Diplomarbeit

Lehrstuhl Prof. Tichy
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe (TH)

Erweiterung des Kantenkonzepts deklarativer Graphersetzungssysteme von Einfachkanten über Hyperkanten zu „Superkanten“

Oliver Denninger

Betreuer
Prof. Walter F. Tichy
Tom Gelhausen

Karlsruhe, 29. März 2007

Erklärung der Selbständigkeit

Hiermit erkläre ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Alle nicht von mir stammenden Inhalte sind durch Angabe der Quelle kenntlich gemacht.

Karlsruhe, den 29. März 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Aktuelle Modelltransformationswerkzeuge können nur mit einfachen, gerichteten Kanten umgehen, für Hyperkanten und „Superkanten“ bieten sie keine Unterstützung. In dieser Arbeit wird das Konzept der „Superkante“ als Erweiterung der Hyperkante vorgestellt und implementiert. Damit lassen sich Texte als Graph darstellen und weiterverarbeiten.

Modellgetriebene Entwicklung ist derzeit ein wichtiger Trend in der Softwareentwicklung. Die notwendigen Modelle werden oft auf der Grundlage textueller Spezifikationen erzeugt. Für die Darstellung von natürlichsprachigen Texten als Modell gibt es jedoch kaum angemessene Werkzeugunterstützung: Der Grund dafür liegt nicht allein bei den verfügbaren Implementierungen, sondern in den Beschränkungen des Kantenkonzepts der Graphentheorie.
Diese Arbeit führt das Konzept der Superkante ein. Superkanten sind eine Erweiterung von Hyperkanten und können nicht nur mehrere Knoten miteinander verbinden, sondern auch Kanten. Eine Erweiterung des Graphersetzungssystems GrGen implementiert Superkanten und bietet dem Anwender ein Werkzeug zur Darstellung von natürlichsprachigen Texten als Modell, sowie deren Weiterverarbeitung.

1.1 Motivation

Anforderungsdokumente bilden die Grundlage vieler Softwareprojekte. Sie beinhalten bereits in einer sehr frühen Phase viele für ein Projekt wichtige Informationen. Es ist deshalb naheliegend, die in Anforderungsdokumenten enthaltenen Informationen für die Softwareentwicklung zu nutzen. Das Ziel ist die automatische Erzeugung von Softwaremodellen aus Anforderungsdokumenten. Diese Arbeit versucht nicht Texte direkt in Softwaremodelle zu überführen, sondern stellt die Texte zuerst als Modell (Graph) dar und verarbeitet sie dann mittels Graphersetzung weiter. Dieser Zwischenschritt ist sinnvoll, da für die Weiterverarbeitung auf bestehende Graphersetzungssysteme und die zugehörige Theorie zurückgegriffen werden kann.
Bei der Darstellung eines Textes in einem Modell bildet die Semantik des Textes einen besonders kritischen Punkt. Kann der dem Modell zugrunde liegende Graph die Semantik nur unzureichend darstellen, so erhöht dies das Risiko, dass eine aus diesem Modell erzeugte Softwarearchitektur die durch den Text spezifizierten Anforderungen nicht erfüllt.
Ein Problem bei der Darstellung von Texten als Graph sind die Einschränkungen des Kantenkonzepts der Graphentheorie. Das folgende Beispiel soll dem Leser eine Vorstellung vermitteln, wie ein Text in einen Graph überführt werden kann. Zugleich zeigt

es die Einschränkungen durch das Kantenkonzept auf. Der Beispielsatz stammt aus der Spezifikation des WHOIS-Protokolls[1] (RFC 3912 [Dai04]). Die hellgrau unterlegten Teile skizzieren den Kontext des Satzes.

> ... Anfragen werden mit ASCII CR und dann ASCII LF beendet. Die Antwort kann mehr als eine Textzeile enthalten, weshalb das Auftreten des ASCII CR- oder ASCII LF-Zeichens nicht das Ende einer Antwort markiert. Der WHOIS-Server beendet ...

Als erster Schritt der Überführung werden den Satzelementen semantische Rollen [vP85] zugeordnet. Dabei kommt das am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelte Rollensystem [Bru07] zum Einsatz.

> ... Anfragen werden mit ASCII CR und dann ASCII LF beendet. Die Antwort(OMN) kann mehr als eine Textzeile(PARS) enthalten, weshalb das Auftreten(AG) des ASCII CR- oder ASCII LF-Zeichens nicht das Ende(PAT) einer Antwort markiert(ACT). Der WHOIS-Server beendet ...

Der Satz besteht aus zwei Teilsätzen. Im erste Teilsatz werden die Rollen „Omnium“ und „Pars“ zugeordnet. „Omnium“ beschreibt ein Ganzes, das aus Teilen besteht und „Pars“ beschreibt ein Teil des Ganzen. Im zweiten Teilsatz sind die Rollen „Agens“, „Patiens“ und „Actus“ zugeordnet. „Agens“ beschreibt den Handelnden, „Patiens“ den von einer Handlung Betroffenen und „Actus“ die Handlung. Dies ist nur ein Teil, der in diesem Satz auftretenden Rollen[2].
Als zweiter Schritt der Überführung wird jeder Teilsatz als eine Kante mit mehr als zwei Endpunkten dargestellt (eine Einführung in die grundlegenden Begriffe der Graphentheorie befindet sich in Kapitel 2). Abbildung 1.1 zeigt die Darstellung der Teilsätze als Hyperkanten. Die semantischen Rollen sind in der Abbildung an den „Enden“ der Hyperkanten dargestellt.
Jede der Hyperkanten repräsentiert einen Teilsatz mit den entsprechenden Satzelementen und ihren semantischen Rollen. Allerdings drückt der Graph den Zusammenhang zwischen den Teilsätzen nicht aus. Im Beispiel haben die beiden Teilsätze einen kausalen Zusammenhang: Der Inhalt des ersten Teilsatzes begründet die Aussage des zweiten Teilsatzes. Dieser Zusammenhang wird im Graph durch eine zusätzliche Kante ausgedrückt. Diese Kante verbindet die beiden bestehenden Hyperkanten, wobei die linke Hyperkante die Rolle „Causa“ und die rechte Hyperkante die Rolle „Actus“ hat. „Causa“ beschreibt den Grund für eine Handlung. Abbildung 1.2 zeigt den Graphen mit der zusätzlichen Hyperkante, die den kausalen Zusammenhang darstellt.
Der Graph in Abbildung 1.2 ist kein „gültiger“ Graph im Sinne der Graphentheorie, denn das Kantenkonzept der Graphentheorie lässt Kanten nur als Verbindung zwischen Knoten zu. Kanten, die neben Knoten auch andere Kanten verbinden, sind in der Graphentheorie nicht vorgesehen. Dieses neue, in der Literatur bisher nicht benannte Konzept

[1] Der Beispielsatz wurde für die Verwendung in dieser Arbeit übersetzt. Das Originaldokument ist in Englisch verfasst.
[2] Es lassen sich noch weitere Rollen zuordnen. Darauf wurde aber verzichtete, um das Beispiel möglichst übersichtlich zu gestalten. Eine Liste aller in dieser Arbeit verwendeten Rollen findet sich im Anhang

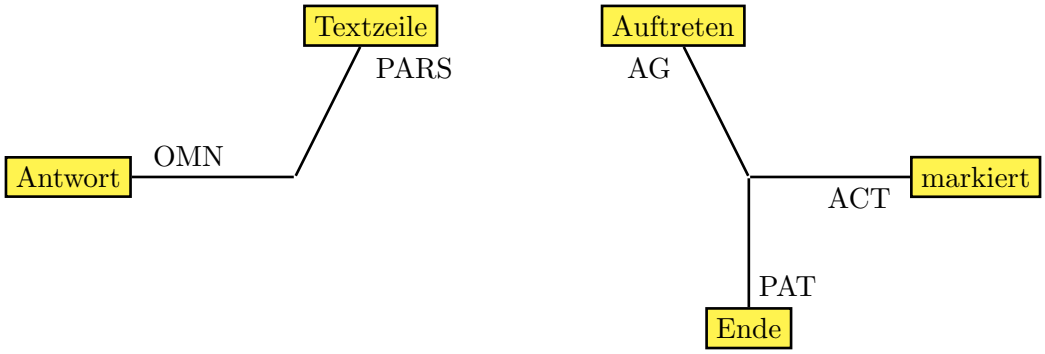


Abbildung 1.1: Einführung – Motivationsbeispiel

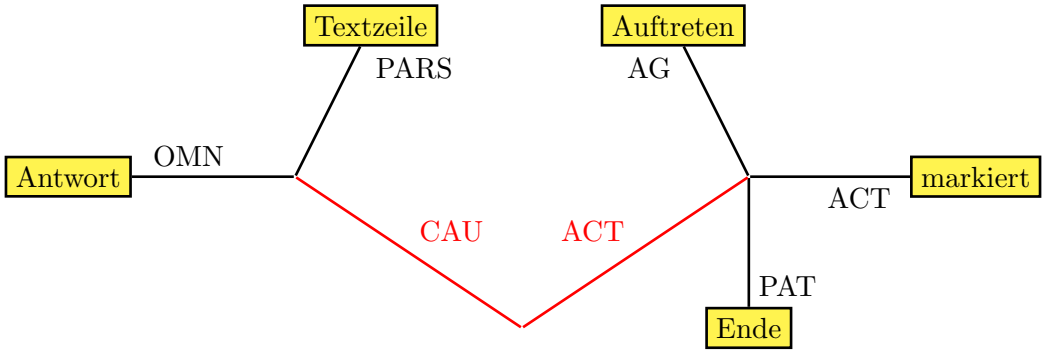


Abbildung 1.2: Einführung – vollständiges Motivationsbeispiel

erhält den Namen *Superkante*. Der Name ist an Hyperkante angelehnt, um zu zeigen, dass es sich um eine Erweiterung des Konzepts der Hyperkante handelt.
Als Konsequenz der Beschränkung des bisherigen Kantenkonzepts gibt es momentan keine Graphersetzungssysteme, die Superkanten darstellen können. Die Darstellung von natürlichsprachigen Texten als Graphen wird durch Superkanten wesentlich eleganter und unmittelbarer. Diese Arbeit behandelt deshalb Superkanten und ihre Realisierung. Die exakte Formulierung der Ziele dieser Arbeit folgt im nächsten Abschnitt:

1.2 Zielbestimmung

Das Ziel dieser Arbeit ist die Entwicklung einer Lösung der Aufgaben:

- **Repräsentation eines natürlichsprachigen Textes als Graph**
- **Weiterverarbeitung des Graphen durch Graphersetzung**

Für den Anwender wird die Darstellung von Texten als Graph durch die mangelnde Unterstützung durch Graphersetzungssysteme erschwert. Verfügbare Graphersetzungssysteme können weder Hyperkanten noch Superkanten darstellen und bieten kein Konzept für die Verwendung von Rollen. Es müssen deshalb die folgenden Funktionen realisiert werden.

- Die Erweiterung des Konzepts der Kanten als Verbindung zwischen Knoten auf ein allgemeineres Konzept, das Kanten auch als Verbindung zwischen anderen Kanten zulässt. Dies ist notwendig, um beispielsweise kausale oder temporale Zusammenhänge zwischen verschiedenen Sätzen eines Dokumentes darstellen zu können.

- Die Zuordnung von Rollen zu Graphelementen, wie sie zum Beispiel in Topic Maps [BBN99] möglich ist. Im Falle einfacher, gerichteter Graphen sind die Rollen der beiden durch eine Kante verbundenen Graphelemente klar: Eingangs- und Zielknoten. Bei Hypergraphen liegt die Rollenverteilung leider nicht mehr so eindeutig vor – sie muss explizit vorgenommen werden. Eindeutige Rollen sind aber wichtig um die Bedeutung von Objekten eines Satzes in Graphen darstellen zu können.

- Die in Texten auftretenden Adjektive und Adverbien beschreiben Objekte des Satzes. Im Graph sollen sie als Attribute des Knotens, der das entsprechende Objekt repräsentiert, dargestellt werden. Dabei muss beachtet werden, dass Anzahl und Art der Attribute eines beliebigen Textes unbekannt sind. Das Graphersetzungssystem muss deshalb ein universelles Konzept für Attribute bieten.

- Die Semantik von Mengen, wie sie zum Beispiel durch Aufzählungen in Texten enthalten ist, soll im Graph dargestellt werden. Dabei können in natürlicher Sprache logische Aussagen zwischen den Elementen einer Aufzählung formuliert oder Elemente ausgeschlossen werden.

Für die genannten Funktionen muss zuerst ein geeigneter Formalismus entwickelt werden (Kapitel 4). Danach müssen die Funktionen auf die von einem verfügbaren Graphersetzungssystem gebotene Funktionalität zurückgeführt werden. Zu diesem Zweck evaluiert Kapitel 3 bestehende Graphersetzungssysteme, um ein System zu finden, das möglichst gut als Basis für die Erweiterungen geeignet ist. (Die Neuentwicklung eines Graphersetzungssystems ist weder sinnvoll noch im Rahmen dieser Arbeit möglich.) Die Syntax der Erweiterungen wird in Kapitel 5 vorgestellt, die Implementierung in Kapitel 7 erklärt und in Kapitel 6 evaluiert.

2 Formale Grundlagen

In diesem Kapitel werden Grundlagen der Graphentheorie und der Graphersetzung erläutert, die für das Verständnis dieser Arbeit wichtig sind. Die Notationen sind größtenteils [EEPT06] entnommen.

2.1 Graphentheorie

Ein Graph besteht aus Knoten und Kanten zwischen den Knoten. Dabei verbindet eine Kante grundsätzlich genau zwei Knoten miteinander. Formal besteht ein Graph $G = (V, E)$ aus einer Menge von Knoten $V(vertice)$ und einer Menge von Kanten $E(edge)$ mit $E \subseteq V \times V$. Ein Element $(y, z) \in E$ beschreibt eine Kante zwischen den Knoten y und z. Abbildung 2.1 zeigt einen allgemeinen Graph $G = (V, E)$ mit $V = \{w, x, y, z\}$ und $E = \{(y, z), (w, z), (w, x), (x, z)\}$.

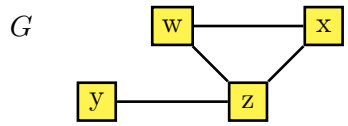


Abbildung 2.1: Grundlagen – allgemeiner Graph

Gerichteter Graph Neben der Notation eines Graphen durch $G = (V, E)$ ist auch die Notation durch $G = (V, E, s, t)$ weit verbreitet. Dabei ist V die Menge der Knoten und E die Menge der Kanten, wobei die beiden Funktionen $s, t : E \to V$ einer Kante jeweils einen Quell- (source) und einen Zielknoten (target) zuordnen. Somit erhalten die Kanten eine Richtung. Ein Graph dessen Kanten gerichtet sind, heißt gerichteter Graph. Die Darstellung im Graph erfolgt durch einen Pfeil. Abbildung 2.2 zeigt den gerichteten Graphen $G_S = (V_S, E_S, s_S, t_S)$ mit $V_S = \{w, x, y, z\}$, $E_S = \{a, b, c, d\}$, $s_S : E \to V : a, b \mapsto w; c \mapsto z; d \mapsto y$ und $t_S : E \to V : a, c \mapsto x; b, d \mapsto z$.

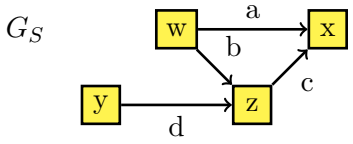


Abbildung 2.2: Grundlagen – gerichteter Graph

Multigraph Die Notation durch $G = (V, E, s, t)$ erlaubt gegenüber der Notation durch $G = (V, E)$ die Definition von mehreren Kanten zwischen zwei Knoten. Ein Graph bei dem zwei Knoten durch mehr als eine Kante (im gerichteten Fall mehr als eine gleich gerichtete Kante) verbunden sind, nennt man einen Multigraphen. Die Formalisierung von Multigraphen unterscheidet sich nicht von der Formalisierung von gerichteten Graphen. Abbildung 2.3 zeigt den Multigraphen $G_S = (V_S, E_S, s_S, t_S)$ mit $V_S = \{w, x, y, z\}$, $E_S = \{a, b, c, d, e, f\}$, $s_S : E \to V : a, b \mapsto y; c, e \mapsto w; d \mapsto z; f \mapsto x$ und $t_S : E \to V : a, b \mapsto w; e, d \mapsto x; c, f \mapsto z$.

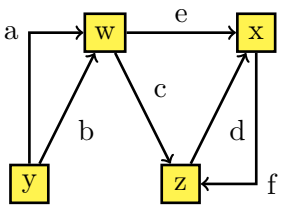


Abbildung 2.3: Grundlagen – Multigraph

Neben gerichteten Graphen und Multigraphen gibt es weitere Varianten des allgemeinen Graphmodells. Nachfolgend werden Varianten vorgestellt, die für das Verständnis dieser Arbeit von Bedeutung sind. Dazu wird zuerst der Begriff des Graphmorphismus eingeführt.

Graphmorphismus Für zwei Graphen $G_1 = (V_1, E_1, s_1, t_1), G_2 = (V_2, E_2, s_2, t_2)$ ist $f : G_1 \to G_2, f = (f_V, f_E)$ mit $f_V : V_1 \to V_2, f_E : E_1 \to E_2$ ein Graphmorphismus, wenn f_V, f_E die Quell- und Zielfunktionen der beiden Graphen erhalten. Es muss also gelten:

$$f_V \circ s_1 = s_2 \circ f_E \text{ und } f_V \circ t_1 = t_2 \circ f_E$$

Typisierter Graph Um große Graphen besser strukturieren zu können, verwendet man Typen. Dazu werden zusätzlich zum Graphen Typen für Knoten und Kanten definiert. Jedem Knoten und jeder Kante im Graph wird dann ein Typ zugeordnet. Wie bei Klassen und Objekten, in der Objektorientierten Programmierung, können die Knoten und Kanten eines Graphen, als Instanz ihres Typs betrachtet werden.
Ein Tupel $(G, type)$ bestehend aus einem Graphen G und einem Graphmorphismus $type : G \to TG$, der einen Graphen auf einen Typgraphen TG abbildet, nennt man einen typisierten Graphen. Ein Typgraph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ besteht aus einer Knotentypmenge V_{TG}, einer Kantentypmenge E_{TG}, sowie den Funktionen s_{TG}, t_{TG}, die den Typ der Quell- und Zielknoten einer Kante festlegen.
Abbildung 2.4 zeigt den Typgraphen $G_T = (V_T, E_T, s_T, t_T)$ mit $V_T = \{s, t, u\}$, $E_T = \{e, f, g\}$, $s_T : E_T \to V_T : e, f \mapsto s; g \mapsto u$ and $t_T : E_T \to V_T : e, g \mapsto t; f \mapsto u$.
Der Graph G_S aus Abbildung 2.2 ergibt zusammen mit dem Graphmorphismus $type$ und dem Typgraphen G_T aus Abbildung 2.4 den typisierten Graphen $G = (G_S, type)$ in Abbildung 2.5. Dabei gilt $type = (type_V, type_E) : G_S \to G_T$ mit $type_V : V_S \to V_T$:

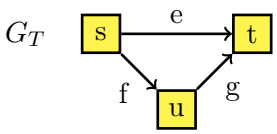


Abbildung 2.4: Grundlagen – Typgraph

$w, y \mapsto s; x \mapsto t; z \mapsto b$ und $type_E : E_S \to E_T : a \mapsto e; b, d \mapsto f; c \mapsto g$. Der Typ eines Knoten bzw. einer Kante wird durch die Darstellung $: Typ$ angegeben.

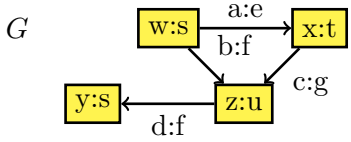


Abbildung 2.5: Grundlagen – typisierter Graph

Im Kontext der Graphersetzung wird der Typgraph meist als „Modell“ bezeichnet, denn er legt fest welche Typen in einem Graph, der diesem Modell entspricht, auftreten dürfen. Im Kontext der Modelltransformation wird der Typgraph als „Metamodell“ bezeichnet, da bereits Graphen den Begriff Modell tragen.
Nicht alle Graphersetzungswerkzeuge stellen den Typgraphen in Form eines Graphen dar, sondern halten die Typdefinitionen in textueller Form fest. Die vom Nutzer definierten Typen ergeben allerdings immer einen Typgraphen, auch wenn dieser nicht explizit dargestellt wird.

Markierter Graph Bisher wurden den Knoten und Kanten in den Abbildungen immer Namen zugeordnet um sie identifizieren zu können. Diese Zuordnung lässt sich auch formal ausdrücken: Man spricht dann von einem markierten Graphen (labeled graph). Ein markierter Graph $G = (V, E, s, t, l_V, l_E)$ besteht aus einem Graphen $G = (V, E, s, t)$, einem Markierungsalphabet $L = (L_V, E_V)$, bestehend aus Mengen von Knotenmarkierungen L_V und Kantenmarkierungen L_E, sowie zwei Markierungsfunktionen:

$$l_V : V \to L_V \text{ und } l_E : E \to L_E$$

Abbildung 2.6 zeigt einen markierten Graph mit $L_V = \{a, b, c, d\}$ und $L_E = \{s, t, u, v\}$.

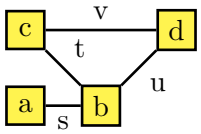


Abbildung 2.6: Grundlagen – markierter Graph

Attributierter Graph In einem attributierten Graphen können den Knoten und Kanten Attribute zugeordnet werden. Zum Beispiel kann einer Kante ein Attribut vom Typ Ganzzahl zugeordnet werden. Das bekannte Beispiel „Suche des kürzesten Weges eines Geschäftsreisenden“ (Travelling-Salesman-Problem) verwendet solche Attribute, um die Kosten der verschiedenen Wege (die durch Kanten realisiert werden) darzustellen.
Die Notation eines attributierten Graphen erfolgt in der Form:

$$G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (s_j, t_j)_{j \in \{G,NA,EA\}}).$$

V_G sind die Graphknoten und V_D die Datenknoten, während E_G, E_{NA}, E_{EA} die Graph-, Knotenattribut- und Kantenattributkanten sind. Für jede der drei Kantenmengen gibt es entsprechende Quell- und Zielfunktionen. Die Graphkanten verlaufen wie in den bereits vorgestellten Graphen zwischen Graphknoten: $s_G, t_G : E_G \to V_G$. Die Knotenattributkanten ordnen Datenknoten zu Graphknoten zu: $s_{NA} : E_{NA} \to V_G, t_{NA} : E_{NA} \to V_D$. Kantenattributkanten ordnen Datenknoten zu Kanten zu: $s_{EA} : E_{EA} \to E_G, t_{EA} : E_{EA} \to V_D$.
Den attributierten Graphen $G = (V_S, V_D, E_S, E_{NA}, (s_S, t_S, s_{NA}, t_{NA}))$ ist in Abbildung 2.7 dargestellt, wobei $G_S = (V_S, E_S, (s_S, t_S))$ dem Graph aus Abbildung 2.2 entspricht und $V_D = \{s, t\}$, $E_{NA} = \{g, h\}$, $s_{NA} : E_{NA} \to V_G : g \mapsto z; h \mapsto x$ und $t_{NA} : E_{NA} \to V_D : g \mapsto t, h \mapsto s$ gilt. Kantenattribute treten in diesem Beispiel nicht auf.

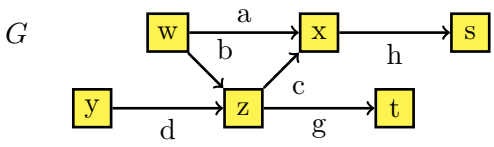


Abbildung 2.7: Grundlagen – attributierter Graph

Rollen Versucht man das Konzept einer gerichteten Kante zu verallgemeinern, so erkennt man, dass eine gerichtete Kante ihren Endpunkten spezielle Semantik zuweist. Einer der Endpunkte erhält die Markierung „Start“ und der andere die Markierung „Ziel“. Die Endpunkte erhalten also Rollen, die sie in dieser Kante einnehmen. Abbildung 2.8 zeigt eine gerichtete Kante, deren Richtung durch Rollen definiert ist.

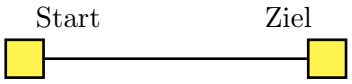


Abbildung 2.8: Grundlagen – Rollen

Hyperkanten In allen bisher vorgestellten Konzepten hatten Kanten immer exakt zwei Endpunkte. Um diese Einschränkung aufzuheben gibt es die Erweiterung von Einfachkante (mit zwei Endpunkten) zu Hyperkanten, die beliebig viele Endpunkte verbinden können. Ein zu einer Hyperkante gehörender Knoten wird, wie bei Einfachkanten, als

Endpunkt bezeichnet, während die Kombination aus Endpunkt und dem Teil der Kante, der an diesem Endpunkt endet, als Tentakel bezeichnet wird. Abbildung 2.9 veranschaulicht die Begriffe Endpunkt und Tentakel.

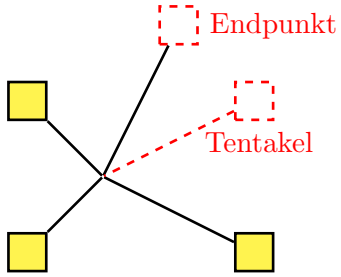


Abbildung 2.9: Grundlagen – Hyperkante

Eine Hyperkante wird durch die Menge der Knoten, die sie verbindet, beschrieben. Bei einer Menge stellt sich oft die Frage, wie eine Ordnung auf dieser Menge definiert werden kann? Dazu werden den einzelnen Tentakeln der Hyperkante Rollen zugewiesen. Abbildung 2.10 zeigt eine Hyperkante mit fünf Tentakeln, die durch Rollen geordnet sind.

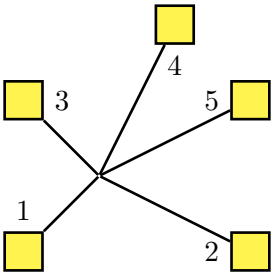


Abbildung 2.10: Grundlagen – Hyperkante mit Rollen

Rollen werden nicht nur zum Ordnen von Tentakeln oder Richten von Kanten eingesetzt, sondern stellen ein universelles Konzept zur Zuordnung von Semantik dar, das an vielen Stellen eingesetzt wird. So sind Rollen z.B. von Assoziationen in UML-Klassendiagrammen oder aus Topic Maps [BBN99] bekannt.
Als abschließende Bemerkung zur Graphentheorie, sei darauf verwiesen, dass Hyperkanten eine Verallgemeinerung, von Einfachkanten sind. Eine Einfachkante, lässt sich immer auch als Hyperkante mit zwei Tentakeln darstellen. Abbildung 2.11 zeigt die gleiche Kante, dargestellt als Einfachkante und als Hyperkante.

Abbildung 2.11: Grundlagen – Hyperkanten als Verallgemeinerung

2.2 Graphersetzung

Unter dem Begriff Graphersetzung versteht man in erster Linie die Anwendung von Graphersetzungsregeln auf einen Graphen. Für eine vollständige Graphersetzung müssen allerdings unter Anderem auch noch ein Modell und eine Reihenfolge für die Regelabarbeitung definiert werden. Diese sind aber nicht Gegenstand dieses Abschnitts.
Eine Ersetzungsregel besteht aus einer linken und einer rechten Seite, wobei die Ersetzungsregel die linke auf die rechte Seite abbildet. Abbildung 2.12 zeigt die linke und rechte Seite einer Regel. Die Nummern dienen der Identifizierung der Knoten. Der Knoten mit der Nummer „1“ tritt nur auf der linken Seite auf und wird deshalb bei der Ersetzung gelöscht. Die Knoten mit den Nummern „3“ und „4“ treten nur auf der rechten Seite auf und werden deshalb bei der Ersetzung neu erzeugt.

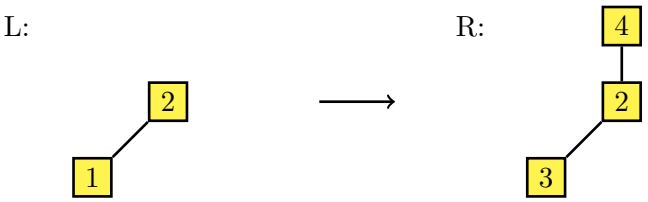


Abbildung 2.12: Grundlagen – Ersetzungsregel

Der Ablauf einer Regelanwendung erfolgt in zwei Schritten:

1. **Linke Seite L im Wirtsgraphen suchen.**
2. **Linke Seite L durch rechte Seite R ersetzen.**

Als Wirtsgraph wird der Graph, auf den die Regel angewendet werden soll, bezeichnet. Abbildung 2.13 zeigt einen Graphen G, der als Wirtsgraph dient.

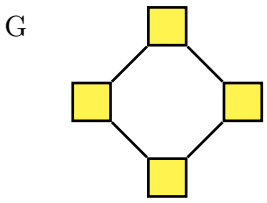


Abbildung 2.13: Grundlagen – Wirtsgraph

Im ersten Schritt wird eine Übereinstimmung der linken Seite im Wirtsgraphen gesucht. Dazu muss jedem Element der linken Seite ein entsprechendes Element im Wirtsgraph zugeordnet werden. Ist eine Übereinstimmung gefunden, so kann der zweite Schritt ausgeführt werden. Man sieht, dass es mehrere mögliche Übereinstimmungen der linken Seite gibt. Die meisten Graphersetzungssysteme wählen in diesem Fall indeterministisch eine der möglichen Übereinstimmungen aus. In Abbildung 2.14 ist im Graphen G eine Übereinstimmung der linken Seite markiert.

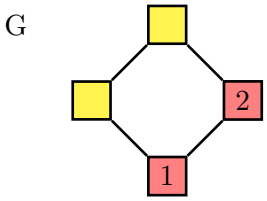


Abbildung 2.14: Grundlagen – Übereinstimmung

In Abbildung 2.15 ist der Graph nach der Ersetzung der linken durch die rechte Seite dargestellt. Das Ergebnis ist kein gültiger Graph mehr, da „lose“ Kanten (mit nur einem Endpunkt) im Graph auftreten. Um dieses Problem zu lösen gibt es zwei Ansätze:

Single Pushout (SPO) Bei diesem Ansatz werden die „losen“ Kanten nach der Ersetzung einfach gelöscht. Dadurch entsteht wieder ein gültiger Graph.

Double Pushout (DPO) Bei diesem Ansatz darf eine Regel nur ausgeführt werden, wenn keine „losen“ Kanten zurückbleiben. Eine genaue Erklärung dieses Ansatzes folgt im Abschnitt 3.2.

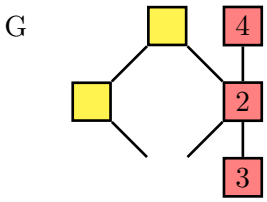


Abbildung 2.15: Grundlagen – Regelanwendung

Ein wichtiger Bestandteil der linken Seite sind *negative Anwendungsbedingungen (NAC)*. Sie definieren Graphelemente die im Wirtsgraph **nicht** auftreten dürfen und sind somit das Gegenstück, zu den Graphelementen auf der linken Seite (die für eine Übereinstimmung im Graph vorhanden sein müssen). Damit eine Regel anwendbar ist, müssen alle negativen Anwendungsbedingungen erfüllt sein. Negative Anwendungsbedingungen sind meist mit den Graphelementen der linken Seite verbunden, um den Kontext einer Übereinstimmung einzuschränken. Abbildung 2.16 zeigt die linke Seite der Regel aus Abbildung 2.12 erweitert um eine negative Anwendungsbedingung (gestrichelt dargestellt).

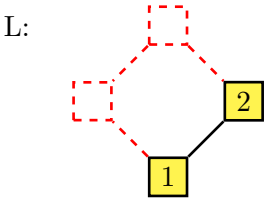


Abbildung 2.16: Grundlagen – negative Anwendungsbedingung

Die negative Anwendungsbedingung verhindert die Anwendung der Regel aus Abbildung 2.16 auf den Graphen aus Abbildung 2.13.

3 Verwandte Arbeiten

Dank der zunehmenden Popularität modelgetriebener Softwareentwicklung [MDD], gibt es heute eine große Zahl von Modelltransformationswerkzeugen. Viele dieser Werkzeuge verwenden Graphtransformationen, die ihren Ursprung in der Theorie der Graphgrammatiken und dem Übersetzerbau haben.
Dieses Kapitel gibt einen Überblick über die am weitest verbreiteten Werkzeuge und prüft ihre Eignung für die in der Einleitung genannten Anforderungen. Zuerst werden einige Arbeiten betrachtet, die versuchen verschiedene Modelltransformationsansätze und -werkzeuge gegeneinander abzugrenzen und zu klassifizieren. Anschließend werden daraus für einen Vergleich wichtige Kriterien erarbeitet. Anhand der Kriterien wird dann ein Überblick über verschiedene Ansätze und Werkzeuge, die sich für Graphtransformationen eignen, gegeben. Am Ende werden einige konkrete Werkzeuge vorgestellt und ihre Funktionsweise anhand eines Beispiels demonstriert.

3.1 Klassifikationsansätze

Czarnecki und Helsen haben in ihrem 2003 erschienenen Artikel „Classification of Model Transformation Approaches“ [CH03] den Versuch unternommen, Ansätze sowie verfügbare Werkzeuge zum Thema *Modell*transformationen zu klassifizieren. In den Jahren nach der Veröffentlichung wurde diese Klassifikation zur Grundlage vieler Publikationen im Bereich der Modelltransformationen. Als Reaktion veröffentlichten Czarnecki und Helsen 2006 im IBM Systems Journal eine überarbeitete und erweiterte Version [CH06]. Diese stellt derzeit den umfassendsten Überblick zum Thema Modelltransformationen dar und diente im Vorfeld dieser Diplomarbeit als Grundlage für die Betrachtung bestehender Ansätze.

In [MvGKV05] werden *Graph*transformationen klassifiziert. Dazu wird eine Klassifikation für Modelltransformationen angewendet. Die Autoren ordnen alle Graphtransformationsansätze in vier Kategorien ein. Allgemeine Graphersetzungswerkzeuge (AGG [AGG], PROGRES [aS]), auf Graphtransformation basierende Reengineering-Werkzeuge (Fujaba [Fuj], VARLET [Vara]), Modelltransformationswerkzeuge (GReAT [GRe], MOLA [MOL]) und Modellprüfungs- und verifikations-Werkzeuge (VIATRA [Varb], GROOVE [GRO], CheckVML [SV03]). Die Kriterien der Klassifikation sind in vier Gruppen unterteilt. Die erste Gruppe von Kriterien klärt die Frage: „Was soll in was transformiert werden?“. Hierunter fallen zum Beispiel Angaben zu den Quell- und Zielmodellen, die Darstellung der Modelle, sowie die Art der Transformation. Die zweite Gruppe von Kriterien behandelt die wichtigen Charakteristiken einer Transformation, zum Beispiel wie

stark die Automatisierung ist oder die Komplexität der Transformation. Die dritte Gruppe von Kriterien betrachtet die für einen produktiven Einsatz wichtigen Anforderungen an ein Transformationswerkzeug. Dazu gehören die Möglichkeit die Regelanwendung zu steuern, Regeln wiederzuverwenden und die Bidirektionalität von Regeln. Die letzte Gruppe bilden Kriterien zu Qualitätsanforderungen an Transformationswerkzeuge. Dazu zählen Nutzbarkeit, Performanz und Erweiterbarkeit.

In [EGdL+05] findet sich ein detaillierter Vergleich zwischen den vier Graphtransformationswerkzeugen AGG [AGG], AToM [dL], VIATRA [Varb] und VMTS [VMT]. Am Ende findet ein zusätzlicher Vergleich mit QVT [QVT] statt. Die Graphtransformationswerkzeuge werden alle anhand eines Beispielszenarios, das eine Transformation zwischen UML-Klassendiagrammen und ER-Diagrammen vornimmt, demonstriert.

Sendall und Kozaczynski nehmen in [SK03] eine kurze Einordnung von verschiedenen Modelltransformationsansätzen vor. Sie unterscheiden zwischen:

Direkter Modellmanipulation Dabei wird die Darstellung von Modellen hinter einer API verborgen, die den Zugriff auf das Modell und die Manipulation des Modells regelt. Dafür wird meist eine imperative Programmiersprache verwendet. Dieser Ansatz findet sich namensgleich in [CH06] wieder.

Zwischendarstellungen Hierunter versteht man den Export der Modelle in ein Standardformat wie z.B. XML [XMLa]. Die Transformation findet extern z.B. mittels XSLT [XSL] statt.

Transformationssprachenunterstützung Bei diesen Ansätzen kommen spezielle Sprachen zum Einsatz, die sowohl die Spezifikation der Regeln als auch der Regelanwendung ermöglichen. Jedes Werkzeug, das einen derartigen Ansatz verfolgt bringt in der Regel seine eigene Sprache mit und kann meist die so spezifizierten Transformationen direkt ausführen. Für die Transformationssprachen nennt der Artikel vier wichtige Kriterien:

- die Unterstützung von Vorbedingungen für Transformationsregeln,
- die Möglichkeiten zur Komposition von Regeln,
- eine geeignete Form für die Regeln (grafische Regeldarstellungen sind ihrer Meinung nach textuellen vorzuziehen, da sie einfacher zu verstehen und anzuwenden sind),
- und die Benutzerfreundlichkeit der Sprachen. Hybride Ansätze, die sowohl deklarative als auch imperative Regelangaben zulassen, seien ihrer Meinung nach am besten geeignet, da deklarative Regeln kurz und präzise sind und imperative Regeln einfache Kompositionen unterstützen.

Whittle beschäftigt sich in [Whi02] mit Transformationen zwischen UML-Klassenmodellen, die zusätzlich mit OCL-Zusicherungen angereichert sind. Der Fokus liegt auf Refaktorisierungen und der Überprüfung von Zusicherungen. Die Umsetzung erfolgt in der

Logiksprache MAUDE [Mau], für die bereits ein UML-Metamodell vorhanden ist.

In [GGKH03] werden die bis zu diesem Zeitpunkt eingereichten Entwürfe für die QVT-Spezifikation der OMG [OMG] betrachtet. Die wichtigste Errungenschaft des Artikels ist allerdings die Definition einiger Begriffe, die in allen zukünftigen Publikationen der OMG zum Thema Modelltransformationen verwendet werden sollen. Als *Transformation* wird die Erzeugung eines Zielmodells aus einem Quellmodell betrachtet. Dabei können die Modelle *abhängig* oder *unabhängig* sein, je nachdem ob nach der Transformation noch Abhängigkeiten (in Form von gemeinsamen Modellelementen oder Kanten zwischen Modellelementen) zwischen dem Quell- und Zielmodell bestehen oder nicht. Eine Transformation wird als *top-down* bezeichnet, wenn das Zielmodell nach der Generierung nicht verändert wird. Änderungen werden immer nur am Quellmodell vorgenommen und dann transformiert. Eine *unidirektionale (one-way)* Transformation erlaubt nur die Generierung des Zielmodells aus dem Quellmodell, während eine *bidirektionale (two-way)* Transformation beide Richtungen erlaubt. Es können also sowohl am Quell- als auch am Zielmodell Änderungen vorgenommen und dann transformiert werden. Dieser Vorgang, in dem aus dem Zielmodell das Quellmodell verändert wird, nennt man *Aktualisierung (update)*. Da sich bei bidirektionalen Transformationen die Begriffe Quell- und Zielmodell nicht immer klar zuordnen lassen, werden die Begriffe *linksseitiges (left-hand) Modell* und *rechtsseitiges (right-hand) Modell* verwendet. Die Spezifizierung von Transformationen wird analog zu den allgemein für Programmiersprachen üblichen Kriterien als *imperativ* oder *deklarativ* bezeichnet. Darüber hinaus gibt es *hybride* Ansätze, die sowohl deklarative als auch imperative Komponenten besitzen. Eine Transformation besteht aus verschiedenen *Regeln*, die jeweils einen bestimmten Ausschnitt aus einem Modell transformieren. Eine *Implementierung* ist eine imperative Spezifikation einer Regel, die explizit Elemente im Zielmodell erzeugt. Implementierungen sind generell unidirektional. Sofern Bidirektionalität erwünscht ist, muss die Rücktransformation explizit angegeben werden. Eine *Übereinstimmung (match)* liegt dann vor, wenn die linke Seite einer Regel in einem Modell auftritt und außerdem alle Zusicherungen der linken Seite zutreffen. Wenn nach einer Änderung des Quellmodells nur die Regeln angewendet werden, deren linke Seite von den Änderungen betroffen ist, so spricht man von *inkrementeller* Transformation.

3.2 Klassifikationskriterien und -kategorien

Bisher wurden Arbeiten vorgestellt, die sich mit der Klassifikation von Modelltransformationsansätzen und -werkzeugen beschäftigen. Im nächsten Abschnitt werden Kriterien vorgestellt, die für die Bewertung von Werkzeugen im Kontext dieser Arbeit wichtig sind. Anschließend gibt es einen Überblick über die grundsätzlichen Modelltransformationskategorien.

Klassifikationskriterien

Czarnecki und Helsen verwenden in [CH06] ein Merkmalsmodell (feature model) [CE00], um sämtliche Aspekte und Zusammenhänge zwischen den Aspekten ihrer Klassifikation darzustellen. Der nachfolgende Abschnitt beschränkt sich auf eine Auswahl an Aspekten, die für diese Arbeit von Interesse sind. Es werden außerdem ein paar zusätzliche, nicht in [CH06] enthaltene Aspekte betrachtet.
Die Kriterien lassen sich in drei Kategorien einteilen: die Domäne der Modelle, die Transformationsregeln und die Anwendung der Regeln.

Domäne

Dieser Bereich betrachtet zuerst die Domänen der Regeln. Unter dem Begriff Domäne versteht man ein Metamodell, das die Struktur eines gültigen Modells beschreibt[1]. In den meisten Fällen gehören das Quell- und Zielmodell jeweils einer eigenen Domäne an, die jedoch auch identisch sein kann. Die Domäne eines Modells wird durch die Angabe eines Metamodells, dem alle darauf basierenden Modelle entsprechen müssen, spezifiziert. Ein Beispiel für eine domänenübergreifende Transformation ist die Übersetzung von UML-Diagrammen in die vor allem aus dem Datenbankentwurf bekannten ER-Diagramme. Eine weitere domänenübergreifende Transformation ist die Erzeugung von Quelltext aus UML-Diagrammen, während Refaktorisierungen auf dem Quelltext eine domäneninterne Transformation darstellen.

Anzahl der Quell- und Zielmodelle Dieses Kriterium gibt an, in welchem Verhältnis Quell- und Zielmodelle auftreten können. Betrachtet wird dabei nur die Anzahl der Modelle und nicht der Typ der Modelle. Mögliche Verhältnisse sind z.B. Umsetzung eines Modelles auf ein anderes (1:1), Trennung eines Modelles in mehrere verschiedene (1:n) oder Vereinigung mehrerer Modelle zu einem einzigen (n:1). Einige Werkzeuge erlauben auch allgemeine Transformationen (n:m).

Art der Transformation Hier gibt es die beiden Möglichkeiten *endogen* und *exogen*. Bei einer endogenen Transformation wird innerhalb des gleichen Metamodells transformiert. Bei einer exogenen Transformation gehören Quell- und Zielmodell zu unterschiedlichen Metamodellen, d.h. es ist ein Übergang zwischen verschiedenen Domänen möglich.

Angabe des Metamodells Um das Metamodell zu spezifizieren, empfiehlt sich ein als allgemeiner Standard akzeptiertes Datenformat wie z.B. das OMG-Format MOF [MOF]. Allerdings nutzen die meisten Werkzeuge eigene Formate und können nur in wenigen Ausnahmefällen MOF-Modelle importieren bzw. exportieren. Dies liegt zum einen daran, dass viele Werkzeuge älter sind als MOF und zum anderen, dass einige Werkzeuge in ihren Metamodellen Graphen zulassen, die sich durch MOF nur unzureichend spezifizieren lassen.

[1] Der Begriff Domäne wird nur im Zusammenhang mit Modelltransformationen verwendet. Bei Graphtransformationen spricht man explizit von Metamodellen oder Graphmodellen.

Angabe von Variablen/Mustern Dieses Kriterium beschreibt das Datenformat der eigentlichen Modelle und der linken und rechten Seiten von Ersetzungsregeln. Variablen können beliebige Elemente der Quell- oder Zielmodelle enthalten, sowie eventuell auftretende temporäre Elemente. Muster setzen sich aus mehreren Variablen zusammen und können entweder textuell oder grafisch angegeben werden.

Angabe der Logik Unter Logik werden Berechnungen und Zusicherungen für Modellelemente verstanden. Bei neu zu erzeugenden Elementen müssen z.B. oft Attribute berechnet werden. Um die Logik in einem Modell zu beschreiben gibt es verschiedene Möglichkeiten. Für Zusicherungen bietet sich OCL [OCL] an, während sich für Berechnungen imperative Sprachkonstrukte anbieten. Allerdings setzen nur wenige Werkzeuge auf OCL, sondern nutzen ihre eigenen Mechanismen.

Typisierung Sofern die Typisierung von Modellen unterstützt wird, kann zwischen *syntaktischer* und *semantischer* Typisierung unterschieden werden. Im Fall der syntaktischen Typisierung wird einer Variablen ein Element eines Metamodells zugewiesen, so dass die Variable nur eine Instanz des Elements aufnehmen kann. Bei der semantischen Typisierung können weitere Eigenschaften zugesichert werden, zum Beispiel die Wohlgeformtheit einer Regel oder ein bestimmtes Verhalten einer Variablen. Wird keine Typisierung unterstützt, so können nur die vom Werkzeug bereitgestellten Typen (i.d.R. Knoten und Kante) unterschieden werden.

Knotenattribute Um Modelle sinnvoll einsetzen zu können, sind in den meisten Fällen zusätzlich zu den reinen Knoten auch Attribute notwendig. Jedes Attribut kann grundsätzlich durch einen zusätzlichen Knoten, der mit dem zu attributierenden Knoten verbunden ist, ersetzt und so in jedem Graph ausgedrückt werden. Diese Darstellung macht allerdings das Modell unübersichtlich und schwer verständlich. Um Modelle kompakter und besser überschaubar zu machen, ist es wünschenswert, dass Attribute nicht als zusätzliche Knoten sondern als Teil der vorhandenen Knoten dargestellt werden. Sofern Typisierung unterstützt wird, betrachtet dieses Kriterium auch die möglichen Attributtypen.

Kantenattribute Neben der Attributierung von Knoten ist auch die Attributierung von Kanten wünschenswert. Allerdings bieten nicht alle Werkzeuge die Möglichkeit dazu.

Hyperkanten/Superkanten Hyperkanten, also Kanten mit mehr als zwei Endpunkten, sind ein in der Graphentheorie seit langem bekanntes Konzept, allerdings bietet kein Ersetzungswerkzeug Unterstützung dafür. Superkanten, also Kanten zwischen anderen Kanten, wurden bisher in der Graphentheorie nicht betrachtet, trotzdem können einige Werkzeuge damit zumindest rudimentär umgehen.

Transformationsregeln

Typ Für die Angabe von Transformationsregeln kommen analog den Programmiersprachen zwei grundlegende Paradigmen in Frage: *deklarativ* und *imperativ*.

Angabe Dieses Kriterium beschreibt, wie Regeln angegeben werden. Dies kann z.B. durch einen grafischen Editor oder eine spezielle Definitionssprache erfolgen. Im Abschnitt 3.3 finden sich Beispiele für die Eingabe von Regeln in verschiedenen Werkzeugen.

