in seiner textuellen Repräsentation in Form von Regeln, sowie in der grafischen Visualisierung als Node-Link-Diagramm, anzeigen und interaktiv bearbeiten lässt. Dabei sind Änderungen in der einen Ansicht sofort in der anderen sichtbar. Die grafische Visualisierung bietet dem Benutzer die Möglichkeit Elemente sowohl selber nach eigenen Kriterien anzuordnen, als auch automatisch zu layouten.

Der Unterbau des Plugins ist sehr generisch gehalten, so dass es recht einfach ist, dies auch für andere Anwendungen, in denen Graphen durch textuelle Regeln beschrieben werden, zu nutzen.

Ausblick

Diese Arbeit beschäftigt sich mit der Visualisierung und interaktiven Bearbeitung des Faktengraphen. Es gibt eine Vielzahl von Möglichkeiten dies zu erweitern und zu verbessern. Ein möglicher Ansatzpunkt ist das Layouting. Hier kann der Fruchterman/Reingold-Algorithmus [FR91] mit weiteren Layoutverfahren verglichen werden um gegebenenfalls ein besseres automatisches Layoutverfahren zu erhalten. Um die Einstiegshürden bei der Benutzung des Faktengraph weiter zu senken wäre ein Wizardfunktion sinnvoll, die den Benutzer Schritt für Schritt durch die Erstellung eines Modells leitet und dabei den erstellten Graph auf semantische Fehler überprüft um Lösungsvorschläge aufzuzeigen. Auch sind andere Visualisierungsarten und damit verbundene interaktive Bearbeitungskonzepte möglich, die in einer Benutzerstudie gegeneinander evaluiert werden können.

Die Visualisierung und Bearbeitung des Faktengraphen ist nur eine Aufgabe im Umgang mit Anforderungsspezifikationen. Hier gibt es eine Vielzahl von Möglichkeiten die Anwendung

mit weiteren Features zu erweitern: Eine Möglichkeit ist das aus dem Faktengraph erstellte Modell mit Testfällen automatisiert zu überprüfen, sowie aus dem Modell Testfälle zu erstellen um das entatandene System zu testen. Weiter denkbar im Rahmen der modellgetriebenen Entwicklung ist die Codeerzeugung.

Literaturverzeichnis

[Ecl12] Eclipse. About Eclipse, 2012. URL http://www.eclipse.org/org. (Zitiert auf Seite 15)

[ELMS91] P. Eades, W. Lai, K. Misue, K. Sugiyama. Preserving the Mental Map of a Diagram. *Proceedings of COMPUGRAPHICS*, 91(9):24–33, 1991. (Zitiert auf Seite 20)

[FR91] T. M. J. Fruchterman, E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21:1129–1164, 1991. doi:10.1002/spe.4380211102. (Zitiert auf den Seiten 6, 13, 43, 44 und 55)

[GFC04] M. Ghoniem, J.-D. Fekete, P. Castagliola. A Comparison of the Readability of Graphs Using Node-Link and Matrix-Based Representations. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pp. 17–24. IEEE Computer Society, Washington, DC, USA, 2004. doi:10.1109/INFOVIS.2004.1. (Zitiert auf Seite 19)

[HIF10] D. Holten, P. Isenberg, J. Fekete. Performance Evaluation of Tapered , Curved , and Animated Directed-Edge Representations in Node-Link Graphs. *Evaluation*, 2010. (Zitiert auf Seite 13)

[Pur02] H. C. Purchase. Metrics for Graph Drawing Aesthetics. *J. Visual Lang. and Comp.*, 13(5):501–516, 2002. (Zitiert auf Seite 13)

[Rob07] J. Roberts. State of the Art: Coordinated Multiple Views in Exploratory Visualization. In *Coordinated and Multiple Views in Exploratory Visualization, 2007. CMV '07. Fifth International Conference on*, pp. 61 –71. 2007. doi:10.1109/CMV.2007.20. (Zitiert auf Seite 14)

[Shn96] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pp. 336–. IEEE Computer Society, Washington, DC, USA, 1996. (Zitiert auf Seite 14)

[Som11] R. Sommer. *Automatisierte Formalisierung von Anforderungen an eingebettete Systeme im Automobilbau*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011. (Zitiert auf den Seiten 9, 11, 20, 32 und 43)

[xte12] Xtext, 2012. URL http://www.eclipse.org/Xtext/. (Zitiert auf Seite 43)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Andreas Paul)