Richtung Regeln können sowohl unidirektional als auch bidirektional sein. Deklarative Regeln sind immer bidirektional, da man linke und rechte Seite der Regel tauschen kann. Lediglich falls die Abbildung zwischen Quell- und Zielmodell nicht injektiv ist, also z.B. zwei verschiedene Elemente des Quellmodells auf ein Element des Zielmodells abgebildet werden, sind die Regeln nicht automatisch bidirektional. Bei imperativen Regeln muss die Rückrichtung explizit angegeben werden.

Voraussetzungen Eine Regel wird angewendet, wenn die linke Seite der Regel im Quellmodell gefunden wird. In vielen Fällen ist es jedoch sinnvoll, Vorraussetzungen anzugeben, wann eine Regel angewendet werden kann. Mögliche Voraussetzungen sind zum Beispiel *negative Anwendungsbedingungen (NAC - negative application condition)*, die Modellelemente spezifizieren, die nicht vorhanden sein dürfen oder für Attribute zulässige Wertebereiche vorgeben. Eine andere Möglichkeit sind Zusicherungen an Knoten- und Kantentypen oder Attribute.

Parametrisierung Einige Werkzeuge bieten parametrisierbare Regeln. Möglich sind dabei einfache Kontrollparameter, Generizität oder sogar Regeln höherer Ordnung, d.h. Regeln, die sich mit Regeln parametrisieren lasse. Diese Möglichkeit bietet bisher allerdings nur das Termersetzungssystem Stratego [Vis].

Quell-Ziel-Abhängigkeiten Quell- und Zielmodell können auf zwei verschiedene Arten voneinander abhängen. Zum einen kann es sich um unterschiedliche Modelle handeln. Zum Beispiel ein UML-Modell als Quelle und als Ziel ein Modell für Java-Quellcode. Die Transformation überführt dann Elemente des UML-Modells in Java-Quellcode. Zum anderen kann es sich um ein einzelnes Modell handeln, das sowohl Quelle als auch Ziel ist. Zum Beispiel ein UML-Klassendiagramm, auf dem mit Hilfe von Transformationsregeln Refaktorisierungen ausgeführt werden. Die Transformationen verändert also direkt das UML-Klassenmodell. Diesen Vorgang nennt man: *in-place-update*. Als Variation gibt es die Möglichkeit das Modell vor der Transformation zu kopieren und die Änderungen auf der Kopie auszuführen. Das Originalmodell bleibt dabei unverändert.

Regelanwendung

Bei der Regelanwendung gibt es zwei wichtige Freiheitsgrade: der Ort, an dem eine Regel angewendet wird und die Auswahl der Regel, die angewendet werden soll.

Anwendungsort Was passiert falls eine Regel an mehreren Stellen des Quellmodells angewendet werden kann? Die Auswahl der Stelle kann *deterministisch*, zum Beispiel mittels Tiefensuche, *nichtdeterministisch* oder *interaktiv* durch Benutzereingaben erfolgen.

Bei der Regelauswahl sind zwei Aspekte zu beachten: Zum einen muss bei mehreren anwendbaren Regeln eine ausgesucht werden und zum anderen muss die Reihenfolge, in der die Regeln ausgeführt werden, festgelegt werden.

Regelauswahl Die Auswahl der anzuwendenden Regeln, kann bei mehreren möglichen Regeln, durch *explizite Bedingungen*, *Nichtdeterminismus*, *Konfliktauflösung* mittels Prioritäten oder *interaktiv* erfolgen.

Ablaufsteuerung Bei der Ablaufsteuerung ist zwischen *impliziter* und *expliziter Steuerung* zu unterscheiden. Sofern die Reihenfolge implizit durch das Werkzeug bestimmt wird, hat der Nutzer nur die Möglichkeit, durch das gezielte Entwerfen der Regeln, die Abarbeitungsreihenfolge zu beeinflussen. Bei Werkzeugen, die es ermöglichen eine explizite Reihenfolge festzulegen, gibt es zwei verschiedene Ansätze. *Externe Ansätze* bieten spezielle Methoden um die Reihenfolge festzulegen. Ein Beispiel dafür ist VIATRA [Varb], das mit Hilfe von endlichen Zustandsautomaten die Reihenfolge der Regel festlegt. Bei *internen Ansätzen* ist die Festlegung der Reihenfolge direkt in die Angabe der Regeln integriert, so bietet z.B. ATL [Val] die Möglichkeit aus einer Regel heraus direkt andere Regeln aufzurufen.

Angabe der Ablaufsteuerung Der Mechanismus mit dem die explizite Steuerung des Regelablaufs erfolgt.

Regelwiederholung Oft ist es sinnvoll eine Regel mehrfach auszuführen. Dabei soll nicht jede Regelausführung in der Ablaufsteuerung angeben, sondern direkt die Wiederholung spezifiziert werden. In der Praxis kommen Rekursion, Schleifen oder Fixpunktiteration zum Einsatz.

Terminierungsprüfung Eine wünschenswerte Eigenschaft von Graphtransformationssystemen ist die Zusicherung der Terminierung. Dieses Problem ist allerdings für allgemeine Graphtransformationssysteme nicht entscheidbar sondern nur für eingeschränkte Transformationsregeln. Ein Beweis für die Unentscheidbarkeit bei Verwendung des DPO-Ansatzes findet sich in [Plu98]. In [Ass00] finden sich Beispiele für eingeschränkte Graphersetzungssysteme, bei denen die Terminierung entscheidbar ist.

Modelltransformationskategorien

Nachfolgend werden die grundsätzlichen Modelltransformationskategorien beschrieben. Die Bezeichnungen sind [CH06] entnommen. In anderen Arbeiten werden die Kategorien teilweise mit leicht unterschiedlichen Namen verwendet.

Ansätze mit direkter Manipulation

Bei den Ansätzen mit direkter Manipulation wird die Darstellung des Modells hinter einer API versteckt. Der Nutzer verwendet dann die von der API zur Verfügung gestellten

Schnittstellen um Modellmanipulationen zu programmieren. Neben den Transformationen muss der Nutzer auch die Steuerung des Regelablaufs programmieren. Der Vorteil dieses Ansatzes liegt in der Verwendung gewohnter, imperativer Programmiersprachen. Allerdings muss der Nutzer alle Transformationen und deren Steuerung von Grund auf selbst programmieren. Die Unterstützung durch Werkzeuge beinhaltet nur die Darstellung der Modelle.

Operationale Ansätze

Operationale Ansätze sind zu den Ansätzen mit direkter Manipulation ähnlich, verfügen aber über gezielte Unterstützung für Modelltransformationen. Hierzu zählen zum Beispiel Metamodellierungsansätze, die mit Funktionalität zur Angabe von Berechnungen erweitert werden. „QVT Operations“ ist ein Beispiel dafür.

Relationale Ansätze

Bei relationalen Ansätzen handelt es sich um deklarative Transformationssysteme, die als Grundlage mathematische Relationen verwenden. Dabei werden zwischen Elementen des Quell- und Zielmodells Beziehungen unter Verwendung von Zusicherungen spezifiziert. Diese Zusicherungen sind in der Regel nicht ausführbar, sondern müssen erst interpretiert werden. Auf Grund ihrer mathematischen Grundlage sind die relationalen Ansätze bidirektional. Dadurch entsteht aber auch die zwingende Trennung zwischen Quell- und Zielmodell. Ersetzungen auf einem einzigen Modell sind somit nicht möglich. Beispiele sind „QVT Relations“ [QVT] oder „Tefkat“ [Law].

Auf Graphtransformationen basierend Ansätze

Diese Gruppe setzt auf die Theorie der Graphtransformation auf. Dabei wird ausgehend von einem Graph ein Teilgraph (linke Seite) durch einen durch eine Ersetzungsregel spezifizierten anderen Graphen (rechte Seite) ersetzt. Eine Transformation besteht somit aus zwei Graphmustern, der linken und rechten Seite. Beispiele sind AGG, AToM3, VIATRA, GReAT, BOTL, MOLA und Fujaba.
Da Graphtransformationen schon lange Gegenstand intensiver Forschung sind und die meisten der verwendeten Werkzeuge auf Graphtransformationen basieren, wird diese Kategorie noch genauer betrachtet. Die folgenden Ansätze stammen aus Rozenberg [Roz97]. Diese Arbeit bot als erster einen Überblick über alle Graphtransformationsansätze. Sie ist heute die Standardreferenz im Bereich der Graphtransformation und wird in vielen Arbeiten als „Graphtransformationsbibel“ bezeichnet.

Ansatz der Ersetzung von markierten Knoten Bei der Verwendung dieses Ansatzes werden einzelne Knoten (manchmal auch ganze Teilgraphen) durch einen neuen Graphen ersetzt. Formal wird eine solche Ersetzung durch (L, G ,R) angegeben, wobei L die linke Seite der Ersetzungsregel darstellt. Kann die linke Seite L auf einen Graphen H abgebildet werden, so ist die Ersetzungsregel anwendbar. Dabei wird die Abbildung von L aus dem Graphen H entfernt und durch die rechte Seite R

der Regel ersetzt. Die Ersetzungsvorschrift G erzeugt neue Kanten zwischen der neu eingesetzten rechten Seite R und dem ursprünglichen Graphen H.

Ansatz der Ersetzung von Hyperkanten Bei diesem Ansatz werden keine Knoten oder Graphmuster sondern Hyperkanten ersetzt. Hyperkanten sind eine Verallgemeinerung von Kanten, indem sie beliebig viele Kantenendpunkte zulassen. Eine Hyperkante wird jeweils durch einen Graphen ersetzt, wobei die ehemaligen Endknoten der Hyperkante zu Endpunkten neuer Kanten mit Knoten aus dem neu eingefügten Graphen werden. Es können bei dieser Art der Ersetzung keine Knoten gelöscht werden und auch Kanten können nicht wirklich gelöscht, sondern nur durch neue Teilgraphen ersetzt werden.

Algebraischer Ansatz Beim algebraischen Ansatz werden die Kanten zwischen dem zu bearbeitenden Graph und dem zu ersetzenden Teilgraph nicht gelöscht und dann neu erzeugt, sondern bleiben vorhanden. Dabei sind zwei verschiedene Verfahren zu unterscheiden.

Beim *Double-Pushout (DPO)* besteht eine Ersetzungsregel aus den drei Teilen (L, K, R) wobei L die linke Seite und R die rechte Seite angibt. K stellt einen gemeinsamen Kontext dar, also alle Knoten die sowohl in L als auch in R vorkommen.

Abbildung 3.1 greift das Beispiel aus Abschnitt 2.2 aus dem Grundlagenkapitel wieder auf. Der Kontext K besteht bei dieser Regel nur aus dem Knoten mit der Bezeichnung „2“.

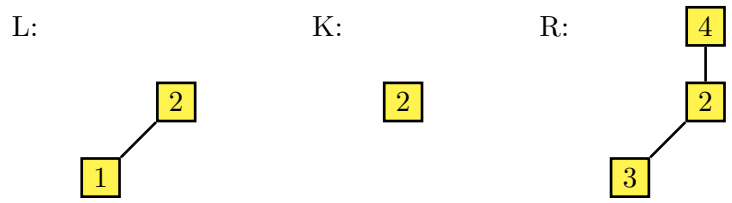


Abbildung 3.1: Verwandte Arbeiten – Double-Pushout (Kontext)

Hat man eine Übereinstimmung der linke Seite L im Wirtsgraphen H gefunden, so dürfen keine Kanten zwischen der Übereinstimmung der linken Seite und dem Rest von H verlaufen, sofern sie nicht von oder zu einem Knoten gehen, der auch im Kontext K liegt. Nachfolgend ist jeweils eine Übereinstimmung, die im Sinne des DPO-Ansatzes zulässig (H2) bzw. nicht zulässig (H1) ist, dargestellt (Abbildung 3.2).

In Übereinstimmung H1 gibt es eine Kante zu Knoten „1“ der nicht im Kontext liegt. Da dieser Knoten bei der Regelausführung gelöscht wird, hätte die Kante nach der Ausführung keine zweiten Endpunkt mehr. Deshalb ist die Übereinstimmung im Sinne des DPO-Ansatzes nicht zulässig.

In Übereinstimmung H2 gibt es nur Kanten zu Knoten „2“, der im Kontext liegt. Aus diesem Grund bleiben sämtliche Kanten auch nach der Ausführung erhal-

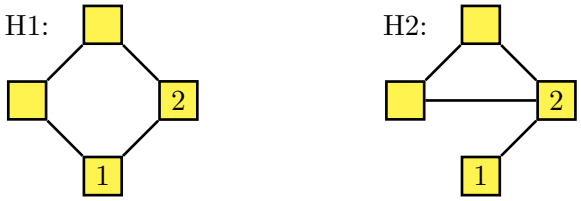


Abbildung 3.2: Verwandte Arbeiten – Double-Pushout (Übereinstimmungen)

ten, da die Knoten von K auch in der rechten Seite R wieder auftauchen. Die Übereinstimmung ist also im Sinne des DPO-Ansatzes zulässig.

Beim *Single-Pushout (SPO)* gibt es keinen Kontext K, sondern die Regeln bestehen nur aus linker und rechter Seite. Für den Fall, dass Kanten zwischen der Übereinstimmung der linke Seite und dem Rest des Wirtsgraphen H verlaufen und die beteiligten Knoten auf der rechten Seite nicht wieder auftreten, werden diese Kanten gelöscht. Eine SPO-Regel ist also immer anwendbar, wenn es eine Übereinstimmung der linken Seite mit dem Wirtsgraphen H gibt. Eine detaillierte Einführung zum algebraischen Ansatz und den DPO- und SPO-Methoden gibt [EEPT06].

Ansatz der programmierten Graphersetzung Da sich die bisher genannten Ansätze nur auf die Ersetzungsregeln konzentrieren, dabei aber die Anwendung der Regeln und die Erzeugung der zugrunde liegenden Graphen vernachlässigen, wurden programmierte Graphersetzungssysteme entwickelt. Diese ermöglichen es die Reihenfolge der Anwendung von Ersetzungsregeln zu steuern.

In der Praxis werden oft Mischungen aus algebraischen und programmierten Ansätzen verwendet, um die Vorteile von beiden nutzen zu können.

Strukturgetriebene Ansätze

Diese Ansätze bauen ein Zielmodell anhand seiner Struktur auf. Sie bestehen aus zwei Phasen. In der ersten Phase wird die hierarchische Struktur des Zielmodells erzeugt, während in der zweiten Phase die Attribute und Referenzen im Zielmodell gesetzt werden. Die Strategie zur Regelanwendung wird in der Regel von den Werkzeugen vorgegeben. Ein Beispiel hierfür ist OptimalJ [Opt].

Auf Templates basierende Ansätze

Ansätze, die auf Templates basieren, kommen meist bei Transformationen von Modellen nach Text zum Einsatz. Dabei ist der zu erzeugende Text als Template schon vor der Übersetzung vorhanden und es werden bei der Übersetzung nur noch die fehlenden Teile im Template durch Elemente des zu übersetzenden Modells ergänzt. Oft ist in dem Text Code eingebettet, der bei der Instanziierung die fehlenden Teile aus dem Modell berechnet. Zur Angabe des Codes werden oft OCL-Annotationen verwendet. Ein bekanntes Werkzeug, das diesen Ansatz unterstützt ist „openArchitectureWare“ [oAW].

Hybride Ansätze

Stellen eine Mischung aus den vorangegangenen Ansätzen dar. Möglich sind zum Beispiel Ansätze mit verschiedenen Komponenten, die aus verschiedenen Kategorien stammen. Ein Beispiel hierzu ist QVT [QVT] mit den Komponenten „QVT Relations“, „QVT Operational“ und „QVT Core“. Wobei QVT Relations und QVT Core deklarative Ansätze sind, während QVT Operational ein programmierter Ansatz ist. Sinnvolle hybride Ansätze erlauben die deklarative Spezifikation von Transformationsregeln und sowohl deklarative als auch imperative Spezifikation der Ablaufsteuerung. Imperative Sprachkonstrukte ermöglichen die Verwendung sehr komplexer Ablaufsteuerungen.

Andere Ansätze

Ein gänzlich anderer Ansatz ist die Darstellung der Modelle mittels eines Standardformats wie XML [XMLa] beziehungsweise des Modellformats XMI [XMI]. Die Modelltransformationen werden dann durch XSLT [XSL] durchgeführt. Allerdings bringt dies die Einschränkung mit sich, dass keine Änderungen am Modell durchgeführt werden können, sondern das Quellmodell nach jeder Änderung vollständig transformiert werden muss. Es handelt sich also um ein Inplace-Update mit Kopieren.

3.3 Werkzeuge

Die Zahl der Modelltransformationswerkzeuge ist sehr groß, allein in [CH06] werden zwei Dutzend verschiedene Werkzeuge und Ansätze genannt. Davon wurden einige ausgewählt und genauer betrachtet. Für die Auswahl war vor allem wichtig, dass die Werkzeuge aktuell sind, also noch weiterentwickelt werden. Außerdem mussten sie tatsächlich verfügbar und durften nicht nur in Arbeiten erwähnt worden sein. Des Weiteren wurde auf die Bekanntheit der Werkzeuge geachtet. Es wurden also Werkzeuge gewählt, die in einer großen Zahl von weiteren Arbeiten erwähnt werden.

Übriggeblieben sind aus dem Bereich der Modelltransformationswerkzeuge GReAT [GRe], VIATRA [Varb] und Fujaba [Fuj]. Dazu kommt die Graphtransformationswerkzeug AGG [AGG] und GrGen [GrG]. Zum Vergleich werden Prolog [Prob], Haskell [Has] und OWL [OWL] herangezogen, da sich damit ebenso Graphtransformationen spezifizieren lassen.

Übersicht

Nachfolgend geben die Tabellen 3.1 und 3.2 einen Überblick über die acht gewählten Werkzeuge. Danach wird ein Beispiel präsentiert, das zur Demonstration mit jedem Werkzeug umgesetzt wurde. Neben der jeweiligen Umsetzung gibt es zu jedem Werkzeug auch noch ausführliche Kommentare.

	AGG	GrGen	GReAT	VIATRA
Domäne				
Anzahl Quell-/Zielmodelle	1:1	n:m	n:m	n:m
Art der Transformation	endogen	exogen	exogen	exogen
Angabe Metamodell	nein (nur Konsistenzbedingungen möglich)	spezielle Sprache	UML-Klassendiagramm	spezielle Sprache (MOF-Import möglich)
Angabe Variablen/Muster	grafischer Editor	spezielle Sprache	UML-Klassendiagramm, OCL	spezielle Sprache
Angabe Logik	Java-Expressions	spezielle Sprache	OCL	spezielle Sprache
Typisierung	ja	ja, mit Mehrfachvererbung	ja, mit Mehrfachvererbung	ja, mit Mehrfachvererbung
Knotenattribute	beliebige Java-Objekte	einfache Typen	einfache Typen, Enumeration	einfache Typen, eigene Typen definierbar
Kantenattribute	wie Knoten	wie Knoten	nein	wie Knoten
Hyperkanten / Superkanten	nein / nein	nein / nein	nein / nein	nein / ja
Transformationsregeln				
Typ	deklarativ SPO	deklarativ SPO / imperativ	imperativ	deklarativ / imperativ
Angabe	grafischer Editor	spezielle Sprache	grafischer Editor	spezielle Sprache
Richtung	unidirektional	unidirektional	unidirektional	unidirektional
Voraussetzungen	NAC	NAC, Attribut- und Typzusicherungen	NAC, Attribut. und Typzusicherungen	NAC, Attribut- und Typzusicherungen
Parametrisierung	nein	nein	nein	alle Typen, Generizität
Quell-Ziel-Abhängigkeiten	In-place Update	In-place Update	In-place Update, getrennt , Kopieren	In-place Update
Regelanwendung				
Anwendungsort	indeterministisch (interaktiv möglich)	indeterministisch	indeterministisch	indeterministisch
Regelauswahl	indeterministisch (interaktiv möglich)	deterministisch (indeterministisch möglich)	deterministisch oder indeterministisch	deterministisch oder indeterministisch
Ablaufsteuerung	implizit	explizit	explizit	explizit
Angabe der Ablaufsteuerung	verschiedene Layer	ähnlich zu regulären Ausdrücken	Ablaufdiagramme	abstrakte Zustandsmaschine
Regelwiederholung	Fixpunktiteration	Schleife, Fixpunktiteration	Schleifen, Rekursion	rekursiv, Schleife, Fixpunktiteration
Terminierungsprüfung (siehe AGG-Beschreibung)	für eingeschränkte Grammatiken möglich			

Tabelle 3.1: Übersicht Werkzeuge - Teil 1

	Fujaba	OWL	Prolog
Domäne			
Anzahl Quell-/Zielmodelle	1:1	siehe Abschnitt OWL	n:m
Art der Transformation	exogen		exogen
Angabe Metamodell	UML-Klassendiagramm		nicht möglich
Angabe Variablen/Muster	Javacode / Storydiagramme		Prolog-Fakten
Angabe Logik	Java-Expressions		Prolog-Arithmetik
Typisierung	ja, mit Vererbung		ja
Knotenattribute	beliebige Java-Objekte		Prolog-Atome (Zahlen, Zeichenketten)
Kantenattribute	wie Knoten		wie Knoten
Hyperkanten / Superkanten	nein / nein		ja / ja
Transformationsregeln			
Typ	imperativ		deklarativ
Angabe	Storydiagramme		Prolog-Regeln
Richtung	bidirektional		unidirektional
Voraussetzungen	NAC, Attribut- und Typzusicherungen		NAC, Attribut- und Typzusicherungen
Parametrisierung	alle Typen		Prolog-Atome
Quell-Ziel-Abhängigkeiten	In-place Update, getrennt, Kopieren		In-place Update
Regelanwendung			
Anwendungsort	indeterministisch		deterministisch (Reihenfolge der Definition der Graphelemente)
Regelauswahl	deterministisch		deterministisch (Reihenfolge der Definition der Regeln)
Ablaufsteuerung	explizit		explizit
Angabe der Ablaufsteuerung	Storydiagramme		Reihenfolge der Regeln
Regelwiederholung	Schleife		Rekursion
Terminierungsprüfung			

Tabelle 3.2: Übersicht Werkzeuge - Teil 2

Beispielszenario

Das Beispiel, dem sich alle Werkzeuge, die ausführlich besprochen werden, stellen müssen, ist nachfolgend abgebildet. Es besteht aus zwei Regeln und einem Startgraphen 3.5. Die erste Regel 3.3 beschreibt eine einfache Graphtransformation und sollte von allen Werkzeugen ohne Probleme gemeistert werden. Die zweite Regel 3.4 beschreibt eine Transformation, bei der eine Kante zwischen zwei anderen Kanten entsteht. Ziel dieser Regel ist es, zu zeigen ob die Werkzeuge mit Kanten zwischen Kanten umgehen können.
Die Knoten und Kanten des Beispiels sind typisiert. Die Bezeichner geben den Typ an, wobei Kleinbuchstaben für Knotentypen und Großbuchstaben für Kantentypen stehen.

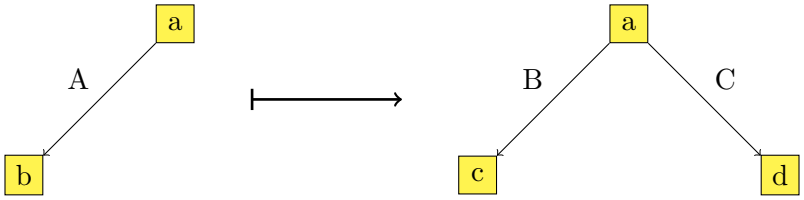


Abbildung 3.3: Transformationsbeispiel – Regel 1

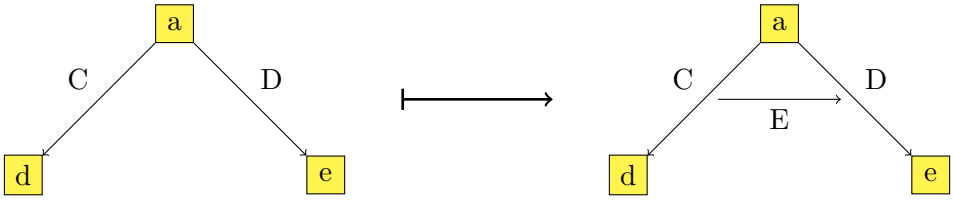


Abbildung 3.4: Transformationsbeispiel – Regel 2

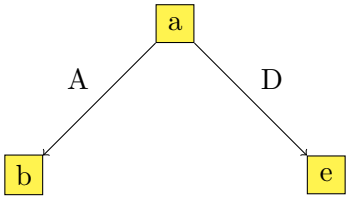


Abbildung 3.5: Transformationsbeispiel – Startgraph

3.3.1 AGG – Attributed Graph Grammars

AGG [AGG] stellt eine komplette Graphersetzungsumgebung inklusive grafischem Editor und Transformationsvisualisierung bereit. Knoten und Kanten können mit beliebigen Java-Objekten attributiert werden. Die Angabe eines Metamodells wird nicht unterstützt. Es lassen sich lediglich Konsistenzbedingungen für den Startgraphen und die Ersetzungsregeln angeben, die dann automatisch zu Beginn und während der Ausführung überprüft werden. Für Ersetzungsregeln lassen sich negative Anwendungsbedingungen (NAC) angeben. Die Ersetzungen verwenden den Single-Pushout (SPO) Ansatz. Die Ablaufsteuerung kann nur durch die Anordnung der Regeln in Layern beeinflusst werden. Dazu können bis zu zehn Layer definiert und jede Regel einem Layer zugeordnet werden. Innerhalb eines Layers werden die Regeln indeterministisch ausgeführt, während die Reihenfolge der Layer vom Benutzer vorgegeben ist. Zusätzlich gibt es einen interaktiven Modus, in dem sowohl die Auswahl der Regel als auch des Anwendungsorts durch den Benutzer erfolgt.

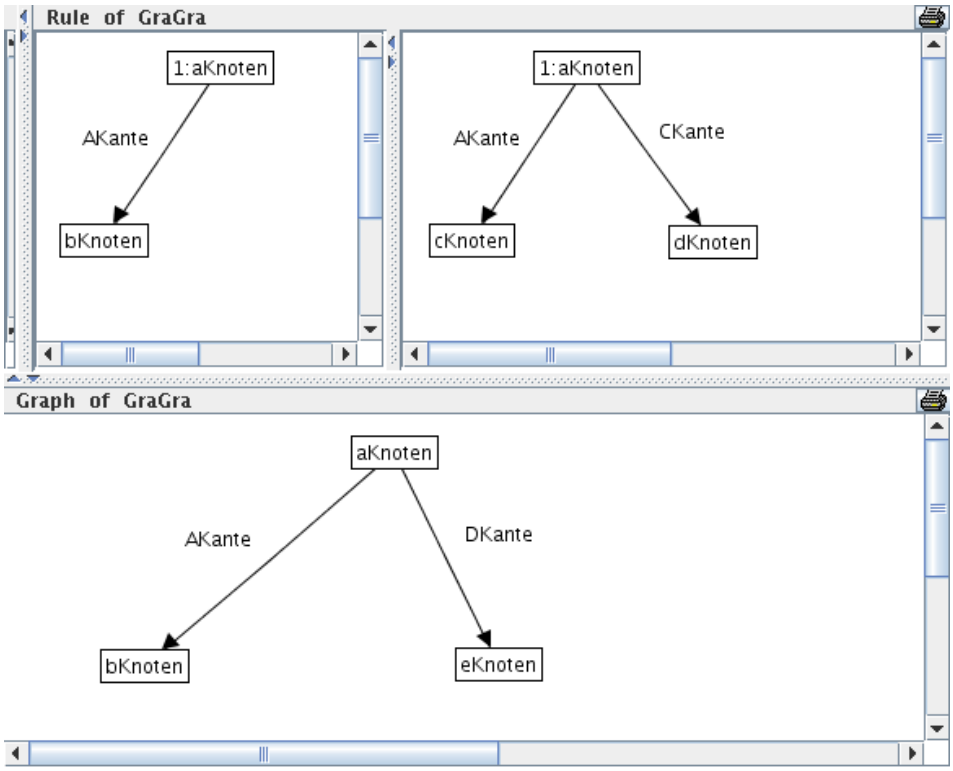


Abbildung 3.6: AGG – Regel 1 und Startgraph

AGG bietet die automatische Ermittlung von „kritischen Paaren“: Bei kritischen Paaren handelt es sich um zwei Regeln, bei denen die Ausführung der einen, die Ausführung der anderen verhindert. Dies kann in drei Fällen passieren: Regel A löscht ein Element, das für Regel B Voraussetzung ist; Regel A erzeugt ein Element, das für Regel B nicht vor-

handen sein darf oder Regel A ändert Attribute, so dass Regel B nicht mehr anwendbar ist.
Sofern die Grammatiken einigen Einschränkungen genügen, ist auch eine automatische Terminierungsprüfung möglich. Die genauen Einschränkungen an die Grammatiken sind in [EEdL+05] zu finden. Darüber hinaus unterstützt AGG das Zerteilen von eingeschränkten Grammatiken, näheres dazu findet sich in [BTS00].
Abbildung 3.6 zeigt die Modellierung von Regel 1 und des Startgraphen. Die linke und rechte Seite der Regel sind in der oberen Hälfte der Abbildung gezeigt. Knoten und Kanten, die auf beiden Seiten der Regel auftreten, sind durch entsprechende Zahlen zugeordnet. In der unteren Hälfte der Abbildung befindet sich der Startgraph. Die Markierungen der Knoten und Kanten sind mit einem Attribut *name* vom Typ char realisiert. Regel 2 lässt sich nicht umsetzen, da AGG keine Kanten zwischen Kanten darstellen kann.
Für das Beispiel wurde AGG in Version 1.5.0 verwendet.

3.3.2 GrGen – Graph Rewrite GENerator

GrGen [GrG] ist ein aus verschiedenen Werkzeugen bestehendes Graphersetzungspaket. Es besteht hauptsächlich aus dem Übersetzer *grgen* und der Ausführungsumgebung *grShell*. Der Übersetzer verarbeitet sowohl das Metamodell des Graphen als auch die Regeln zu ausführbarem Code. Der Startgraph wird erst in der Ausführungsumgebung *grShell* zur Verfügung gestellt. Als Alternative zur *grShell* steht mit der *grLib* Bibliothek auch eine Programmierschnittstelle bereit, mit dem sich GrGen aus anderen Anwendungen ansteuern lässt. Damit lassen sich Ersetzungsregeln auch imperativ angeben. Das Metamodell und die Graphersetzungsregeln werden in einer speziellen Sprache angegeben. Abbildung 3.7 zeigt ein simples Metamodell für das Beispielszenario, während Abbildung 3.8 die Umsetzung von Regel 1 zeigt.

```
 1  model BeispielModell;
 2
 3  node class aKnoten;
 4  node class bKnoten;
 5  node class cKnoten;
 6  node class dKnoten;
 7  node class eKnoten;
 8
 9  edge class AKante
10          connect aKnoten [*] -> bKnoten [*];
11  edge class BKante
12          connect aKnoten [*] -> cKnoten [*];
13  edge class CKante
14          connect aKnoten [*] -> dKnoten [*];
15  edge class DKante
16          connect aKnoten [*] -> eKnoten [*];
```

Abbildung 3.7: GrGen – Metamodell

```
 1  actions BeispielRegel using BeispielModell;
 2
 3  rule Regel1 {
 4          pattern {
 5                  a:aKnoten -A:AKante-> b:bKnoten;
 6          }
 7          replace {
 8                  a -B:BKante-> c:cKnoten;
 9                  a -C:CKante-> d:dKnoten;
10          }
11  }
```

Abbildung 3.8: GrGen – Beispielregel 1

GrGen ist typisiert und unterstützt Attribute aus den einfachen Datentypen `bool, int, string`. Für Kantentypen können Zusicherungen definiert werden, die regeln zwischen welchen Knotentypen diese Kante wie oft auftreten darf. Das Typsystem unterstützt Mehrfachvererbung, wobei die Kantenzusicherungen nicht vererbt werden.

Die Ersetzungsregeln folgen dem SPO-Ansatz und dürfen Attributbedingungen, Typzusicherungen und negative Anwendungsbedingungen (NAC) enthalten. Die Ablaufsteuerung der Regeln muss explizit angegeben werden. Dazu wird eine, an reguläre Ausdrücke angelehnte, Notation verwendet. Abbildung 3.9 zeigt die Erzeugung eines Startgraphen in der *grShell* und die Angabe der Ablaufsteuerung. Die Ablaufsteuerung umfasst nur Regel 1, da GrGen keine Kanten zwischen Kanten darstellen und somit nicht mit Regel 2 umgehen kann.

```
 1  select backend "lgspbackend.dll"
 2  new graph "lgsp-BeispielModellModel.dll" "Beispiel"
 3  select actions "lgsp-BeispielRegelActions.dll"
 4
 5  new a:aKnoten($="Knoten a")
 6  new b:bKnoten($="Knoten b")
 7  new e:eKnoten($="Knoten e")
 8
 9  new a -:AKante-> b
10  new a -:DKante-> e
11
12  grs Regel1*
```

Abbildung 3.9: GrGen – Erzeugung des Startgraphen und Regelausführung

Zur Visualisierung von Graphen steht das Tool *ycomp* zur Verfügung. Das Hauptaugenmerk bei der Entwicklung von GrGen lag auf der effizienten Ausführung der Ersetzungen. Dies wird mit Hilfe von Suchplänen erreicht, die das Auffinden einer Übereinstimmung erheblich beschleunigen. Näheres dazu findet sich in [GBG+06].

3.3.3 GReAT – Graph Rewrite and Transform

GReAT [GRe] ist ein grafisches Modelltransformationswerkzeug. Es besteht aus drei Komponenten. Die *grafische Modellierungsumgebung GME*, bietet einen grafischen Editor zur Erstellung von (Meta-)Modellen (Abbildung 3.10). Das *universelle Datenmodell UDM* speichert alle Modelle und Transformationen. *GReAT* erweitert GME um Funktionen zur Erstellung von Transformationen, sowie deren Ausführung. Alle Komponenten werden von der Vanderbilt Universität in Nashville, Tennessee entwickelt und können von [ISI] bezogen werden.

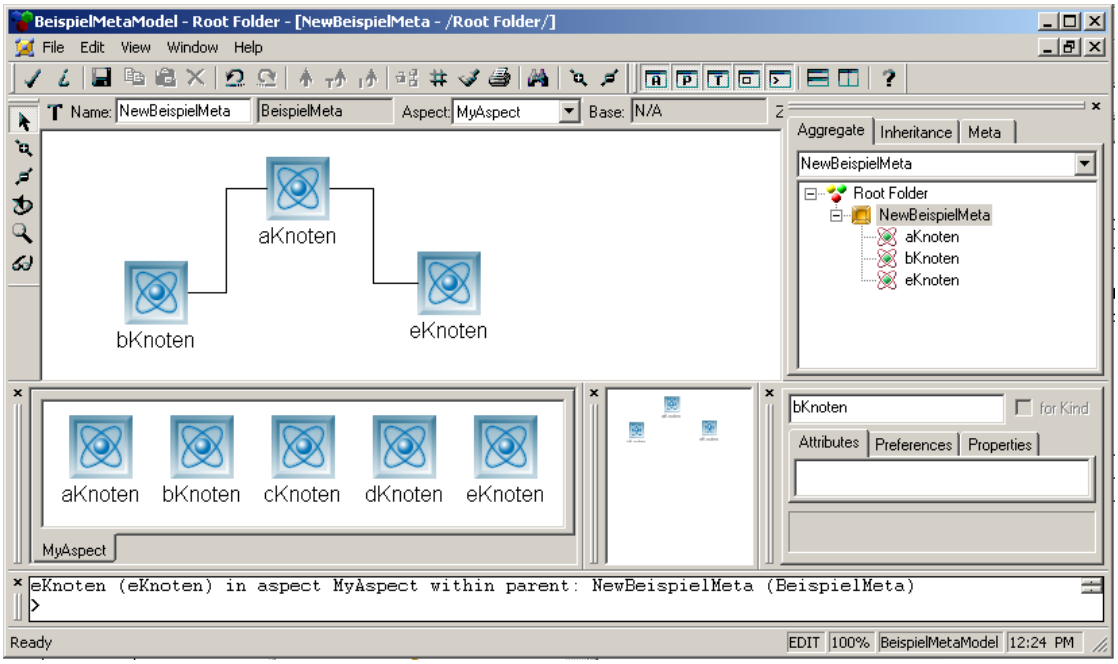


Abbildung 3.10: GReAT – GME mit Startgraph

Als Grundlage für eine Transformation muss zuerst ein Metamodell erzeugt und danach als *Paradigma* in GME registriert werden. Paradigmen dienen dann als Schema zur Erzeugung von Transformationen oder Modellen. Abbildung 3.11 zeigt das Metamodell für das Beispielszenario. Das zentrale Element ist das Modell, das eine Komposition aus verschiedenen Knoten und Kanten ist. Jede Kante ist mit einer Beziehung assoziiert, die Start- und Endpunkt der Kante angibt. Zur Demonstration von Vererbung gibt es jeweils eine Knoten- und Kantenoberklasse. Interessanterweise lassen sich in Metamodellen Kanten zwischen Kanten definieren. So stellt die „EKante“ eine Kante zwischen einer „CKante“ und einer „DKante“ dar. Allerdings kann die „EKante“ nicht in Modellen oder Transformationen verwendet werden.
Abbildung 3.10 zeigt die GME-Umgebung und den Startgraphen im mittleren Fenster. Im linken unteren Fenster sieht man die durch das Metamodell bereitgestellten Knotentypen. Die Kantentypen werden nicht explizit angegeben, sondern treten lediglich als

Verbindung im Graphen auf. Verbindungen lassen sich aber nur entsprechend dem Metamodell erzeugen. Eine Verbindung zwischen „bKnoten“ und „eKnoten“ ist zum Beispiel nicht möglich.

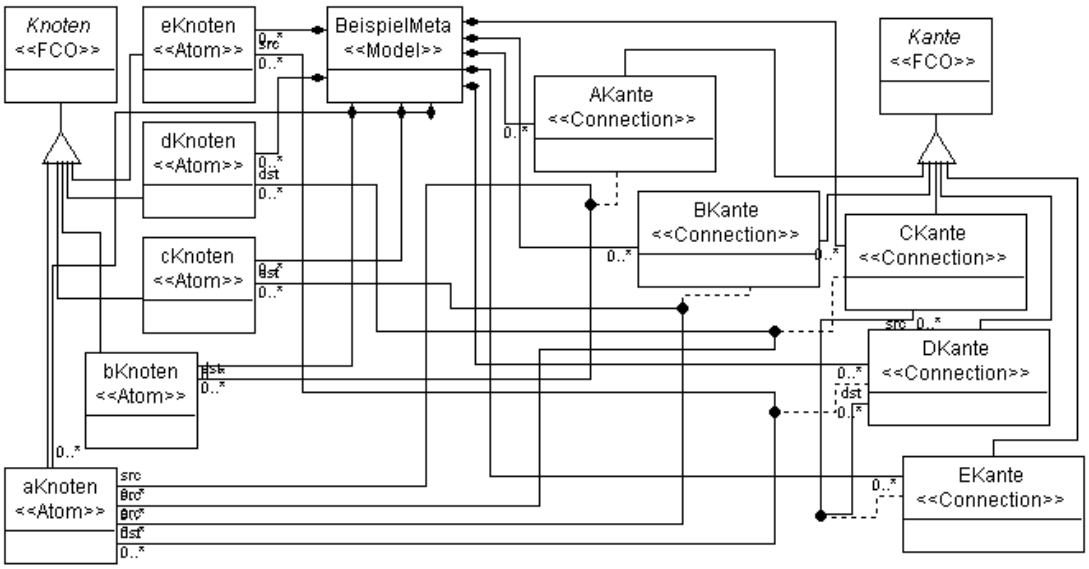
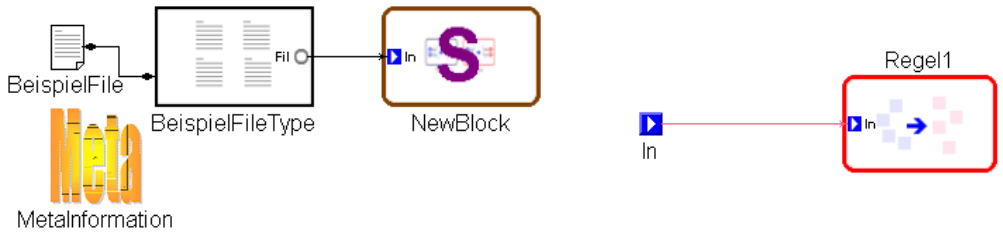


Abbildung 3.11: GReAT – Metamodell



(a) Regelkonfiguration (b) Regelsteuerung

Abbildung 3.12: GReAT – Regelkonfiguration und -steuerung

In Abbildung 3.12(a) ist die Konfiguration einer Transformation, in Abbildung 3.12(b) die Definition des Ablaufschemas einer Regel angegeben. Da es sich im Beispiel um eine Transformation mit In-place Update handelt, gibt es nur eine Datei und einen Dateityp. Die Datei stellt das zu bearbeitende Modell dar, während der Dateityp das Metamodell angibt. Diese Informationen sind in der Abbildung nicht zu sehen, sondern werden in der GME als Attribute der entsprechenden Symbole gesetzt. Das Symbol *NewBlock* stellt das Regelsystem dar und zeigt, dass das Modell an dieses Regelsystem übergeben wird. Der rechte Teil der Abbildung zeigt das Regelsystem *NewBlock*. Es besteht aus einem

Eingabeport und Regel 1. Der Eingabeport erhält die in der Konfiguration angegebenen Modelle und gibt diese an Regel 1 weiter. Wenn mehrere Regeln vorhanden sind, so können diese parallel oder sequentiell Eingabedaten erhalten. Zusätzlich lassen sich Regeln gruppieren. Informationen zur Erzeugung von Transformationen finden sich im *GReAT Step-by-Step Guide*, der in der Installation enthalten ist. Für Informationen zur Ablaufsteuerung ist das ebenfalls enthaltene *GReAT User Manual* besser geeignet.

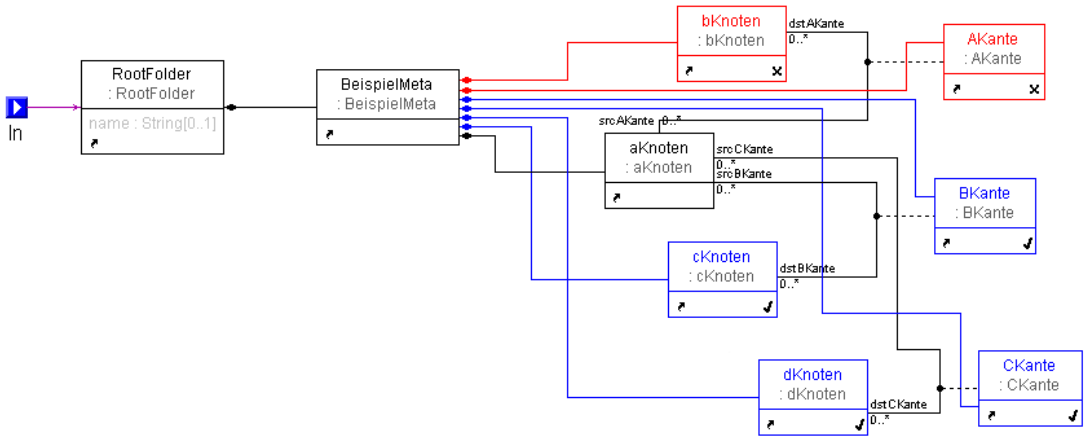


Abbildung 3.13: GReAT – Regel 1

Abbildung 3.13 zeigt die Definition von Regel 1. Links ist der Eingabeport zu sehen. Er erhält das gesamte Eingabemodell, dargestellt durch *RootFolder*. Im Eingabemodell wird ein Modell, das dem Metamodell *BeispielMeta* entspricht erwartet. Innerhalb dieses Modells wird ein Muster, bestehend aus einem *aKnoten*, einem *bKnoten* und einer *AKante* gesucht. Davon werden die rot dargestellten und mit einem Kreuz markierten Elemente gelöscht, während die blau dargestellten und mit einem Häckchen markierten Elemente neu erzeugt werden.
Da die Regeln nicht durch linke und rechte Seite angegeben, sondern die Modellelemente explizit gelöscht oder erzeugt werden, ist dieser Ansatz als imperativ einzustufen. Durch die zahlreichen Kanten in den Mustern und den Modellen kann die grafische Darstellung sehr schnell unübersichtlich werden. Ein großer Nachteil ist, dass bei einer Erweiterung des Metamodells die damit erzeugten Modelle und Transformationen ungültig werden. Die Modelle können von GME teilweise automatisch aktualisiert werden. Bei Transformationen funktioniert dies leider nicht. Bereits vorhandene Transformationen können dann nicht mit dem geänderten Metamodell genutzt werden.
Für dieses Beispiel kamen die folgenden Softwareversionen zum Einsatz: GReAT 1.6.0, GME r6.11.9 und UDM 3.1.1.

3.3.4 VIATRA – VIsual Automated model TRAnsformations

VIATRA [Varb] versteht sich als allgemeines Modelltransformationswerkzeug, das den gesamten Modellierungs- und Transformationsprozess unterstützt. Für die Bewertung wird die aktuelle Version 2, die als eclipse-Plugin realisiert ist, verwendet. Alle Artefakte, also Metamodelle, Modelle, Transformationen und die Transformationssteuerung, werden in einem sogenannten Modellraum gespeichert. Da der von der OMG spezifizierte Modellraum MOF (Meta-Object Facility) einige Einschränkungen hat, verwendet VIATRA einen eigenen, im Gegensatz zu MOF erweiterten, Modellraum. Dieser Modellraum names VPM (Visual and Precise Metamodelling) besteht aus zwei Teilen.

```
1  entity (beispiel) {
2          entity (metamodel) {
3                  entity (aKnoten);
4                  entity (bKnoten);
5                  entity (cKnoten);
6                  entity (dKnoten);
7                  entity (eKnoten);
8                  relation (AKante, aKnoten, bKnoten);
9                  relation (BKante, aKnoten, cKnoten);
10                 relation (CKante, aKnoten, dKnoten);
11                 relation (DKante, aKnoten, eKnoten);
12                 relation (EKante, aKnoten.CKante, aKnoten.DKante);
13         }
14         entity(models);
15 }
```

Abbildung 3.14: VIATRA – Metamodell

VTML (VIATRA Textual (Meta)Modelling Language) dient zur Spezifizierung von Metamodellen sowie deren konkrete Instanzen (Modelle). Ein Modellraum kann dabei beliebig viele verschiedene Metamodelle und Modelle aufnehmen. Abbildung 3.14 zeigt das Metamodell für die Beispieltransformation, während Abbildung 3.15 den Startgraphen zeigt. VTML kennt nur zwei verschiedene Elemente. Entitäten können sowohl Objekte als auch Namensräume darstellen und Relationen stellen Verbindungen zwischen den Entitäten her. Die Attribute einer Entität werden durch eine Relation mit einem Datentypen realisiert. Alle Modelle können zerteilt und somit auf Konformität gegenüber ihrem Metamodell geprüft werden. Metamodelle lassen sich auf innere Konformität prüfen.
Der zweite Bestandteil ist VTCL (VIATRA Textual Command Language) zur Spezifizierung der Transformationen und zur Angabe der Ablaufsteuerung. Eine Transformationsregel besteht aus zwei Mustern, die jeweils die linke und rechte Seite der Ersetzungsregel darstellen. In VTCL werden diese Vor- und Nachbedingung genannt. Elemente, die nur in der Vorbedingung auftreten, werden automatisch gelöscht und Elemente, die nur in der Nachbedingung auftreten, werden automatisch erzeugt. Alternativ zu dieser deklarativen Definition können Transformationsregeln auch imperativ mittels Erzeugungs- und Löschanweisungen realisiert werden. Es können auch deklarative Definitionen mit imperativen Anweisungen kombiniert werden.
Abbildung 3.16 zeigt die Definition der beiden Beispielregeln. Dabei ist besonders zu be-

```
1  namespace beispiel.models;
2  import beispiel.metamodel;
3
4  aKnoten(a);
5  bKnoten(b);
6  eKnoten(e);
7  aKnoten.AKante(A, a, b);
8  aKnoten.DKante(D, a, e);
```

Abbildung 3.15: VIATRA – Startgraph

achten, dass VIATRA auch die zweite Beispielregel beherrscht, also mit Kanten zwischen Kanten umgehen kann. Die Angabe der Ablaufsteuerung erfolgt als abstrakte Zustandsmaschine. Dabei entsprechen die einzelne Regeln (auch Befehle werden als Regeln betrachtet) den Zuständen. In einem *seq* Block werden die enthaltenen Regeln sequentiell ausgeführt, in einem *random* Block in zufälliger Reihenfolge.
Die Ausführung einer Regel kann auf zwei verschiedene Arten erfolgen. Die Anweisung *choose* sucht eine mögliche Belegung des Musters der linken Seite der Regel und wendet die Regel dann auf diese Belegung an. Mit *forall* werden zuerst alle möglichen Belegungen ermittelt, danach wird die Regel auf alle Belegungen ausgeführt. Dies ist fehleranfällig, da Regeln Seiteneffekte enthalten können. Außerdem können am Ende noch oder wieder Belegungen des Musters vorhanden sein. Um Fixpunktiteration zu erreichen, verwendet man die Anweisung *iterate choose*, welche die Regel so lange wiederholt, bis keine Belegung des Musters im aktuellen Graph mehr vorhanden ist.
Neben den Vor- und Nachbedingungsblöcken, darf in einer Transformationsregel auch ein *action* Block stehen, in dem beliebige Regelaufrufe (inklusive Rekursion) erlaubt sind. Eine gute Einführung in VIATRA findet sich in [VPB+06]. Für das Beispiel wurde VIATRA in Version 2.0.0 unter Eclipse 3.2 eingesetzt.

```
1  import beispiel.metamodel;
2
3  machine beispielmachine {
4          gtrule regel1 (in A, in B, out C, out D, out BK, out CK) = {
5                  precondition pattern lhs (A, B, AK) = {
6                          aKnoten(A); bKnoten(B);
7                          aKnoten.AKante(AK, A, B); }
8                  postcondition pattern lhr (A, B, C, D, BK, CK) = {
9                          aKnoten(A); cKnoten(C); dKnoten(D);
10                         aKnoten.BKante(BK, A, C);
11                         aKnoten.CKante(CK, A, D); } }
12
13         gtrule regel2 (in A, in D, in E, out EK) = {
14                 precondition pattern lhs (A, D, E, CK, DK) = {
15                         aKnoten(A); dKnoten(D); eKnoten(E);
16                         aKnoten.CKante(CK, A, D);
17                         aKnoten.DKante(DK, A, E); }
18                 postcondition pattern lhr (A, D, E, CK, DK, EK) = {
19                         aKnoten(A); dKnoten(D); eKnoten(E);
20                         aKnoten.CKante(CK, A, D);
21                         aKnoten.DKante(DK, A, E);
22                         aKnoten.CKante.EKante(EK, CK, DK); } }
23
24         rule main() = seq {
25                 let C = undef in
26                 let D = undef in
27                 let BK = undef in
28                 let CK = undef in
29                 forall X below beispiel.models, Y below beispiel.models
30                         with apply regel1(X, Y, C, D, BK, CK) do
31                         seq {
32                                 rename(C, "c");
33                                 move(C, beispiel.models);
34                                 rename(D, "d");
35                                 move(D, beispiel.models);
36                                 rename(BK, "B");
37                                 rename(CK, "C"); }
38                 let EK = undef in
39                 forall X below beispiel.models, Y below beispiel.models,
40                         Z below beispiel.models
41                         with apply regel2(X, Y, Z, EK) do
42                         seq {
43                                 rename(EK, "E"); }
44 }       }
```

Abbildung 3.16: VIATRA – Beispielregeln und Ablaufsteuerung

3.3.5 Fujaba – From UML to Java and back again

Fujaba [Fuj] ist ein Werkzeug für die Anwendungsmodellierung und automatische Codeerzeugung, das besonderen Wert auf den Aspekt des Round-Trip-Engineering legt. Zu diesem Zweck kommen Modelltransformationen zum Einsatz, die ein durch UML-Diagramme spezifiziertes Modell in Javacode überführen. Grundlage dieser Transformationen sind Storydiagramme [FNTZ98].
Da Fujaba die durch Storydiagramme spezifizierten Transformationen nicht direkt ausführen, sondern nur in Javacode übersetzen kann, ist es für Graphtransformationen nur bedingt geeignet. Denn die übersetzten Modelle und Transformationen müssen durch ein Javaprogramm instanziiert und ausgeführt werden. Die Ausführungsreihenfolge lässt sich zwar ebenfalls durch ein Storydiagramm modellieren, entlastet den Anwender (im Endeffekt) allerdings nicht davon, durch handgeschriebenen Javacode die Instanziierung und Ausführung anzustoßen.

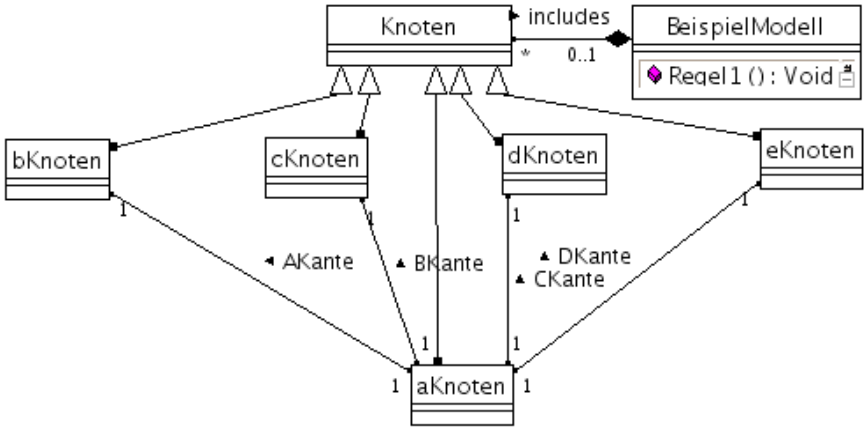


Abbildung 3.17: Fujaba – Metamodell

Abbildung 3.17 zeigt das Metamodell für das Beispiel. In der Klasse *Beispielmodell* ist eine Methode *Regel1* enthalten, die Beispielregel 1 repräsentiert. Die exakte Spezifizierung von Regel 1 erfolgt durch das in Abbildung 3.18 dargestellte Storydiagramm.
Im Storydiagramm sind linke und rechte Seite der Transformationsregel in einem einzigen Graphen dargestellt. Dabei sind Knoten bzw. Kanten, die nur auf der linken Seite auftreten, rot markiert und mit dem Zusatz *destroy* versehen, während Knoten bzw. Kanten, die nur auf der rechten Seite auftreten, grün markiert und mit dem Zusatz *create* versehen sind. Das *this*-Objekt ist an die Instanz des *Beispielmodells*, auf der *Regel1()* aufgerufen wird, gebunden. Für die restlichen Objekte der rechten Seite wird eine Übereinstimmung mit den entsprechenden Knoten des aktuellen Modells gesucht. Wird eine Übereinstimmung gefunden, so wird die Ersetzung ausgeführt.
Abbildung 3.19 zeigt den Javacode zur Instanziierung des Beispielmodells und zur Ausführung von Regel 1.

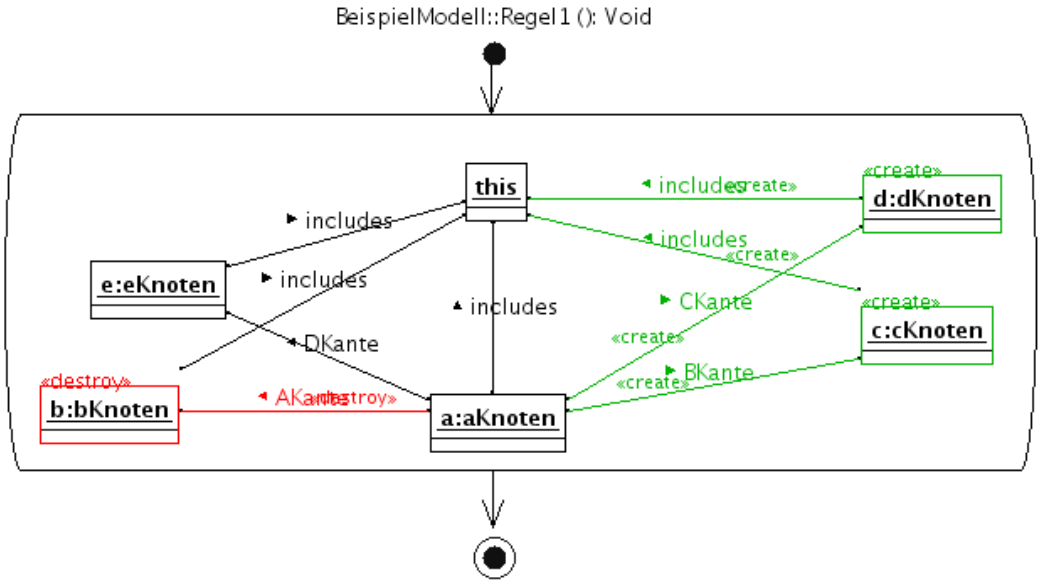


Abbildung 3.18: Fujaba – Regel 1

MoTE (Model Transform Engine) ist ein Plugin für Fujaba, dass TGG-Diagramme (TripleGraphGrammar) [Sch94] bereitstellt. TGG-Regeln dienen allerdings ausschließlich der Synchronisierung von verschiedenen Modellen und können nicht zur Manipulation eines einzelnen Modells verwendet werden, da sie nur Modellelemente erzeugen und nicht löschen können. Bei TGG-Regeln kann auf das Löschen von Elementen verzichtet werden, da die TGG-Regeln immer auf einen Graphen angewendet werden und dann den jeweils anderen Graphen daraus neu generieren. Aus den TGG-Regeln werden automatisch Storydiagramme erzeugt, die dann wiederum in Javacode übersetzt werden. Zu MoTE gehört allerdings auch noch das Plugin MoRTEn (Model Round-Trip Engineering) mit dessen Hilfe die Transformationen dann direkt innerhalb der grafischen Oberfläche von Fujaba ausgeführt werden können.
Für das Beispiel wurde die Fujaba Tool Suite in Version 4.3.2 verwendet.

```
 1  package beispiel;
 2
 3  public class test {
 4          public static void main(String args[]) {
 5                  // Startgraph erzeugen
 6                  BeispielModell modell = new BeispielModell();
 7
 8                  // Knoten für den Startgraph erzeugen und hinzufügen
 9                  aKnoten a = new aKnoten();
10                  modell.addToKnoten(a);
11                  bKnoten b = new bKnoten();
12                  modell.addToKnoten(b);
13                  eKnoten e = new eKnoten();
14                  modell.addToKnoten(e);
15
16                  // Kanten für den Startgraph erzeugen
17                  a.setBKnoten(b);        // AKante
18                  a.setEKnoten(e);        // DKante
19
20                  // Regel 1 ausführen
21                  modell.Regel1();
22          }
23  }
```

Abbildung 3.19: Fujaba – Instanziierungscode

3.3.6 Prolog – PROgramming in LOGic

Prolog [Prob] ist der bekannteste Vertreter unter den Logikprogrammiersprachen. Auf den ersten Blick hat Prolog nur wenig mit Graphtransformationen gemeinsam. Allerdings handelt es sich beim Ableiten von neuem Wissen mittels Regeln um den prinzipiell gleichen Vorgang wie bei Graphtransformationen. Dabei repräsentiert der Graph das bereits vorhandene Wissen und die Transformationsregeln legen fest, welches neue Wissen daraus generiert werden kann.
Abbildung 3.20 zeigt wie Attribute, NACs und Parameter in Prolog umgesetzt werden. Das erste Faktum legt fest, dass Knoten vom Typ „a_knoten“ aus einem Bezeichner und einem Attribut mit Zahlenwert bestehen. Die definierte Regel hat einen Parameter, der auf das Attribut des Knotens addiert wird. Dazu muss das Faktum, das den Knoten repräsentiert, gelöscht und mit dem neuen Attributwert wieder erzeugt werden. Negative Anwendungsbedingungen können mit Hilfe des Operators \+ definiert werden. Dieser lässt eine Abarbeitung fehlschlagen, falls das hinter dem Operator stehende Prädikat zu wahr ausgewertet wurde. Dies kann sowohl auf Elemente des Graphen als auch auf Attribute von Elementen angewendet werden. Die definierte Regel wird nur ausgeführt, wenn kein Knoten vom Typ „c_knoten“ existiert und ein Knoten vom Typ „b_knoten“ gefunden wird, dessen Attributwert nicht kleiner als zehn ist.

```
 1  a_knoten(knoten1,0).
 2
 3  regel(P) :-
 4          /* Elemente der linken Seite der Transformationsregel binden */
 5          a_knoten(Xan,Y),
 6
 7          /* Attribut um Paramterwert erhöhen */
 8          Y1 is Y+P,
 9          retract(a_knoten(Xan,Y,Z)),
10          assertz(a_knoten(Xan,Y1,Z)),
11
12          /* auf NAC prüfen */
13          \+c_knoten(Xcn),
14          b_knoten(Xbn, Z),
15          \+Z<10.
```

Abbildung 3.20: Prolog – Attribute, NACs und Parameter

Prolog kann zwar Hyper- und Superkanten definieren und verarbeiten, hat aber zwei gravierende Nachteile. Der erste Nachteil ist das Fehlen von Metamodellen. Dadurch werden Transformationen ermöglicht, die das Modell beschädigen. Prolog prüft zwar nach dem ersten Auftreten eines Prädikats alle nachfolgenden Verwendungen auf Übereinstimmung bezüglich der Anzahl und Art der Attribute, allerdings führen Tippfehler bei Prädikatnamen umgehend zur Definition eines neuen Prädikats und nicht zu einem Fehler. Der zweite Nachteil ist die eingeschränkte Mächtigkeit bei der Definition der Reihenfolge der Regelabarbeitung. So werden Regeln in genau der Reihenfolge, in der sie aufgerufen werden, abgearbeitet. Die einzige Möglichkeit Regeln zu wiederholen besteht darin, die Regeln rekursiv zu definieren.

Abbildung 3.21 zeigt die Umsetzung des Beispiels. Die Knoten und Kanten des Startgraphen werden als Fakten angegeben. Ein Knoten besteht aus einem Bezeichner und einer Zuordnung zu einem Knotentyp. Das Faktum „a_knoten(knoten1)“ definiert zum Beispiel einen Knoten vom Typ „a_knoten“ mit dem Bezeichner „knoten1“. Eine Kante besteht aus einem Bezeichner und den beiden Endknoten, die durch ihre Bezeichner identifiziert werden. So definiert zum Beispiel „a_kante(knoten1, kante3, knoten14)“ eine Kante vom Typ „a_kante“ mit dem Bezeichner „kante3“ zwischen den beiden Knoten „knoten1“ und „knoten14“.
Eine Transformationsregel wird in zwei Schritten in eine Prologregel überführt. Zuerst müssen mittels der entsprechenden Prädikate die Elemente der linken Seite der Transformationsregel an Variablen gebunden werden. Danach werden die auf der rechten Seite der Ersetzungsregel neu auftretenden Elemente erzeugt und die nicht mehr vorhandenen gelöscht. Für die neu erzeugten Fakten muss jeweils ein eindeutiger Bezeichner generiert werden. Dazu wird die Funktion *gensym* verwendet, die [CM03] entnommen ist. Sie erwartet als Parameter einen Bezeichner und eine Variable und weist bei jedem Aufruf der Variable den um eine fortlaufende Nummer erweiterten Bezeichner zu. Die neu erzeugten Fakten werden mit der Funktion *assertz* der Wissensbasis hinzugefügt. Zum Entfernen von Fakten wird die Funktion *retract* verwendet.
Der Code wurde mit GNU Prolog Version 1.2.18 [Prob] getestet und verwendet die für diese Prologvariante gültige Syntax und Funktionen.

```
1  :- dynamic(a_knoten/1,b_knoten/1,c_knoten/1,d_knoten/1,e_knoten/1,
2                  a_kante/3,b_kante/3,c_kante/3,d_kante/3,e_kante/3).
3
4  /* Startgraph */
5  a_knoten(a_knoten01). b_knoten(b_knoten01). e_knoten(e_knoten01).
6  a_kante(a_knoten01, a_kante01, b_knoten01).
7  d_kante(a_knoten01, d_kante01, e_knoten01).
8
9  /* Regel 1 */
10 regel1 :-
11         a_knoten(Xan),
12         b_knoten(Xbn),
13         a_kante(Xan, Xae, Xbn),
14         gensym(c_knoten,Xcn),
15         gensym(d_knoten,Xdn),
16         gensym(b_kante,Xbe),
17         gensym(c_kante,Xce),
18         assertz(c_knoten(Xcn)),
19         assertz(d_knoten(Xdn)),
20         assertz(b_kante(Xan, Xbe, Xcn)),
21         assertz(c_kante(Xan, Xce, Xdn)),
22         retract(a_kante(Xan, Xae, Xbn)),
23         retract(b_knoten(Xbn)).
24
25 /* Regel 2 */
26 regel2 :-
27         a_knoten(Xan),
28         d_knoten(Xdn),
29         e_knoten(Xen),
30         c_kante(Xan, Xce, Xdn),
31         d_kante(Xan, Xde, Xen),
32         gensym(e_kante,Xee),
33         assertz(e_kante(Xce, Xee, Xde)).
34
35 /* Transformation */
36 transform :-
37         regel1, regel2.
38
39 /* Graph ausgeben */
40 ausgeben :-
41         listing(a_knoten), listing(b_knoten), listing(c_knoten),
42         listing(d_knoten), listing(e_knoten),
43         listing(a_kante), listing(b_kante), listing(c_kante),
44         listing(d_kante), listing(e_kante).
```

Abbildung 3.21: Prolog – Startgraph und Beispielregeln

3.3.7 OWL – Web Ontology Language

Im vorangegangenen Abschnitt wurde gezeigt, dass es sich bei Graphen prinzipiell um Wissensrepräsentation (KR – Knowledge Representation) und bei Transformationen um Ableitung neuen Wissens handelt. Als Alternative zu der bereits vorgestellten Sprache Prolog wird jetzt OWL [OWL] betrachtet. OWL wurde 2003 von der OMG [OMG] als Sprache zur Repräsentation von Wissen für das Semantikweb entwickelt.
Dieser sehr spezifische Einsatzzweck von OWL bringt allerdings einige Einschränkungen mit sich. Da OWL rein für die Wissensrepräsentation entwickelt wurde, ist die Sprache statisch, das heißt, es können im Gegensatz zu Prolog keine neuen Fakten erzeugt bzw. abgeleitet werden. Ein Werkzeug zur automatischen Ableitung von Wissen (Reasoner) kann in OWL lediglich zusätzliches Wissen über bereits definierte Fakten ableiten.

OWL kennt drei wichtige Entitäten: Klassen, Individuen und Eigenschaften. Klassen sind Mengen von Individuen, die gemeinsame Eigenschaften besitzen. Individuen können als Instanzen von Klassen betrachtet werden. Allerdings können in OWL, im Gegensatz zum Konzept der Objektorientierung, auch direkt Individuen ohne eine zugehörige Klasse definiert werden. Eigenschaften (Properties) setzen entweder Individuen zueinander in Beziehung oder sie realisieren ein Attribut, indem sie ein Individuum mit einem Datentyp in Beziehung setzen. Dazu können die aus XML-Schema [XMLb] bekannten Datentypen verwendet werden. Abbildung 3.22 zeigt den Startgraph des Beispiels in OWL.
Darin werden die Klassen „A-Knoten“, „B-Knoten“, „C-Knoten“ sowie die Oberklasse „Knoten“ definiert. Zusätzlich werden die beiden im Startgraphen vorhandenen Kanten als Eigenschaften modelliert. Am Ende wird von jeder Knotenklasse eine Instanz erzeugt und die entsprechenden Kanten werden gesetzt.
Mit dieser Ontologie ist zwar der Startgraph korrekt repräsentiert, allerdings bietet OWL keine Möglichkeit darauf aufbauend Transformationen auszuführen, da es keine Möglichkeit gibt neue Individuen zu erzeugen. Eine Prologregel, wie z.B. „kante(X,Y) :- knoten(X), knoten(Y)“, kann in OWL nicht realisiert werden.
Der Reasoner hat unter OWL zwei Aufgaben. Zum einen muss er die Konsistenz der Ontologie, also die Korrektheit der Vererbungshierarchie sowie aller Zusicherungen, überprüfen. Zum anderen muss er Anfragen an die Ontologie beantworten können. Dazu gehören z.B. das Zuordnen eines Individuums zu den Klassen, zu denen es gehört oder die Auflistung aller Oberklassen einer gegebenen Klasse.

Der Grund, dass OWL ungeeignet für Graphtransformationen ist, liegt im Ursprung von OWL, der in der Beschreibungslogik liegt. Beschreibungslogik hat das Ziel Wissen zu repräsentieren, während im Gegensatz dazu Prolog, das aus der Logikprogrammierung stammt, Wissen nicht nur repräsentiert, sondern auchverarbeitett. DKonsequenzenz daraus ist, dass Prolog im Gegensatz zu OWL nicht mehr entscheidbar ist.[2] Ein Prolog-Reasoner kann im Gegensatz zu einem OWL-Reasoner nicht garantieren, dass er jede mögliche Ableitung von Wissen auch tatsächlich findet.

[2]Tatsächlich sind nur die beiden OWL-Teilmengen OWL Lite und OWL DL entscheidbar. OWL Full ist auf Grund seiner Mächtigkeit nicht mehr entscheidbar.

Seit dem Jahr 2000 gibt es die *Rule Markup Initiative* [Rul], die sich das Ziel gesetzt hat, eine Regelsprache zu entwickeln, die auf RDF/OWL aufsetzend auch die Ableitung von Wissen mittels Regel ermöglicht. Die Entwicklung von *RuleML* befindet sich noch im Entwurfsstatus, so dass es bisher keine Spezifikation bzw. passenden Werkzeuge gibt. Für die Umsetzung des Startgraphen wurde das Ontologieeditor Protégé [Proa] in Version 3.2.1 verwendet. Für die Konsistenzprüfung der Ontologie wurde der OWL-Reasoner Pellet [Pel] in Version 1.4-RC1 eingesetzt.

```
 1  <?xml version="1.0"?>
 2  <rdf:RDF
 3      xmlns="http://www.owl-ontologies.com/Ontology1168193502.owl#"
 4      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 5      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 6      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 7      xmlns:owl="http://www.w3.org/2002/07/owl#"
 8    xml:base="http://www.owl-ontologies.com/Ontology1168193502.owl">
 9    <owl:Ontology rdf:about=""/>
10    <owl:Class rdf:ID="E-Knoten">
11      <rdfs:subClassOf>
12        <owl:Class rdf:ID="Knoten"/>
13      </rdfs:subClassOf>
14    </owl:Class>
15    <owl:Class rdf:ID="A-Knoten">
16      <rdfs:subClassOf rdf:resource="#Knoten"/>
17    </owl:Class>
18    <owl:Class rdf:ID="B-Knoten">
19      <rdfs:subClassOf rdf:resource="#Knoten"/>
20    </owl:Class>
21    <owl:ObjectProperty rdf:ID="D-Kante">
22      <rdfs:domain rdf:resource="#A-Knoten"/>
23      <rdfs:range rdf:resource="#E-Knoten"/>
24    </owl:ObjectProperty>
25    <owl:ObjectProperty rdf:ID="A-Kante">
26      <rdfs:range rdf:resource="#B-Knoten"/>
27      <rdfs:domain rdf:resource="#A-Knoten"/>
28    </owl:ObjectProperty>
29    <A-Knoten rdf:ID="A-Knoten_1">
30      <D-Kante>
31        <E-Knoten rdf:ID="E-Knoten_1"/>
32      </D-Kante>
33      <A-Kante>
34        <B-Knoten rdf:ID="B-Knoten_1"/>
35      </A-Kante>
36    </A-Knoten>
37  </rdf:RDF>
```

Abbildung 3.22: OWL – Startgraph

3.4 Werkzeugauswahl

Von den vorgestellten Werkzeugen können nur VIATRA und GReAT ansatzweise mit Superkanten umgehen. VIATRA kann Superkanten in Modellen definieren und in Regeln verwenden, aber keine Graphen mit Superkanten erzeugen. In GReAT können Superkanten nur in Modellen definiert, aber weder in Regeln noch in Graphen verwendet werden. Um mehr Informationen zu erhalten, wieso beide Werkzeuge Superkanten teilweise unterstützen, wurden die Verantwortlichen für die beiden Werkzeuge kontaktiert.
Bei VIATRA entsteht die Möglichkeit, Kanten zwischen Kanten zu definieren, durch die Art der internen Repräsentation von Graphen. Allerdings handelt es sich hierbei um ein unbeabsichtigtes Nebenprodukt der Entwicklung, weshalb die Eingabesyntax für Graphen keine Möglichkeit zur Eingabe von Superkanten vorsieht.
In GReAT ist die Möglichkeit, in Modellen Superkanten zu definieren, ebenfalls ein unbeabsichtigtes Nebenprodukt. Ein Konzept für Superkanten gibt es nicht und auch die Unterstützung von Superkanten in Regeln und Graphen ist nicht vorgesehen.
Leider beherrscht keines der Werkzeuge den Umgang mit Hyperkanten, weshalb für die Umsetzung dieser Arbeit nicht nur eine Erweiterung um Superkanten erfolgen muss, sondern als erster Schritt eine Erweiterung um Hyperkanten notwendig ist.
Bei der Wahl eines geeigneten Werkzeuges als Grundlage für die geplanten Erweiterungen kamen VIATRA und GrGen in die engere Auswahl. AGG eignet sich nicht, da die Ablaufreihenfolge von Regeln nur eingeschränkt beeinflusst und kein Metamodell angegeben werden kann. Außerdem macht es der grafische Editor für die Regel- und Grapheingabe schwierig, Erweiterungen zu realisieren. GReAT scheitert aus, da die Eingabe von Regeln nicht deklarativ möglich ist und ebenfalls durch einen grafischen Editor erfolgt. Fujaba ist ein Werkzeug zur Codeerzeugung und nur bedingt für Graphersetzungen geeignet. Prolog kann zwar prinzipiell alle Arten von Graphersetzungen ausführen, bietet aber keinerlei Unterstützung für den Anwender, wie zum Beispiel durch spezielle Syntax für Modelle oder Regeln. OWL ist nur zur Darstellung von Graphen, aber nicht zur Weiterverarbeitung geeignet.
Im Gegensatz zu GrGen lässt VIATRA beliebige Typen als Attribute und Parameter von Regeln zu. Diese beiden Punkte spielen für die Umsetzung der Anforderungen allerdings keine Rolle und wurden deshalb nur gering gewichtet. Ansonsten bieten VIATRA und GrGen die gleiche Funktionalität. Die Entscheidung fiel zu Gunsten von GrGen, da es die einfachere und besser strukturierte Syntax für die Eingabe von Regeln bietet. Außerdem gibt es bei GrGen sehr engen Kontakt mit den Entwicklern. Dies ist ein großer Vorteil, da die meisten der verfügbaren Werkzeuge nur sehr rudimentäre Dokumentationen bieten.

4 Formalisierung von Superkanten

In diesem Kapitel wird ein Formalismus für Superkanten vorgestellt. Danach wird ein Konzept für die Umsetzung von Superkanten und die bei der Zielbestimmung im Abschnitt 1.2 besprochenen Graphelemente auf die in GrGen verfügbaren Graphelemente (Knoten, gerichtete Kanten und Typkonzept mit Attributierung) entwickelt. Zur Erinnerung, es müssen die folgenden Elemente behandelt werden:

- Superkanten (mit Tentakeln und entsprechenden Rollen)
- Dynamische Attribute
- Mengensemantik (für Tentakel und dynamische Attribute)

4.1 Formalismus

Dieser Abschnitt stellt die reine Formalisierung kompakt und übersichtlich dar. Die ausführlich Beschreibung inklusive Motivation folgt in Abschnitt 4.2.

4.1.1 Formale Definition einer Superkante

$G = (E, R, S, rol)$ definiert einen Graph mit Superkanten. Dabei ist E eine Menge von Graphelementen und R eine Menge von Rollen.
Für die Menge S gilt:

$$S \subseteq \{(e, T) \mid e \in E \wedge T \in \mathcal{P}(E)\}$$

und

$$\forall s_1, s_2 \in S : (s_1 = (e_1, T_1), s_2 = (e_2, T_2), e_1 = e_2) \Rightarrow (s_1 = s_2 \wedge T_1 = T_2).$$

Für die Abbildung rol gilt:

$$rol : \mathcal{T} \rightarrow R$$

mit $\mathcal{T}$ als Menge von „Tentakeln“:

$$\mathcal{T} = \{(e_1, e_2) \mid e_1 \in E \wedge e_2 \in \tau(e_1)\},$$

wobei die Funktion τ einer Kante e die Menge ihrer Tentakel zuordnet:

$$\tau(e) = \{t \mid t \in T \text{ mit } (e, T) \in S\}.$$

4.1.2 Anmerkungen

Die Menge E enthält alle Knoten und alle Kanten. Unterschieden werden sie anhand der Anzahl ihrer Tentakel (Knoten haben null Tentakel). Die Menge S besteht aus Tupeln, die einem Graphelement eindeutig seine Tentakel zuordnen. Dabei kann die Menge der zugeordneten Tentakel leer sein. Ein Knoten ist folglich ein Graphelement, dem durch S keine Tentakel zugeordnet sind. Für die Menge der Knoten V gilt also:

$$V = \{e \in E \mid \forall T \in \mathcal{P}(E) \setminus \emptyset : (e, T) \notin S\}.$$

Die Funktion *rol* ordnet einem Tentakeln eine Rolle zu. Ist *rol* surjektiv, so muss jedem Tentakel einer Kante eine Rolle zugeordnet sein. Die Formalismen für Typen, Attribute und Markierungen können analog Kapitel 2 auch auf einen Graphen mit Superkanten angewendet werden.

4.1.3 Beispiel

Abbildung 4.1 zeigt den Graph $G = (E_S, R_S, S_S, rol_S)$ mit $E = \{a, b, c, s, t\}$, $S = \{(s, \{a, b\}), (t, \{b, c, s\})\}$, $R = \{Q, R\}$ und $rol : (s, a) \mapsto R; (s, b) \mapsto Q$.

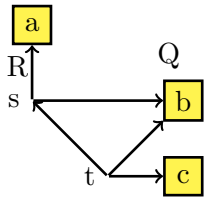


Abbildung 4.1: Formalismus – Beispiel eines Supergraphen

4.2 Konzept der Umsetzung

Für die Umsetzung der Erweiterungen auf die von GrGen unterstützten Graphelemente wird zuerst ein Konzept für Hyperkanten entwickelt. Dieses Konzept wird danach Schritt für Schritt auf Superkanten, Rollen, Attribute und Mengensematik erweitert.

4.2.1 Hyperkanten

Betrachtet man die Darstellung einer Hyperkante, so kommt man schnell auf die Idee, diese durch Einfügen eines speziellen Verbindungskonten auf normale Knoten und Kanten zurückzuführen. Der Verbindungsknoten erhält einen speziellen Typ, der ihn von allen anderen Knoten im Graph unterscheidet. Somit ist es möglich den Verbindungsknoten durch ein Graphersetzungssystem vor dem Anwender zu verbergen. Außerdem kann gewährleistet werden, dass ein Verbindungsknoten nicht irrtümlich als Zielknoten einer anderen Kante verwendet wird. Abbildung 4.2 zeigt die Realisierung einer Hyperkante durch Einfügen eines Verbindungsknoten.

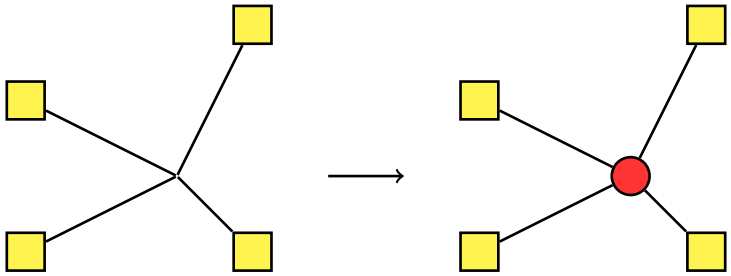


Abbildung 4.2: Konzept – Realisierung einer Hyperkante

Die Realisierung einer Hyperkante durch einen zusätzlichen Knoten kann der Anwender auch selbst vornehmen. Allerdings bringt die Unterstützung durch das Graphersetzungssystem eine deutliche Arbeitserleichterung, da der Anwender bei der Grapheingabe nur ein einziges Graphelement eingeben muss. Bei der Realsierung von Hand müsste der Anwender neben dem Verbindungsknoten für jedes Tentakel eine eigene Kante eingeben. Dieser Vorteil gilt analog für die Definition von Regeln. Auch dabei muss sich der Anwender keine Gedanken um die Verbindungskanten machen, sondern kann eine Hyperkante als Menge von Tentakeln eingeben.

4.2.2 Superkanten

Realisiert man Hyperkanten durch Einfügen eines Verbindungsknotens, so lassen sich daraus Superkanten erzeugen, indem der Verbindungsknoten der Hyperkante als Endpunkt einer anderen Kante verwendet wird. Allerdings ist dabei ein Problem zu beachten, zu dessen Veranschaulichung der folgende Sachverhalt dient: Eine Hyperkante mit den Knoten C und D soll Tentakel einer anderen Hyperkante sein, die zusätzlich die Knoten A und B enthält. In der in Abschnitt 5.2.1 vorgestellten Syntax wird der Sachverhalt folgendermaßen ausgedrückt: `[A B [C D]]`.
Eine mögliche Realisierung ist in Abbildung 4.3 dargestellt.

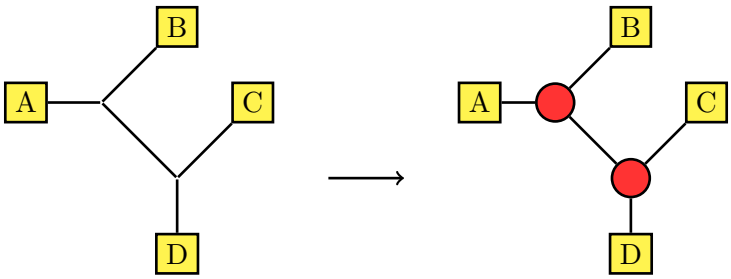


Abbildung 4.3: Konzept – Realisierung einer Superkante (Versuch)

Der Betrachter stellt sofort fest, dass er den dargestellten Sachverhalt nicht verstehen kann: Die Abbildung bringt nicht zum Ausdruck, welche Hyperkante ein Tentakel der anderen ist, sondern verbindet beide Hyperkanten gleichberechtigt. Dieses Problem betrifft allerdings nicht nur die auf der rechten Seite dargestellte Realisierung, sondern auch die

Ausgangsdarstellung auf der linken Seite. Um das Problem zu lösen, muss die Kante, die beide Hyperkanten verbindet, eine Richtung erhalten. Damit ein einheitliches Konzept entsteht, werden alle Kanten zwischen dem Verbindungsknoten und den Endpunkten einer Hyperkante gerichtet (siehe Abbildung 4.4).

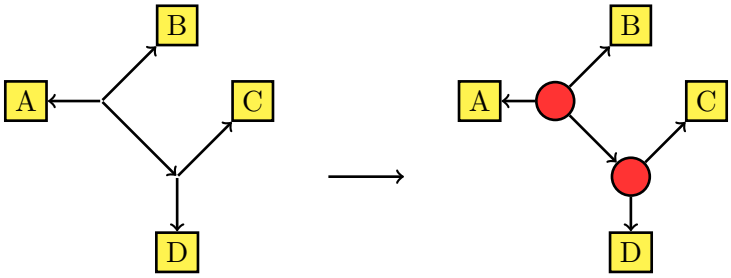


Abbildung 4.4: Konzept – Realisierung einer Superkante

Damit ist eine Superkante durch einen Verbindungsknoten, der mit den Endpunkten durch eine gerichtete Kante verbunden ist, realisiert. Da es sich bei Superkanten um eine Verallgemeinerung von Kanten und Hyperkanten handelt, wird in Zukunft nicht mehr zwischen Hyperkanten und Superkanten unterschieden, es kommen nur noch Superkanten zum Einsatz. Der Verbindungsknoten bekommt den Typ *SUPEREDGE*, damit er sich von anderen Knoten unterscheidet und das Graphersetzungssystem ihn vor dem Anwender verbergen kann.

4.2.3 Rollen

Bereits in Abschnitt 2.1 „Grundlagen“ wurde erklärt, dass bei der Verwendung von Hyperkanten Rollen zur Verfügung stehen müssen, damit den Tentakeln der Hyperkante Semantik zugewiesen werden kann. Dies gilt für natürlich auch für Superkanten.
Bevor die Frage der Realisierung von Rollen geklärt werden kann, muss zuerst untersucht werden, an welcher Stelle im Graph eine Rolle tatsächlich auftritt. Dazu ein Beispiel, bestehend aus den beiden Sätzen:

> „Ein Programmierer|AG arbeitet|ACT“ und „Ein Programmierer|RECP erhält|ACT Gehalt|HAB“.

In Abbildung 4.5 sind die Sätze dargestellt und bereits mit Rollen versehen.

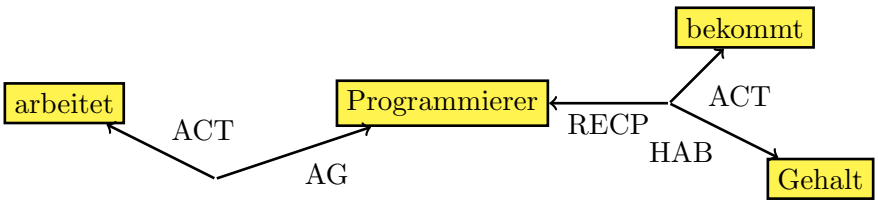


Abbildung 4.5: Konzept – Rollenzuordnung

Man sieht, dass der Knoten „Programmierer“ zwei verschiedene Rollen hat. Von jeder Superkante, an der er beteiligt ist, erhält er eine Rolle. Im Allgemeinen kann ein Knoten also beliebig viele Rollen haben, da er an beliebig vielen Superkanten beteiligt sein kann. Darüber hinaus zeigt die Darstellung, dass eine Rolle auch nicht einfach der Superkante zugeordnet werden kann, da eine Superkante für jeden zugehörigen Knoten eine Rolle enthalten kann. Die Rollen gehören also an das Tentakel (die Kombination aus Knoten und dem Teil der Superkante, der am Knoten endet).
Bei der Realisierung einer Superkante durch einen zusätzlichen Verbindungsknoten stellt die Zugehörigkeit einer Rolle zum Tentakel kein Problem dar, da ein Tentakel durch eine eigene Kante, die zum entsprechenden Knoten führt, realisiert wird. Eine Rolle muss also der Kante zwischen Verbindungsknoten und dem eigentlichen Knoten zugeordnet werden. Die nächstliegende Lösung wäre, der Kante ein Attribut zuzuordnen, das die Rolle aufnimmt. Es gibt aber eine elegantere Lösung, die ohne ein Attribut auskommt: Die Rolle wird als Typ der Kante realisiert. Dies ist in Abbildung 4.6 dargestellt (zur Verdeutlichung sind auch die Typen der anderen Graphelemente angegeben).

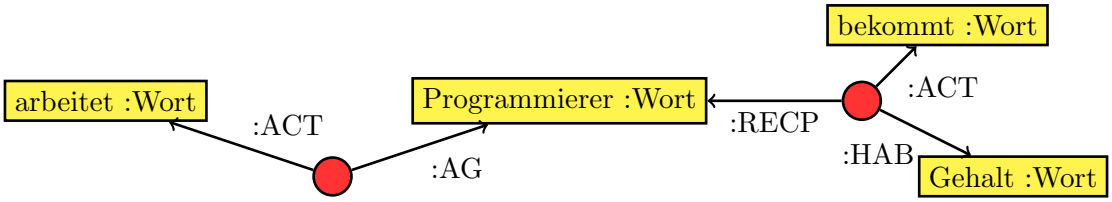


Abbildung 4.6: Konzept – Realisierung von Rollen

4.2.3.1 Rollenlisten

Nachdem die Entscheidung gefallen ist, eine Rolle als Typ der Kante zwischen Verbindungsknoten und Tentakelknoten darzustellen, stellt sich die Frage: Wie lassen sich mit dieser Lösung einem Knoten mehrere Rollen innerhalb einer Superkante zuordnen? Als Beispiel dient der Satz:

> „Der Client|{AG, DON} schickt|ACT eine Anfrage|{HAB, PAT} an den Server|RECP.“

Um mehrere Rollen darzustellen wird jede Rolle als eigene Kante dargestellt (siehe Abbildung 4.7).

4.2.3.2 Form der Rollennutzung

Da mache Rollen sowohl in positiver Form als auch in negativer Form verwendet werden können, muss eine Möglichkeit gefunden werden, dies im Graph auszudrücken. Die einfachste Lösung besteht darin, eine Rolle, die in zwei verschiedenen Formen verwendet werden kann, auch in zwei Formen anzulegen. Diese Vorgehensweise erhöht allerdings die Zahl der Rollen deutlich. Um keine zusätzlichen Rollen anlegen zu müssen, wird die

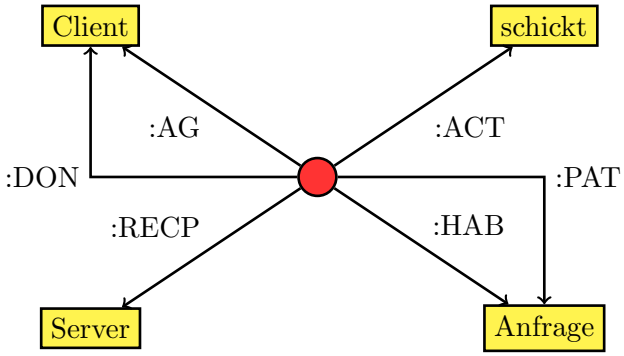


Abbildung 4.7: Konzept – Realisierung von Rollenlisten

Art der Rollenverwendung deshalb in einem Attribut *STATUS* der Rollen gespeichert. Das nachfolgende Beispiel zeigt die Verwendung einer Rolle in verschiedenen Formen:

„Der Server| ¬COMP ist leistungsstärker als die Workstation|COMP.“

Abbildung 4.8 zeigt die entsprechende Darstellung der Umsetzung durch ein Attribut.

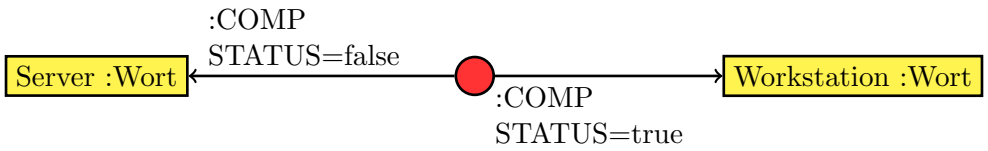


Abbildung 4.8: Konzept – Realisierung der Verwendungsart einer Rolle

Die bis hierher eingeführten Superkanten und Rollen genügen bereits, um ein *allgemeines* Graphersetzungssystem so zu erweitern, dass Kanten als Endpunkt anderer Kanten verwendet werden können.
Speziell für die Darstellung von Texten als Graph sind jedoch noch weitere Konzepte wünschenswert, die für allgemeine Graphersetzungssysteme wenig Nutzen haben. Diese erleichtern dem Anwender die Arbeit und werden nachfolgend vorgestellt.

4.2.4 Dynamische Attribute

In der Grundlagen der Graphentheorie (siehe 2) wurden attributierte Graphen definiert, bei denen Attribute zu den Typen der Graphelemente zugeordnet und dann im Graph belegt werden. Dieses Attribute haben den Nachteil, dass sie „statisch“ sind. Das heißt, alle Attribute müssen bei der Definition der Modellelemente (genauer der Definition der Typen der Modellelemente) bereits festgelegt werden. In Bezug auf die Darstellung von Texten wird man mit Attributen vor allem im Text auftretende Adjektive und Adverbien realisieren wollen. Adjektive und Adverbien können in einem Text aber in beliebiger Anzahl und an beliebiger Stelle auftreten. Es muss also eine Möglichkeit geschaffen werden, jedem Graphelement eine beliebige Zahl von Attributen zuzuordnen.

4.2.4.1 Erster Lösungsversuch

Als erstes überlegt man sich, dass eine beliebige Zahl von Attributen nur durch einen Mengentyp realisiert werden kann. Um Elemente in einem Satz als Knoten darzustellen und ihnen Attribute zuordnen zu können, definiert man sich einen Knotentyp „Wort“ mit einem Attribut „AttList“ vom Typ „Liste“. Die Signatur in einer Pseudodarstellung sieht dann folgendermaßen aus:

```
node type Wort { AttList: Liste }
```

Abbildung 4.9 zeigt die Darstellung des folgenden Satzes:

„Das Projekt ist teuer und aufwändig“

Das Prädikat „ist“ wird hier nicht als eigener Knoten dargestellt, da es keine Aussage hat, sondern lediglich die Adjektive dem Subjekt zuordnet.

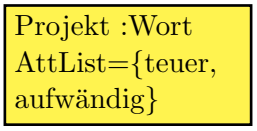


Abbildung 4.9: Konzept – Realisierung von Attributen (Versuch)

Die Realisierung der Attribute als Listentyp kann zwar beliebig viele Attribute darstellen, scheitert aber am folgenden Beispiel:

„Das Projekt ist sehr teuer.“

Bei „sehr“ handelt es sich nicht um ein Adjektiv, das sich direkt auf das Subjekt bezieht, sondern es bezieht sich auf das nachfolgende Adjektiv. In der Attributliste des Knoten müsste also neben dem Attribut „teuer“ eine Referenz gespeichert werden, die auf „sehr“ verweist. Dies würde eine Graphersetzungssystem voraussetzen, das für Attribute komplexe Datentypen zulässt. Außerdem würde die komplette Struktur von Attributen und ihren Beziehungen untereinander nicht durch die Struktur des Graphen ausgedrückt.

4.2.4.2 Bessere Lösung

Wie bei Rollen besteht auch bei Attributen der Wunsch, diese direkt und in vollem Umfang im Graph sichtbar zu machen. Deshalb werden Attribute als eigenständige Knoten realisiert. Um sie von anderen Knotentypen zu unterscheiden, wird ein spezieller Typ *ATTRIBUTE* verwendet. Die Verbindungskanten haben den Typ *ATT*. In Abbildung 4.10 ist die Realisierung des bereits im ersten Lösungsversuch verwendeten Beispiels dargestellt.
Man beachte, dass die Kante, entgegen der üblichen Konvention, nicht vom Attribut zum Knoten verläuft sondern in die andere Richtung. Der Grund dafür liegt in der Mengensemantik (siehe nächster Abschnitt). Um Mengensemantik einheitlich sowohl auf Knoten als auch auf Attribute anwenden zu können, müssen die Kanten die gleiche Richtung haben.

Abbildung 4.10: Konzept – Realisierung von Attributen

4.2.4.3 Gültigkeit von dynamischen Attributen (Kontext)

Wenn man Adjektive und Adverbien eines Textes als Attribute realisiert, muss man sich Gedanken über den Gültigkeitsbereich einzelner Attribute machen. Das folgende Beispiel verdeutlicht die Problematik der Gültigkeit von Attributen.

> „Das Projekt ist sehr teuer. Der Teamleiter sagt, das Projekt sei abgeschlossen.“

Das Attribut „teuer“ ist eine allgemeine Aussage über das „Projekt“, während das Attribut „abgeschlossen“ eine Aussage ist, die bezogen auf einen ganzen Text erst ab dem Zeitpunkt ihres Auftretens gilt. Damit die Gültigkeit eines Attributs im Graph dargestellt werden kann, wird eine spezielle Kontextkante eingeführt, die sofern ein Attribut innerhalb einer Superkante definiert wurde, auf diese Superkante zeigt. Die Kante ist vom Typ *ATT_CONTEXT*. Die Realisierung des Satzes ist in Abbildung 4.11 zu sehen.

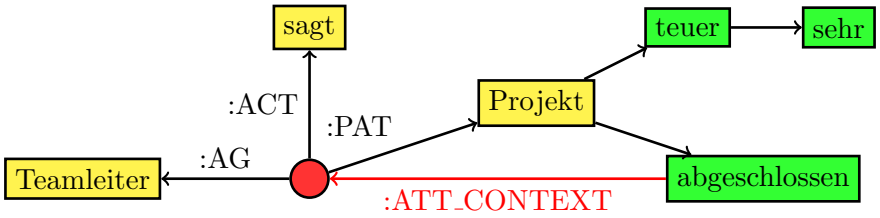


Abbildung 4.11: Konzept – Realisierung von Attributen (mit Kontext)

Das gerade vorgestellte Konzept der Realisierung von Attributen durch eigenständige Knoten wird zur besseren Unterscheidung von Attributen die einem Typ zugeordnet sind, als *dynamische Attribute* bezeichnet.

4.2.5 Mengensemantik

Um die Semantik eines Textes möglichst vollständig in einem Graphen darstellen zu können, fehlt noch eine Möglichkeit, die in natürlicher Sprache ausdrückbare Mengensemantik darzustellen. Dazu ein Beispiel:

> „Der Tester|AG oder der Teamleiter|AG, aber nicht der Entwickler|AG führen|ACT die Tests|PAT aus.“

In diesem Satz gibt es drei Subjekte mit der Rolle Agens. Dabei handelt es sich allerdings nicht einfach um eine Aufzählung, sondern die Subjekte stehen in einem ganz speziellen Verhältnis zu einander. Der Satz sagt aus, dass entweder der Tester oder der Teamleiter, aber nicht der Entwickler die Tests ausführen. Als logische Aussage dargestellt sieht der Sachverhalt folgendermaßen aus:

$$(Tester \vee Teamleiter) \wedge \neg Entwickler$$

Wie kann diese Semantik in einem Graph dargestellt werden? Dazu werden die Mengenoperatoren als eigenständige Knoten realisiert (siehe Abbildung 4.12).

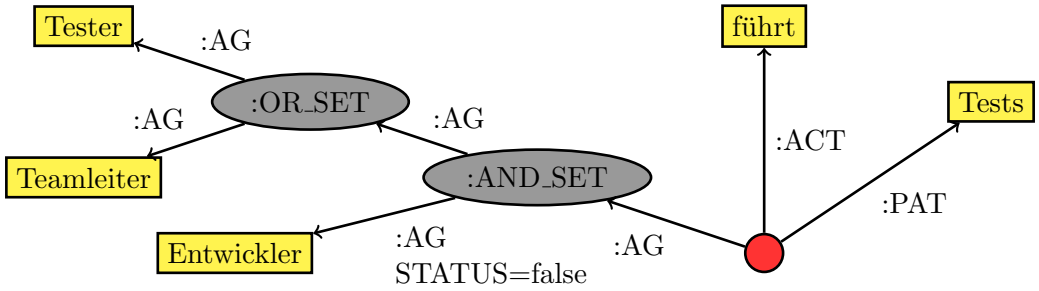


Abbildung 4.12: Konzept – Realisierung der Mengensemantik

Die Mengenknoten haben einen speziellen Typ *SET* von dem die beiden Untertypen *AND_SET* und *OR_SET* abgeleitet sind. Die Rollen in Form des Kantentyp werden an alle in der Menge verwendeten Kanten weitergereicht. Alle Elemente einer Menge müssen also immer die gleiche Rolle (oder Rollen) haben, sonst handelt es sich nicht um eine gemeinsame Menge. Die Negation wird durch das *STATUS*-Attribut der Rolle ausgedrückt. Das heißt, eine Negation in einer Menge wechselt den Wert des Attribut von wahr nach falsch oder umgekehrt. Wird eine Rolle also negativ verwendet und zusätzlich einem Knoten zugeordnet, der in einer Menge mit negativer Semantik auftritt, so führt die doppelte Negation dazu, dass die Verwendung insgesamt wieder positiv ist.

4.2.5.1 Mengensemantik in Ersetzungsregeln

Die vorgestellte Form der Realisierung einer Menge stellt zwar die komplette Semantik im Graph dar, bringt aber Probleme bei der Definition von Ersetzungsregeln mit sich. Bei der Definition von Regeln hat der Anwender keine Kenntnis über die konkrete Form der Menge in einem Graphen. So ist die folgende logische Aussage mit dem vorigen Beispiel äquivalent:

$$(Tester \wedge \neg Entwickler) \vee (Teamleiter \wedge \neg Entwickler)$$

Die Darstellung als Graph ist in Abbildung 4.13 zu sehen und hat trotz Äquivalenz zu Abbildung 4.12 eine andere Graphstruktur und sogar einen Knoten mehr.
Um den Anwender im Umgang mit Mengen zu unterstützen, bietet die Erweiterung für die Eingabe von Graphen die Möglichkeit, Mengen automatisch in disjunktive Normalform zu überführen (siehe 5.2.4).

Neben dem Problem der fehlenden Eindeutigkeit gibt es bei der gewählten Realisierung von Mengen noch eine weitere Problematik:
Bei der hier verwendeten Realisierung einer Menge handelt es sich um einen Baum, bei dem die Elemente der Menge die Blätter des Baums sind.

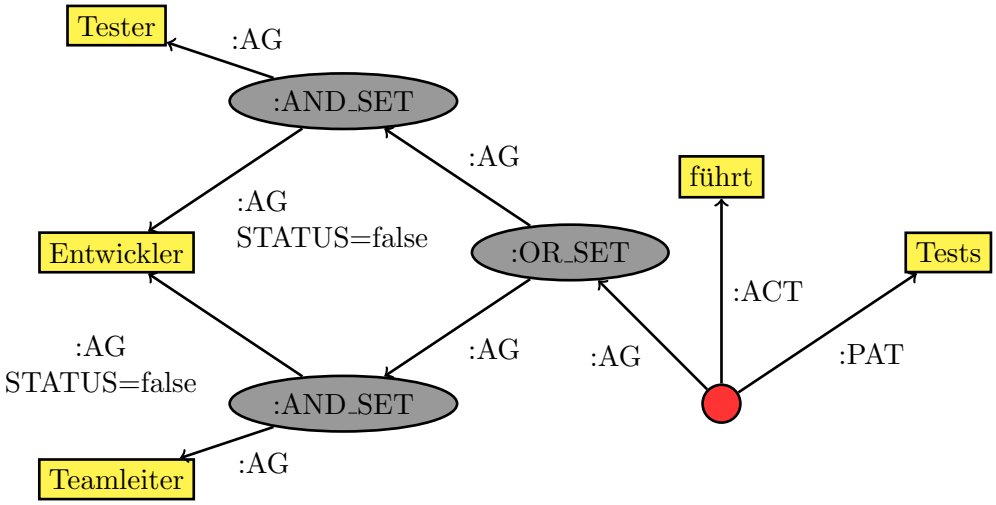


Abbildung 4.13: Konzept – Realisierung der Mengensemantik 2

Dies stellt den Anwender vor ein Problem, wenn er in der linken Seite einer Ersetzungsregel auf ein Element einer Menge zugreifen möchte. Abbildung 4.14 zeigt die grafische Darstellung der linke Seite einer Regel:

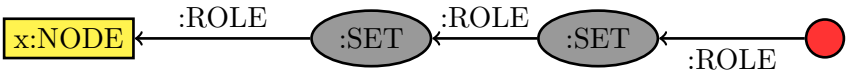


Abbildung 4.14: Konzept – Regel zur Suche von Mengenelementen

Die Regel sucht ein Element aus einer Menge, das auf der zweiten Stufe des Mengenbaums steht. Mengen können aber mit einer beliebigen Tiefe eingegeben werden. Die Suche aller Elemente eines beliebigen Baums ist also mit einer endlichen Anzahl von Regeln nicht möglich.
Der Anwender kann Mengen automatisch in disjunktive Normalform transformieren lassen und so die Tief auf genau zwei Ebenen beschränken. Um aber selbst diese zwei Ebenen nicht unterscheiden zu müssen, wird für jede Menge die „Hülle“ gebildet. Jedes Element der Menge wird durch eine Kante direkt mit der Wurzel der Menge verbunden. Abbildung 4.15 zeigt das Beispiel aus Abbildung 4.12, erweitert um die Hüllenkanten.
Mit Hilfe dieser Kanten vom Typ *SET_CLOSURE* kann der Anwender mit einer einzigen Regel beliebige Element aus einer Menge suchen. Die linke Seite einer solchen Regel ist in Abbildung 4.16 dargestellt. Die Variable „x“ wird vom Graphersetzungssystem mit einem beliebigen Element aus der Menge belegt, egal wo das Element im Mengenbaum auftritt.

4.2.5.2 Mengensemantik bei Attributen

Die Mengensematik gilt nicht nur für Tentakel, sondern kommt in gleicher Form bei Attributen zum Einsatz. Das bereits einmal verwendete Beispiel

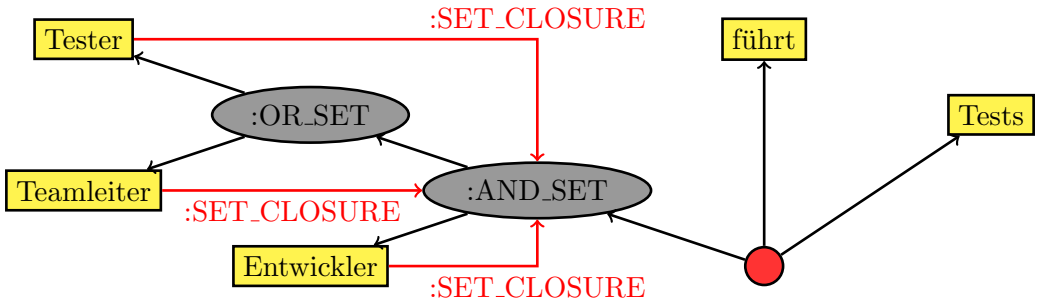


Abbildung 4.15: Konzept – Realisierung der Hülle einer Menge

Abbildung 4.16: Konzept – Regel zur Suche von beliebigen Mengenelementen

„Das Projekt ist aufwändig und teuer.“

ist mit Mengensemantik in Abbildung 4.17 dargestellt.

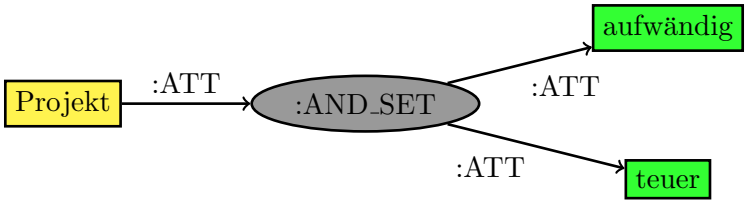


Abbildung 4.17: Konzept – Realisierung von Attributmengen

4.3 Charakterisierung der GrGen-Erweiterung

In diesem Abschnitt werden die grundlegenden Charakteristiken des in dieser Arbeit entwickelten Graphersetzungssystems vorgestellt.
Für ein Graphersetzungssystem, das auf dem Konzept von Einfach- bzw. Hyperkanten basiert, ist beim Entwurf zu entscheiden, ob SPO- oder DPO-Semantik für die Ersetzung verwendet wird. Beim Entwurf der Erweiterung um Superkanten, gibt es jedoch noch eine dritte Möglichkeit, da der Formalismus für Superkanten, im Gegensatz zu Einfach- bzw. Hyperkanten, auch Kanten ohne Tentakel zulässt: Eine Kanten kann ohne Tentakel existieren, da nicht gefordert werden muss, dass Kanten ohne Tentakel gelöscht werden müssen (SPO) oder nicht erzeugt werden dürfen (DPO).
Das zugrunde liegende Graphersetzungssystem GrGen verwendet SPO-Semantik (siehe Kapitel 3.3.2). Für die Erweiterung um Superkanten wurde entschieden, die Möglichkeit zu nutzen Kanten ohne Tentakel zuzulassen. Das Ziel war dabei die Entscheidung dem

Anwender zu überlassen. Das implementierteGraphersetzungssystemm lässt Kanten ohne Tentakel zu, kann aber mit minimalem Aufwand um SPO-Semantik ergänzt werden. Dazu reicht die in Abbildung 4.18 dargestellt Ersetzungsregel aus.

```
1  rule SPO {
2          pattern {
3                  s:SUPEREDGE;
4                  negative {
5                          s[:NODE];
6                  }
7          }
8          replace {
9          }
10 }
```

Abbildung 4.18: Charakterisierung – Ersetzungsregel für SPO-Semantik

Die Regel muss in der Form `grs SPO*` ausgeführt werden (so oft wie möglich), da durch das Löschen einer Kante, weitere Kanten ihr letztes Tentakel verlieren können. Diese Situation ist in Abbildung 4.19 dargestellt.

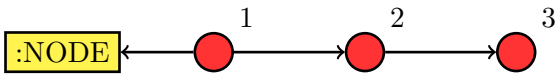


Abbildung 4.19: Charakterisierung – Ausführungssequenz für SPO-Semantik

Die rechte Superkante (3) hat keine Tentakel, sie wird nur als Tentakel von der mittleren Superkante (2) verwendet. Die SPO-Regel entfernt die rechte Superkante (3). Danach gilt für die mittlere Superkante (2), dass sie keine Tentakel hat. Die SPO-Regel muss also ein zweites Mal ausgeführt werden, um auch die mittlere Superkante (2) zu entfernen, so dass am Ende nur die linke Superkante (1) erhalten bleibt, da sie noch ein Tentakel besitzt.

Gibt es für eine Regel mehrere Übereinstimmungen, so wird indeterministisch eine der Übereinstimmungen ausgewählt. Die Reihenfolge in der die verschiedenen Regeln ausgeführt werden, muss vom Anwender angegeben werden. Diese beiden Charakteristiken unterscheiden sich nicht von GrGen.

5 Erweiterte Syntax

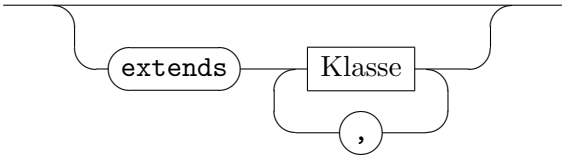
Die neuen Graphelemente werden nicht direkt in die Syntax von GrGen integriert, sondern es wird eine „erweiterte Syntax“ entwickelt, die dann automatisch in GrGen-Syntax übersetzt wird. Damit ist die Erweiterung um Superkanten, von der Weiterentwicklung von GrGen unabhängiger als bei einer direkten Integration (siehe auch: Einführung GrGen in Kapitel 6.2).
Nachfolgend bezeichnet der Begriff „GrGen-Syntax“ die Syntax des Graphersetzungssystems GrGen, während der Begriff „erweiterte Syntax“ die neue, um Superkanten, dynamische Attribute, etc. erweiterte Syntax bezeichnet.
Bei der Formalisierung steht die Erhaltung der GrGen-Syntax im Vordergrund. Dies hat zwei praktische Gründe. Zum einen kann so der Aufwand für die Übersetzung der erweiterten Syntax in die GrGen-Syntax gering gehalten werden. Zum anderen kann bei einer nur minimal geänderten Syntax die Semantik weitestgehend beibehalten werden. Somit werden Probleme bei der Übersetzung vermieden, die durch unterschiedliche semantische Ausdrucksmöglichkeiten zwischen erweiterter Syntax und GrGen-Syntax entstehen.

Ein Graphersetzungssystem gliedert sich in vier logische Einheiten: Modelldefinition, Regeldefinition, Grapheingabe und Definition der Ausführungsreihenfolge der Regeln. Die Umsetzung dieser logischen Einheiten in Softwarekomponenten wird in verschiedenen Graphersetzungswerkzeugen unterschiedlich gehandhabt. GrGen verfügt über einen kombinierten Modell- und Regelübersetzer, sowie eine kombinierte Eingabe von Graphen und Ausführungsreihenfolgen von Regeln. Eine detaillierte Einführung und Beschreibung von GrGen findet sich am Anfang von Kapitel 6.
Die Definition der Ausführungsreihenfolge von Regeln wird durch die erweiterte Syntax nicht verändert, sondern direkt von GrGen übernommen. Die erweiterte Syntax für Modell- und Regeldefinitionen, sowie der Grapheingabe ist nachfolgend beschrieben.

5.1 Modelldefinition

Bei der erweiterten Syntax für Modelldefinitionen handelt es sich um eine echte Erweiterung der GrGen-Syntax. Die Syntax für die Definition von Knoten und Kanten bleibt also vollständig erhalten und wird um die Möglichkeiten zur Definition von Superkanten und Rollen erweitert. Sowohl für Superkanten als auch für Rollen erfolgt die Angabe von Vererbung und Attributen in der gleichen Syntax wie bei GrGen. Ebenfalls analog gilt für Superkanten und Rollen, dass Mehrfachvererbung erlaubt ist.
Vererbung: Innerhalb einer Definition einer Typklasse wird die Angabe von Vererbungsbeziehungen durch das Schlüsselwort `extends` eingeleitet. Danach folgen durch Komma getrennt alle vererbten Klassen der neuen Typklasse:

Vererbung

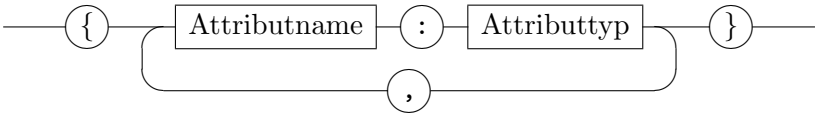


Das folgende Beispiel definiert einen Knotentyp „Quadrat“, der vom Typ „Rechteck“ erbt.

```
node class Quadrat extends Rechteck;
```

GrGen-Attribute: Die GrGen-Attribute einer Typklasse werden in geschweiften Klammern angegeben. Ein GrGen-Attribut hat immer einen Namen und eine Typ:

Attribute



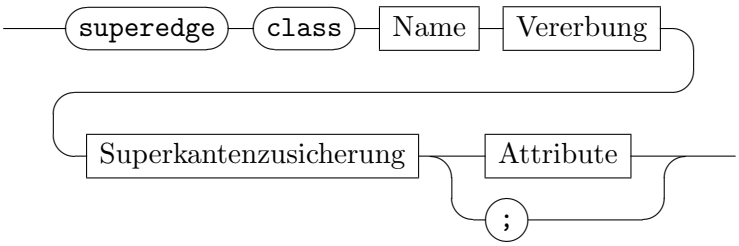
Das folgende Beispiel definiert für den Knotentyp „Quadrat“ ein GrGen-Attribut „Seitenlaenge“ vom Typ Integer.

```
node class Quadrat extends Rechteck { Seitenlaenge : int };
```

5.1.1 Superkanten

Die Definition einer Superkante beginnt mit den Schlüsselwörtern *superedge* und *class*. Anschließend kommt der Name der Superkante, sowie optional Vererbungsbeziehungen, Zusicherungen und GrGen-Attribute:

Superkante



Jede Superkante erbt automatisch von *NODE* (der Oberklasse aller Knoten der Erweiterung) die Attribute „NAME“ und „VALUE“. Im Attribut „NAME“ ist der Namen der Superkante und in „VALUE“ die Namen, der im Rumpf der Superkante enthaltenen Elemente, gespeichert (siehe 5.2.2).

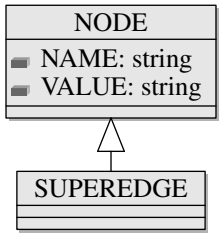


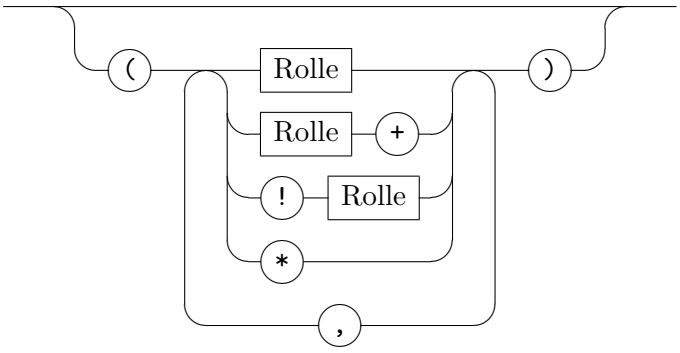
Abbildung 5.1: Modelldefinition – SUPEREDGE

Die Ableitung aller Superkanten von *NODE* zeigt, dass es sich konzeptuell bei einer Superkante eigentlich nur um den Verbindungsknoten handelt, der mit einer beliebigen Zahl von Tentakel angereichert werden kann. Ist kein Tentakel vorhanden, so ist die Superkante ein normaler Knoten.

5.1.1.1 Zusicherungen

Durch die Zusicherungen wird festgelegt, welche Rollen (jeweils gebunden an ein Tentakel) für diesen Superkantentyp zulässig sind. Dabei sind die Modifikatoren +, ! und * erlaubt. Eine Rolle ohne Modifikator bedeutet, diese muss genau einmal als Tentakel auftreten. Der Modifikator + erzwingt, dass die Rolle mindestens einmal auftritt, während mit ! festgelegt wird, dass die Rolle nicht auftreten darf. Der Modifikator * wird nicht in Kombination mit einer Rolle verwendet, sondern steht allein und regelt, dass alle Rollen beliebig oft auftreten dürfen. Die Modifikatoren haben die folgende absteigende Priorität !, kein Modifikator, +, *. Der +-Modifikator steht jeweils hinter der Rolle, während der !-Modifikator immer davor steht.

Superkantenzusicherung



Das folgende Beispiel verdeutlicht die Verwendung der Modifikatoren:

```
superedge class Satz (ACT,*, AG+, RECP, !RECP);
```

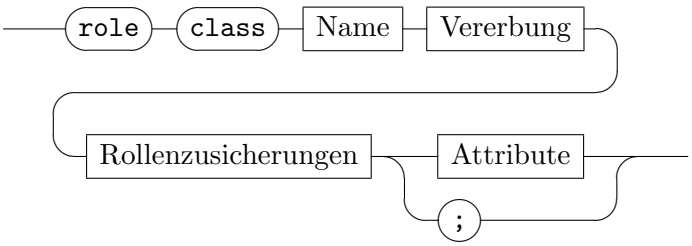
In diesem Beispiel gilt für die Rolle „RECP“, dass sie nicht auftreten darf, da

sich der ! Modifikator durchsetzt. „AG“ muss mindestens einmal und „ACT“ genau einmal auftreten. Alle anderen Rollen dürfen beliebig oft auftreten.

5.1.2 Rollen

Die Definition einer Rolle wird durch die Schlüsselwörter *role* und *class* eingeleitet. Danach kommen der Name der Rolle und optional Vererbungsbeziehungen, Zusicherungen und Attribute.

Rolle



Jede Rolle erbt automatisch von der Klasse *ROLE*, in der das Attribut *STATE* definiert ist. Durch die Verwendung einer gemeinsamen Oberklasse für alle Rollen kann bei der Definition von Mustern in Ersetzungsregeln durch die Typangabe *ROLE* nach beliebigen Rollen gesucht werden. Rollen vererben nur ihre Attribute und nicht ihre Zusicherungen. Dieses Verhalten entspricht der Semantik von GrGen und ist auch für Rollen erwünscht, da sonst eine Unterrolle lediglich gleichviel oder mehr Endpunkte zulassen kann, als die entsprechende Oberrolle. Dies würde aber der bei Vererbung üblichen Spezialisierungssemantik widersprechen.

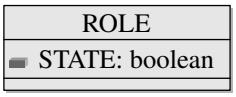
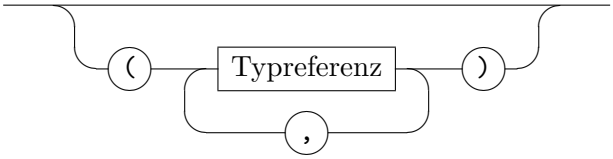


Abbildung 5.2: Modelldefinition – ROLE

Das Attribut *STATE* gibt an, ob die Rolle positiv oder negativ verwendet wird. Standardmäßig werden Rollen positiv verwendet ($STATE = true$), außer die Rolle wurde bei der Grapheingabe durch ! explizit als negativ ($STATE = false$) gekennzeichnet (siehe 5.2.2.4).

5.1.2.1 Zusicherungen

Rollenzusicherungen



Durch die Zusicherungen wird angegeben, an welche Typen von Graphelementen die Rolle gebunden werden kann. Die konkrete Bindung in einem Graph erfolgt zwischen einer Instanz der Rolle und einer Instanz eines zulässigen Typs (Typpreferenz). Die Definitionen von Rollenzusicherungen werden in Zusicherungen übersetzt, die von GrGen ausgewertet werden können (siehe Abschnitt 6.1). Wenn keine Endpunkte angegeben sind, kann die Rolle an alle Knoten- und Superkantentypen gebunden werden. Um gezielt alle Superkantentypen zuzulassen, kann *SUPEREDGE* verwendet werden.
Beispiel:

> `role class ACT extends SemantischeRolle(Wort,SUPEREDGE);`
> Definiert eine Rolle mit dem Namen „ACT“, die von „SemantischeRolle“ erbt und auf die Graphelemente vom Typ „Wort“ bzw. alle Superkantentypen angewendet werden kann.

5.1.3 Unzulässige Typ- und Attributnamen

Sowohl GrGen als auch die Erweiterung definieren interne Typen, deren Namen nicht vom Anwender für eigene Typen verwendet werden können (die Typen an sich können vom Anwender durchaus verwendet werden). Für GrGen sind die vordefinierten Typen: *Node* und *Edge*. Für die Erweiterung sind es *NODE*, *SUPEREDGE*, *SET*, *AND_SET*, *OR_SET*, *ATTRIBUTE*, *ROLE*, *ATT*, *ATT_CONTEXT* und *SET_CLOSURE*.
Die in der Erweiterung definierten Typen besitzen jeweils die Attribute „NAME“, „VALUE“ und „STATE“ (nicht jeder Typ besitzt jedes der drei Attribute, der Anwender sollte aber trotzdem diese drei Namen nicht für eigene Attribute verwenden). GrGen verwendet für interne Attribute Namen, die mit einem Unterstrich beginnen und erlaubt dem Anwender deshalb generell keine mit Unterstrich beginnenden Attributnamen.

5.2 Grapheingabe

Für die Eingabe von Graphen mit Superkanten kann die GrGen-Syntax aus konzeptuellen Gründen nicht beibehalten, sondern muss grundlegend geändert werden. In GrGen besteht jede Kante aus zwei Endpunkten (jeweils einem expliziten Start- und Endknoten), wodurch die Kante automatisch gerichtet ist. Superkanten haben hingegen weder eine Richtung noch eine feste Anzahl von Endpunkten. Somit lassen sich die Endpunkte nur als Menge aufzählen. Auf eine Richtung für Superkanten wird ebenso wie auf

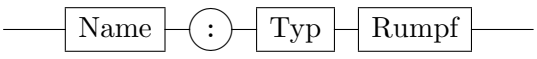
die Ordnung der Endpunktmenge verzichtet. Beides lässt sich durch die Bindung entsprechender Rollen an die Endpunkte ohne großen Aufwand realisieren: Jedem Endpunkt können beliebig viele Rollen zugeordnet werden. Ein bestehendes Graphmodell kann also durch Definition zweier Richtungsrollen (Start- und Endpunktrolle) in ein Graphmodell mit gerichteten Superkanten überführt werden.
Neben der Definition einer Superkante durch Aufzählung ihrer Tentakel (genauer die Endpunkte der Tentakel), muss eine sinnvolle Definition von dynamischen Attributen, sowie der Mengensemantik bei Tentakeln und Attributen gefunden werden. Die Idee, dynamische Attribute analog zu GrGen-Attributen direkt an die Definition des zugehörigen Graphelements anzuschließen, ist nicht realisierbar, da dynamische Attribute nicht nur Knoten und (Super-)Kanten, sondern auch anderen dynamischen Attributen zugeordnet werden können. Somit müssen dynamische Attribute ebenfalls als eigenständige Graphelemente definiert werden (siehe 4.2.4).
Da die erweiterte Syntax gegenüber der GrGen-Syntax grundlegend neu entworfen werden muss, erfolgt dies unter dem Gesichtspunkt der möglichst einfachen Eingabe von Anforderungsdokumenten. Eingegeben wird natürlich nicht das Dokument, sondern eine Graphbeschreibung, die möglichst nahe am Dokument ist. Dieses Ziel soll allerdings nicht zu einer Syntax führen, die für die Eingabe allgemeiner Graphen nicht notwendigen Ballast mit sich bringt.
Die erweiterte Syntax zur Grapheingabe sieht nur ein einziges Graphelement vor, das je nach Ausprägung einen Knoten oder eine Superkante darstellen kann. Die Eingabe von „herkömmlichen“ Kanten ist nicht vorgesehen, da im Rahmen dieser Arbeit kein Bedarf dafür besteht. Die erweiterte Syntax, sowie die zugehörigen Übersetzungswerkzeuge sollten sich aber ohne grundlegende Änderungen entsprechend erweitern lassen.

Nachfolgend wird zuerst das Konzept des Graphelements vorgestellt, danach werden die einzelnen Bestandteile einer Definition erklärt.

5.2.1 Definition Graphelement

Die Definition jedes Elements eines Supergraphen besteht aus drei Teilen: Einem Namen für das Element, einem Typ, sowie einem Rumpf. Im Rumpf können Attribute definiert werden, sowie Tentakel bei Superkanten. Die Unterscheidung zwischen Knoten und Superkanten erfolgt anhand der Tentakel: Ist im Rumpf mindestens ein Tentakel definiert, so handelt es sich um eine Superkante, sonst ist das Graphelement ein Knoten.

Definition



Es ist wünschenswert, Teile der Definition eines Graphelements optional zu machen, um dem Anwender Schreibarbeit zu ersparen. Dabei sind vor allem anonyme Definitionen, sowie Definitionen ohne Rumpf von Interesse.
Bei der Eingabe von Anforderungsdokumenten haben alle Knoten den Typ „Wort“ und alle Superkanten den Typ „Satz“. Weshalb es wünschenswert ist, diese Typen nicht bei

jeder Definition angeben zu müssen, sondern automatisch einfügen zu lassen. Diesem Wunsch trägt die Anweisung *auto node <Knotentyp>, superedge <Kantentyp>;* Rechnung, die im Kopf einer Graphdefinitionsdatei angegeben werden kann (siehe 5.2.4). Nach der Angabe von Standardtypen ist auch die Typangabe optional. Der Übersetzer ergänzt bei allen Graphelementen ohne Tentakel den Standardknotentyp und bei allen Graphelementen mit Tentakel den Standardsuperkantentyp.

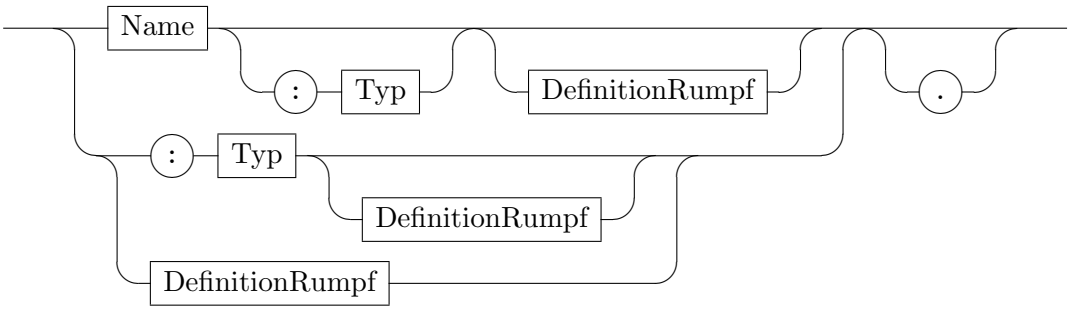
```
1  auto node Wort, superedge SUPEREDGE;
2
3  // fehlerhaft
4  Programmierer
5  Software
6  Unternehmen
7  [@Programmierer entwickelt @Software]
8
9  // richtig
10 Programmierer.
11 Software.
12 Unternehmen.
13 [@Programmierer entwickelt @Software]
```

Abbildung 5.3: Grapheingabe – mehrdeutiges Beispiel

Durch die optionale Typangabe wird allerdings die der Syntax zugrunde liegende Grammatik mehrdeutig. Die Mehrdeutigkeit wird in Kauf genommen, um dem Anwender die Eingabe von Graphen zu erleichtern. Durch die Mehrdeutigkeit kann der Zerteiler nicht unterscheiden, ob der Nutzer durch die Angabe von Namen und Rumpf ein oder zwei Graphelemente definieren will (erstes Element mit Namen und automatischem Typ, zweites Element anonym mit automatischem Typ und Rumpf). In Abbildung 5.3 ist dieses Problem anhand eines Beispiels demonstriert. In dem mit fehlerhaft kommentierten Abschnitt hatte der Nutzer die Absicht „Unternehmen“ als Wort zu definieren und anschließend eine neue Superkante, die den Satz „Programmierer entwickelt Software“ darstellt, zu definieren. Der Zerteiler würde hier aber eine Superkante mit dem Namen „Unternehmen“ und dem Satz als Rumpf erkennen. Dieses Problem kann nur durch die Markierung des Endes einer Graphelementdefinition mittels eines speziellen Symbols gelöst werden. Allerdings bringt die generelle Verwendung eines Symbols unerwünschten Ballast mit sich. Als Lösung erhält der Anwender die Möglichkeit das Ende einer Graphelementdefinition durch einen Punkt explizit anzugeben. Dies bedeutet:
Sofern die Option zur automatischen Typisierung verwendet wird, sollte der Anwender alle Graphelemente, die keinen Rumpf besitzen, mit einem Punkt abschließen!

Die tatsächliche Form eines Graphelements sieht mit sämtlichen optionalen Teilen folgendermaßen aus:

Definition



Nachfogend finden sich einige Beispiele zur Veranschaulichung von Graphelementdefinitionen. Einige der Beispiele setzen voraus, dass Standardtypen festgelegt wurden.

`Programmierer`
Definiert einen Knoten mit dem Namen „Programmierer“ und dem Standardknotentyp.

`:Wort`
Definiert einen Knoten mit einem automatisch generierten Namen und dem Typ „Wort“.

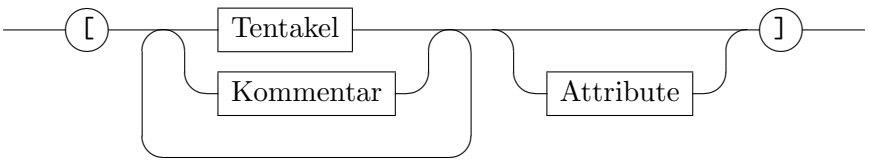
`[]`
Definiert einen Knoten mit einem automatisch generierten Namen und dem Standardknotentyp.

`Satz[Unternehmen beschäftigt]`
Definiert eine Superkante mit dem Namen „Satz“ und dem Standardsuperkantentyp. Die Superkante hat die beiden Tentakel „Unternehmen“ und „beschäftigt“, die wiederum sind Knoten vom Standardknotentyp.

5.2.2 Rumpf eines Graphelements

Handelt es sich bei einer Graphelementdefinition um einen Knoten, so können im Rumpf die zum Knoten gehörenden Attribute definiert werden. Bei einer Superkante dürfen im Rumpf außerdem Tentakel und Kommentare angegeben werden. Einem Tentakel kann zusätzlich eine Rolle zugeordnet werden. In jeder Superkante kann außerdem ein Tentakel als Kopfelement ausgezeichnet werden. Dadurch wird geregelt, wie eine Superkante mit ihrer Umgebung verbunden ist. In diesem Abschnitt werden zuerst Tentakeln und danach Attribute und Kommentare behandelt.

DefinitionRumpf



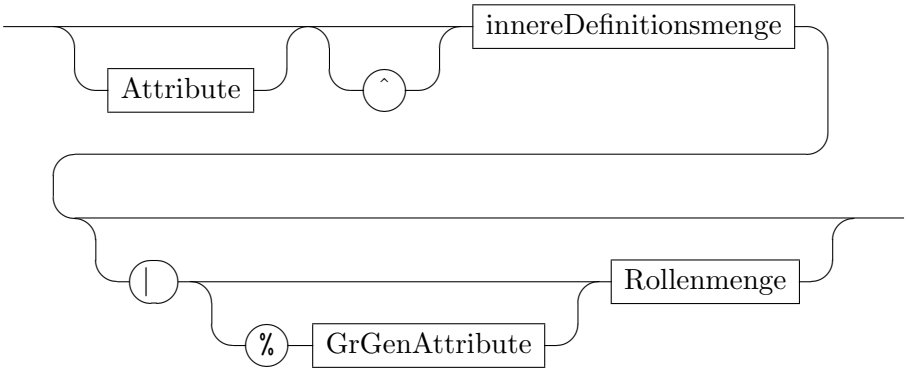
Man beachte, dass die Attribute einer Graphelementdefinition im Rumpf immer an letzter Stelle stehen müssen. Dies wird durch die Rechts-Zuordnung der Attribute notwendig (siehe dazu Definition Attribute 5.2.2.5).
Beispiel:

> `Satz[#Der Programmierer arbeitet #mit #dem PC $Satzteil=1]`
> Definiert eine Superkante vom Typ Satz, mit drei Tentakeln und vier Kommentaren (markiert durch #). Außerdem wird der Superkante das dynamische Attribut „Satzteil“ zugeordnet und auf den Wert „1“ gesetzt (dynamische Attribute werden durch $ markiert).

5.2.2.1 Tentakel

Im Rumpf einer Superkante können beliebig viele Tentakel definiert werden. Die Syntax zur Definition eines Tentakels hat die folgende Form:

Tentakel



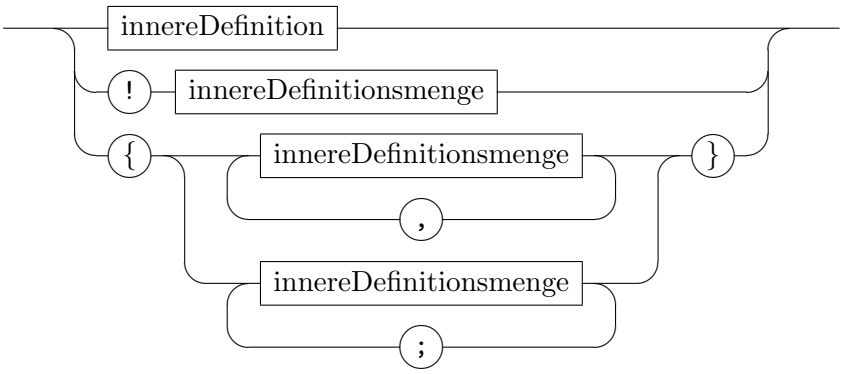
Die einzelnen Teile eines Tentakel werden nachfolgend erklärt. Beispiel:

> `$fleißig ^{Entwickler, Anwender}| %typ=primär ACT`
> Definiert ein Tentakel, dass aus zwei Graphelementen („Entwickler“ und „Anwender“) besteht und die Rolle „ACT“ hat. Außerdem ist dem Tentakel das dynamische Attribut „fleißig“ zugeordnet und es ist durch das ^-Zeichen als Kopfelement der Superkante ausgezeichnet. Das GrGen-Attribut „typ“ wird der Rolle „ACT“ zugeordnet und auf den Wert „primär“ gesetzt.

5.2.2.2 Tentakelmenge

Ein Tentakel muss nicht aus genau einem Graphelement bestehen, sondern kann aus einer Menge von Graphelementen bestehen. Mengen von Graphelementen haben die Syntax:

innereDefinitionsmenge



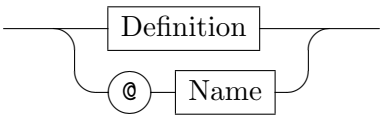
Eine Definitionsmenge von Graphelementen kann aus einer inneren Definition oder einer negierten Definitionsmenge bestehen. Außerdem können Mengen durch „und“ (Komma) bzw. „oder“ (Semikolon) verbunden werden. Das folgende Beispiel zeigt die Eingabe der in Abschnitt 4.2.5 als Beispiel verwendeten Tentakelmenge:

```
{{Tester; System}, !Teamleiter}
```

5.2.2.3 Referenzierung

Bei einem Tentakel kann es sich entweder um die Referenzierung eines bereits zuvor definierten Graphelements oder um ein neu definiertes Graphelement handeln. Die Referenzierung eines Graphelements erfolgt durch das @-Zeichen und den Namen des Graphelements. Wird als Tentakel ein neues Graphelement definiert, so gilt die gleiche Syntax wie für Graphelemente, die außerhalb des Rumpfs einer Superkante definiert werden. In einer Superkante können also als Tentakel rekursiv wieder Superkanten definiert werden.

innereDefinition



Die folgenden Beispiele zeigen die Verwendung von Referenzen:

`Satz1[@Server Workstation:Wort PC]`
Definiert eine Superkante mit dem Namen „Satz1“ und dem Standardsuperkantentyp, die drei Tentakel enthält. Zuerst wird das Graphelement „Server“ als Tentakel referenziert, dieses muss bereits zuvor definiert worden sein (z.B. durch: `Server:Wort[].`). Als zweites und drittes Tentakel werden zwei neue Knoten definiert. Der Knoten „Workstation“ ist vom Typ „Wort“ während der Knoten „PC“ den Standardknotentyp erhält.

`[Konferenz Team[@Entwickler @Teamleiter] Termin]`
Definiert eine Superkante ohne Namen, die drei Tentakel enthält. Das erste und das letzte Tentakel sind jeweils neu definierte Knoten. Das zweite Tentakel ist eine neue Superkante die den Namen „Team“ trägt und als Tentakel die beiden Graphelemente Entwickler und Teamleiter referenziert.

`[@Team Mitglieder]`
Dieses Beispiel demonstriert, dass auch Superkanten referenziert werden können. Dabei wird die Superkante aus dem vorherigen Beispiel referenziert.

Abbildung 5.4 zeigt, dass zweite und dritte Beispiel in einem gemeinsamen Graphen dargestellt:

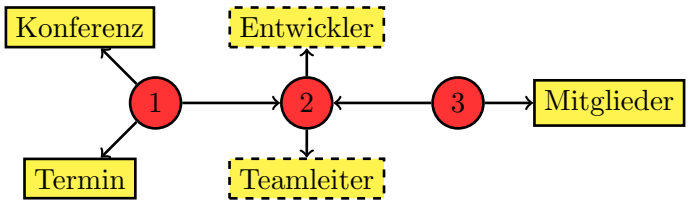


Abbildung 5.4: Grapheingabe – Beispiel einer Referenzierung

Bei der mittleren Superkante(3) handelt es sich um die Superkante „Team“ aus dem zweiten Beispiel. Die rechte Superkante(3) ist die anonyme Superkante aus dem dritten Beispiel und bei der linken Superkante(1) handelt es sich um die anonyme Superkante aus dem zweiten Beispiel. Die Knoten „Entwickler“ und „Teamleiter“ sind gestrichelt dargestellt, da sie in den Beispielen nur referenziert werden (ihre Definition wird als gegeben vorausgesetzt). Die Darstellung der beiden Referenzen als Knoten, ist eigentlich nicht korrekt, da es sich auch um Referenzen auf Superkanten handeln könnte (der Name einer Referenz erlaubt keine Rückschlüsse auf den Typ).

5.2.2.4 Überdeckung von Definitionen

In einem Text können verschiedene Elemente den gleichen Namen haben. Das folgende Beispiel verdeutlicht diesen Sachverhalt:

```
Der Entwickler schreibt ein Programm. ... Für die Entwicklung kommt
ein Programm zum Einsatz.
```

Mit dem Namen „Programm“ sind hier zwei verschiedene Programme gemeint. Damit der Anwender bei der Überführung in eine Grapheingabe möglichst geringen Aufwand hat, gilt: **Wird ein bereits zuvor definierter Name als Tentakel verwendet, ohne ihn durch @ zu referenzieren, so wird der ursprüngliche Name überdeckt.** Der Übersetzer generiert dann einen neuen, eindeutigen Namen, indem er eine fortlaufende Nummer anhängt. Jede danach auftretende Referenz bezieht sich dann auf die zuletzt erfolgte Überdeckung. Der Benutzer erhält bei der Transformation eine Warnung, um auf eventuell unbeabsichtigtes Überdecken hinzuweisen.

```
Programm.
[Programm System]
[@Programm Fehler]
```

In der ersten Zeile wird ein Knoten mit dem Namen „Programm“ definiert. Danach wird in der zweiten Zeile eine Superkante mit zwei Tentakeln definiert. Bei beiden Tentakeln handelt es sich um neu definierte Knoten. Da der Namen „Programm“ bereits verwendet wurde, ändert der Übersetzer den Namen des ersten Knotens automatisch in „Programm_1“. Die Referenzierung in der dritten Zeile bezieht sich dann auf die zuletzt erfolgte Definition von „Programm“ also auf „Programm_1“. Der Übersetzer gibt bei diesem Beispiel die folgende Warnung aus: „WARNING: Name has already been defined, renaming: Programm_1 (line:1,Column:2)“

Kopfelement Das Kopfelement einer Superkante wird durch das Zeichen ^ markiert. Ist in einer Superkante ein Tentakel als Kopfelement ausgezeichnet, so wird die Superkante durch dieses Tentakel mit einer eventuell vorhandenen umschließenden Superkante verbunden. Ist kein Kopfelement ausgezeichnet, so wird die Superkante durch ihren inneren Verbindungsknoten mit einer äußeren Superkante verbunden. Abbildungen 5.5 und 5.6 verdeutlichen den Unterschied (Zum besseren Verständnis wurden die Superkanten nummeriert):

```
1[Entwickler|AG benutzt|ACT 2[^PC|HAB Kollegen|POSS]|HAB]
```

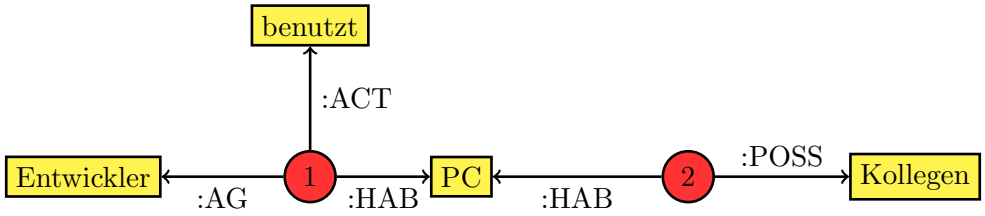


Abbildung 5.5: Grapheingabe – Superkante mit Kopfelement

```
1[2[Entwickler|AG beendet|ACT Arbeit|PAT]|ACT #danach 3[@Entwickler|AG
geht|ACT #zum Arzt|LOC]|CAU]
```

Das #-Zeichen markiert eine Kommentar, der nicht im Graph dargestellt wird (siehe Abschnitt 5.2.2.6).

Rollen Im Rumpf einer Superkante kann jedem Tentakel eine Menge von Rollen zugeordnet werden. Eine Rollenmenge wird durch das vorangestellte | -Zeichen eingeleitet. Eine Rollenmenge hat die folgende Syntax:

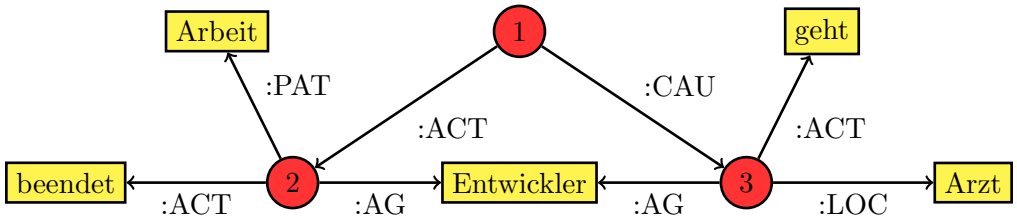
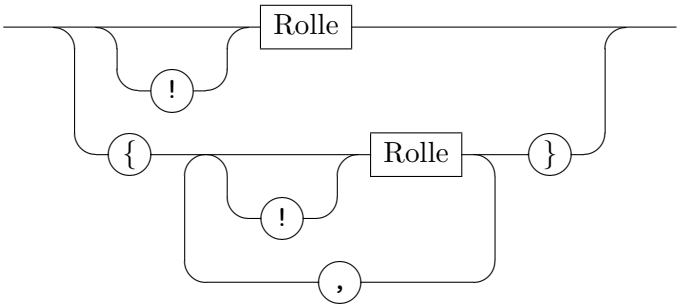


Abbildung 5.6: Grapheingabe – Superkante ohne Kopfelement

Rollenmenge

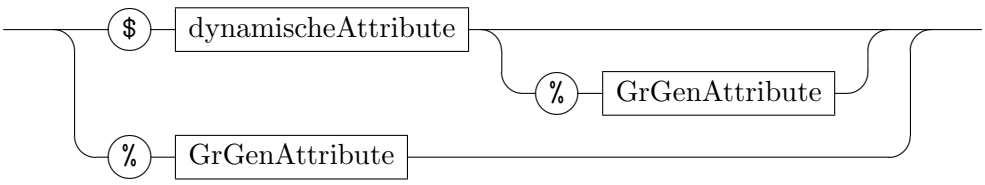


Bei der Rolle muss es sich um ein im Graphmodell definierten Rollentyp handeln. Im Kopf der Graphdefinitionsdatei kann mit der Angabe `force roles;` erzwungen werden, dass jedem im Graph definierten Tentakel eine Rolle zugeordnet werden musst (siehe 5.2.4).

5.2.2.5 Attribute

Im Rumpf einer Definition können sowohl GrGen-Attribute als auch dynamische Attribute definiert werden. Mit dem $-Zeichen wird ein dynamisches Attribut und mit dem %-Zeichen ein GrGen-Attribut definiert.

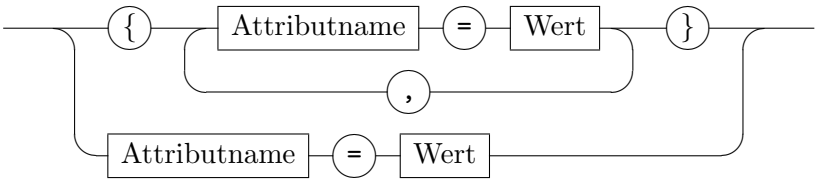
Attribute



Dynamische Attribute stehen immer vor GrGen-Attributen. Mehrere Attribute vom gleichen Typ müssen geklammert werden (siehe die Beispiel am Ende dieses Abschnitts).

GrGen-Attribute GrGen-Attribute dürfen in beliebiger Zahl auftreten. Name und Typ der GrGen-Attribute müssen der Definition im Graphmodell entsprechen. Es ergibt sich für die Eingabe von GrGen-Attributen die Form:

grGenAttribute



Dynamische Attribute Ein dynamisches Attribut kann im Gegensatz zu GrGen-Attributen auch ohne eine Wertzuweisung eingegeben werden. Das Attribut wird dann als Variable mit Wahrheitswert interpretiert. Der Wert wird automatisch auf „wahr“ gesetzt. Steht vor der Attributdefinition ein „!“, so wird der Wert des Attributs auf „falsch“ gesetzt.Neben dem Wert des Attributs ändert „!“ auch den Wahrheitswert der Attributskante (gespeichert im Attribut „STATE“).
Hat ein Attribut sowohl einen Wert als auch ein „!“ voranstehen, so wird der Wert des Attributs auf den angegebenen Wert gesetzt und die Attributskante auf „falsch“. Abbildung 5.7 zeigt die Umsetzung des folgenden Satzes, der zusätzlich auch als Grapheingabe dargestellt ist:

„Die Farbe des Servers ist nicht rot“

```
Server:Wort[$!Farbe=rot]
```

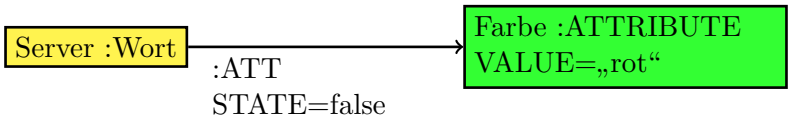
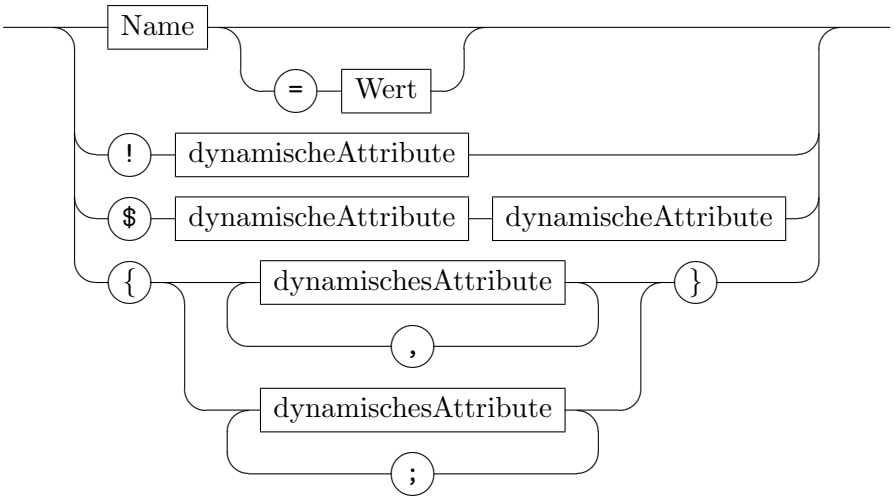


Abbildung 5.7: Grapheingabe – negiertes Attribut mit Wert

dynamischeAttribute



Zuordnung von Attributen Attribute werden immer nach rechts zugeordnet. Sie stehen also vor dem Element, auf das sie sich beziehen. Dies entspricht der Semantik der deutschen und englischen Sprache[1]. Deshalb müssen Attribute immer vor dem Tentakel stehen, auf das sie sich beziehen. Dies ist auch der Grund, weshalb die Attribute von Superkanten am Ende des Rumpfs definiert werden müssen.
Die folgenden Beispiele veranschaulichen die Verwendung von Attributen:

`Programm[${komplex,LOC=120}]`
Definiert einen Knoten mit den dynamischen Attributen „komplex“ und „LOC“.

`Satz[$Schriftart=Arial %Länge=10 Teilsatz | %Wert=10 AG $Farbe=rot %Länge=20]`
Definiert eine Superkante mit dem Namen „Satz“ und den Attributen „Farbe“ und „Länge“. Dabei ist „Länge“ ein GrGen-Attribut und muss entsprechend im Graphmodell definiert sein. Erinnerung: Die zur Superkante gehörenden Attribute müssen am Ende des Rumpf definiert werden (dynamische Attribute vor GrGen-Attributen). Die Superkante enthält einen Knoten „Teilsatz”, dem die Attribute „Schriftart“ und „Länge“ zugeordnet sind. Außerdem hat der Knoten die Rolle „AG“. Dieser Rolle ist das Attribut „Wert“ zugeordnet. **Rollen können nur GrGen-Attributen zugeordnet werden!**[2]

`Projekt:Wort[${sehr teuer}]`
Definiert einen Knoten „Projekt“ vom Typ „Wort“, dem das Attribut „teuer“ zugeordnet ist. Diesem Attribut ist wiederum ein Attribut „sehr“ zugeordnet (die Darstellung des Graphen befindet sich in Abbildung 4.10). Nur dynamische Attribute können sich auf andere dynamische Attribute beziehen, da sie als eigenständige Graphelemente realisiert werden. GrGen-Attribute sind immer dem Graphelement in dessen Typ sie definiert sind zugeordnet.

`Projekt:Wort[${teuer,langwierig}]`
Definiert einen Knoten „Projekt“ vom Typ „Wort“, dem die Attribute „teuer“ und „langwierig“ zugeordnet sind. Die Attribute werden in einer Menge gespeichert. Die Zuordnung mehrere GrGen-Attribute zu einem Graphelement erfolgt analog.

5.2.2.6 Kommentare

Im Rumpf einer Graphelementdefinition können neben Tentakeln und Attributen auch Kommentare auftreten. Der Anwender kann damit auch „Füllwörter“ eines Satzes, die nicht im Graph dargestellt werden sollen, in die Grapheingabe schreiben. Diese werden durch das #-Zeichen definiert. Das Kommentarzeichen bezieht sich immer nur auf die nachfolgende Zeichenkette, bis zum nächsten Leerzeichen.

[1] Im Französischen zum Beispiel stehen die Adjektive nach dem Bezugselement.
[2] Da eine Rolle durch eine Kante realisiert wird, können ihr nur GrGen-Attribute zugeordnet werden und keine dynamischen Attribute.

```
[#Der Entwickler|AG arbeitet||ACT #an #einer Anwendung|PAT ]
```

5.2.3 Sonderzeichen

Bei der Definition von Graphen dürfen die folgenden Umlaute verwendet werden: ä, ö, ü, Ä, Ö, Ü, ß. Diese werden vom Übersetzer der Grapheingabe in erweiterter Syntax automatisch durch ae, oe, ue, Ae, Oe, Ue, ss ersetzt. Dies dient der erleichterten Eingabe deutscher Texte.

5.2.4 Optionen für die Grapheingabe

Im Kopf einer Graphdefinitionsdatei können Standardtypen für Knoten und Superkanten festgelegt werden. Außerdem kann erzwungen werden, dass jedem in einer Superkante definierten Tentakel eine Rolle zugeordnet werden muss. Mengen können automatisch in disjunktive Normalform gebracht werden. Die Anweisungen müssen vor der ersten Definition eines Graphelements erfolgen.

```
auto node Wort, superedge SUPEREDGE;
force roles; disjunctive normal form;
```

Mit *auto node <Knotentyp>, superedge <Kantentyp>;* werden die Standardtypen für Knoten und Superkanten festgelegt (im Beispiel sind dies „Wort“ und *SUPEREDGE*). Danach ist es möglich, Knoten und Superkanten ohne Typangabe zu definieren. Sie erhalten dann jeweils den Standardknotentyp bzw. den Standardsuperkantentyp. Erfolgt eine Definition ohne Typangabe obwohl keine Standardtypen festgelegt wurden, so führt dies zu einem Übersetzungsfehler.
Durch *force roles;* wird erzwungen, dass der Anwender für jedes Tentakel eine Rolle angibt. Fehlen Rollen, so bricht die Übersetzung der Grapheingabe mit einem Fehler ab. Verwendet der Anwender *disjunctive normal form*, so werden alle Mengen in der Grapheingabe automatisch in disjunktive Normalform (DNF) gebracht. Da der Algorithmus für die Transformation in disjunktive Normalform im schlimmsten Fall exponentiellen Aufwand hat, ist diese Option standardmäßig ausgeschaltet[3]. Die bei der Darstellung von Texten zu erwartenden Mengen, bestehen jeweils nur aus wenigen Elementen, so dass der Aufwand des Algorithmus hier vernachlässigt werden kann.

5.3 Regeldefinition

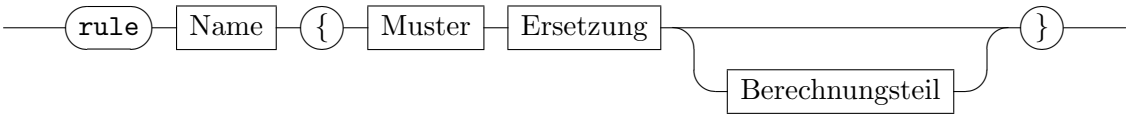
Die erweiterte Syntax für die Definition von Ersetzungsregeln behält die grundlegende Struktur der GrGen-Syntax bei und ändert nur die Definition von Kanten.

[3]Der Aufwand ist genau dann exponentiell, wenn die Eingabe in konjunktiver Normalform ist.

5.3.1 Regel

Eine Regel wird durch das Schlüsselwort *rule* sowie den Namen der Regel eingeleitet. Danach folgen linke und rechte Seite der Regel und der Berechnungsteil. Die linke Seite wird bei GrGen als Muster (*pattern*) bezeichnet und die rechte Seite als Ersetzung (*replace*). Im Berechnungsabschnitt (*eval*) können die Werte von Attributen geändert werden.

Regeldefinition



Die drei Teile einer Regel sind in Beispiel 5.8 dargestellt.

```
1  actions Beispielregeln using SpracheModell;
2
3  rule Beispiel {
4          pattern {
5                  x:Wort;
6                  if { x.NAME == "Server"; }
7                  negative {
8                          y:SUPEREDGE[x|AG];
9                  }
10         }
11         replace {
12                 z[x|AG];
13         }
14         eval {
15                 z.NAME="Satz";
16         }
17 }
```

Abbildung 5.8: Regeldefinition – Regelschema

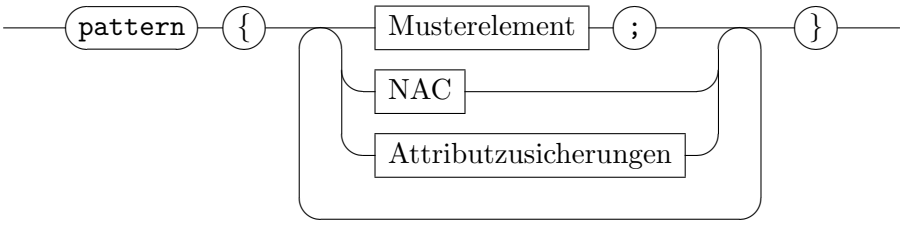
Die Regel trägt den Namen „Beispiel“. In der ersten Zeile wird der Name der Regeldatei als „Beispielregeln“ festgelegt, sowie das Graphmodell „SpracheModell“ eingebunden (siehe Abschnitt 6.1). Im linken Teil der Regel wird ein Knoten vom Typ „Wort“ gesucht, dessen Attribut „NAME“ den Wert „Server“ hat. Durch die negative Anwendungsbedingung wird sichergestellt, dass der Knoten nicht mit der Rolle „AG“ in einer Superkante auftritt. Auf der rechten Seite, wird der Knoten in eine neue Superkante eingefügt und es wird ihm die Rolle „AG“ zugeordnet. Im Berechnungsteil wird das Attribut „NAME“ der Superkante auf „Satz“ gesetzt.

5.3.2 Linke Seite

Durch die linke Seite einer Regel werden Graphelemente definiert, die im Wirtsgraphen vorhanden sein müssen, damit die Ersetzungsregel angewendet werden kann. Zusätzlich

können Zusicherungen an die Attribute der Elemente angegeben werden. Darüber hinaus sind negative Anwendungsbedingungen möglich (siehe dazu Abschnitt 2.2 der Grundlagen). Die linke Seite einer Regel wird durch das Schlüsselwort *pattern* eingeleitet.

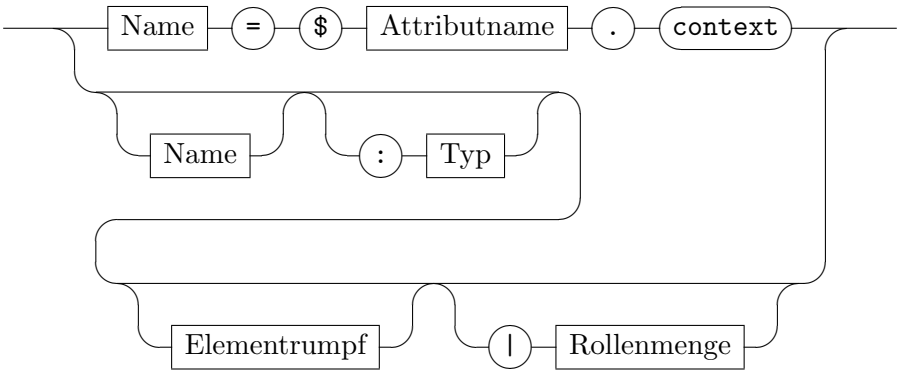
Muster



5.3.2.1 Musterelement

Ein Musterelement ist entweder ein Graphelement oder eine Kontextabfrage. Die Syntax für Graphelemente ist ähnlich wie bei der Eingabe von Graphen. Name, Typ, Rumpf und Rollenmenge sind jeweils optional und können in beliebigen Variationen verwendet werden.

Musterelement



Wird bei der Definition eines Musterelements kein Typ angegeben, so wird als Typ *NODE* angenommen. Das Element kann also sowohl auf einen Knoten, als auch auf den Verbindungsknoten einer Superkante abgebildet werden, da *NODE* die Oberklasse aller Knoten und Superkanten (Verbindungskoten) ist. Die folgenden Beispiele veranschaulichen die Definition von Musterelementen. Neben der erweiterten Syntax für die Definition der Musterlemente ist auch die graphische Darstellung der Übersetzung in die GrGen-Syntax angegeben.

`x;`
Definiert ein Graphelement vom Typ *NODE*. Es kann also sowohl auf einen Knoten als auch eine Superkante abgebildet werden.

Abbildung 5.9: Regeldefinition – Beispiel eines Musterelements 1

`:Wort|AG;`
Definiert ein Graphelement vom Typ „Wort“, dem die Rolle „AG“ zugeordnet ist. Dazu muss die Superkante, in der das Element die Rolle hat, ergänzt werden.

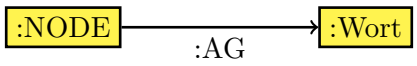


Abbildung 5.10: Regeldefinition – Beispiel eines Musterelements 2

5.3.2.2 Kontextabfrage

Mit Hilfe der Kontextabfrage kann die Superkante, die den Kontext eines Attributs bildet (siehe Abschnitt 4.2.4.3), ermittelt werden. Hat bei der Kontextabfrage das entsprechende Attribut keinen Kontext, so wird das Muster nicht gefunden und die Regel nicht ausgeführt. Nachfolgend ist eine beispielhafte Kontextabfrage angegeben:

```
x = $fleissig.context;
```

Das aus dieser Abfrage vom Übersetzer erzeugte Graphmuster ist in Abbildung 5.11 angegeben. Der Typ des Knoten „x“ ist *SUPEREDGE* und wurde aus Gründen der Übersichtlichkeit nicht dargestellt.

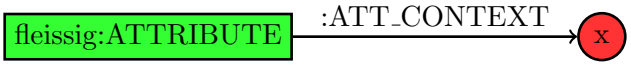


Abbildung 5.11: Regeldefinition – Kontextabfrage

Der folgende Satz dient als Beispiel zur Veranschaulichung einer Kontextabfrage:

```
Satz[$fleissig Entwickler]
```

In Abbildung 5.12 ist der entsprechende Graph dargestellt. Der Satz wird als Superkante mit einem Tentakel dargestellt. Diesem Tentakel ist das Attribut „fleissig“ zugeordnet, dass im Kontext der Superkante definiert ist.
Die oben definierte Kontextabfrage würde angewendet auf diesen Graphen die Variable „x“ mit dem Graphelement „Satz“ belegen.

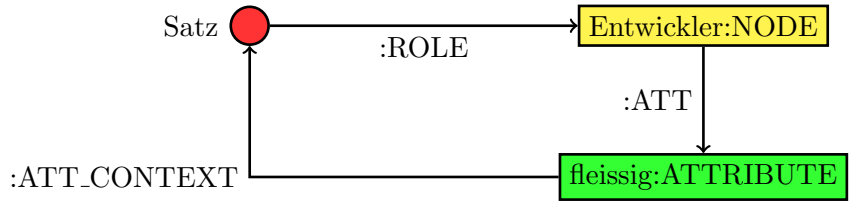
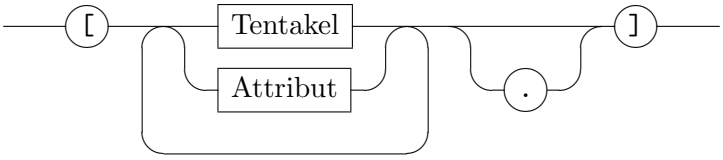


Abbildung 5.12: Regeldefinition – Beispiel Kontextabfrage

5.3.2.3 Elementrumpf

Im Rumpf eines Elements können beliebig viele Tentakel und Attribute definiert werden. Dabei ist zu beachten, dass, wie bei der Eingabe von Graphen, die Attribute nach rechts zugeordnet werden. Ein Attribut vor einem Tentakel wird also immer diesem Tentakel zugeordnet.

Elementrumpf



Der optionale Punkt am Ende eines Elementrumpfs bietet die Möglichkeit die Aufzählung der Tentakel im Rumpf einer Superkante als vollständig zu markieren. Dies bedeutet, dass neben den aufgezählten Tentakeln keine weiteren Tentakel in der Superkante auftreten dürfen. Um dies sicherzustellen, erzeugt der Übersetzer eine negative Anwendungsbedingung, die weitere Tentakel ausschließt. Das folgende Beispiel veranschaulicht diesen Sachverhalt:

`Satz:SUPEREDGE[x:Wort y:Wort .]`
Neben dem Muster, das aus der Superkante sowie ihren beiden Tentakeln besteht, generiert der Übersetzer eine negative Anwendungsbedingung, um sicherzustellen, dass die Superkante keine weiteren Tentakel hat. Der gesamte, vom Übersetzer generierte Code, ist nachfolgend zu sehen:

```
1   ZZx4:NODE;
2           ZZx4 -:ROLE-> x:NODE;
3           ZZx4 -:ROLE-> y:NODE;
4           negative {
5                   ZZx4 -:ROLE-> ZZx5:NODE;
6           }
```

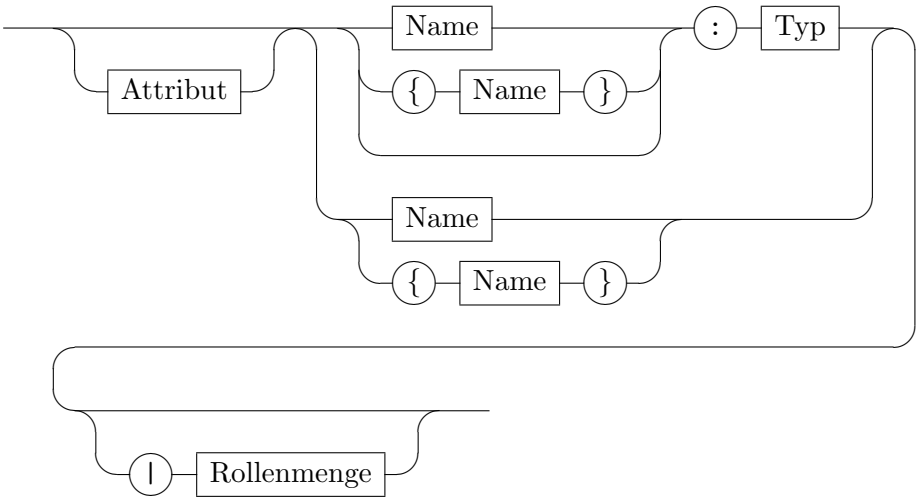
Die beiden Variablennamen „ZZx4“ und „ZZx5“ werden vom Übersetzer automatisch generiert. In der ersten Zeile wird die Superkante definiert,

bevor in der zweiten und dritten Zeile die beiden Tentakel definiert werden. Beide haben den Typ *NODE* und die Rolle *ROLE*. In der vierten bis sechsten Zeile folgt dann die negative Anwendungsbedingung, die garantiert, dass es keine weiteren Tentakel gibt. Dies funktioniert, da GrGen isomorphe Übereinstimmungen verwendet (ein Graphelement kann also nicht zwei Variablen zugewiesen werden).

5.3.2.4 Tentakel

Ein Tentakel besteht in erster Linie aus einem Namen, dem optional Attribute, ein Typ sowie eine Menge von Rollen zugeordnet werden können. Die Syntax für Rollenmengen entspricht der Syntax bei der Grapheingabe (siehe 5.2.2.4). Die Syntax von Attributen wird im nachfolgenden Abschnitt erklärt. Der Namen kann optional in geschweiften Klammern stehen oder, sofern ein Typ angegeben ist, ganz fehlen. Steht der Name in geschweiften Klammern, so wird auf ein Element einer Tentakelmenge zugegriffen. Siehe hierzu Abschnitt 4.2.5 zur Erklärung von Mengensemantik.

Tentakel



Die folgenden Beispiele verdeutlichen die Definition von Tentakeln:

`[x:Wort|ACT];`
Definiert eine Superkante mit einem Tentakel vom Typ „Wort“ und der Rolle „ACT“. Wird eine Übereinstimmung gefunden, so wird das gefundene Element der Variable „x“ zugewiesen.

`[{y}|AG];`
Definiert eine Superkante mit einem Tentakel aus einer Tentakelmenge, dem die Rolle „AG“ zugeordnet ist. Der Übersetzer generiert daraus den folgenden GrGen-Code:

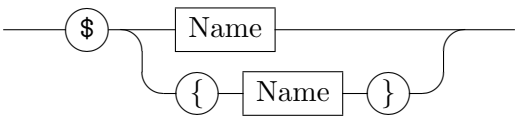
```
1   ZZx2:NODE;
2           ZZx2 -:AG-> ZZx3:SET;
3           y:NODE -:SET_CLOSURE-> ZZx3;
```

ZZx2 und ZZx3 sind automatisch generierte Variablennamen. In der ersten Zeile wird eine Superkante definiert. In der zweiten Zeile folgt dann ein Knoten vom Typ *SET*, der die Rolle „AG“ hat. In der dritten Zeile folgt die Hüllenkante vom Zielknoten y zum in der zweiten Zeile definierten Mengenknoten.

Wird ein Knoten aus einer Menge gesucht, so kann nicht angegeben werden, von welchem konkreten Mengentyp die Menge sein soll.

Dynamische Attribute Da bei Attributen wie bei Tentakeln Mengensemantik erlaubt ist, kann ein Attribut entweder direkt oder als Mengenelement angegeben werden.

Attribut



Das folgende Beispiel zeigt die Verwendung eines Attributs aus einer Menge:

```
[$ {x} ]
```

Abbildung 5.13 zeigt die Umsetzung in ein Graphmuster durch den Übersetzer.

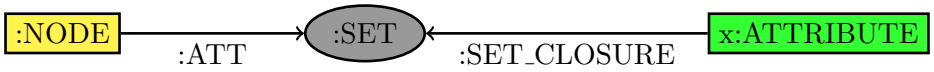


Abbildung 5.13: Regeldefinition – Attribut aus einer Menge

Die erweiterte Syntax bietet keine spezielle Unterstützung, um in einem Muster ein Attribut anzugeben, das einem anderen Attribut zugeordnet ist. Damit derartige Attribute angegeben werden können, ist ein kleiner Trick notwendig, der nachfolgend erklärt wird:

```
1   [$teuer Projekt:Wort];
2   teuer[$sehr];
```

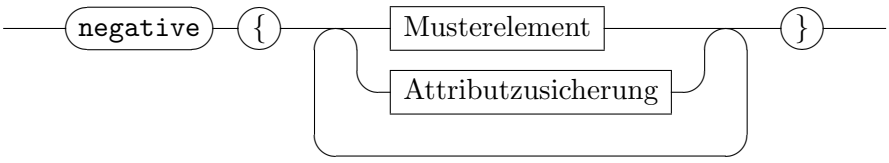
In der erste Zeile wird eine Superkante mit einem Tentakel vom Typ „Wort“ definiert. Dem Tentakel ist ein Attribut „teuer“ zugeordnet. In der zweiten Zeile wird dann ein zweites Attribut „sehr“ dem zuvor definierten Attribut „teuer“ zugeordnet. Abbildung 5.14 zeigt die Darstellung der Umsetzung durch den Übersetzer.

Abbildung 5.14: Regeldefinition – Attribut eines Attributs

5.3.2.5 Negative Anwendungsbedingungen (NAC)

Eine negative Anwendungsbedingung definiert Musterelemente, die im Wirtsgraphen, auf den die Regel angewendet werden soll, nicht vorkommen dürfen. Negative Anwendungsbedingungen sind also Voraussetzungen, die erfüllt sein müssen, damit eine Regel angewendet werden kann. Jede negative Anwendungsbedingung (siehe Kapitel 2.2) wird durch das Schlüsselwort *negative* eingeleitet. Danach folgen beliebig viele Musterelemente oder Zusicherungen.

NAC



Das folgende Beispiel verlangt, dass im Wirtsgraph keine zwei verschiedenen Knoten vom Typ „Wort“ vorkommen, die den gleichen Namen haben und von denen der eine die Rolle „AG“ und der andere die Rolle „ACT“ hat, damit die Regel „Beispiel“ nicht angewendet wird. Dabei müssen die beiden Superkanten unterschiedlich sein. Würden die beiden Knoten mit ihren Rollen und identischen Namen in einer Superkante auftreten, so wäre die negative Anwendungsbedingung erfüllt und die Regel nicht anwendbar.

```
1   rule Beispiel {
2           pattern {
3                   negative {
4                           x[a:Wort|AG];
5                           y[b:Wort|ACT];
6                           if { a.NAME == b.NAME; }
7                   }
8                   ...
9           }
10  }
```

5.3.2.6 Attributzusicherungen

Attributzusicherungen sind eine Art Erweiterung von Musterelementen. Mit Musterelementen werden Voraussetzungen an die Struktur des Wirtsgraphen gestellt, während mit Zusicherungen Voraussetzungen an die GrGen-Attribute der einzelnen Graphelemente gestellt werden. Die Zusicherungen an die Attribute werden durch logische Formeln ausgedrückt. Attributzusicherungen können sowohl in Mustern als auch in negativen

Anwendungsbedingungen auftreten. Die Syntax ist direkt von GrGen übernommen. Für Details siehe [GBG+06].
Das folgende Beispiel zeigt eine Attributzusicherung:

```
if { ( x.wert > 2 && x.wert < 10 ) || x.wert == 12; }
```

5.3.3 Rechte Seite

Da GrGen SPO-Semantik verwendet (siehe Kapitel 4.3), müssen Graphelemente, die nicht gelöscht werden sollen, auf der rechten Seite wiederholt werden. Dies ist besonders bei Kontextabfragen, Mengenabfragen oder Definitionen von Elementen mit Tentakeln, bei denen der Übersetzer zusätzlich Graphelement in die linke Seite eingefügt, problematisch. Das folgende Beispiel verdeutlicht diese Problematik anhand der Definition einer Mengenabfrage auf der linken Seite:

```
1  rule Beispiel {
2          pattern {
3                  [{x}:Wort];
4          }
5          replace {
6                  {x};
7          }
8  }
```

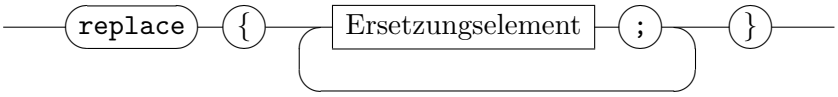
Auf der linken Seite wird ein Element vom Typ „Wort“ aus einer Menge gesucht. Mengenabfragen können nur im Rumpf eines Graphelements gestellt werden, da Mengen auch nur im Rumpf einer Elementdefinition erzeugt werden können. Diese Regeldefinition in erweiterter Syntax wird vom Übersetzer in die folgende GrGen-Syntax überführt.

```
1  rule Beispiel {
2          pattern {
3                  ZZx0:NODE;
4                          ZZx0 -:ROLE-> ZZx1:SET;
5                          x:Wort -:SET_CLOSURE-> ZZx1;
6          }
7          replace {
8                  ZZx0 -:ROLE-> ZZx1;
9                  x -:SET_CLOSURE-> ZZx1;
10         }
11 }
```

Der Übersetzer erzeugt mehrere Knoten und Kanten, die dem Anwender der Erweiterungen verborgen bleiben. Es ist deshalb Aufgabe des Übersetzers, diese zusätzlich erzeugten Elemente auf der rechten Seite zu wiederholen, wenn der Anwender das eigentliche Element auf der rechten Seite angegeben hat.

Die rechte Seite einer Regel (Ersetzung) beginnt mit dem Schlüsselwort *replace*, gefolgt von beliebig vielen Ersetzungselementen.

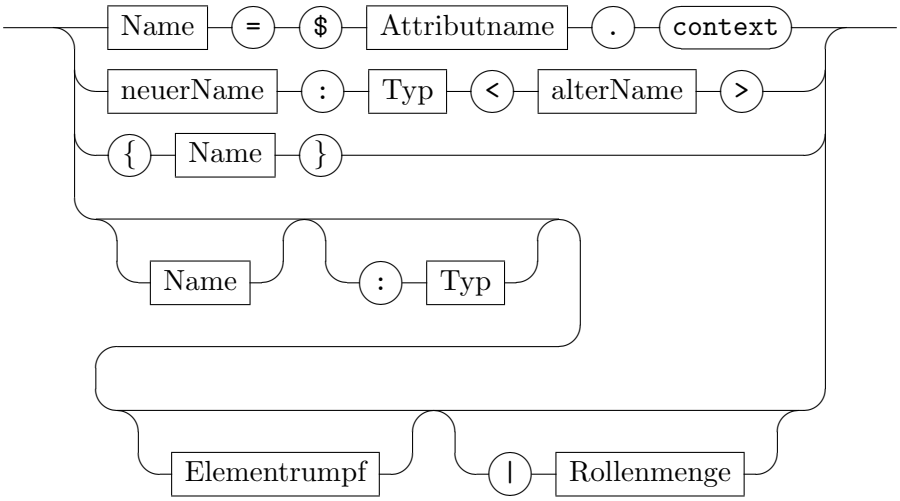
Ersetzung



5.3.3.1 Ersetzungselement

Bei einem Ersetzungselement kann es sich genau wie bei einem Musterelement entweder um eine Kontextabfrage oder ein Graphelement mit einem Elementrumpf handeln. Zusätzlich gibt es eine vereinfachte Notation, um Mengenabfragen zu reproduzieren und die Möglichkeit Graphelemente zu retypisieren. Ein Graphelement hat einen optionalen Namen, Typ und Elementrumpf sowie eine optionale Rollenmenge.

Ersetzungselement



Damit alle auf der linken Seite einer Regel definierten Graphelemente auch auf der rechten Seite wieder definiert werden können (und somit vor der Löschung bewahrt werden) stehen dem Benutzer folgende Möglichkeiten zur Verfügung:

1. Knoten und Superkanten können analog zur linken Seite definiert werden. Es gilt die gleiche Syntax.

2. Kontextabfragen werden ebenfalls analog zur linken Seite definiert. **Jede auf der linken Seite verwendete Kontextabfrage muss auf der rechten Seite ebenfalls definiert sein, sonst werden Teile des Graphen gelöscht.**

3. **Mengenabfragen müssen ebenfalls auf der rechten Seite wieder definiert werden, da sonst die Hüllenkante und der vom Übersetzer eingefügte**

Mengenknoten gelöscht würden. Zur Vereinfachung genügt die Notation {x}, wobei „x“ das Element der Menge ist (siehe vorangegangenes Beispiel).

Darüber hinaus hat der Anwender die Möglichkeit, den Typ eines Knotens oder einer Superkante zu ändern. Alle Möglichkeiten sind im in Abbildung 5.15 dargestellten Beispiel verdeutlicht.

```
 1  rule Beispiel2 {
 2          pattern {
 3                  x[{y} $z];
 4                  c = $z.context;
 5                  a:Wort|AG;
 6          }
 7          replace {
 8                  {y};
 9                  c = $z.context;
10                  a|AG;
11                  b:NODE<a>;
12          }
13  }
```

Abbildung 5.15: Regeldefinition – Rechte Seite

Auf der linken Seite werden zuerst eine Superkante „x“ mit einer Mengenabfrage sowie einem der Superkante zugeordneten Attribut „z“ definiert. Danach wird der Kontext des Attributs „z“ abgefragt. Zuletzt wird ein Graphelement vom Typ „Wort“ definiert, das die Rolle „AG“ hat. Um den Graphen auf der rechten Seite vollständig zu reproduzieren, sind die folgende Definitionen notwendig: Die Mengenabfrage kann durch die Angabe des abgefragten Elements in geschweiften Klammern erreicht werden (Zeile acht). Die Superkante muss dabei nicht explizit definiert werden. Das Attribut wird durch die Wiederholung der Kontextabfrage reproduziert. Danach wird das Graphelement mit der Rolle „AG“ wiederholt.
Außerdem wird der Typ des Graphelements vom Typ „Wort“, den es auf der rechten Seite zugewiesen bekommen hat, in den Typ *NODE* geändert. Typänderungen sind nur zu Obertypen möglich. Dabei werden alle dem Knoten oder der Superkante zugeordneten Kanten und Attribute, beibehalten (siehe [GBG+06]).

5.3.4 Berechnungsteil

Wenn für eine Regel alle Voraussetzungen erfüllt sind und die linke durch die rechte Seite ersetzt wurde, können im Berechnungsteil die Werte von Attributen geändert werden. Dabei kann schreibend nur auf die Attribute von Graphelementen zugegriffen werden, die in der rechten Seite der Ersetzungsregel vorkommen. Lesend kann auch auf Elemente der linken Seite zugegriffen werden. Die Syntax ist die gleiche wie bei GrGen. Für Details siehe [GBG+06].

Das folgende Beispiel zeigt die Verwendung des Berechnungsteils:

```
1  eval {
2          x.NAME = y.NAME + "_new";
3          x.Wert = x.Wert - 3;
4  }
```

Die zweite Zeile zeigt die Änderung eines Attributs eines Zeichenkettentyps und die dritte Zeile eine Berechnung auf einem Attribut eines Zahlentyps.

In diesem Kapitel wurde für die Erweiterungen von GrGen eine geeignete Syntax entwickelt. Im nächsten Kapitel folgt die Implementierung der Erweiterungen.

6 Implementierung

Im Kapitel 3 wurde das Graphersetzungssystem GrGen als Basis für die Erweiterung um Superkanten vorgestellt. Das Konzept für Superkanten und Rollen wurde zusammen mit der zugehörigen Syntax im Kapitel 4 vorgestellt. In diesem Kapitel wird die Implementierung der Erweiterungen behandelt. Zu Beginn werden Aufbau und Funktionsweise von GrGen vorgestellt, danach der Aufbau der Erweiterungen erklärt. Am Ende werden die Erweiterungen für Modelle, Graphen und Regeln einzeln besprochen.

6.1 Einführung in GrGen

GrGen wurde ursprünglich in C geschrieben und war für die Ausführung auf Linuxsystemen konzipiert. Für die Erweiterungen wird jedoch eine .NET-Portierung von GrGen verwendet. Diese befindet sich momentan noch in der Testphase und ist noch nicht öffentlich verfügbar, wird aber voraussichtlich noch im ersten Halbjahr 2007 unter der Adresse, von der auch die C-Version [GrG] bezogen werden kann, veröffentlicht. Eine kurze Einführung findet sich in [GBG+06]. Die vollständige Dokumentation wird zusammen mit der .NET-Portierung veröffentlicht werden [BG07].

Aufbau von grGen

Jedes Graphersetzungssystem verarbeitet vier durch den Anwender erzeugte Artefakte. Dies sind: Modelle, Regeln, Graphen und Regelsequenzen. Modelle definieren die Struktur von gültigen Graphen, während Graphen eine zu verarbeitende Instanz eines Modells darstellen. Durch die Regeln gibt der Anwender die möglichen Verarbeitungsschritte für ein dem Modell entsprechenden Graphen vor. Eine Regelsequenz definiert die Reihenfolge, in der einzelne Regeln auf einen Graphen angewendet werden. Regeln müssen immer an ein oder mehrere Modelle gebunden sein, da sie die in den Modellen definierten Knoten- und Kantentypen verwenden.
Wie diese vier Artefakte von einem Graphersetzungssystem verarbeitet werden und ob sie überhaupt voneinander unterschieden werden, ist von Graphersetzungssystem zu Graphersetzungssystem unterschiedlich. Manche System integrieren die Reihenfolge der Ausführung in die Definition der Regeln, andere vermischen die Definition von Modell und Graph.
GrGen trennt alle vier Aspekte klar voneinander und verarbeitet diese in zwei unterschiedlichen Komponenten. Der eigentliche Kern von GrGen ist der Übersetzer „GrGen“, der Modelle und Regeln in ausführbaren Code übersetzt. Die zweite Komponente ist die Ausführungsumgebung „grShell“. In ihr kann der Anwender den Code für die Modellen und Regeln laden, Graphen eingeben und Regelsequenzen definieren, die dann auf

den Graphen angewendet werden. Abbildung 6.1 zeigt den Zusammenhang der beiden Komponenten und der vier Artefakte.

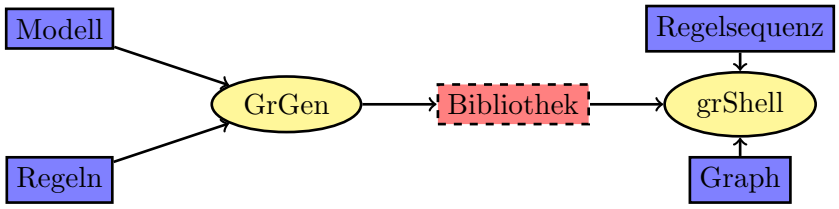


Abbildung 6.1: GrGen

Für alle schematischen Darstellungen in diesem Kapitel gilt: Artefakte, die der Anwender erzeugen muss, sind als (blaue) Rechtecke mit durchgehenden Rand dargestellt. Automatisch generierte Artefakte sind als (rote) Rechtecke mit gestricheltem Rand dargestellt. Werkzeuge werden als (gelbe) Ellipsen dargestellt.
Alternativ zur Ausführungsumgebung gibt es eine Programmierschnittstelle um eine Graphersetzung aus einer Anwendung heraus zu steuern. Für die Erweiterungen von GrGen kommt nur die Ausführungsumgebung „grShell“ zum Einsatz. Trotzdem lassen sich die Erweiterungen auch in Verbindung mit der API verwenden, bieten allerdings selbst keine eigene Programmierschnittstelle an.

Übersetzer Der Übersetzer besteht aus zwei Teilen. Zum einen aus dem auf einer ANTLR-Grammatik [Par] basierenden Javaprogramm „grgen.jar“, das aus den Modellen und Regeln C#-Quellcode erzeugt. Zum anderen aus dem in .NET realisierten Steuerprogramm „grgen.exe“, das zuerst die Codegenerierung und dann die Übersetzung des erzeugten Code in .NET-Assemblies anstößt. Abbildung 6.2 zeigt den Übersetzer schematisch dargestellt.

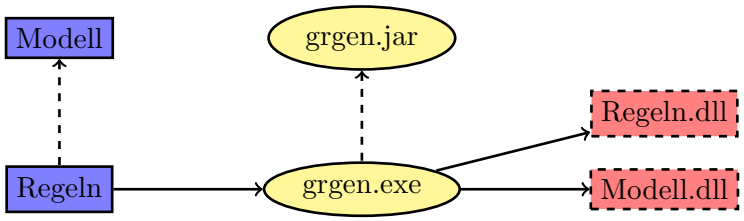


Abbildung 6.2: GrGen – Übersetzer

Der Übersetzer „grgen.exe“ erhält als Eingabeargument die Datei mit den Regeldefinitionen (in der die entsprechenden Modelle referenziert werden) und erzeugt daraus unter Verwendung von „grgen.jar“ .NET-Assemblies für die Regeln und Modelle (alle referenzierten Modelle werden in ein Assembly gepackt).
Abbildung 6.3 zeigt eine Modelldefinition, die entsprechend dem in der ersten Zeile definierten Namen, in einer Datei mit dem Namen „UMLModell.gm“ gespeichert werden muss. Die Erweiterung „.gm“ steht für „Graph Model“. In Zeile drei wird ein Knotentyp mit dem Namen „Klasse“ definiert und in den Zeilen fünf und sechs ein Kantentyp mit

```
1  model UMLModell;
2
3  node class Klasse;
4
5  edge class Vererbung
6          connect Klasse[1] -> Klasse[1];
```

Abbildung 6.3: GrGen – Modelldefinition

den Namen „Vererbung“. Kanten von diesem Typ verbinden jeweils genau zwei Knoten vom Typ „Klasse“ miteinander.

```
1  actions UML using UMLModell;
2
3  rule erzeugeKind {
4          pattern {
5                  k:Klasse;
6          }
7          replace {
8                  kind:Klasse;
9                  k -:Vererbung-> kind;
10         }
11 }
```

Abbildung 6.4: GrGen – Regeldefinition

Abbildung 6.4 zeigt eine Regeldefinition. Analog zu Modellen ist auch bei Regeln der Name in der ersten Zeile definiert. Die Regeldefinition muss in der Datei „UML.grg“ abgespeichert werden. Die Erweiterung „.grg“ steht für „Graph Grammar“. Die definierte Regel erhält den Namen „erzeugeKind“ und erwartet auf der linken Seite einen Knoten vom Typ „Klasse“. Die rechte Seite erzeugt einen neuen Knoten vom Typ „Klasse“ und eine Kante vom Typ „Vererbung“ zwischen dem neuen Knoten und dem auf der linken Seite definierten Knoten.
Die Übersetzung der Modell- und Regeldefinitionen erfolgt mit dem Befehl `grgen.exe UML.grg`. Als Ergebnis werden die beiden .NET-Assemblies „lgsp-UMLModel.dll“ und „lgsp-UMLActions.dll“ erzeugt.

Ausführungsumgebung Die Ausführungsumgebung „grShell“ besteht nur aus der in .NET realisierten Anwendung „grShell.exe“. Nach ihrem Start muss zuerst ein Graph initialisiert werden. Dazu muss ein Modell in Form des entsprechenden .NET-Assemblies geladen werden. Danach kann das Assembly für die Regeln geladen werden. Die einzelnen Elemente des Graphen können direkt in der Ausführungsumgebung erzeugt werden oder aus einer Datei geladen werden. Das gleiche gilt für die Regelsequenz. Abbildung 6.5 zeigt einen beispielhaften Eingabedialog für die Ausführungsumgebung.
In der ersten Zeile wird zuerst das Backend geladen. Das Backend enthält den für die Suche von Übereinstimmungen zwischen der linken Seite einer Regel und dem Wirtsgraphen notwendigen Algorithmus. In der zweiten Zeile wird ein leerer Graph erzeugt

```
1  >select backend "lgspbackend.dll"
2  >new graph "../lib/lgsp-UMLModel.dll" "UMLGraph"
3  >select actions "../lib/lgsp-UMLActions.dll"
4  >
5  >new k1:Klasse
6  >
7  >grs erzeugeKind
```

Abbildung 6.5: GrGen – Ausführungsumgebung

und es wird ihm der Name „UMLGraph“ zugewiesen. Anschließend werden die Regeln geladen. In der Zeile fünf wird ein Knoten vom Typ „Klasse“ erzeugt. In der letzten Zeile wird die Regel mit dem Namen „erzeugeKind“ auf den Graph angewendet.

Visualisierung Die Ausführungsumgebung kann selbst keine Graphen visualisieren. Da aber die visuelle Darstellung der Graphen von großem Nutzen für den Anwender ist, kommt das Werkzeug „yComp“ zum Einsatz. Es kann den aktuellen Arbeitsgraph der Ausführungsumgebung visuell darstellen bzw. auch gespeicherte Graphen laden. Zusätzlich verfügt „yComp“ über einen interaktiven Modus, mit dessen Hilfe man Grapherersetzungsregeln schrittweise ausführen kann. Dabei wird nicht nur das Ergebnis der Ausführung dargestellt, sondern zuerst die Übereinstimmung der linken Seite im aktuellen Graph gezeigt und dann die Änderungen durch die Ersetzung mit der rechten Seite. Abbildung 6.6 zeigt die Darstellung des vorangegangenen Beispiels mittels „yComp“, nach der Ausführung der „erzeugeKind“-Regel.

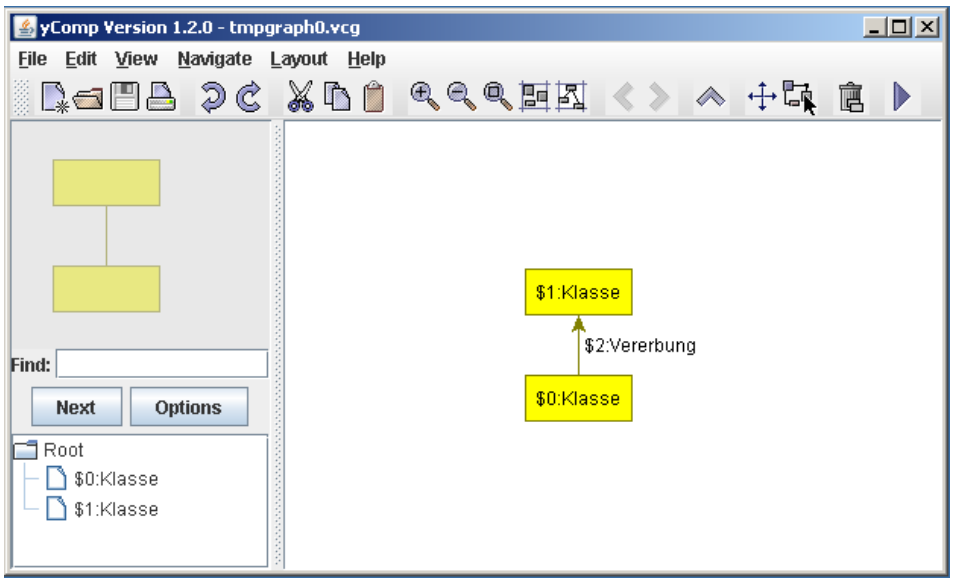


Abbildung 6.6: GrGen – Visualisierung

Um „yComp“ zu starten muss in der Ausführungsumgebung der Befehl `show graph ycomp` eingegeben werden. Mit `debug enable` startet der interaktive Modus.

Modellprüfung

Die Ausführungsumgebung prüft einen Graphen nicht direkt bei der Eingabe auf Gültigkeit gegenüber dem vorgegebenen Modell, sondern macht dies erst auf Anweisung des Anwenders. Es können also Graphen erzeugt werden, die nicht dem vorgegebenen Modell entsprechen. Um die Prüfung anzustoßen, wird der Befehl `validate strict` verwendet. Er liefert als Ausgabe alle Graphelemente des Graphen, die gegen das Modell verstoßen. Abbildung 6.7 zeigt das Ergebnis der Modellprüfung des in Abbildung 6.6 dargestellten Graphen.

```
1  > validate strict
2  The graph is NOT valid:
3    CAE: Klasse "$1" [0<1] -- Vererbung  --> Klasse
4    CAE: Klasse -- Vererbung  --> Klasse "$0" [0<1]
```

Abbildung 6.7: GrGen – Modellprüfung

Der Graph ist gegenüber dem Modell in Abbildung 6.3 nicht gültig, da nur eine Kante zwischen den Knoten existiert. Das Modell definiert aber für jeden Knoten vom Typ „Klasse“ genau eine eingehende und eine ausgehende Kante.

6.2 Erweiterung von GrGen

Bereits zuvor wurde erklärt, dass sämtliche Erweiterungen auf die bestehende Funktionalität von GrGen zurückgeführt werden, um den Aufwand zu begrenzen. Trotzdem gibt es zwei grundsätzliche Möglichkeiten dies zu realisieren:

- Die Erweiterungen können direkt in GrGen integrierte werden. Dazu muss der Zerteiler für den Übersetzer und die Ausführungsumgebung angepasst werden. Die Transformation der Erweiterungen auf die Funktionalität von GrGen, findet dann innerhalb von GrGen statt.
- Die Erweiterungen können durch einen eigenständigen Übersetzer realisiert werden. Dieser übersetzt die erweiterte Syntax in die Syntax von GrGen. Bei dieser Vorgehensweise werden an GrGen selbst keine Änderungen vorgenommen.

Da sich GrGen insgesamt und momentan speziell die Portierung auf .NET noch in ständiger Weiterentwicklung befindet, wurde, in Rücksprache mit den Entwicklern von GrGen, die zweite Variante gewählt. Sie verhindert Probleme, die bei der parallelen Weiterentwicklung entstehen würden. Der eigenständige Übersetzer ist bei der Übergabe der Daten an GrGen nur auf die Eingabesyntax von GrGen angewiesen und diese ist, im Gegensatz zu den internen Schnittstellen von GrGen, deutlich weniger Änderungen unterworfen. Somit sind die Abhängigkeiten bei der zweiten Variante geringer.

Die geplanten Erweiterungen von GrGen betreffen nur drei der vier Artefakte, die der Anwender erzeugen muss. Die Syntax der Regelsequenzen wird von den Erweiterungen nicht beeinflusst, während die Syntax der Modell- und Regeldefinition, sowie der Grapheingabe erweitert oder sogar komplett verändert wird. Für diese drei Artefakte muss jeweils ein Übersetzer entwickelt werden, der die erweiterte Syntax in die Syntax von GrGen überführt. Das Konzept für die Überführung der Syntax wurde in Kapitel 4 vorgestellt. Abbildung 6.8 zeigt, die aus den drei Übersetzern bestehende Erweiterung von GrGen.

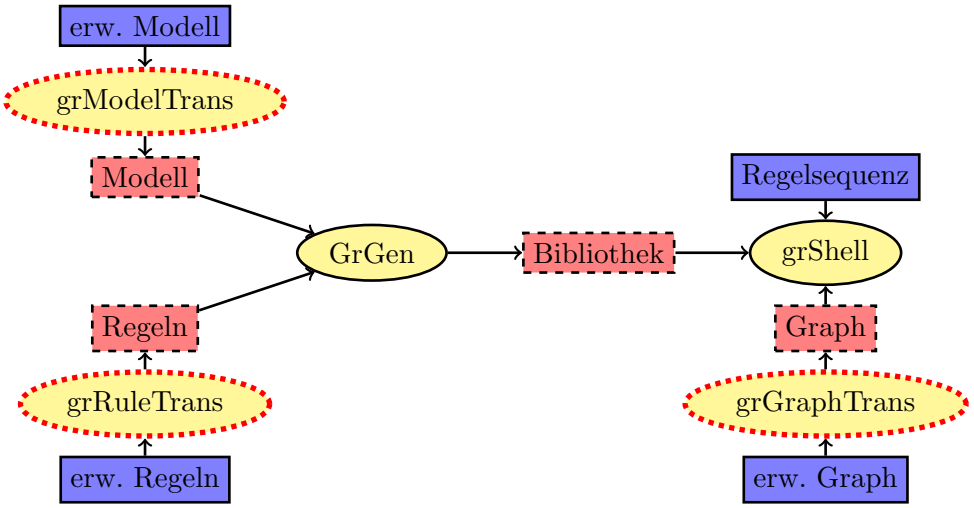


Abbildung 6.8: GrGen – Erweiterung

Der Übersetzer für die Modelle hat den Namen „grModelTrans“, der Übersetzer für die Regeln den Namen „grRuleTrans“ und der Übersetzer für die Grapheingabe den Namen „grGraphTrans“. Die Dateinamen der Artefakte sind gegenüber der Konvention für GrGen um ein „e“ für „extended“ erweitert worden:
Tabelle 6.1 zeigt die Dateierweiterungen von GrGen und der Erweiterung, sowie die Bedeutungen.

Artefakt	GrGen-Syntax	erweiterte Syntax
Modelle	*.gm	*.egm (extended graph model)
Regeln	*.grg	*.egrg (extended graph grammar)
Graphen	*.grs	*.egrs (extended graph rewrite sequence)

Tabelle 6.1: GrGen – Dateierweiterungen

Der Begriff Graphersetzungssequenz (graph rewrite sequence) bezieht sich nicht nur auf Graphen, sondern auf alle Eingaben, die in der Ausführungsumgebung möglich sind und damit auch auf Regelsequenzen.

Die Übersetzer „grRuleTrans“ und „grGraphTrans“ sind jeweils Java-Anwendungen, die auf einer ANTLR-Grammatik zum Zerteilen der erweiterten Syntax basieren. Für den Übersetzer der erweiterten Modellsyntax „grModelTrans“ wurde ein anderer Ansatz verwendet: Er basiert ebenfalls auf einer ANTLR-Grammatik, nutzt diese allerdings nicht in einer Java-Anwendung, sondern zerteilt die erweiterte Modellsyntax in einen abstrakten Syntaxbaum. Dieser wird dann mittels Graphersetzung in einen abstrakten Syntaxbaum entsprechend der GrGen-Syntax übersetzt. Dieser Vorgang wird im folgenden Abschnitt detailliert beschrieben.
Die Idee für die Realisierung mittels Graphersetzung, entstand während eines gemeinsamen Treffens mit meinem Betreuer und dem verantwortlichen Entwickler von GrGen. Ziel war, die theoretisch mögliche Realisierung eines Übersetzers mittels Graphersetzung, durch eine Implementierung praktisch nachzuweisen. Außerdem erste Erkenntnisse in der „Programmierung durch Graphersetzung“ mit GrGen zu sammeln. Meine persönlichen Erfahrungen habe ich im Anhang 8 beschrieben.

6.3 Modelltransformation

Die Transformation der erweiterten Modellsyntax in die Modellsyntax von GrGen erfolgt in drei Schritten (die alle innerhalb der Ausführungsumgebung von GrGen ablaufen):

- Zerteilung der erweiterten Syntax in einen abstrakten Syntaxbaum (AST). Dazu wird das Modul „ASTdapter“ verwendet, das mit Hilfe eines Zerteilers die Eingabedatei mit der erweiterten Syntax zerteilt und als abstrakten Syntaxbaum (in Form eines Graphen) in der Ausführungsumgebung zur Verfügung stellt.
- Transformation des abstrakten Syntaxbaums. Durch Graphersetzungsregeln wird der abstrakte Syntaxbaum (AST), so transformiert, dass er der Syntax von GrGen entspricht (AST*). Die Transformation erfolgt anhand des in Kapitel 4 entwickelten Konzepts zur Übersetzung von erweiterten Modelldefinitionen.
- Erzeugung der GrGen-Syntax aus dem abtrakten Syntaxbaum (AST*). Der abstakte Syntaxbaum existiert als Graph in der Ausführungsumgebung und wird mit Hilfe des Moduls „grIO“ wieder in eine Datei geschrieben, so dass der GrGen-Übersetzer diese Modelldefinitionsdatei verarbeiten kann.

Abbildung 6.9 veranschaulicht die drei Schritte einer Modelltransformation. AST beschreibt den abstrakten Syntaxbaum der aus der erweiterten Syntax erzeugt wird und AST* den abstrakten Syntaxbaum, entsprechend der GrGen-Syntax, beschreibt.

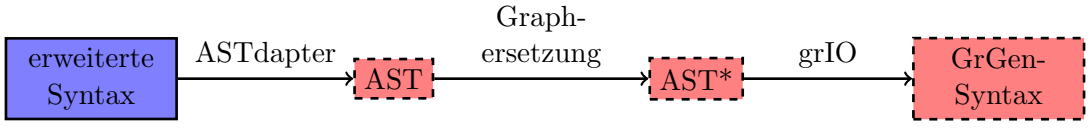


Abbildung 6.9: Implementierung – Modellübersetzer (grModelTrans)

Der gesamte Übersetzungsvorgang erfolgt ausschließlich durch ein Graphersetzungssystem und zeigt, dass ein Übersetzer allein durch Graphersetzung realisiert werden kann. Eine persönliche Bewertung dieser Vorgehensweise, im Gegensatz zur Realisierung durch eine imperative Programmiersprache, findet sich wie bereits erwähnt im Anhang 8.
Die drei Schritte der Modelltransformation werden nachfolgend einzeln beschrieben.

6.3.1 Zerteilung der erweiterten Syntax

Die Zerteilung der erweiterten Syntax erfolgt durch das Modul „ASTdapter“. Dieses benötigt allerdings einen Zerteiler, in Form eines .NET-Assemblies. Der Zerteiler (grModelParser) wird aus einer ANTLR-Grammatik (grModelParser.g, grModelLexer.g) erzeugt. Die Grammatik spezifiziert die erweiterte Syntax für Modelldefinitionen. Da es sich bei der erweiterten Syntax um eine echte Erweiterung handelt, enthält sie automatisch auch die GrGen-Syntax für Modelldefinitionen, so dass für den transformierten abstrakten Syntaxbaum (AST*) kein eigenes Modell benötigt wird. Die Erzeugung des Zerteilers ist in Abbildung 6.10 als „1“ gekennzeichnet. Die als (grüne) Rechtecke mit gepunktetem Rand dargestellten Artefakte, werden nicht vom Anwender erstellt, sondern wurden vom Entwickler des Systems erstellt. Der Code für die Erzeugung des Zerteilers ist in Zeile zwei bis vier von Abbildung 6.11 dargestellt.
Für die Transformation des abstrakten Syntaxbaums müssen neben dem Zerteiler auch Assemblies für das zugrunde liegende Graphmodell und die Transformationsregeln erzeugt werden. Dies ist nur einmalig notwendig. Nur bei Änderungen an der erweiterten Syntax für Modelldefinitionen oder den Transformationsregeln für die Modellübersetzung, müssen die Assemblies (Zerteiler, Modell und Regeln) neu erzeugt werden.

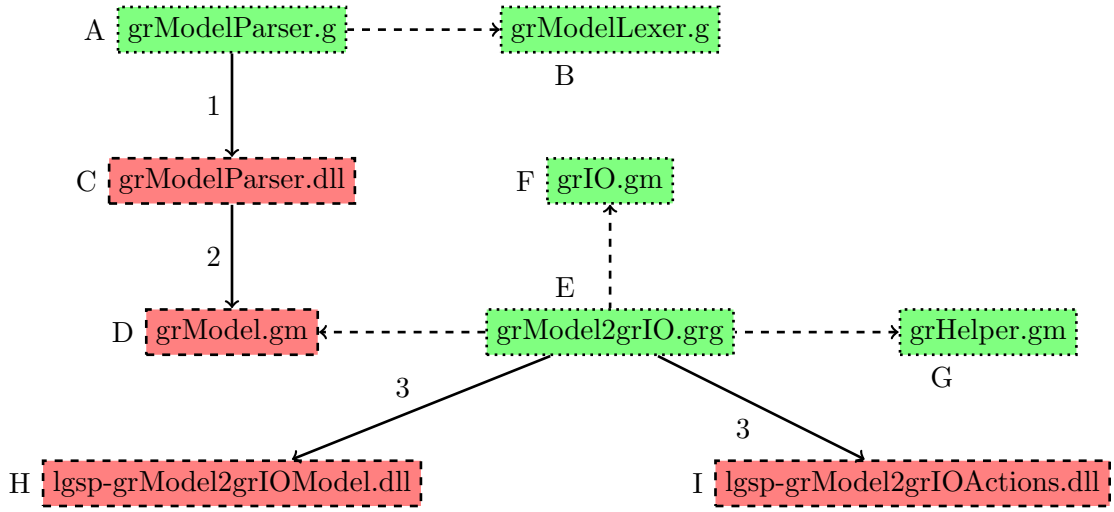


Abbildung 6.10: Implementierung – Generierung des Zerteilers (grModelTrans)

Das Graphmodell für die erweiterte Syntax (D) kann automatisch aus dem Zerteiler (C) generiert werden. Dieser Schritt ist mit „2“ in Abbildung 6.10 gekennzeichnet. Der zugehörige Befehl steht in Zeile sieben der Abbildung 6.11. Zusammen mit den Regel-

definitionen (E), dem Graphmodell (F) für das grIO-Modul und einem weiteren Graphmodell (G), das einige Hilfsgraphelemente definiert, kann das Graphmodell (D) mit dem GrGen-Übersetzer in die benötigten Assemblies (H, I) übersetzt werden (Markierung „3“ in 6.10, Zeile neun in 6.11).

```
1  REM step 1
2  java -cp antlr.jar antlr.Tool grModelLexer.g
3  java -cp antlr.jar antlr.Tool grModelParser.g
4  csc /t:library /reference:antlr.runtime.dll /out:grModelParser.dll *.cs
5
6  REM step 2
7  ASTdapter.exe grModelParser.dll
8
9  REM step 3
10 grgen.exe grModel2grIO.grg
```

Abbildung 6.11: Implementierung – Generierung des Zerteilers (makeModelParser.bat)

Das in Abbildung 6.11 dargestellte Skript „makeModelParser.bat“ erzeugt die für die Modelltransformation notwendigen Assemblies (Zerteiler, Modell, Regeln). Es muss nur ausgeführt werden, wenn Änderungen an einer der in Abbildung 6.10 grün mit gepunktetem Rand dargestellten Dateien erfolgt sind.

Die eigentliche Zerteilung der erweiterten Syntax ist in Abbildung 6.12 dargestellt.

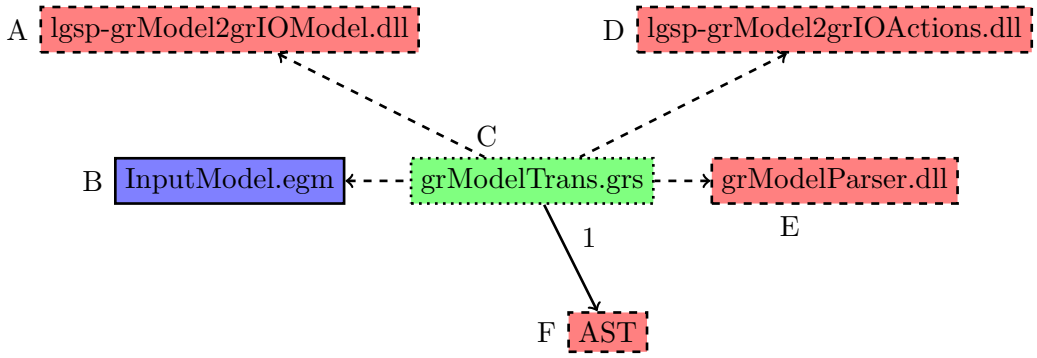


Abbildung 6.12: Implementierung – Transformationsschema (grModelTrans)

Dort sieht man den Zerteiler (E) und die Assemblies für das Graphmodell (A) und die Regeln (D), sowie das Steuerungsskript (grModelTrans.grs). Das zu transformierende Modell (in erweiterter Syntax) befindet sich in der Datei „InputModel.egm“ (B) und wird durch Schritt 1 in den abstrakten Syntaxbaum überführt. Der Code dazu befindet sich in Abbildung 6.13.

6.3.2 Transformation des abstrakten Syntaxbaums

Im vorangegangenen Schritt wurde das Modell (in erweiterter Syntax) zerteilt und liegt jetzt als abstrakter Syntaxbaum (Graph) in der Ausführungsumgebung vor. In diesem

```
1  select backend "lgspbackend.dll"
2  new graph "../lib/lgsp-grModel2grIOModel.dll" "grModelTrans"
3  select actions "../lib/lgsp-grModel2grIOActions.dll"
4  select parser "grModelParser.dll" "text"
5  parse file "InputModel.egm"
```

Abbildung 6.13: Implementierung – Transforamtion Teil 1 (grModelTrans.grs)

Schritt werden die im abstrakten Syntaxbaum enthaltenen, erweiterten Graphelemente (Superkanten, Rollen) in die von GrGen unterstützten Graphelemente (Knoten, Kanten) transformiert.

```
1  model Beispiel;
2
3  role class Endpunkt (Klasse) {
4      Multiplizitaet : string;
5  }
```

Abbildung 6.14: Implementierung – Rollenbeispiel (Definition)

Um eine Vorstellung des abstrakten Syntaxbaums zu vermitteln, ist als Beispiel (Abbildung 6.15) der AST der Definition des Rollentyps „Endpunkt“ aus Abbildung 6.14 dargestellt. Bei der Abbildung handelt es sich um einenBildschirmausdruckk von yComp, das zwar eine automatische Anordnung in Baumformunterstütztz, allerdings in diesem Fall scheitert, da sich der Baum aufgrund seiner Breite nicht auf einer Bildschirmseite darstellen lässt. Die Abbildung ist deshalb nicht in Baumform.
Die Nummern verdeutlichen die Reihenfolge der Graphelemente. Ein „IDENT“-Knoten steht für einen beliebigen Bezeichner, „LPAREN“ und „RPAREN“ sind die öffnende und schließende runde Klammer. „LBRACE“ und „RBRACE“ sind analog die geschweiften Klammern. „COLON“ ist der Doppelpunkt und „SEMI“ der Strichpunkt.
In der Mitte sieht man die beiden nicht nummerierten Knoten „MODEL“ und „IDENT“, sie stehen für die Definition des Modells. Die nummerierten Knoten sind die einzelnen Elemente der Rollendefinition. Die abstrakten Syntaxbäume für Knoten, Kanten und Superkanten sehen entsprechend aus.

Die gesamte Transformation wird durch das Skript „grModelTrans.grs“ gesteuert. Der Anfang des Skripts, der die Zerteilung des Eingabemodells in den abstrakten Syntaxbaums steuert, wurde bereits in Abbildung 6.13 dargestellt. Nachfolgend werden jeweils einzelne Abschnitte des Skripts erklärt.

Obertypen einfügen Alle Knotentypen erben automatisch von *NODE* (siehe Abbildung 6.16).
Der gemeinsame Obertyp ermöglicht es, die beiden Attribute „NAME“ und „VALUE“ für alle Knoten verfügbar zu machen. Außerdem können in Regeln mit Hilfe des Obertyps beliebige Knoten gesucht werden (dies wäre auch schon durch den von GrGen zur

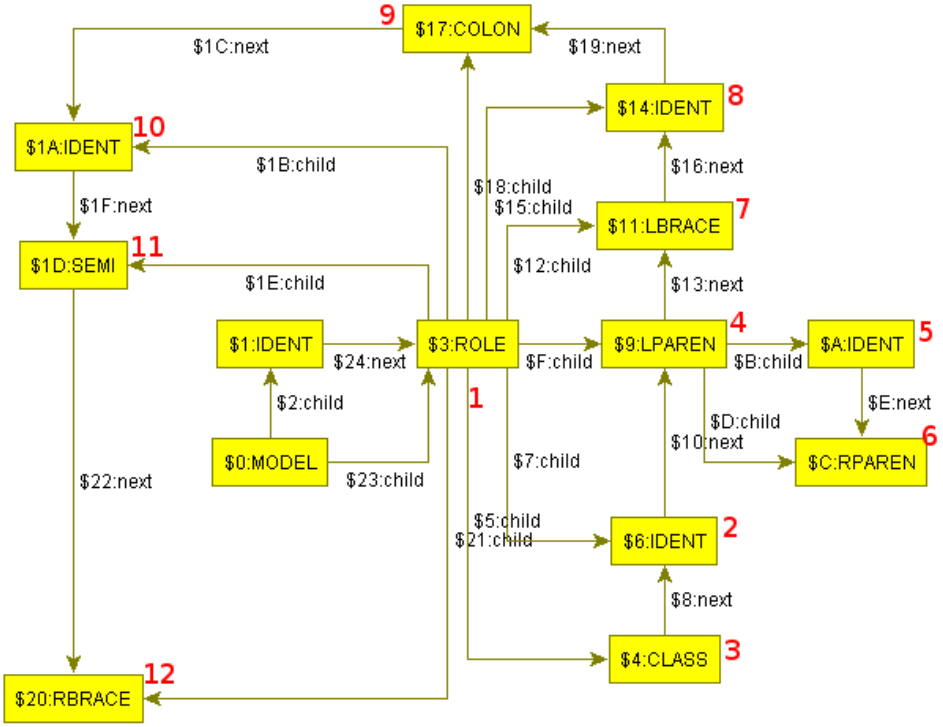


Abbildung 6.15: Implementierung – Rollenbeispiel (abstrakter Syntaxbaum)

Verfügung gestellten Knotenobertyp *Node* möglich, allerdings fehlen dann die beiden Attribute). Das Attribut „NAME“ speichert den vom Benutzer in einem Graphen für das Graphelement vergebenen Namen. Das Attribut „VALUE“ ist speziell für die Darstellung von Texten als Graph gedacht: Es speichert den „Inhalt eines Knoten“. Dies sind bei einem Knoten die zugeordneten Attribute (sowohl GrGen-Attribute als auch dynamische Attribute) und bei einer Superkante alle Tentakel und Attribute. Das folgende Beispiel verdeutlicht die Bedeutung der Attribute:

Satz[#Der Entwickler|AG arbeitet|ACT $fleißig].

Die Superkante „Satz“ enthält in diesem Beispiel die Attribute NAME=„Satz“ und VALUE=„#Der |AG |ACT $fleissig“. Abbildung 6.17 zeigt das Beispiel als Graph.
Damit auch die vom Anwender definierten Knotentypen den Obertyp *NODE* erben, muss dieser in alle Knotendefinitionen, die nicht von einem anderen Typ erben, eingefügt werden. Typen, die von einem anderen Typ erben, bekommen von diesem auch den Obertyp vererbt. Das Einfügen des Knotenobertyps erfolgt durch die Regel *grStdNodeExtends* (Abbildung 6.18).
Analog zu Knoten haben auch Superkanten einen gemeinsamen Obertyp: *SUPEREDGE*. Attribute hat *SUPEREDGE* nicht, erbt aber die Attribute von *NODE*, da eine Superkante ja in einen „Verbindungsknoten“ übersetzt wird. Damit in Regeln nach beliebigen

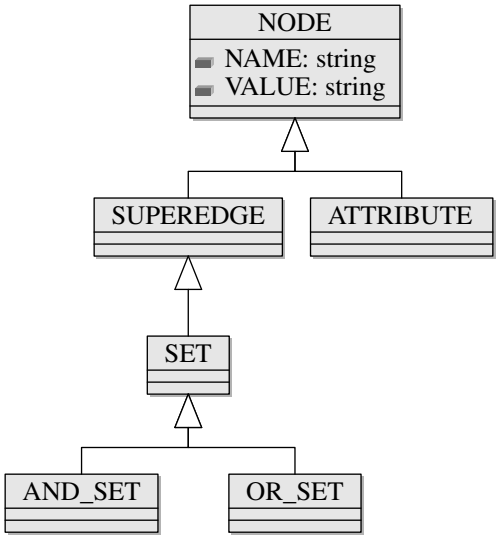


Abbildung 6.16: Implementierung – Standard-Knotentypen

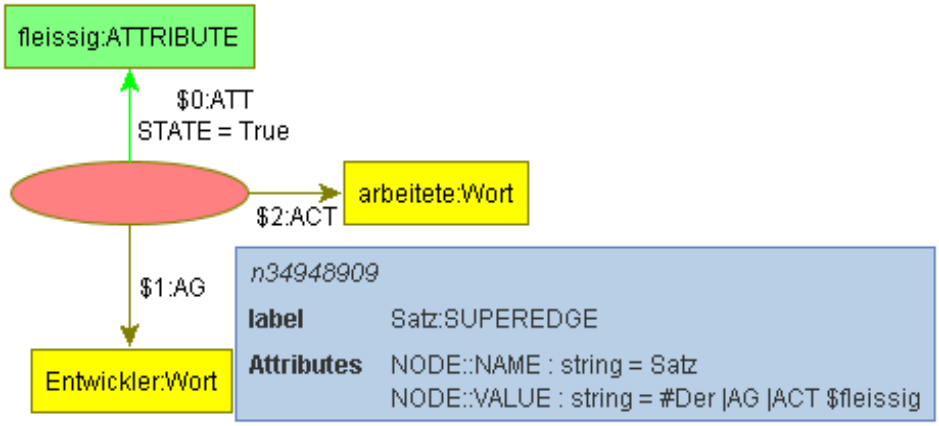


Abbildung 6.17: Implementierung – Beispiel mit Knotenattributen

Superkanten gesucht werden kann, wird ein eigener Superkantenobertyp zur Verfügung gestellt. Die Regel *grStdSEdgeExtends* fügt bei Bedarf den Obertyp in Definitionen von Superkanten ein.
Für Rollen gibt es ebenfalls eine Obertyp mit dem Namen *ROLE* (siehe Abbildung 6.19). Der Obertyp hat das Attribut „STATE“, das die Art der Verwendung der Rolle beschreibt. Rollen können sowohl positiv als auch negativ verwendet werden. Mit Hilfe des Obertyps kann in Regeln nach beliebigen Rollen gesucht werden. Der Rollenobertyp wird durch die Regel *grStdRoleExtends* eingefügt. Analog zu Knoten und Superkanten, wird er nur bei Rollen eingefügt, die bisher keinen Obertyp haben.

```
6   # Insert the "extends NODE" to nodes without extends and "extends SUPEREDGE" to
7   # superedges without extends
8   grs grStdNodeExtends* ; grStdSEdgeExtends* ; grStdRoleExtends*
```

Abbildung 6.18: Implementierung – Transformation Teil 2 (grModelTrans.grs)

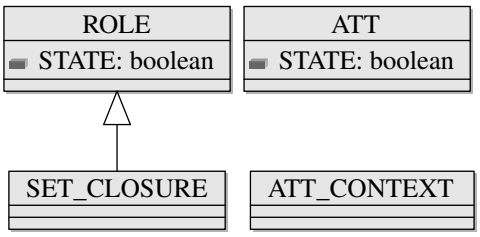


Abbildung 6.19: Implementierung – Standard-Kantentypen

Für Kanten gibt es keinen Obertyp, da der Anwender in der Erweiterung von GrGen nur Superkanten nutzen kann. Es ist zwar möglich, in erweiterten Modellen Kanten zu definieren, allerdings kann der Anwender die Kanten nicht in der erweiterten Grapheingabe oder erweiterten Regeldefinition nutzen. Der Grund für diesen inkonsequenten Entwurf liegt darin, dass durch die Tolerierung von Kanten in der erweiterten Syntax für Modelle eine gemeinsame Grammatik für die erweiterte Syntax und die GrGen-Syntax verwendet werden kann. Dies vereinfacht die Transformation von Modellen.

Einheitliche Form für Zusicherungen von Rollen und Superkanten Bei der Definition von Graphelementtypen können für Rollen und Superkanten Zusicherungen definiert werden. Um diese Zusicherungen mit möglichst wenigen Regeln weiterverarbeiten zu können, werden sie in diesem Schritt in eine einheitliche Form gebracht. Abbildung 6.20 zeigt den abstrakten Syntaxbaum, der aus der folgenden Rollendefinition entsteht:

```
role class AG (Wort, SUPEREDGE);
```

Die drei vom Anwender gewählten Bezeichner sind in der Abbildung markiert: „AG“ (1), „Wort“ (2) und „SUPEREDGE“ (3). Bei den beiden letzten handelt es sich um Knotentypen, die als Zusicherung an die Rolle verwendet werden (sie müssen im Modell zuvor bereits definiert worden sein).
Um Zusicherungen in eine einheitliche Form zu bekommen, wird ein Knoten vom Typ „CONNECT“ eingefügt. Dabei handelt es sich um einen der in „grHelper.gm“ definierten Hilfstypen, die nur für die Transformation benötigt werden. Das Schema der einheitlichen Form für Rollen ist in Abbildung 6.21 dargestellt. Um den Bezug zum vorangegangenen Beispiel herzustellen, sind die beiden Zusicherungen markiert. Die restlichen Knoten des Beispiels („:CLASS“, „:IDENT“ (3) und „SEMI“) sind nicht dargestellt.
Abbildung 6.22 zeigt das Schema für Superkanten. Analog zu Rollen wird auch hier ein Knoten vom typ „CONNECT“ eingefügt.

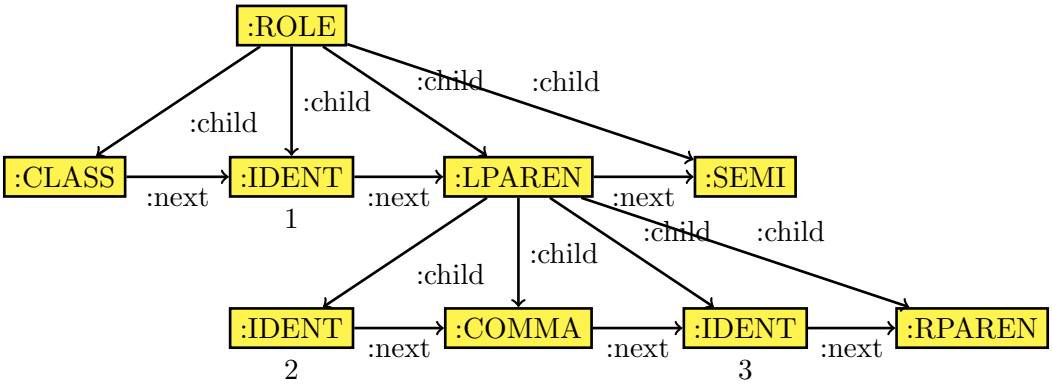


Abbildung 6.20: Implementierung – Zusicherungen bei Rollen

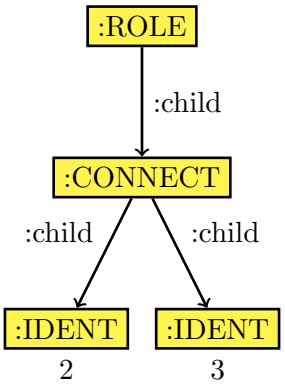


Abbildung 6.21: Implementierung – Zusicherungen bei Rollen (Schema)

Um das Schema zu verdeutlichen, wurde die Zusicherungen aus dem folgenden Beispiel nummeriert und im Schema markiert:

```
superedge class Satz (AG, !ACT, PAT+, *);
```

Die Zuordnung ist: „AG“ (1), „ACT“ (2), „PAT“ (3) und „*“ (4).

Entfernen doppelter Zusicherungen Der Anwender hat die Möglichkeit, bei der Definition von Zusicherungen von Superkanten die gleiche Rolle mehrfach zu verwenden. Dies kann durch die Syntaxprüfung des Zerteilers nicht erkannt werden. Deshalb entfernen die Regeln *grSEdgeRemDblSTAR*, *grSEdgeRemDblPLUS*, *grSEdgeRemDblID* und *grSEdgeRemDblNOT* mehrfach auftretende Zusicherungen (siehe Abbildung 6.24).
Das folgende Beispiel stellt das Problem doppelter Zusicherungen dar:

superedge class Satz (ACT*,!ACT,AG+,AG+);

Das Beispiel definiert eine Superkante mit dem Namen „Satz“, die Rolle „ACT“ ausschließt und Rolle „AG“ als Tentakel mindestens einmal erfordert. Die doppelte Verwendung von „ACT“ ist kein Problem, da unterschiedliche Modifikatoren verwendet werden

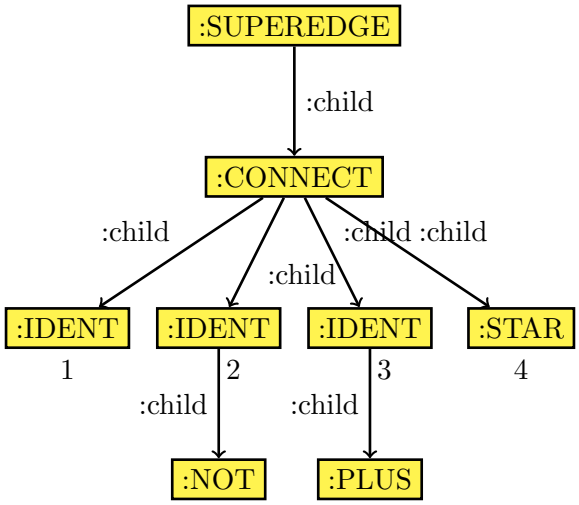


Abbildung 6.22: Implementierung – Zusicherungen bei Superkanten (Schema)

```
 9  # Generate role assertions and superedge assertions
10  grs grRoleInsCONNECTwAss* ; grRoleInsCONNECTwAtt* ; grRoleInsCONNECTwSemi*
11          ; grRoleInsCONNECTwExt* ; grRoleRemRPAREN* ; grSEdgeInsCONNECT*
12          ; grSEdgeRewSTAR* ; grSEdgeRewNOT* ; grSEdgeRewPLUS* ; grSEdgeRemRPAREN*
```

Abbildung 6.23: Implementierung – Transformation Teil 3 (grModelTrans.grs)

und sich der strengere Modifikator durchsetzt (siehe Abschnitt 5.1.1.1). Die doppelte Verwendung von „AG“ mit dem gleichen Modifikator würde aber die weitere Transformation stören und muss deshalb entfernt werden.

Standardtypen einfügen In diesem Transformationschritt werden die in den Abbildungen 6.16 und 6.19 dargestellten Standardtypen in den abstrakten Syntaxbaum hinein kopiert, damit sie für die Transformation der Zusicherungen zur Verfügung stehen (die Zusicherungen der Standardtypen werden in die Transformation mit einbezogen). Die Regeln dazu sind in Abbildung 6.25 zu sehen.

Zusicherungen transformieren Die Zusicherungen von Rollen und Superkanten wurden in einem früheren Schritt in eine einheitliche Form gebracht, damit sie einfacher transformiert werden können. Das folgende Beispiel verdeutlicht den Zweck der Transformation: In Abbildung 6.26 ist die Definition eines Graphmodells gezeigt (es handelt sich um einen Ausschnitt aus dem in 7.1.1 zur Darstellung von Texten verwendeten „Sprachmodell“).
In diesem Modell wird in Zeile drei der Knotentyp „Wort“ definiert. In den Zeilen fünf und sechs folgen zwei Rollentypen, die jeweils auf den Knotentyp „Wort“ oder auf den Superkantentyp „Satz“ angewendet werden können („AG“ zusätzlich auf beliebige Mengen). Zuletzt wird in Zeile acht ein Superkantentyp „Satz“ definiert, der als Tentakel die beiden zuvor definierten Rollen „AG“ und „ACT“ zulässt.

```
13  # Remove double modifiers on superedges
14  grs grSEdgeRemDblSTAR* ; grSEdgeRemDblPLUS* ; grSEdgeRemDblID*
15          ; grSEdgeRemDblNOT*
```

Abbildung 6.24: Implementierung – Transformation Teil 4 (grModelTrans.grs)

```
16  # Generate STD node and edges TYPES
17  grs grStdSUPEREDGE ; grStdNODE ; grStdATTRIBUTE ; grStdATT ; grStdATT_CONTEXT
18          ; grStdROLE ; grStdSET ; grStdANDSET ; grStdORSET ; grStdSET_CLOSURE
```

Abbildung 6.25: Implementierung – Transformation Teil 5 (grModelTrans.grs)

Abbildung 6.27 zeigt das Modell nach der Übersetzung in die GrGen-Syntax.
Bei der Transformation in die GrGen-Syntax werden aus Superkanten Knoten und aus Rollen Kanten. Da es in der GrGen-Syntax nur Kantenzusicherungen gibt, müssen die Zusicherungen der Rollentypen und der Superkantentypen vereint werden. Auf der linken Seite der neuen Kantenzusicherungen stehen die Superkantentypen, die in ihrer Definition die entsprechende Rolle zulassen. Auf der rechten Seite stehen die Knotentypen, die als zulässige Endpunkte der Rollen definiert wurden. Insgesamt hat eine Rolle $n*(m+1)$ Zusicherungen, wobei n die Zahl der für die Rolle zulässigen Knotentypen und m die Zahl der Superkantentypen, die als Tentakel die Rolle zulassen, ist. Die Konstante 1 steht für den Typ *SET*, der alle Rollen zulässt. (Mengen können beliebige Superkantentypen aufnehmen, deshalb ist der Mengentyp immer zulässig).
In Abbildung 6.28 sind die Regeln für die Transformation von Zusicherungen dargestellt. Da die Modifikatoren unterschiedliche Priorität haben (der strengste Operator setzt sich durch, siehe Abschnitt 5.1.1.1), werden die Zusicherungen nach aufsteigender Priorität verarbeitet. Zuerst wird der „*“-Modifikator (Zeile 20), danach der „+“-Modifikator (Zeile 21) verarbeitet. Anschließend folgen Rollen ohne Modifikator (Zeile 22) und der „!“-Modifikator (Zeile 23). Durch diese Reihenfolge wird sichergestellt, dass sich der strengste Modifikator durchsetzt, da eventuell bereits verarbeitete nicht so strenge Modifikatoren überschrieben werden.

Graph bereinigen Nachdem die Zusicherungen transformiert wurden, können leere „CONNECT“ Knoten im Graph zurückbleiben. Dies ist der Fall, wenn eine Rolle keine Zusicherungen hat. Sofern sie Zusicherungen hat, muss das Komma nach der letzten Zusicherung entfernt werden (die letzte Zusicherung wird mit einem Semikolon abgeschlossen). Zuletzt werden die „CONNECT“ Knoten entfernt, da sie nur zur Transformation benötigt wurden. Die Regeln zur Bereinigung sind in Abbildung 6.29 zu sehen.

Standardtypen wieder entfernen In diesem Schritt werden die Standardtypen wieder entfernt. Dies verwundert etwas, da die Standardtypen zuvor erst eingefügt wurden. Die Standardtypen waren für die Transformation der Zusicherungen notwendig. Außerdem müssen die Standardtypen in das vom Anwender erzeugte Modellassembly gelangen, sonst kann der Anwender nicht damit arbeiten. Würden die Standardtypen aber in

```
1  model Sprache;
2
3  node class Wort;
4
5  role class AG (Wort, Satz, SET);
6  role class ACT (Wort, Satz);
7
8  superedge class Satz (AG,ACT);
```

Abbildung 6.26: Implementierung – Beispiel für Zusicherungen Teil 1 (Sprache.egm)

```
 1  model Sprache;
 2
 3  node class Satz extends SUPEREDGE;
 4
 5  node class Wort extends NODE;
 6
 7  edge class ACT extends ROLE
 8      connect SET[0:*] ->  Satz[*],
 9              SET[0:*] ->  Wort[*],
10              Satz[1:1] ->  Satz[*],
11              Satz[1:1] ->  Wort[*];
12
13  edge class AG extends ROLE
14      connect SET[0:*] ->  SET[*],
15              SET[0:*] ->  Satz[*],
16              SET[0:*] ->  Wort[*],
17              Satz[1:1] ->  SET[*],
18              Satz[1:1] ->  Satz[*],
19              Satz[1:1] ->  Wort[*];
```

Abbildung 6.27: Implementierung – Beispiel für Zusicherungen Teil 2 (Sprache.gm)

jede Modelldefinition geschrieben, so käme es später zu Doppeldefinitionen, wenn der Anwender mehrere Modelldefinitionen in eine Regeldatei einbinden möchte. Abbildung 6.30 verdeutlicht diese Problematik (sie ist aus dem Evaluierungsbeispiel in Abschnitt 7.1.1 und wurde leicht vereinfacht).
Die beiden erweiterten Graphmodelle „SpracheModell.egm“ und „UMLModell.egm“ werden in entsprechende GrGen-Graphmodelle übersetzt und in „ZuUML.grg“ eingebunden. Wären die Standardtypen in beiden Graphmodelle enthalten, so würde die Übersetzung von „ZuUML.grg“ wegen doppelt definierter Namen fehlschlagen. Deshalb werden die Standardtypen nicht in die Graphmodelle gepackt, sondern bei der Übersetzung der erweiterten Regeln verlinkt (siehe Abbildung 6.31).
Die Graphdefinitionsdatei „grModelStdTypes.gm“ enthält die Standardtypen. Sie muss nicht vom Anwender erzeugt oder eingebunden werden, sondern wird durch *grRuleTrans* automatisch eingebunden.
Abbildung 6.32 zeigt die Regeln zum Entfernen der Standardtypen aus dem abstrakten Syntaxbaum.

```
19  # Rewrite role and edge assertions
20  grs ( [grRoleRewSTAR] ) & grRoleRemSTAR*
21  grs ( [grRoleChaPLUS] ; grSEdgeRemMarker* ; [grRoleRewPLUS] ) ; grRoleRemPLUS*
22  grs ( grRoleChaID* ; grSEdgeRemMarker* ; [grRoleRewID] ) ; grRoleRemID*
23  grs ( [grRoleChaNOT] ; grSEdgeRemMarker* ; [grRoleRewNOT] ) ; grRoleRemNOT*
24  grs grRoleMovAssertion*
```

Abbildung 6.28: Implementierung – Transformation Teil 6 (grModelTrans.grs)

```
25  # Remove connect nodes without any assertions
26  grs grRoleRemEmptyCONNECT*
27
28  # Remove comma between last connect statement and semi of role definition
29  grs grRoleRemCOMMAafterAssertion*
30
31  # Remove role and superedge connect statements
32  grs grRoleRemChildCONNECT* ; grSEdgeRemChildCONNECT*
```

Abbildung 6.29: Implementierung – Transformation Teil 7 (grModelTrans.grs)

Superkanten und Rollen transformieren In diesem Schritt werden die Knoten vom Typ „ROLE“ in Knoten vom Typ „Edge“ und die Knoten vom Typ „SUPEREDGE“ in Knoten vom Typ „NODE“ retypisiert. Damit werden die Rollen zu Kantentypen und die Superkanten zu Knotentypen (die den Verbindungsknoten repräsentieren). Die Retypisierung findet durch die beiden Regeln *grRoleRetype* und *grSEdgeRetype* statt (siehe Abbildung 6.33).
Nach diesem Schritt ist die Transformation des abstrakten Syntaxbaums abgeschlossen. Der abstrakte Syntaxbaum entspricht jetzt der Syntax von GrGen. Im nächsten Schritt muss der abstrakte Syntaxbaum, der immer noch als Graph in der Ausführungsumgebung vorliegt, wieder in eine Datei geschrieben werden.

6.3.3 Erzeugung der GrGen-Syntax

Um den abstrakten Syntaxbaum in eine Datei zu schreiben, kommt das Modul *grIO* zum Einsatz. Die Ausführungsumgebung kann einen Graphen zwar in eine Datei schreiben, verwendet dazu aber ein spezielles Format. Die Ausgabe des abstrakten Syntaxbaums muss aber in der Syntax von GrGen erfolgen, damit die Graphdefinition weiterverarbeitet werden kann.
Das Modul *grIO* besteht aus zwei Teilen: Zum einen aus der Graphdefinitionsdatei „grIO.gm“, die für die Ausgabe notwendige Graphelemente definiert, und zum anderen aus in die Ausführungsumgebung integrierten Code, der die Graphelemente in eine Datei schreibt. Die Graphdefinition „grIO.gm“ definiert die in Abbildung 6.34 dargestellten Typen.
Die Knoten „grIO_OUTPUT“ und „grIO_File“ sind beide Voraussetzung für die Nutzung von *grIO*. Durch das Attribut „path“ wird der Name der Ausgabedatei angegeben. Durch die Kante zwischen den beiden Knoten wird definiert, ob die Ausgabedatei,

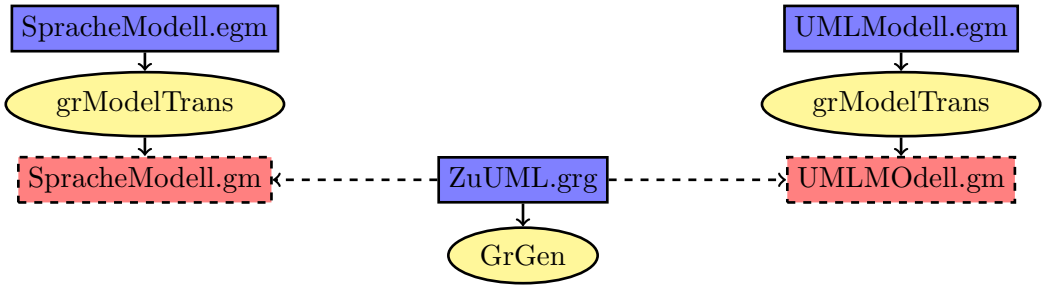


Abbildung 6.30: Implementierung – Problematik bei Standardtypen Teil 1

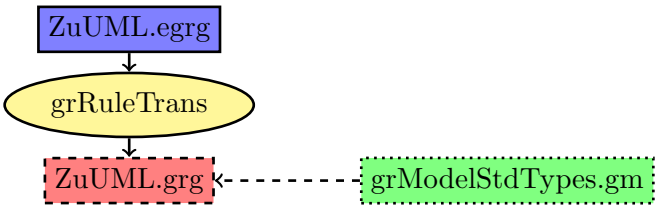


Abbildung 6.31: Implementierung – Problematik bei Standardtypen Teil 2

sofern sie bereits existiert, überschrieben („grIO_CreateOrOverwrite“) oder erweitert („grIO_CreateOrAppend“) werden soll. Für die Modelltransformation ist dieser Schritt in Abbildung 6.35 zu sehen.
Für die eigentliche Ausgabe müssen Knoten vom Typ „grIO_File_Line“ erzeugt werden und der auszugebende Inhalt als Wert des Attributs „content“ gesetzt werden. Mittels einer „grIO_File_ContainsLine“-Kante können die Ausgabezeilen einem Dateiknoten zugeordnet werden. Mit „grIO_File_NextLine“ können die Ausgabezeilen in eine Reihenfolge gebracht werden.
Abbildung 6.36 zeigt eine Beispielregel, die einen Knoten vom Typ „IDENT“ im Graph sucht und den Namen des gefundenen Knotens in die Ausgabedatei schreibt.
Die Regel löscht den gefundenen Knoten und erzeugt eine neue Ausgabezeile mit dem Namen des Knotens als Inhalt. Der Namen eines Graphelements steht im abstrakten Syntaxbaum immer im Attribut „value“.
Um ein Graphmodell, das als abstrakter Syntaxbaum (in GrGen-Syntax) vorliegt, in eine Datei zu schreiben, sind die in Abbildung 6.37 enthaltenen Regeln notwendig.
Die Regeln werden nicht einzeln vorgestellt, sondern nur das Prinzip erklärt, nach dem die Transformation eines Modells in Ausgabezeilen erfolgt.
Abbildung 6.38 zeigt die Visualisierung der Regel *grStart*. Diese Regel verarbeitet die Definition des Modells (die erste Zeile eines Modells, z.B.: `model UML;`) und fügt eine Ausgabezeile ein. Die Ausgabezeile wird durch eine Kante vom Typ „grIO_File_Pointer“ mit dem Knoten vom Typ „MODEL“ verbunden. Diese Kante markiert die zuletzt erzeugte Ausgabezeile und realisiert dadurch einen Dateizeiger, der immer an das Ende der Ausgabedatei (die bisher nur als Graph existiert) zeigt. Gleichzeitig markiert der Dateizeiger die aktuelle Stelle im abstrakten Syntaxbaum.
Abbildung 6.39 zeigt die erste Regel, um einen Knotentyp aus dem abstrakten Syntax-

```
33  # Remove all standard types. They are included later by grModelStdTypes.gm.
34  # Otherwise they would be part of every graph model, leading to double
35  # definitions if two or more models are linked in the same rule file
36  grs grStdRemSUPEREDGE ; grStdRemNODE ; grStdRemATTRIBUTE ; grStdRemATT
37          ; grStdRemATT_CONTEXT ; grStdRemROLE ; grStdRemSET ; grStdRemANDSET
38          ; grStdRemORSET ; grStdRemSET_CLOSURE
```

Abbildung 6.32: Implementierung – Transformation Teil 8 (grModelTrans.grs)

```
39  # Retype roles and superedges
40  grs grRoleRetype* ; grSEdgeRetype*
```

Abbildung 6.33: Implementierung – Transformation Teil 9 (grModelTrans.grs)

baum in die Ausgabedatei (den Graphen der Ausgabedatei) zu transformieren.
Damit die Regel ausgeführt werden kann, muss der Dateizeiger auf den Knoten vom Typ „MODEL“ zeigen. Die Regel verarbeitet den Anfang einer Knotendefinition (z.B. `node class Wort ...`) und schiebt den Dateizeiger weiter. Die bereits verarbeiteten Knoten treten nicht mehr auf der rechten Seite der Regel auf und werden somit gelöscht.
Indem mehrere Regeln dieser Art hintereinander ausgeführt werden (Zeile 47 bis 48 in Abbildung 6.37), kann eine vollständige Definition eines Knotentyps verarbeitet werden. Für Kantentypen gibt es analoge Regeln (Zeile 49 bis 52 in Abbildung 6.37).
Am Ende muss noch der Dateizeiger durch eine Regel entfernt werden (Abbildung 6.40). Nach dieser Regel ist der gesamte abstrakte Syntaxbaum der Modelldefinition in einen *grIO*-Graph transformiert und kann durch den Befehl `sync io` in die Ausgabedatei geschrieben werden (Zeile 55 in Abbildung 6.37).
Damit ist die Transformation einer Modelldefinition (in erweiterter Syntax) in eine Modelldefinition (in GrGen-Syntax) abgeschlossen. Insgesamt besteht die Transformation des abstrakten Syntaxbaums aus 55 Regeln und die Transformation in den grIO-Graph aus 26 Regeln.
Das Modul *grIO* wurde nicht im Rahmen dieser Arbeit entwickelt, lediglich der Kantentyp „grIO_File_Pointer“ wurde erweitert. Dieser Typ ist deshalb auch nicht in „grIO.gm“ sondern in der Datei „grHelper.gm“ definiert.

6.4 Übersetzung der Grapheingabe

Die Transformation der erweiterten Syntax für Grapheingaben erfolgt durch ein Java-Programm. Dieses basiert auf einer ANTLR-Grammatik der erweiterten Syntax für Grapheingaben. Die Grammatik besteht aus den beiden Teilen „grGraphTransParser“ und „grGraphTransLexer“, die im Klassendiagramm in Abbildung 6.41 oben dargestellt sind. Im Klassendiagramm sind die Klassen, die durch ANTLR aus der Grammatik erzeugt werden, dargestellt. Beide werden durch die Klasse „Trans“, die den Programmeinstiegspunkt enthält, aufgerufen.

```
 1  node class grIO_OUTPUT {
 2      timestamp : int;
 3  }
 4
 5  node class grIO_File {
 6      path : string;
 7  }
 8
 9  edge class grIO_CreateOrAppend
10      connect grIO_OUTPUT [0:1] -> grIO_File [0:*];
11
12  edge class grIO_CreateOrOverwrite
13      connect grIO_OUTPUT [0:1] -> grIO_File [0:*];
14
15  node class grIO_File_Line {
16      content : string;
17  }
18
19  edge class grIO_File_ContainsLine
20      connect grIO_File [0:1] -> grIO_File_Line [0:*];
21
22  edge class grIO_File_NextLine
23      connect grIO_File_Line [0:1] -> grIO_File_Line [0:1];
```

Abbildung 6.34: Implementierung – Ausgabemodul (grIO.gm)

```
41  new out:grIO_OUTPUT
42  new filenode:grIO_File
43  filenode.path="OutputModel.gm"
44  new out -:grIO_CreateOrOverwrite-> filenode
```

Abbildung 6.35: Implementierung – Transformation Teil 10 (grModelTrans.grs)

6.4.1 Grundprinzip

Der Übersetzer verarbeitet immer eine Graphelementdefinition nach der anderen. Dazu wird die gesamte Definition zerteilt und dann als entsprechende Definition in GrGen-Syntax wieder ausgegeben. Bevor mit der Erzeugung der Ausgabe begonnen wird, muss die Definition vollständig zerteilt sein (inklusive aller rekursiv – in Form von Tentakeln – enthaltenen Definitionen). Der Grund dafür ist die Möglichkeit, den Typ der Definition automatisch vom Übersetzer ermitteln zu lassen. Zur Erinnerung: Enthält die Definition mindestens ein Tentakel, wählt der Übersetzer den Standardtyp für Superkanten, sonst den Standardtyp für Knoten (siehe Abschnitt 5.2.1). Diese Entscheidung kann aber erst nach der vollständigen Zerteilung der Definition getroffen werden.
Damit ergibt sich die Problemstellung, alle Bestandteile einer Graphdefinition während der Zerteilung zu speichern, bis die Ausgabe daraus erzeugt wurde. Da in einer Graphelementdefiniton beliebig viele andere Elemente in Form von Tentakeln enthalten sein können, ist die Zahl der zu speichernden Elemente nicht begrenzt. Deshalb wird jedes einzelne Element als eigenes Objekt gespeichert. Dazu werden Klassen verwendet,

```
 1  rule grExtNext {
 2          pattern {
 3                  id:IDENT;
 4                  filenode:grIO_File;
 5          }
 6          replace {
 7                  l:grIO_File_Line;
 8          }
 9          eval {
10                  filenode -:grIO_File_ContainsLine-> l;
11                  l.content = id.value;
12          }
13  }
```

Abbildung 6.36: Implementierung – Anwendungsbeispiel grIO

```
45  # Rewrite AST to grIO AST
46  grs grStart
47  grs ( grNode ; grExt ; grExtNext* ; grAtt; grAttDef*
48          ; ( grAttEnd | grNodeEnd ) )*
49  grs ( grEdge ; grExt ; grExtNext* ; grEdgeAss ; ( grEdgeAssDef1
50          & ( grEdgeAssDef2 | grEdgeAssDef3 ) & grEdgeAssDef4 & grEdgeAssDef1
51          & ( grEdgeAssDef2 | grEdgeAssDef3 ) ; grEdgeAssDef5 )* ; grAtt
52          ; grAttDef* ; ( grAttEnd | grEdgeEnd ) )*
53  grs grEnd
54
55  sync io
56  exit
```

Abbildung 6.37: Implementierung – Transformation Teil 11 (grModelTrans.grs)

die den Graphelementtypen entsprechen. Die Objekte werden untereinander so verbunden, wie dies auch in einem Graphen erfolgen würde. Eine solche Menge von Objekten, die eine Graphelementdefinition darstellen, wird im weiteren Verlauf des Abschnitts als „Objektstruktur“ bezeichnet.
Die abstrakte Klasse „GraphElement“ definiert ein allgemeines Graphelement, von dem die Klassen „Knoten“, „Superkante“, „Attribut“ und „Menge“ abgeleitet sind. Rollen werden ebenfalls als Objekte gespeichert, sind aber nicht von der abstrakten Klasse „GraphElement“ abgeleitet.

6.4.2 Zerteilung

Die gesamte Logik für die Zerteilung der Grapheingabe und den Aufbau von Objektstrukturen für Definitionen ist in der Grammatik des Zerteilers enthalten. Dazu sind die Produktionen der Grammatik um entsprechenden Javacode erweitert. Abbildung 6.42 zeigt die Startproduktion der Grammatik.
Die Produktion trägt den Namen „text“ und hat einen Rückgabewert vom Typ „StringBuilder“. Der erste Teil der Produktion (Zeile sechs bis neun) prüft die optionale Angabe von Standardtypen. Sofern Standardtypen angegeben wurden, werden die entsprechen-

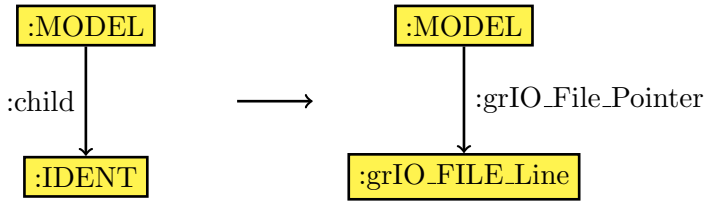


Abbildung 6.38: Implementierung – Transformationsregel (grStart)

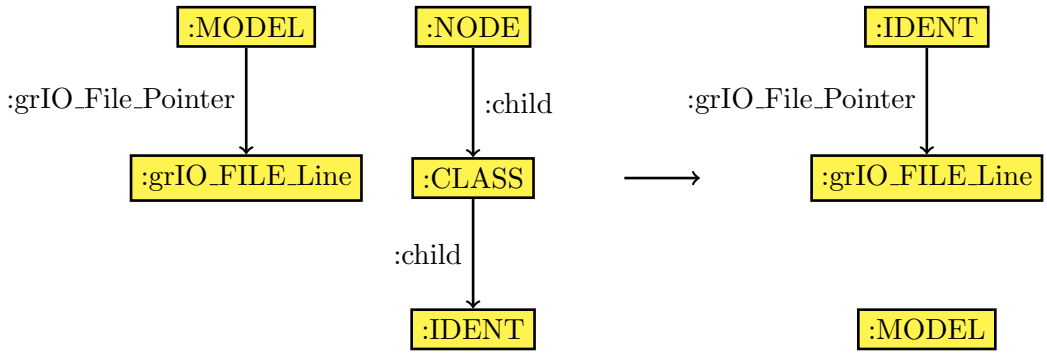


Abbildung 6.39: Implementierung – Transformationsregel (grNode)

den Instanzvariablen vom Zerteiler gesetzt. Als nächstes wird geprüft, ob der Anwender Rollen erzwingen möchte (Zeile 11). In Zeile 13 wird geprüft, ob der Anwender für Mengen disjunktive Normalform erzeugt haben möchte. Danach kommt der wichtigste Teil der Produktion, die Verarbeitung beliebig vieler Definitionen von Graphelementen (Zeile 15 bis 18). Dazu wird beliebig oft die Produktion „definition“ zugelassen. Auf jede Definition wird die Funktion „generateOutput“ aufgerufen, die aus der durch die Definition erzeugten Objektstruktur die Ausgabe erzeugt. Zuletzt wird in Zeile 20 geprüft, ob Fehler aufgetreten sind, ist dies der Fall, wird der Ausgabepuffer zurückgesetzt. Dadurch wird verhindert, dass eine fehlerhafte Ausgabe erzeugt wird. Die Ausgabe der Fehlermeldungen ist bereits durch die Methode „process“ erfolgt.

Um den Aufbau einer Objektstruktur zu verdeutlichen, wird das folgende Beispiel verwendet:

```
[Entwickler|AG Anwendung|ACT]
```

In Abbildung 6.43 ist die daraus erzeugte Objektstruktur zu sehen.
In der Mitte ist die Superkante zu sehen, rechts und links die beiden Tentakel mit den zugeordneten Rollen. Jedem Tentakel wird generell eine Rollenliste zugeordnet, auch wenn es sich nur um eine einzelne Rolle handelt. Die Tentakel werden von der Klasse „Superedge“ in einer „HashMap“ gespeichert. Dabei ist das Graphelement, aus dem das Tentakel besteht, Schlüssel und die Rollenliste Wert des Eintrags.

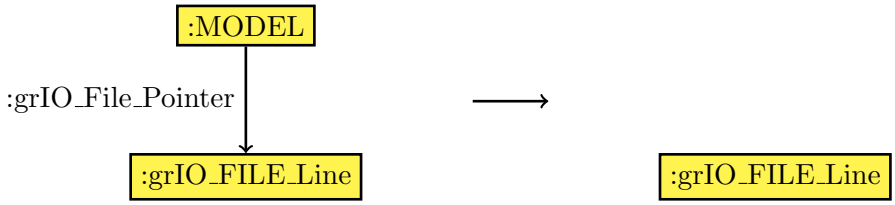


Abbildung 6.40: Implementierung – Transformationsregel (grEnd)

Jedes Graphelement hat die folgenden Attribute: Einen automatisch erzeugten, eindeutigen Identifikator, den vom Benutzer vergebenen Namen (sofern das Element nicht anonym ist), den Namen des Typs (vom Benutzer angegeben, oder ein automatisch zugeordneter Standardtyp), einen Wert, dynamische Attribute und GrGen-Attribute. Außerdem werden die Zeile und Spalte, an der das Element in der Eingabedatei steht, gespeichert, um lokalisierte Fehlermeldungen erzeugen zu können. Die Liste der dynamischen Attribute enthält Graphelemente vom Typ „Attribute“, die jeweils ein dynamisches Attribut realisieren. Die GrGen-Attribute werden dagegen direkt als „HashMap“ gespeichert. Dabei ist der Attributname Schlüssel und der Attributwert der Wert des Eintrags. Dynamische Attribute sind nur fürKnotenn und Superkanten möglich, trotzdem werden sie zur Vereinfachung in der Oberklasse „GraphElement“ gespeichert. Die Bedeutung des Attributs Wert wurde in Abschnitt 6.3.2 erläutert, es speichert den „Inhalt“ des Graphelements in Textform.
Die von „GraphElement“ abgeleitete Klasse „Set“ hat ein Attribut „members“ um die Elemente der Menge zu speichern. Außerdem ein Attribut „type“, das den Typ der Menge (und-verknüpfte Menge, oder-verknüpfte Menge) speichert. In der Klasse „Superdege“ ist das bereits beschriebene Attribut für die Speicherung der Tentakel, sowie ein Attribut für das Kopfelement der Superkante, definiert.
Eine Rolle hat einen Namen und eine Zustand, der angibt, ob die Rolle positiv oder negativ verwendet wird. Eine Rollenliste kann ebenso wie ein Graphelement GrGen-Attribute haben. Diese gelten dann für alle Rollen der Liste. Einzelnen Rollen einer Liste GrGen-Attribute zuzuordnen ist nicht möglich. Die Zuordnung von GrGen-Attributen zu Rollen ist möglich, da Rollen in GrGen-Kanten übersetzt werden und diesen GrGen-Attribute zugeordnet werden können, sofern sie im Graphmodell definiert sind.

6.4.3 Ausgabe erzeugen

Für die Erzeugung der Ausgabe ist in „GraphElement“ die abstrakte Methode „generateOutput“ definiert. Diese Methode generiert die Ausgabe für ein Graphelement und muss von jeder Unterklasse überschrieben werden. Um die Ausgabe für eine Definition, die bereits in eine Objektstruktur zerteilt wurde, zu erzeugen, genügt es auf dem obersten Objekt einer Objektstruktur (das alle anderen Elemente der Definition enthält), diese Methode aufzurufen (Entwurfsmuster Kompositum). Das Weiterreichen der Methode an die anderen Objekte erfolgt automatisch. Ein Objekt vom Typ „Superedge“ reicht die „generateOutput“-Methode an alle Tentakel weiter und ein Objekt vom Typ „Set“ an

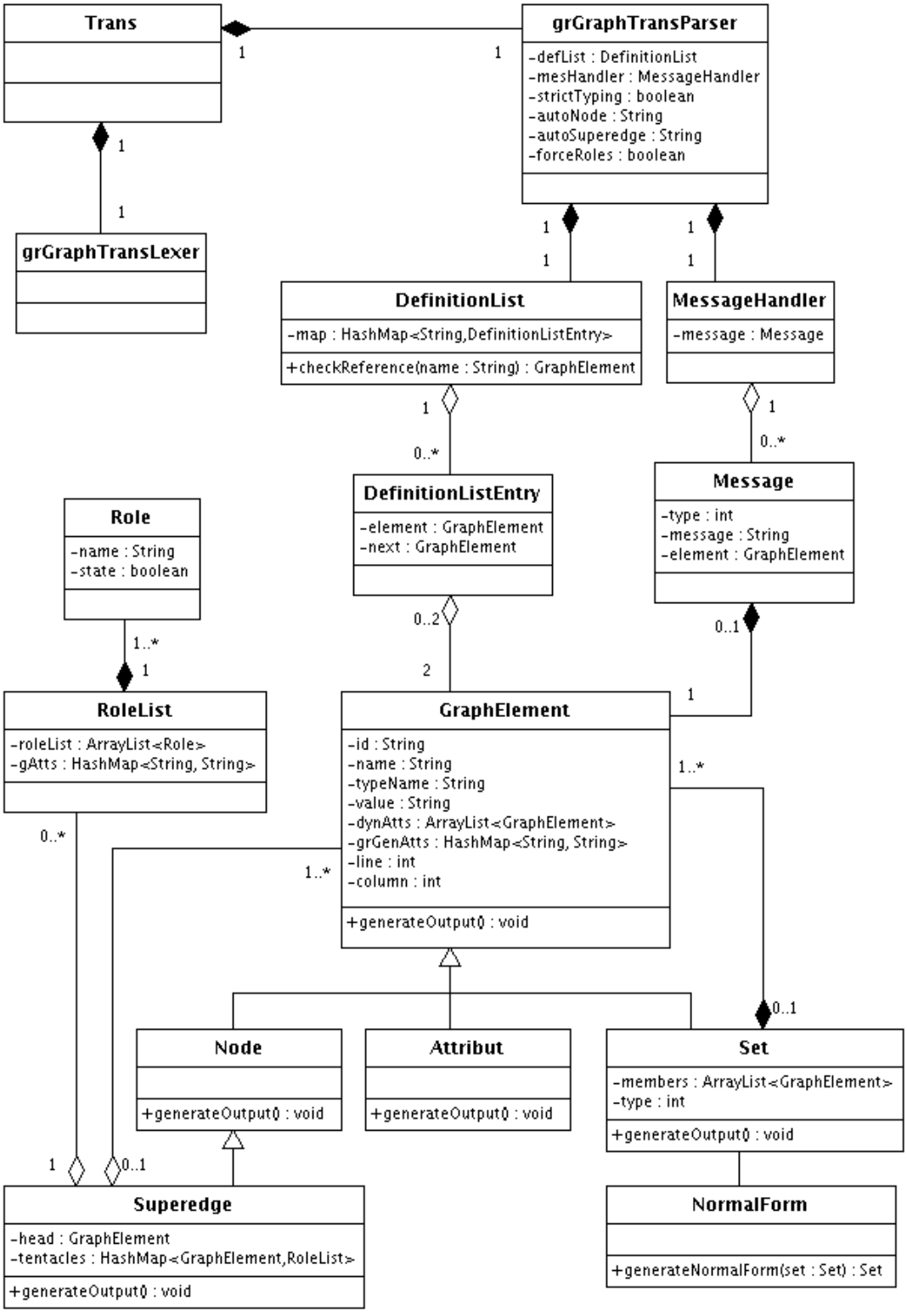


Abbildung 6.41: Implementierung – Klassendiagramm von grGraphTrans

```
 1  text returns[StringBuilder sb  = new StringBuilder()] {
 2                  this.sb = sb;
 3                  GraphElement el = new Node();
 4                  Token node, edge; }
 5          :
 6          ( AUTO NODE node=ident COMMA SUPEREDGE edge=ident SEMI {
 7                          autoNode = node.getText();
 8                          autoSuperedge = edge.getText();
 9                          strictTyping = false; } )?
10
11          ( FORCE ROLES SEMI {forceRoles = true; } )?
12
13          ( DISJUNCTIVE NORMAL FORM SEMI { GraphElement.setNormalForm(true); } )?
14
15          ( el=definition {
16                  el.generateOutput(el, el, null, sb);
17                  sb.append("\n");
18          } )*
19
20          { if (mesHandler.process()) sb = new StringBuilder(); }
21          ;
```

Abbildung 6.42: Implementierung – Beispiel einer Produktion der Zerteilergrammatik

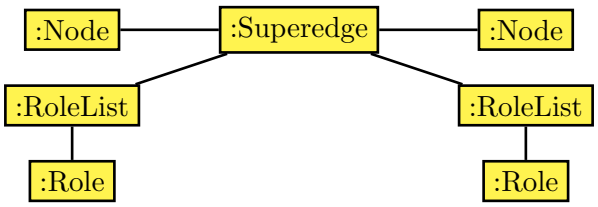


Abbildung 6.43: Implementierung – Beispiel für die Übersetzung der Grapheingabe

alle Elemente in der Menge.

6.4.4 Behandlung von Referenzen

Da die einzelnen Definitionen in einer Grapheingabe nicht unabhängig sind, sondern sich untereinander referenzieren können, ist es nicht möglich alle zu einer Definition gehörenden Objekte nach Verarbeitung der Definition sofort wieder zu löschen. Die Definitionen, die vom Anwender einen Namen erhalten haben, müssen gespeichert werden, um bei einer eventuellen Referenzierung darauf zurückgreifen zu können. Dazu werden die Definitionen mit Namen in einer Definitionsliste vermerkt. Alle Definitionen ohne Namen können vom Anwender nicht weiter referenziert werden und werden deshalb auch nicht gespeichert. Hat der Anwender allen Graphelementen einen Namen gegeben, so wird der komplette Graph im Speicher gehalten.
Das folgende Beispiel verdeutlicht, welche Definitionen gespeichert werden:

```
:Satz[Entwickler:Wort :Wort Teilsatz:Satz[:Wort :Wort]]
```

Die äußere Superkante hat keinen Namen und wird deshalb nicht gespeichert. Das erste Tentakel „Entwickler“ wird gespeichert, während das zweite nicht gespeichert wird. Die innere Superkante „Teilsatz“ wird wegen ihres Namens gespeichert, während ihre beiden Tentakel nicht gespeichert werden.

Um Referenzierungen überprüfen zu können, speichert der Zerteiler eine Instanz von „DefinitionList“, die als Instanzvariable eine „HashMap“ enthält. Als Schlüssel wird der definierte Name und als Wert das zugehörige Graphelement gespeichert. Dadurch lassen sich Anfragen, ob ein Name bereits definiert ist, sehr schnell beantworten. Diese Anfragen erfolgen mit der Methode „checkReference“. Als Parameter wird der zu prüfende Name übergeben und als Rückgabewert wird das entsprechende GraphElement oder, falls der Name nicht definiert ist, der Nullzeiger zurückgegeben.
Die Logik für die Behandlung von Namensüberdeckungen ist ebenfalls in der Definitionslistenklasse enthalten. Überdeckung bedeutet, dass der Anwender für zwei Graphelementen den gleichen Namen vergeben hat und diese Namenskollision automatisch aufgelöst wird (siehe Abschnitt 5.2.2.4). Die Änderung des Namens erfolgt durch Anhängen einer fortlaufenden Nummer. Verwendet der Nutzer den Originalnamen als Referenz nachdem die zweite Definition (die automatisch umbenannt wurde) erfolgt ist, so soll das umbenannte Graphelement referenziert werden (dies wurde so festgelegt). Der geänderte Namen kann nicht als neuer Eintrag in die „HashMap“ aufgenommen werden, denn er würde unter dem Originalnamen nicht gefunden werden. Sondern muss unterhalb des Originaleintrags verlinkt werden. Deshalb hat ein Definitionslisteneintrag „DefinitionListEntry“ neben dem Zeiger auf das eigentliche Element, das er repräsentiert, einen Zeiger auf ein mögliches nachfolgendes Element.
Die Auflösung einer Referenz ergibt sich somit folgendermaßen: Zuerst wird der Namens in der „HashMap“ gesucht und anschließend überprüft, ob der gefundene Definitionslisteneintrag auf einen weiteren Eintrag verweist (dieser Schritt muss wiederholt werden, bis das Ende der Verweiskette erreicht ist). Jeder Eintrag, auf den verwiesen wird, enthält eine Definition, die den Originalnamen überdeckt. Der letzte Eintrag in der Kette, ist die letzte Überdeckung und somit das gewünschte Element.

6.4.5 Fehlerbehandlung

Neben der Definitionsliste speichert der Zerteiler eine Nachrichtenliste (eine Instanz von „MessageHandler“). Diese Nachrichtenliste speichert Fehler und Warnungen, die während der Verarbeitung der Eingabe auftreten.
Der Übersetzer versucht eine Graphdefinition immer vollständig zu verarbeiten und dann anschließend alle gefunden Fehler und Warnungen auszugeben. Eine Fehlermeldung (eine Warnung) besteht aus einem Typ, der die Art des Fehlers (der Warnung) definiert, einem Namen und dem Graphelement, das den Fehler (die Warnung) ausgelöst hat.
Der Zerteiler enthält darüber hinaus noch vier weitere Attribute. „strictTyping“ ist stets auf wahr gesetzt und wird nur geändert, wenn der Benutzer Standardtypen für Knoten und Superkanten definiert (siehe 5.2.4). Die Namen der Standardtypen werden in den Attributen „autoNode“ und „autoSuperedge“ gespeichert. Das Attribut „forceRoles“

wird auf wahr gesetzt, wenn der Benutzer definiert hat, Rollen zu erzwingen.

6.5 Regeltransformation

Für die Übersetzung von Regeln wird, wie bei der Übersetzung von Grapheingaben, ein Java-Programm verwendet, das auf einer Grammatik für die erweiterte Syntax von Regeln basiert.

6.5.1 Grundprinzip

Eine Regel besteht aus den drei Teilen: Linke Seite („pattern“), rechte Seite („replace“) und dem Berechnungsteil („eval“). Abbildung 5.8 aus dem vorherigen Kapitel, zeigt ein Beispiel einer Regel. In der linken Seite können zusätzlich negative Anwendungsbedingungen („negative“) und Attributzusicherungen („if“) auftreten.
Im Berechnungsteil und in den Attributzusicherungen werden nur GrGen-Attribute verwendet[1]. Diese sind von der Erweiterung von GrGen jedoch nicht betroffen und müssen nicht in GrGen-Syntax übersetzt werden. Deshalb ist in der Klasse „Trans“ die Methode „getText“ definiert, die komplette Zeilen aus der Eingabedatei auslesen kann. Diese werden unverändert in die Ausgabedatei übernommen. Somit sind Attributzusicherungen und der Berechnungsteil bereits behandelt und es bleiben nur noch die linke und die rechte Seite übrig. Negative Anwendungsbedingungen müssen nicht getrennt von der linken Seite behandelt werden, da sie sich nur semantisch unterscheiden, die Syntax ist identisch. Die Klasse „Trans“ instanziiert wie bei der Übersetzung der Grapheingabe den Zerteiler. Abbildung 6.44 zeigt das vollständige Klassendiagramm.
Im Unterschied zu der Übersetzung der Grapheingabe, müssen bei Regeln keine Informationen zwischengespeichert werden, sondern können bei der Zerteilung sofort in die Ausgabedatei geschrieben werden. Es wird deshalb keine Objektstruktur aufgebaut.

6.5.2 Ablauf

Jede Regel in einer Regeldefinitionsdatei bildet eine von den anderen Regeln unabhängige Einheit. Informationen müssen, soweit notwendig, nur innerhalb der Übersetzung einer Regel zwischengespeichert werden. Dazu speichert der Zerteiler eine Definitions- und eine Hüllenliste, die nach jeder Regel wieder neu initialisiert werden. Als „Definition“ wird jedes einzelne, auf der linken oder rechten Seite definierte, Graphelement bezeichnet. Eine Definition besteht nur aus einem Namen und einem Verweis auf ein eventuell vorhandenes Vaterelement (bei einem Tentakel ist die Superkante, in dem es definiert ist, das Vaterelement). Außerdem gibt es zwei Attribute mit Wahrheitswerten, die festhalten, auf welcher Seite der Regel die Definition bereits verwendet wurde. Ein Element kann

[1] Um den Wert von dynamischen Attributen zu prüfen oder zu ändern, muss auf der linken Seite zuerst das gewünschte Graphelement vom Typ *ATTRIBUTE* gesucht werden und dann das GrGen-Attribut „VALUE“ dieses Attributs geändert werden.

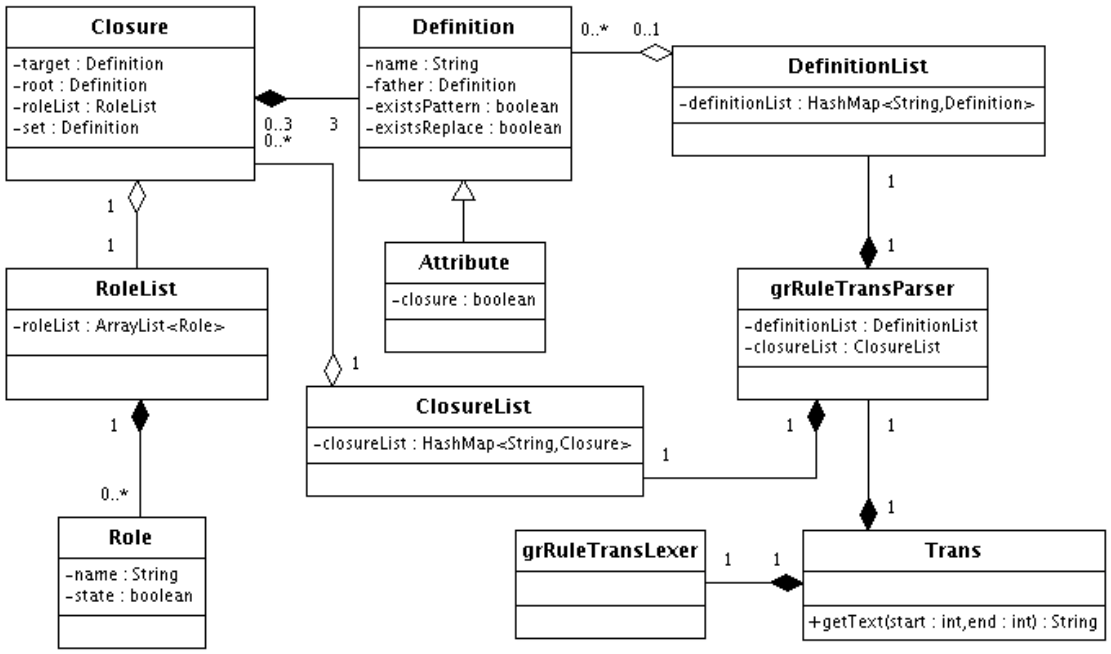


Abbildung 6.44: Implementierung – Klassendiagramm von grRuleTrans

auf der linken, auf der rechten oder auf beiden Seiten definiert werden. Die von „Definition“ abgeleitete Klasse „Attribut“ hat außerdem das Attribut „closure“, das festhält, ob das Attribut in oder außerhalb einer Menge gesucht wird.
Die Notwendigkeit, Definitionen zu speichern, zeigt das folgende Beispiel aus der linken Seite einer Regel.

```
a:Wort|AG;
```

Der Übersetzer erzeugt daraus das in Abbildung 5.10 des vorherigen Kapitels angegebene GrGen-Muster. Er muss einen Verbindungsknoten einer Superkante einfügen, um die Kante erzeugen zu können. Dies ist verständlich, da ein Knoten mit einer Rolle nur als Tentakel einer Superkante definiert werden kann. Der Übersetzer speichert in der Definitionsliste deshalb zwei Objekte ab (siehe Abbildung 6.45).

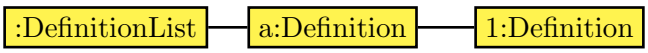


Abbildung 6.45: Implementierung – Beispiel für die Regelübersetzung

Die Definitionsliste enthält als Objekt das Element, das den Knoten vom Typ „Wort“ repräsentiert („a“ in der Abbildung). Dieses Objekt enthält als Vater das Objekt, das den Verbindungsknoten der Superkante repräsentiert („1“ in der Abbildung). Dieser Verbindungsknoten bekommt einen automatisch generierten Namen, der gespeichert werden

muss, damit der Anwender auf der rechten Seite durch die folgende Musterdefinition, die linke Seite reproduzieren kann:

```
a|AG;
```

Um den auf der linken Seite vom Übersetzer eingefügten Verbindungsknoten wieder reproduzieren zu können, muss der Übersetzer den Namen des Knotens kennen. Dies erfolgt mit Hilfe der Definitionsliste und dem Vaterverweis im Objekt „a“.
Die gleiche Problematik gilt für Hüllenabfragen auf der linken Seite. Hier erzeugt der Übersetzer ebenfalls Kontextknoten (siehe Abschnitt 4.2.5), die auf der rechten Seite reproduziert werden müssen. Hier kommt die Hüllenliste zum Einsatz. Eine einzelne Hüllenkante enthält die Attribute Ziel, Quelle, Rollenliste und Menge. Die Bedeutung ist mittels eines Beispiels in Abbildung 6.46 dargestellt:

Abbildung 6.46: Implementierung – Beispiel für Mengensemantik

Die Quelle ist die Superkante, in der die Menge als Tentakel definiert ist. Das Ziel ist der Knoten, auf den die Hüllenkante verweist. Die Rollenliste enthält die Rollen, die der Menge zugewiesen sind. Hier ist die Abbildung ungenau, da der Fall, dass der Mengenknoten mehrere Rollen hat, durch mehrere Kanten ausgedrückt würde. Die Rollenliste repräsentiert in diesem Fall die Gesamtheit dieser Kanten.

7 Ergebnisse

Das Ergebniskapitel behandelt zuerst die Evaluierung der Arbeit und gibt dann einen kurzen Ausblick auf zukünftigen Einsatzmöglichkeiten und Erweiterungen der Werkzeuge.

7.1 Evaluierung der Werkzeuge

Das Hauptziel dieser Arbeit ist die Entwicklung eines Graphersetzungssystems, um Anforderungsdokumente in Modelle (Graphen) überführen und weiterverarbeiten zu können. Für die Evaluierung dieses Gesichtspunkts der Arbeit wird das bereits zu Beginn in der Motivation (Abschnitt 1.1) erwähnte RFC des WHOIS-Protokolls verwendet. Dazu wird der Teil des RFCs, der die Anforderungen an das Protokoll beschreibt, als Graph dargestellt und in ein vereinfachtes UML-Modell überführt.
Das zweite Ziel der Arbeit ist, das Graphersetzungssystem so zu entwickeln, dass es auch für allgemeine Graphersetzungen geeignet ist. Zur Evaluierung dieses Ziels wird, mit Hilfe der Erweiterungen von GrGen, eine Turingmaschine realisiert und ein Programm damit ausgeführt. Außerdem wird die Realisierung eines Zellularautomaten gezeigt.

7.1.1 RFC

Um einen Graphen darstellen zu können, muss immer zuerst ein passendes Graphmodell erzeugt werden. Für die Darstellung eines Textes als Graph muss ein Modell für „Sprachgraphen“ definiert werden. Dieses muss sämtliche semantische Rollen enthalten und soll zusätzlich einen Knotentyp „Wort“ enthalten. Abbildung 7.1 zeigt das Modell für die Darstellung von Texten. In der Abbildung sind, um das Modell übersichtlich zu halten, nur die ersten fünf semantischen Rollen dargestellt. Insgesamt enthält das Modell 46 semantischen Rollen (eine vollständige Liste befindet sich im Anhang 8).

```
 1  model SpracheModell;
 2
 3  node class Wort;
 4
 5  role class ACT (Wort, SUPEREDGE, SET);
 6  role class AG (Wort, SUPEREDGE, SET);
 7  role class BEN (Wort, SUPEREDGE, SET);
 8  role class CAU (Wort, SUPEREDGE, SET);
 9  role class COM (Wort, SUPEREDGE, SET);
10  ...
```

Abbildung 7.1: RFC – Sprachmodell (SpracheModell.egm)

Jede der im Sprachmodell definierten Rollen kann auf Knoten vom Typ „Wort“, auf Superkanten und auf Mengen angewendet werden.
Als Text, der dargestellt werden soll, dient die Spezifikation des WHOIS-Protokolls aus RFC 3912 [Dai04]. Der Originaltext der WHOIS-Protokollspezifikation ist in Abbildung 7.2 dargestellt. Es handelt sich dabei nur um die Protokollspezifikation, der Rest des RFCs ist nicht enthalten und soll auch nicht als Graph dargestellt werden. Es können ohne Einschränkung auch Texte, die keine Spezifikation oder Anforderungsbeschreibung sind, dargestellt werden, allerdings ist dies nicht Fokus dieser Arbeit.

```
1  A WHOIS server listens on TCP port 43 for requests from WHOIS
2  clients.  The WHOIS client makes a text request to the WHOIS server,
3  then the WHOIS server replies with text content.  All requests are
4  terminated with ASCII CR and then ASCII LF.  The response might
5  contain more than one line of text, so the presence of ASCII CR or
6  ASCII LF characters does not indicate the end of the response.  The
7  WHOIS server closes its connection as soon as the output is finished.
8  The closed TCP connection is the indication to the client that the
9  response has been received.
```

Abbildung 7.2: RFC – WHOIS-Protokollspezifikation (whois.txt)

Der Text der Protokollspezifikation muss vor der Darstellung als Graph manuell in die Form einer Grapheingabe gebracht werden. Dazu muss er mit semantischen Rollen versehen werden. Abbildung 7.3 zeigt die WHOIS-Protokollspezifikation nach einer manuellen Überführung in eine Graphdefinition[1].
Mit Hilfe des Graphmodells und der vorliegenden Grapheingabe kann die RFC-Protokollspezifikation als Graph dargestellt werden. Die Darstellung ist in Abbildung 7.4 zu sehen. Knoten vom Typ „Wort“ sind als Rechtecke (gelb) und Superkanten als Ellipsen (rot) dargestellt. Man sieht, dass der Graph zusammenhängend ist, obwohl er aus lauter einzelnen Superkanten besteht. Außerdem ist zu sehen, dass an mehreren Stellen im Graph „echte“ Superkanten, also Kanten, die direkt andere Kanten verbinden, auftreten.
Nachdem die WHOIS-Protokollspezifikation erfolgreich als Modell (Graph) dargestellt wurde, erfolgt im zweiten Schritt die Weiterverarbeitung in ein UML-Modell.
Für die Weiterverarbeitung muss zuerst ein Graphmodell für UML-Modelle (Graphen) definiert werden. Dazu wird ein vereinfachtes UML-Modell verwendet. Die Entwicklung eines vollständigen UML-Modells ist nicht Gegenstand dieser Arbeit. In Abbildung 7.5 ist ein Auszug aus dem Graphmodell für die Erzeugung von UML-Modellen dargestellt. Der Auszug aus dem Modell zeigt die Definition des Knotentyps „Klasse“, sowie des Rollentyps „Endpunkt“ und des Superkantentyps „Assoziation“. Außerdem die beiden Rollentypen „Eltern“ und „Kind“, die für die Darstellung von Vererbungsbeziehungen benötigt werden. Alle Rollen lassen als Endpunkt den Typ „Klasse“ zu. Eine „Assoziation“ muss mindestens ein Tentakel mit der Rolle „Endpunkt“ haben.

Das bereits transformierte UML-Modell ist in Abbildung 7.6 zu sehen.

[1]Die Überführung ist nicht eindeutig, da die Zuordnung von semantischen Rollen leider nicht eindeutig

```
1  auto node Wort, superedge SUPEREDGE;
2  force roles;
3
4  [#A WHOIS_server|{AG,RECP} listen|ACT #on TCP_port_43|LOC #for
5    [^request|OPUS #from WHOIS_client|CREA]|PAT
6  ]
7  [ [#The @WHOIS_client|{AG,DON} sends|ACT #a request|{HAB,PAT} #to #the
8      @WHOIS_server|{RECP} ]|PRE
9    [#The @WHOIS_server|{AG,DON} replies|ACT #with text_content|HAB ]|SUCC
10 ]
11 [#All @request|PAT terminate|ACT
12   [ASCII_CR|PRE #and ASCII_LF|SUCC]|INST
13 ]
14 [ [#The response|OMN #might #contain #more #than #one line_of_text|PAR]|CAU #so
15   [ [#the ^presence|ACT #of ASCII_CR|AG #or ASCII_LF|AG]|ACT $not indicate|ACT
16     [^end|HAB #of_the @response|POSS]|PAT
17   ]|ACT
18 ]
19 [#The @WHOIS_server|AG ^close|ACT
20   [@WHOIS_server|POSS ^connection|HAB]|ACT #as #soon #as
21   [$finished output|PAT]|TEMP
22 ]
23 [ [#The $closed @connection|FIN #is #the ^indication|FIC #to #the
24     @WHOIS_client|RECP]|!THE #that #the
25   [@response|HAB received|ACT]|THE
26 ]
```

Abbildung 7.3: RFC – WHOIS als Graphdefinition (whois.egrs)

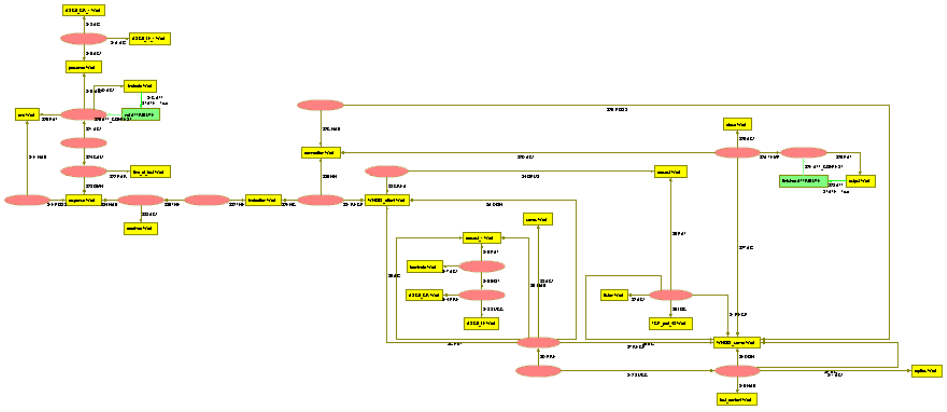


Abbildung 7.4: RFC – dargestellt als Graph

Bei den Ellipsen handelt es sich um Assoziationen oder Vererbungen. Die Knoten sind entweder Klassen, Attribute, Methoden oder Parameter.

ist.

```
1  model BilligUML;
2
3  node class Klasse {
4      Name : string;
5  }
6
7  role class Endpunkt (Klasse) {
8      Multiplizitaet : string;
9  }
10
11 superedge class Assoziation (Endpunkt+) {
12     Name : string;
13 }
14
15 role class Eltern (Klasse);
16 role class Kind (Klasse);
```

Abbildung 7.5: RFC – UML-Modell (UML.egm)

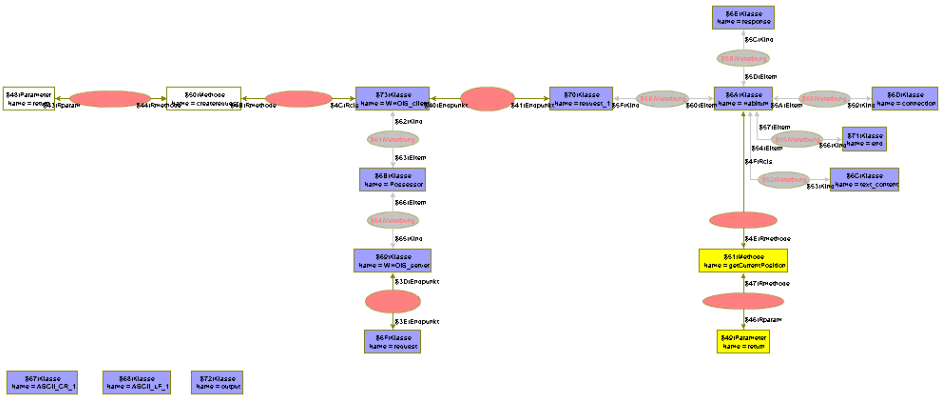


Abbildung 7.6: RFC – transformiert nach UML

Um dieses UML-Modell aus dem RFC-Modell zu erzeugen sind entsprechende Regeln notwendig. In Abbildung 7.7 sind zwei der Transformationsregeln dargestellt. Momentan umfasst die Transformation ungefähr ein Dutzend Regeln.
Die Regeldefinitionsdatei bindet sowohl das Sprachmodell als auch das UML-Modell ein, da die Transformation zwischen den Beiden erfolgen soll. Die erste Regel „DON2class“ sucht einen Knoten vom Typ „Wort“, der die Rolle „DON“ hat (im RFC-Modell) und erzeugt dann einen Knoten vom Typ „Klasse“. Die Klasse erhält den Namen des Wortes. Eine negative Anwendungsbedingung verhindert mit Hilfe des Namensattributs, dass die Regel mehrfach auf das gleiche Wort ausgeführt wird.
Die zweite Regeln „DONinh“ erstellt zwischen einer Klasse (die aus einem Wort mit der Rolle „DON“ erzeugt wurde) und einer Klasse „Possessor“ eine Vererbungsbeziehung. Auch hier gibt es wieder eine negative Anwendungsbedingung, die eine mehrfache

```
1  actions ZuBilligUML using SpracheModell, BilligUML;
2
3  rule DON2class {
4          pattern {
5                  x:Wort|DON;
6                  negative {
7                      z:Klasse;
8                      if { z.Name == x.NAME; }
9                  }
10         }
11         replace {
12                 x|DON;
13                 y:Klasse;
14         }
15         eval {
16             y.Name = x.NAME;
17         }
18 }
19
20 rule DONinh {
21         pattern {
22                 x:Wort|DON;
23                 y:Klasse;
24                 if { y.Name == x.NAME; }
25                 iPoss:Klasse;
26                 if { iPoss.Name == "Possessor"; }
27                 negative {
28                         v:Vererbung[y|Kind iPoss|Eltern];
29                 }
30         }
31         replace {
32                 x|DON;
33                 e:Vererbung[y|Kind iPoss|Eltern];
34         }
35 }
```

Abbildung 7.7: RFC – UML-Transformationsregeln (ZuUML.egrg)

Ausführung der Regel mit den gleichen Klassen verhindert.

Die Übersetzung in UML ist nicht vollständig. Bisher wurden nur einige exemplarische Regeln realisiert, um die Eignung des Graphersetzungssystems zu zeigen.

7.1.2 Turingmaschine

Die Realisierung einer Turingmaschine dient der Evaluierung des zweiten Ziels dieser Arbeit. Nachdem bereits die Eignung des Graphersetzungssystems für die Darstellung und Weiterverarbeitung von Texten gezeigt wurde, wird in diesem Abschnitt die Eignung als universelles Graphersetzungssystem gezeigt.
Eine Turingmaschine besteht aus einem Eingabeband mit unendlich vielen Bandpositionen und einem Bandzeiger, der auf die nächste zu verarbeitende Bandpostion zeigt.
Für die Realisierung mit Hilfe von Superkanten wird hier das gesamte Eingabeband als eine einzige Superkante modelliert. Die einzelnen Bandpositionen werden als Tentakel und der Bandzeiger als Rolle modelliert. Abbildung 7.8 zeigt ein Beispiel eines Eingabebands.

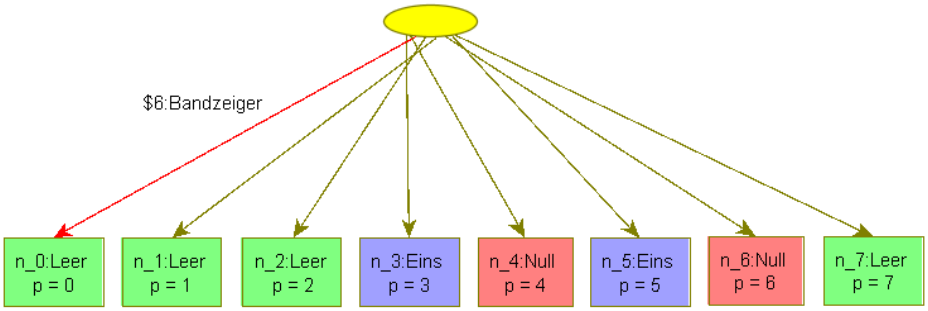


Abbildung 7.8: Turingmaschine – Eingabeband

Das Graphmodell, das dem Graphen zugrunde liegt, ist in Abbildung 7.9 dargestellt. Die verschiedenen Zustände des Eingabebands werden durch Untertypen von „Bandposition“ realisiert. Die Typen „Leer“, „Null“ und „Eins“ verkörpern jeweils die entsprechenden Eingaben der Turingmaschine. Die Rolle „Bandzeiger“ ist im Beispiel dem ersten Tentakel zugewiesen. Da die Tentakel einer Superkante keine Ordnung besitzen, diese für die Reihenfolge der Bandpostionen aber wichtig ist, wird die Reihenfolge durch das Attribut „p“ hergestellt.
Eine Turingmaschine kennt nur drei Operationen: vom Band lesen, auf das Band schreiben und den Bandzeiger bewegen. Die Lese- und Schreiboperationen müssen für alle Zeichen, des von der Turingmaschine verwendeten Alphabets definiert sein. Der Bandzeigers kann sich nach links oder rechts bewegen. Die Umsetzung der Operation in Graphersetzungsregeln ist in Abbildung 7.10 dargestellt (die Abbildung enthält nur zwei der insgesamt elf Regel).
Die Regel „EinsNull“ liest eine Eins vom Band und schreibt eine Null zurück. Sie ist anwendbar, wenn die Rolle „Bandzeiger“ einem Tentakel vom Typ „Eins“ zugewiesen ist. Um die Null zu schreiben, wird der Knoten retypisiert. Der Bandzeiger wird nicht verändert.
Die Regel „Rechts“ bewegt den Bandzeiger eine Position nach rechts. Dazu werden zwei nebeneinanderliegende Tentakel gesucht und der Bandzeiger vom linken auf das rechte

```
1  model TuringModell;
2
3  node class Bandposition {
4          p: int;
5  }
6
7  node class Leer extends Bandposition;
8  node class Null extends Bandposition;
9  node class Eins extends Bandposition;
10
11 role class Bandzeiger(Bandposition);
12
13 superedge class Band;
```

Abbildung 7.9: Turingmaschine – Modell

```
1  actions Turing using TuringModell;
2
3  rule EinsNull {
4          pattern {
5                  x:Eins|Bandzeiger;
6          }
7          replace {
8                  x|Bandzeiger;
9                  y:Null<x>;
10         }
11 }
12
13 rule Rechts {
14         pattern {
15                 n[x:Bandposition|Bandzeiger y:Bandposition];
16                 if { y.p == x.p+1; }
17         }
18         replace {
19                 n[x y|Bandzeiger];
20         }
21 }
```

Abbildung 7.10: Turingmaschine – Regeln

Tentakel übertragen. Voraussetzung ist, dass der Bandzeiger auf das Tentakel mit dem kleineren Attributwert zeigt.
Um die Funktion der Turingmaschine nachzuweisen, wird ein Programm zur Berechnung des Zweierkomplements verwendet. Abbildung 7.11 zeigt die Reihenfolge der Regeln, die das Programm ausdrücken. Das Band wird von links nach rechts durchlaufen, wobei alle Bandpositionen invertiert werden. Danach wird von rechts nach links eine Eins addiert. Das in Abbildung 7.8 dargestellte Eingabeband wird durch die Grapheingabe in Abbildung 7.12 erzeugt. Auf dem Eingabeband ist die Zahl elf kodiert.

```
1  grs (LeerLeer & Rechts)* ; ( EinsNull & Rechts | NullEins & Rechts )*
2  grs (LeerLeer & Links)* ; (EinsEins & Links)* ; (NullEins & Links)
3  grs (NullNull & Links | EinsEins & Links)* ; (LeerLeer & Links)*
```

Abbildung 7.11: Turingmaschine – Programm

```
1  :Band[
2          %p=0 :Leer|Bandzeiger
3          %p=1 :Leer
4          %p=2 :Leer
5          %p=3 :Eins
6          %p=4 :Null
7          %p=5 :Eins
8          %p=6 :Null
9          %p=7 :Leer
10 ]
```

Abbildung 7.12: Turingmaschine – Eingabe

7.1.3 Zellularautomat

Das folgende Beispiel zeigt, wie man mit Superkanten sehr elegant die Felder eines Zellularautomaten modellieren kann:
Ein Zellularautomat besteht aus einfachen Regeln, die auf Felder angewendet werden. Die Felder sind meist in einem Quadrat angeordnet. Alle Felder zusammen werden als „Raum“ des Zellularautomaten bezeichnet. Die Regeln ändern den Zustand der Felder in Abhängigkeit vom aktuellen Zustand. Zellularautomaten werden hauptsächlich in der Simulation eingesetzt.
Abbildung 7.13 zeigt das Modell für die Darstellung des Raums eines Zellularautomaten.

```
1  model AmeiseModell;
2
3  node class Rand;
4
5  superedge class Feld;
6  superedge class Schwarz extends Feld;
7  superedge class Weiss extends Feld;
```

Abbildung 7.13: Zellularautomat – Modell

Die einzelnen Felder des Zellularautomaten werden hier nicht als Knoten, sondern als Superkanten modelliert. Für die Felder wird der Typ „Feld“ definiert und zur Modellierung der Zustände die beiden Typen „Schwarz“ und „Weiss“ abgeleitet. Abbildung 7.14 zeigt das Beispiel eines Raums mit fünf mal fünf Feldern.
Weiße und schwarze Ellipsen stellen die entsprechenden Felder dar. Als Grapheingabe werden nur die Zellen auf der Diagonalen von unten nach links eingegeben. Die Graheingabe ist nachfolgend dargestellt:

```
Z7:Weiss[Z8:Schwarz[Z9:Weiss[Z10:Weiss[Z11:Weiss]]]]
```

Die Superkante „Z7“ entspricht in der Abbildung 7.14 dem untersten Feld. Die Superkante „Z8“ entspricht dann dem links, leicht oberhalb liegenden, schwarzen Feld. Jedes Feld hat genau einen Verweis auf das nächste Feld (in der Reihenfolge der Erzeugung). Die innerste Superkante „Z11“ hat zu Beginn kein Tentakel.
Die Erzeugung des vollständigen Raums erfolgt mit Hilfe von Regel. In Abbildung 7.15 ist als Beispiel die Regel zur Erzeugung der „Rückwärtskante“’ zwischen zwei Feldern abgegeben.

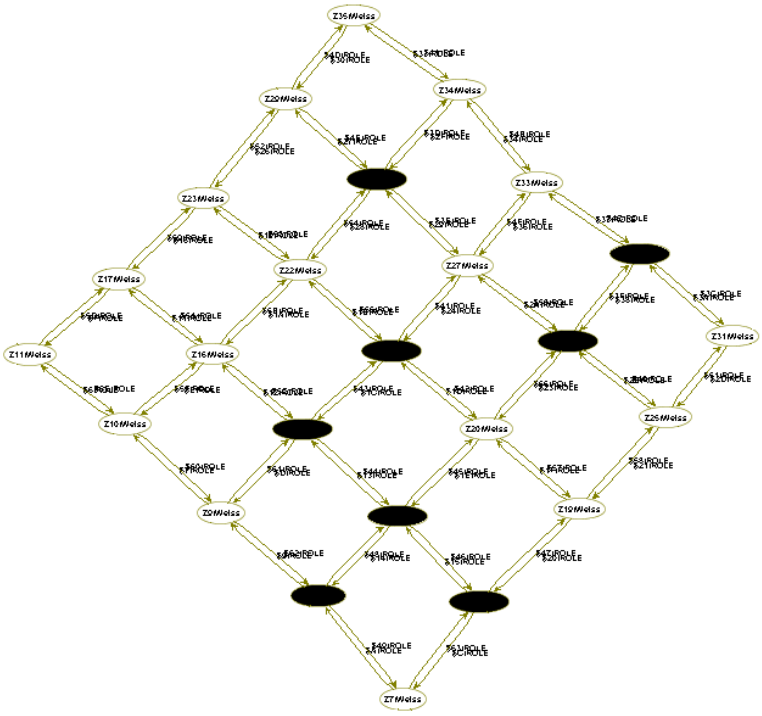


Abbildung 7.14: Zellularautomat – Raum

```
1   rule Felder {
2           pattern {
3                   x:Feld[y:Feld];
4                   negative {
5                           y[x];
6                   }
7           }
8           replace {
9                   x[y];
10                  y[x];
11          }
12  }
```

Abbildung 7.15: Zellularautomat – Regel

7.2 Ausblick

In den vorherigen Abschnitten wurde gezeigt, dass die Erweiterung von GrGen sowohl für die Darstellung und Weiterverarbeitung von Texten, als auch für universelle Graphersetzung geeignet ist.
Sicherlich gibt es neben Turingmaschinen und Zellularautomaten weitere sinnvolle Einsatzzwecke für ein Graphersetzungssystem mit Superkanten. Diese werden sich aber vermutlich erst im Laufe der Zeit zeigen.

Im Bezug auf die Darstellung und Weiterverarbeitung von Texten ist zu erwarten, dass durch die kontinuierliche Weiterentwicklung der Idee, aus Anforderungsdokumenten automatisch Softwaremodelle zu erzeugen, zusätzliche Anforderungen an das Graphersetzungssystem gestellt werden. Das System wurde anhand des RFC-Beispiels getestet, mit besonderem Blick auf die Erhaltung der Semantik des Textes. So sind mit Superkanten, Rollen, Mengen und dynamischen Attributen alle wichtigen Teile eines Satzes im Graph darstellbar. Trotzdem ist es möglich, dass für zukünftige Zwecke weitere Graphelemente notwendig sind.

Änderungen an GrGen, das kontinuierlich weiterentwickelt wird, können ebenfalls Änderungen an der Erweiterung notwendig machen. So ist zum Beispiel von den Entwicklern von GrGen geplant, die Möglichkeit, in Regeln anonyme Kanten zu definieren, zu entfernen. In der Erweiterung verwenden sowohl der Modellübersetzer als auch der Regelübersetzer anonyme Kanten.

Das RFC-Beispiel liefert als Ergebnis einen zusammenhängenden Graphen. Es ist allerdings unklar, ob ein „korrekter“ Spezifikationstext, der einen in sich geschlossenen Sachverhalt beschreibt, immer einen zusammenhängenden Graph ergibt. Im Umkehrschluss ist die Frage interessant, ob aus einem nicht zusammenhängenden Graph auf mögliche „Schwächen“ im Text geschlossen werden kann? Diese Frage kann eventuell in nachfolgenden Arbeiten in diesem Themengebiert geklärt werden.

8 Zusammenfassung

Ausgehend von der Aufgabenstellung, Anforderungsdokumente als Graph darzustellen und mittels Graphersetzung weiterzuverarbeiten, habe ich in dieser Arbeit das Graphersetzungssystem GrGen erweitert. Die besondere Herausforderung dieser Aufgabenstellung war die Notwendigkeit, das Kantenkonzept der Graphentheorie zu erweitern. Denn für die Darstellung von Texten als Graph sind Superkanten notwendig.

Die Arbeit begann mit einer ausführlichen Literaturrecherche (Kapitel 3), um mögliche Hinweise auf Superkanten bzw. ähnliche Konzepte zu finden. Es konnten aber keine Arbeiten gefunden werden, die sich mit Kanten, die andere Kanten verbinden, beschäftigen. Somit stand das weitere Vorgehen fest: Entwicklung eines Konzepts für Superkanten und Umsetzung in einem Graphersetzungssystem.

Um den Aufwand der Arbeit zu begrenzen, sollte ein bestehendes Graphersetzungssystem als Grundlage für die Umsetzung verwendet werden. Dazu habe ich sieben verschiedene Graphersetzungssysteme, auf ihre Eignung als Grundlage der Erweiterung bewertet (Kapitel 3.3). Die Auswahl fiel auf GrGen. Da die Erweiterung um Superkanten grundlegende Änderungen an der internen Struktur von GrGen notwendig gemacht hätte, habe ich mich entschieden mehrere Vorübersetzer zu entwickeln. Diese übersetzen die vom Anwender, in der um Superkanten erweiterten Syntax (Kapitel 5), definierten Artefakte in GrGen-Syntax. Die Erweiterungen werden also intern, vom Anwender verborgen, vollständig auf die von GrGen unterstützen Graphelemente (Knoten, Kanten, Typen und Attribute) zurückgeführt (Kapitel 4.2).

Für die Übersetzung von Graphmodellen in erweiterter Syntax in Graphmodelle in GrGen-Syntax, entstand bei einem Gespräch mit meinem Betreuer und dem verantwortlichen Entwickler von grGen, die Idee, diese durch Graphtransformation zu realisieren. Hintergrund war der Wunsch, die theoretisch mögliche Realisierung eines Übersetzers mittels Graphersetzung durch eine Implementierung praktisch zu zeigen. Der Übersetzer hat dieses Ziel vollständig erfüllt, wenngleich er einige noch bestehende Schwächen dieser Methode aufgezeigt hat. Für die Übersetzung von Regeldefinitionen und Grapheingaben habe ich den konventionellen Weg, die Realisierung mittels imperativer Programmierung, gewählt (Kapitel 6).

Für die Evaluierung (Kapitel 7) der Erweiterungen um Superkanten, wurde ein Anforderungsdokument vollständig in einen Graphen überführt und weiterverarbeitet. Die Weiterverarbeitung wurde nur in Teilen umgesetzt, da im Rahmen dieser Arbeit nur die grundsätzliche Möglichkeit der Weiterverarbeitung realisiert werden sollte. Sowohl bei

der Darstellung des Dokumentes als Text als auch bei der Weiterverarbeitung haben die Erweiterungen alle an sie gestellten Anforderungen erfüllt.
Um neben der Darstellung von Texten auch die Eignung als universelles Graphersetzungssystem mit Superkanten zu zeigen, habe ich eine Turingmaschine und einen Zellularautomaten realisiert, die Superkanten verwenden. Auch dieses Ziel, ein universelles-Graphersetzungssystemm mit Superkanten zu entwickeln, wurde erreicht.

Anhang

A – Nutzungsanleitungen der Werkzeuge

In diesem Abschnitt wird zuerst die Visualisierung von Anforderungsdokumenten, sowie die Änderung des dafür notwendigen „Sprachmodells“ beschrieben. Danach folgt die Beschreibung der einzelnen Werkzeuge.

Visualisierung von Anforderungsdokumenten

Um Anforderungsdokumente als Graphen zu visualisieren, existiert bereits ein passendes Modell für Sprachgraphen („SpracheModell.egm“). Vor der ersten Nutzung bzw. nach einer Änderung des Modells, muss es in die von der Ausführungsumgebung benötigten Bibliotheken übersetzt werden.

Vorbereitung Für die einmalige Erzeugung der Bibliotheken müssen die folgenden Schritte ausgeführt werden:

1. Übersetzen des erweiterten Graphmodells in ein GrGen-Graphmodell.

   ```
   grModelTrans.bat SpracheModell.egm SpracheModell.gm
   ```

2. Übersetzen der erweiterten Regeldefinition in eine GrGen-Regeldefinition. Die Regeldefinitionsdatei enthält lediglich eine Regelattrappe, da der GrGen-Übersetzer immer eine Regeldefinitionsdatei mit mindestens einer Regel erwartet. Für die Visualisierung von Texten sind keine Ersetzungsregeln notwendig, deshalb wird die Attrappe verwendet.

   ```
   grRuleTrans.bat Sprache.egrg Sprache.grg
   ```

3. Übersetzen des GrGen-Graphmodells in GrGen-Bibliotheken.

   ```
   grgen.exe Sprache.grg
   ```

Visualisierung Wurden die Bibliotheken erzeugt, kann ein Anforderungsdokument mit den folgenden Schritten visualisiert werden. Als Beispiel soll ein Anforderungsdokument mit dem Namen „whois.txt“ dienen.

1. Das Anforderungsdokument muss in die erweiterte Graphdefinitionssyntax überführt werden. Dazu müssen die einzelnen Sätze manuell in die Form von Graphdefinitionen gebracht werden. Danach müssen den einzelnen Wörtern und Teilsätzen

semantische Rollen zugeordnet werden (eventuell müssen Kopfelemente in Teilsätzen ausgezeichnet werden). Die erweiterte Graphdefinition wird als „whois.egrs“ gespeichert.

Beispiel:

> A WHOIS server listens on TCP port 43 for a request from a WHOIS client.

Erweiterte Graphdefinition:

> [#A WHOIS_server listen #on TCP_port_43 #for [request #from WHOIS_client]]

Rollenzuordnung:

> [#A WHOIS_server |ACT listen #on TCP_port_43 |LOC #for [ˆrequest |HAB #from WHOIS_client |DON]|ACT]

2. Die erweiterte Graphdefinitionsdatei „Beispiel.egrs“ muss in die GrGen-Graphdefinition „Beispiel.grs“ übersetzt werden:

   ```
   grGraphTrans.bat whois.egrs whois.grs
   ```

3. Mit der Ausführungsumgebung (*GrShell*) wird die Graphdefinition visualisiert. Dazu wird „grShell.exe“ gestartet. In der Ausführungsumgebung wird dann zuerst mittels `include start.txt` ein Skript geladen, das einige Initialisierungsarbeiten ausführt und das Graphmodell lädt. Dieses Skript ist bereits vorhanden und muss nicht vom Anwender erzeugt werden. Danach wird mittels `include whois.grs` die Graphdefinition geladen und der zugehörige Graph im Speicher aufgebaut. Mit `show graph ycomp` wechselt man in den Visualisierungsmodus und kann den Graph betrachten. Durch `validate strict` kann überprüft werden, ob der Graph dem Sprachmodell genügt.

   ```
   grShell.exe
   > include start.txt
   > include whois.grs
   > show graph ycomp
   > validate strict
   ```

„Sprachmodell“ ändern Sind Änderungen am Modell für die Darstellung von Texten notwendig, so müssen diese in der Datei „SpracheModell.egm“ vorgenommen werden. Danach muss der Schritt „Vorbereitung“ erneut ausgeführt werden, damit die Bibliotheken aktualisiert werden.
Änderungen am Sprachmodell sind zum Beispiel notwendig, wenn neue Rollen hinzugefügt werden sollen.

Beschreibung der einzelnen Werkzeuge

Jeder der drei Übersetzer für die erweiterte Syntax (Modelle, Graphen und Regeln) wird einzeln mit allen Aufrufparametern erklärt.

Modellübersetzer

Der Modellübersetzer *grModelTrans* dient zur Übersetzung von Modellen in erweiterter Syntax in Modelle in GrGen-Syntax. Der Aufruf erfolgt mit einem oder zwei Parametern. Der erste Parameter muss immer der Name, des zu übersetzenden Modells (in erweiterter Syntax) sein, der zweite Parameter kann der Name der Ausgabedatei (übersetztes Modell in GrGen-Syntax) sein. Ist kein Name für die Ausgabedatei angegeben, so wird der Name der Eingabedatei um die Erweiterung „.gm“ ergänzt und als Ausgabedatei verwendet.

```
grModelTrans.bat Beispiel.egm Beispiel.gm
```

Übersetzer für Grapheingabe

Der Übersetzer für Grapheingaben *grGraphTrans* dient zur Übersetzung von Grapheingaben in erweiterter Syntax in Grapheingaben in GrGen-Syntax. Der Aufruf erfolgt mit einem oder zwei Parametern. Der erste Parameter muss immer der Name des zu übersetzenden Graphen (in erweiterter Syntax) sein, der zweite Parameter kann der Name der Ausgabedatei (übersetzter Graph in GrGen-Syntax) sein. Ist kein Name für die Ausgabedatei angegeben, so wird der Name der Eingabedatei um die Erweiterung „.grs“ ergänzt und als Name für die Ausgabedatei verwendet (erkennt der Übersetzer die Dateiendung „.egrs“, so wird diese durch „.grs“ ersetzt und nicht damit ergänzt).

```
grGraphTrans.bat Beispiel.egrs Beispiel.grs
```

Regelübersetzer

Der Übersetzer für Regeldefinitionen *grRuleTrans* dient zur Übersetzung von Regeldefinitionen in erweiterter Syntax in Regeldefinitionen in GrGen-Syntax. Der Aufruf erfolgt mit einem oder zwei Parametern. Der erste Parameter muss immer der Name der zu übersetzenden Regeldefinitionen (in erweiterter Syntax) sein, der zweite Parameter kann der Name der Ausgabedatei (übersetzte Regeldefinitionen in GrGen-Syntax) sein. Ist kein Name für die Ausgabedatei angegeben, so wird der Name der Eingabedatei um die Erweiterung „.grg“ ergänzt und als Name für die Ausgabedatei verwendet (erkennt der Übersetzer die Dateiendung „.egrg“, so wird diese durch „.grg“ ersetzt und nicht damit ergänzt).

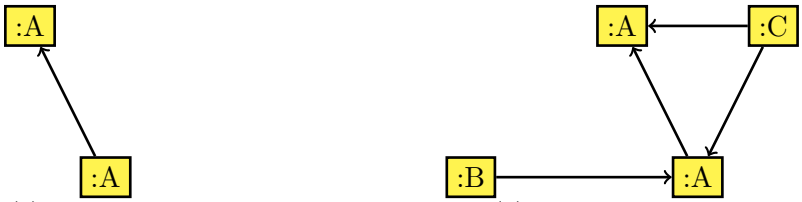
```
grRuleTrans.bat Beispiel.egrg Beispiel.grg
```

B – Imperative Programmierung vs. Graphersetzung

Dieser Abschnitt schildert meine persönliche Erfahrung im Vergleich von imperativer Programmierung und Graphersetzung. Dabei werden die drei im Rahmen dieser Arbeit entwickelten Übersetzer verglichen: *grModelTrans* (Graphersetzung), *grGraphTrans* und *grRuleTrans* (jeweils imperative Programmierung).

Eine quantitative Aussage zwischen den beiden Methoden ist nicht möglich, da ich ca. zehn Jahre Erfahrung in Verwendung imperativer Programmierung habe, mich während der sechs Monate meiner Diplomarbeit aber zum ersten mal mit Graphersetzung beschäftigt habe. Dadurch sind quantitative Aussagen, wie z.B. die Entwicklungszeit der einzelnen Übersetzer nicht vergleichbar. Außerdem hatten die drei Übersetzer unterschiedliche Komplexität. Die Entwicklung der imperativen Übersetzer wurde erschwert, da ich zuvor keine Erfahrung im Umgang mit ANTLR hatte. Der durch Graphersetzung realisierte Modellübersetzer benötigt zwar auch eine Grammatik, allerdings enthält diese lediglich die reinen Produktionen, während die Grammatiken für die imperativen Übersetzter gleichzeitig große Teile der Programmlogik in Form von Javacode, enthalten.
Durch die gründliche Literaturrecherche waren mir bei Beginn der Entwicklung alle wichtigen Konzepte der Graphersetzung bekannt: Typen, Attribute, negative Anwendungsbedingungen, Typzusicherungen, Attributzusicherungen, SPO-Semantik, etc. Außerdem hatte ich bei der Evaluierung verschiedener Graphersetzungssystem erste Erfahrungen bei der Definition von Graphersetzungsregeln gesammelt.

Die ersten Schritte bei der Definition von Graphersetzungsregeln haben sich sehr schwierig gestaltet. Bei den ersten Regeln musste ich mir die linke und rechte Seite der Ersetzung auf Papier skizzieren, erst mit einiger Übung war ich in der Lage, Regeln in Gedanken zu entwerfen.
Ein weiterer deutlicher Lerneffekt war bei der Verwendung von „Entwurfsmustern“ zu beobachten. Eine immer wiederkehrende Aufgabe beim Entwurf von Ersetzungsregeln ist die Frage, wie groß der Kontext der linken Seite sein muss, damit das gesuchte Graphelement eindeutig beschrieben ist. Das Beispiel in Abbildung 8.1 zeigt die Suche eines Musters ohne und mit Kontext.



(a) Muster (b) Muster mit Kontext

Abbildung 8.1: Anhang – Suche eines Musters

Das in Teil (a) der Abbildung beschriebene Muster, ist das gesuchte Muster. Ohne wei-

teren Kontext wird es immer gefunden, sobald im Wirtsgraphen zwei Knoten vom Typ „A“ auftreten und durch eine Kante verbunden sind. Im Teil (b) ist das Muster zusammen mit einem Kontext, der aus zwei Elementen vom Typ „B“ und „C“ besteht, zu sehen. Dieses Muster wird im Wirtsgraphen nur gefunden, wenn auch die Kontextelemente existieren. Zu Beginn war es schwierig, bei der Definition einer Regel, den Kontext eines Musters richtig zu wählen. Nach einiger Übung war dies jedoch kein Problem mehr.

Ein anderes, beim Entwurf von Regeln oft auftretendes Problem, ist die Wiederholung von bereits verarbeiteten Graphelementen. Die Regel in Abbildung 8.2 zeigt dieses Problem.

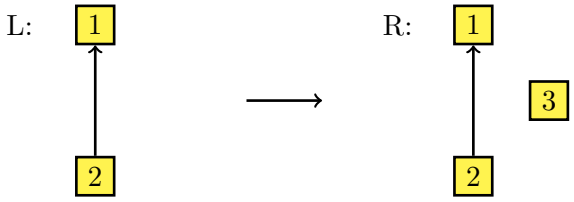


Abbildung 8.2: Anhang – beliebig oft anwendbare Regel

Das Muster der linken Seite kann immer wieder auf die gleiche Stelle im Wirtsgraph angewendet werden und erzeugt jedes mal einen neuen Knoten. Um dies zu verhindern gibt es drei mögliche Lösungsmuster.

1. Modifikation des Musters der linken Seite durch die Regel: Entweder wird ein Teil des Musters der linken Seite durch die Regel gelöscht oder die Regel fügt dem Muster mindestens eine Kante hinzu. In Verbindung mit einer negativen Anwendungsbedingung, wird das Muster so nach der einmaligen Ausführung der Regel (Hinzufügen der Kante) nicht wieder gefunden.

2. Markierung des Musters: Die Regel fügt dem Muster eine Markierung in Form eines speziellen Knoten- oder Kantentyps hinzu. Dieser Typ ist nicht Teil des Graphmodells, sondern wird speziell für die Transformation definiert. Am Ende der Transformation wird die Markierung wieder aus dem Graphen entfernt. Diese Lösung ist der Ersten ähnlich, macht die Markierung aber explizit (durch den speziellen Typ) und für den Leser der Regeldefinitionen leichter verständlich.

3. Parallele Ausführung der Regel: Statt eine Regel hintereinander auf alle Übereinstimmungen im Graph auszuführen, kann dies auch parallel erfolgen. Dadurch wird verhindert, dass die Regel auf die gleiche Übereinstimmung mehrmals angewendet wird. Allerdings funktioniert die parallele Ausführung nur, wenn die Regel keine Seiteneffekte hat: Die Übereinstimmungen im Graph dürfen sich nicht Überschneiden, bzw. bei Überschneidungen, dürfen die betroffenen Graphelemente durch die Regel nicht verändert werden.

Obwohl Graphersetzung aus theoretischer Sicht stark mit funktionaler Programmierung verwandt ist, unterscheidet sich die praktische Anwendung sehr. Das, zum Beispiel aus

Haskell bekannte, Listenkonzept ist im Gegensatz zu Graphen „eindimensional“ (auch bei geschachtelte Listen handelt es sich nach wie vor um eindimensionale Listen). Graphen dagegen sind „multidimensional“, ein Graphelement hat nicht nur einen linken und rechten Listennachbar, sondern beliebig viele Nachbarn.
Der Bezug zwischen Graphersetzung und logischer Programmierung ist dagegen deutlich größer. Die Definition von Aussagen und Ableitungen, wie sie zum Beispiel in Prolog erfolgt, ist ähnlich zur Definition von Graphmustern und Ersetzungsregeln.

Vorteile Graphersetzung

Programmierung durch Graphersetzung zwingt den Entwickler, sich vor Beginn der Entwicklung gründlich Gedanken über die Struktur seiner Lösung zu machen. Unüberlegtes Vorgehen führt schnell in eine Sackgasse, da das, aus der imperativen Programmierung bekannte, Konstrukt – `IF Ausnahmefall THEN ...` – jeweils als eine eigene Regel realisiert werden muss.
Das oberste Ziel beim Entwurf ist es deshalb, eine möglichst allgemeine Lösung zu finden, die nur wenige Ausnahmefälle behandeln muss. Stellt der Entwickler im Nachhinein fest, dass in einer Lösung zu viele Ausnahmefälle auftreten, muss die Lösung grundlegend überarbeitet werden. Die Ausnahmen durch eine entsprechend große Zahl von Regeln zu behandeln, ist oft nicht möglich, da keine Reihenfolge für die Ausführung dieser Regeln gefunden werden kann, so dass am Ende tatsächlich alle Ausnahmen behandelt wurden.

Ein weiterer Vorteil ist, meiner Meinung nach, dass Regeldefinitionen leichter zu warten sind als, imperativer Quellcode. Bei der Graphersetzung sind Ablauf und Logik strikt von einander getrennt. Der Ablauf wird durch die Ausführungsreihenfolge der Regeln bestimmt, während die Logik in den Regeln selbst enthalten ist. In imperativen Programmen sind diese beidenKomponentenn oft stark vermischt.
Bei der Wartung von Graphersetzungsregeln, kann eine zu ändernde Stelle zuerst in der Ausführungsreihenfolge lokalisiert und dann die entsprechenden Regeln geändert werden. Jede Regel bildet für sich eine verständliche Einheit, da das Muster der linken Seite gleichzeitig den Kontext der Regel angibt. In der imperativen Programmierung ist bei einer Methode nicht immer klar, welchen Zweck sie hat und in welchem Zusammenhang sie steht. Durch gute Strukturierung, Bezeichnerwahl und Dokumentation können imperative Programmtexte zwar ebenfalls verständlich gemacht werden, allerdings ist dies vom Entwickler abhängig. Der Kontext von Regeln ist durch die linke Seite hingegen automatisch vorhanden.

Nachteile Graphersetzung

Graphersetzung ist für alle Anwendungen, die Interaktion mit der Umwelt (Anwender, Systemdienste, etc.) voraussetzten, nicht geeignet. Sie kann nur bei Anwendungen, die ohne Interaktion mit dem Anwender eine Eingabedatenstruktur in eine Ausgabedatenstruktur überführen, eingesetzt werden.

Für die Realisierung eines Übersetzers ist Graphersetzung deshalb grundsätzlich geeignet. Allerdings wäre auch bei einem Übersetzer eine Möglichkeit wichtig, Fehler direkt an den Anwender ausgeben zu können. Durch reine Graphersetzung besteht nur die Möglichkeit, einen speziellen Fehlerknoten zu erzeugen und diesen dann in die Ausgabdatei zu schreiben. Direkte Ausgabe auf der Konsole (im Fall von GrGen in der Ausführungsumgebung) ist nicht möglich.

Ein Problem, das nicht grundsätzlich für die Verwendung von Graphersetzung ist, sondern in diesem konkreten Fall auftritt, ist die Unzulänglichkeit des Ausgabemoduls *grIO*. Das Zerteilen der Eingabedatei in einen abstrakten Syntaxbaum wird von *ASTdapter* automatisch ausgeführt, während für die Ausgabe mit *grIO* erst der abstrakten Syntaxbaum in einen *grIO*-Graph transformiert werden muss. Hier wäre ein Werkzeug, das analog zu *ASTdapter*, mit Hilfe einer Grammatik einen abstrakten Syntaxbaum wieder in eine Datei schreibt, wünschenswert.

C – Liste aller Semantischen Rollen

Das am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelte Rollensystem besteht zur Zeit aus 46 Rollen. Die Liste in Abbildung 8.3 zeigt diese Rollen. Das Rollensystem wurde in einer Studienarbeit evaluiert [Bru07], trotzdem befindet es sich in permanenter Weiterentwicklung. Es könnten also in Zukunft sowohl weitere Rollen hinzukommen als auch Rollen entfallen.
Die Abbildung ist [Bru07] entnommen, wo sie auch ausführlich erklärt wird. Sie wird hier nur zum Nutzen des Lesers abgebildet.

Thematische Rollen

Kürzel	Name	Beschreibung	siehe auch
ACT	actus	eine Handlung, die von einer Person oder Sache ausgeführt wird	AG, PAT
AG	agens	der Handelnde, der Aktive; Person oder Sache, die eine Handlung ausführt	ACT, PAT
BEN	beneficiens*	der von einer Handlung Profitierende (+); Person oder Sache, zu deren Vorteil (+) oder Nachteil (-) eine Handlung ausgeführt wird	FAU, FAV
CAU	causa*	Sachverhalt, der die Ursache einer Handlung darstellt (+) oder trotz dessen (-) die Handlung ausgeführt wird	ACT, INT
COM	comes	der Begleiter eines Elements	DUX
COMP	comparand	das Element, mit dem verglichen wird	~~COMP~~
~~COMP~~	compariens	das Element, das verglichen wird	COMP
CONT	contrarius	der Gegner eines Elements	~~CONT~~
~~CONT~~	contrariens	das Element, das einen Gegner hat	CONT
CREA	creator*	Person oder Sache, die etwas erzeugt (+) oder zerstört (-)	OPUS
DEST	destinatio**	ein Ziel oder Endpunkt	ORIG, POS
DIM	dimensio**	das Ausmaß von etwas	LOC, TEMP
DON	donor	Person oder Sache, die eine Sache abgibt	HAB, POSS, RECP
DUX	dux	Person oder Sache, die von einer anderen Person oder Sache begleitet wird oder zusammen mit dieser auftritt	COM
EXP	experior	jemand, der etwas erfährt (z.B. durch Sinneswahrnehmung)	NOT, STIM
FAU	fautor*	Person oder Sache, die eine Handlung zum Vorteil (+) oder Nachteil (-) einer Person oder Sache ausführt	BEN, FAV
FAV	favor*	Der Vorteil (+) oder Nachteil (-) einer Person oder Sache	BEN, FAU
FIC	fictum	eine Rolle, die jemand oder etwas einnimmt	FIN
FIN	fingens	Person oder Sache, die eine Rolle einnimmt	FIC
FREQ	frequens	die Häufigkeit oder die Zeitpunkte einer Handlung	TEMP
HAB	habitum	die Habe, der Besitz; Person oder Sache, die von einer Person oder Sache besessen (auch kurzzeitig), erhalten oder weitergegeben wird	DON, POSS, RECP
INST	instrumentum*	das Hilfsmittel, mit dem (+) oder ohne das (-) eine Handlung durchgeführt wird	ACT, MOD
INT	intentio	der Zweck einer Handlung	ACT, CAU
LIM	limes	der Pfad, den etwas nimmt	LOC
LOC	locus**	ein Ort, ein Platz, eine Gegend	DIM, DST, LIM, ORIG, POS
MOD	modus	die Art und Weise, in der eine Handlung durchgeführt wird	ACT, INST
NOT	notio	der Eindruck, die Vorstellung, der Begriff, das Bild, die Idee oder die Erfahrung, die jemandem übermittelt wird; Sachverhalt, den eine Person oder Sache erfährt (z.B. durch Sinneswahrnehmung)	EXP, STIM
OMN	omnium	Person oder Sache, die ein Ganzes, das aus Teilen besteht, darstellt	PAR
OPUS	opus*	das durch eine Handlung erzeugte (+) oder zerstörte (-) Element	CREA
ORIG	origo**	eine Quelle, eine Herkunft, ein Beginn	DEST, POS
PAR	pars	ein Teil eines Ganzen	OMN
PAT	patiens	Person oder Sache, die von einer Handlung betroffen ist oder an der eine Handlung ausgeführt wird	ACT, AG
POS	positio**	die aktuelle Position (im Sinne eines festen Bezugspunkts) eines Elements (nicht nur lokal!)	DEST, ORIG
POSS	possessor	der (gegenwärtige) Besitzer eines Elements, der "Haber"	DON, HAB, RECP
QUAL	qualitas	die Beschaffenheit eines Elements	~~QUAL~~
~~QUAL~~	qualifitiens	das durch eine Beschaffenheit beschriebene Objekt	QUAL
RECP	recipient	der Empfänger eines Elements	DON, HAB, POSS
STIM	stimulus	Person oder Sache, die etwas (einen Eindruck/Bild/Idee/…) erzeugt, das eine Person oder Sache erfährt (z.B. durch Sinneswahrnehmung)	EXP, NOT
SUB	substitutus	Person oder Sache, die durch eine andere Person oder Sache ersetzt oder vertreten wird	~~SUB~~
~~SUB~~	substituens	Person oder Sache, die eine andere Person oder Sache ersetzt oder vertritt	SUB
SUM	sumptio	die Voraussetzung einer Handlung; die Annahme, unter der eine Handlung ausgeführt wird	ACT
TEMP	tempus**	eine Zeitangabe	DIM, DEST, FREQ, ORIG, POS
THE	thema	der Inhalt, der Gegenstand einer Betrachtung	~~THE~~
~~THE~~	thematiens	das Element, dessen Inhalt oder Gegenstand beschrieben wird	THE

* vorzeichenbehaftet
** reicht als alleinige Klassifizierung nicht aus

Abbildung 8.3: Anhang – Liste aller Semantischen Rollen

Stichwortverzeichnis

Literaturverzeichnis

[AGG] AGG - Attributed Graph Grammar. `http://tfs.cs.tu-berlin.de/agg/`. Fachbereich Informatik, Technischen Universität Berlin. 13, 14, 23, 27

[aS] andy Schürr. PROGRES - PROgrammed Graph REwriting Systems. `http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php?page=Research%3A+Progres`. Lehrstuhl für Informatik 3, Technische Universität Aachen (RWTH Aachen). 13

[Ass00] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, 2000. 19

[BBN99] Michel Biezunski, Martin Bryan, and Steven R. Newcomb. Topic maps: Information technology – document description and markup languages,. *ISO/IEC 13250:2000*, 1999. 4, 9

[BG07] Jakob Blomer and Rubino Geiß. The grgen user manual. Technical report, Universität Karlsruhe, IPD Goos, April 2007. 86

[Bru07] Torben Brumm. Erstellung eines Systems thematischer Rollen mit Hilfe einer Benutzbarkeitsstudie. Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, 2007. 2, 135

[BTS00] Paolo Bottoni, Gabriele Taentzer, and Andy Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 59, Washington, DC, USA, 2000. IEEE Computer Society. 28

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Application*. Addison Wesley, 2000. 16

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003. In press. 13

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. In *IBM SYSTEMS JOURNAL, Vol 45, NO 3*, 2006. In press. 13, 14, 16, 19, 23

[CM03] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer, 5. ed. edition, 2003. 41

[Dai04] L. Daigle. RFC 3912 - WHOIS protocol specification, 2004. 2, 117

[dL] Juan de Lara. AToM - A tool for meta-modelling and model-transforming. `http://moncs.cs.mcgill.ca/MSDL/research/projects/AToM3`. McGill University, School of Computer Science, Montreal, Canada. 14

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005. 28

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 5, 22

[EGdL+05] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005. 14

[FNTZ98] T. Fischer, Jörg Niere, L. Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998. 37

[Fuj] Fujaba - From UML to Java and back again. `http://www.fujaba.de/`. Software Engineering Group, Universität Paderborn. 13, 23, 37

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. Grgen: A fast spo-based graph rewriting tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations - ICGT 2006*, Lecture Notes in Computer Science, pages 383 – 397. Springer, 2006. Natal, Brasilia. 30, 82, 84, 86

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. Query / Views / Transformations submissions and recommendations towards the final standard. 2003. 15

[GRe] GReAT - Graph Rewrite and Transform System. `http://escher.isis.vanderbilt.edu/tools/get_tool?GReAT`. Institute for Software Integrated Systems, Vanderbilt University, Nashville. 13, 23, 31

[GrG] GrGen - Graph Rewrite GENerator. `http://www.info.uni-karlsruhe.de/software/grgen/`. Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe. 23, 29, 86

[GRO] GROOVE - GRaphs for Object-Oriented VErification. `http://groove.sourceforge.net/groove-index.html`. Department of Computer Science, University of Twente. 13

[Has] Haskell - A Purely Functional Language. `http://www.haskell.org/haskellwiki/Haskell`. 23

[ISI] ISIS - Institut for Software Integrated Systems, Vanderbilt University, Nashville, Tennessee. `http://repo.isis.vanderbilt.edu/tools/`. 31

[Law] Michael J Lawley. Tefkat - The EMF Transformation Engine. `http://sourceforge.net/projects/tefkat`. 20

[Mau] Maude - A language for rewriting logic. `http://maude.cs.uiuc.edu/`. Department of Computer Science, University of Illinois at Urbana-Champaign. 15

[MDD] MDD - Model driven development. `http://www.softmetaware.com/oopsla2004/mdsd-workshop.html`. 13

[MOF] MOF - MetaObject Facility. `http://www.omg.org/mof/`. 16

[MOL] MOLA - MOdel transformation LAnguage. `http://mola.mii.lu.lv/`. nstitute of Mathematics and Computer Science, University of Latvia. 13

[MvGKV05] Tom Mens, Pieter van Gorp, Gabor Karsai, and Dániel Varró. Applying a model transformation taxonomy to graph transformation technology. In G. Karsai and G. Taentzer, editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, ENTCS, 2005. In press. 13

[oAW] openArchitectureWare. `http://www.openarchitectureware.org/`. 22

[OCL] OCL - Object Constraint Language. `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14`. 17

[OMG] OMG - Object Management Group. `http://www.omg.org/`. 15, 43

[Opt] OptimalJ - Model-driven development for Java. `http://www.compuware.com/products/optimalj/default.htm`. 22

[OWL] OWL - Web Ontology Language. `http://www.w3.org/TR/owl-features/`. 23, 43

[Par] Terence Parr. An Introduction to ANTLR. `http://www.antlr.org/`. 87

[Pel] Pellet - OWL DL-Reasoner. `http://pellet.owldl.com/`. 44

[Plu98] Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998. 19

[Proa] Protégé. `http://protege.stanford.edu/`. Stanford Medical Informatics, Stanford University School of Medicine. 44

[Prob] GNU Prolog. `http://gnu-prolog.inria.fr/`. 23, 40, 41

[QVT] QVT - Query/View/Transformation. `http://www.omg.org/cgi-bin/doc?ptc/2005-11-01`. 14, 20, 23

[Roz97] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. 20

[Rul] RuleML - Rule Markup Language. `http://www.ruleml.org/`. 44

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994. 38

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. 14

[SV03] Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, San Francisco, CA, USA, October 20-24 2003. Springer. 13

[Val] Patrick Valduriez. ATL - Atlas Transformation Language. `http://www.eclipse.org/gmt/atl/`. Université de Nantes. 19

[Vara] Varlet - database reengineering project. `http://wwwcs.uni-paderborn.de/cs/varlet/`. Universität Paderborn. 13

[Varb] Dániel Varró. VIATRA - VIsual Automated model TRAnsforms. `http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/index.html`. Budapest University of Technology and Economics. 13, 14, 19, 23, 34

[Vis] Eelco Visser. Stratego - Strategies for program transformation. `http://www.program-transformation.org/Stratego`. Delft University of Technology. 18

[VMT] VMTS - Visual Modelling and Transformation System. `http://avalon.aut.bme.hu/~tihamer/research/vmts/`. Budapest University of Technology and Economics. 14

[vP85] P. von Polenz. *Deutsche Satzsemantik: Grundbegriffe des Zwischen-den-Zeilen-Lesens*. de Gruyter, Berlin, 1985. 2

[VPB+06] Dániel Varró, András Pataricza, András Balogh, András Schmidt, Dávid Vágó István Ráth, and Gergely Nyilas. VIATRA - Tutorial and release notes. `http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/doc/viatratut_October2006.pdf`, 2006. 35

[Whi02] Jon Whittle. Transformations and software modeling languages: Automating transformations in UML. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML*, volume 2460 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2002. 14

[XMI] XMI - XML Metadata Interchange. `http://www.omg.org/technology/documents/formal/xmi.htm`. 23

[XMLa] XML - Extensible Markup Language. `http://www.w3.org/XML/`. 14, 23

[XMLb] XML Schema. `http://www.w3.org/XML/Schema`. 43

[XSL] XSLT - XSLTranslations. `http://www.w3.org/TR/xslt/`. 14, 23