Ein Interpreter für Diaplan, eine regelbasierte Sprache zum Programmieren mit Graphen und Diagrammen

Diplomarbeit

von

Raimund Klein

25. November 2004

Universität Bremen · Fachbereich 3 · Studiengang Informatik

Gutachter:
Dr. Berthold Hoffmann
Prof. Dr. Hans-Jörg Kreowski

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Motivation . . . 3
1.2 DiaGen . . . 4
1.3 Ziel dieser Arbeit . . . 5
1.4 Übersicht . . . 5

2 Hypergraphen und Hypergraphtransformation **7**
2.1 Hypergraphen und ihre Bestandteile . . . 7
2.1.1 (Einfache) Knoten und Kanten . . . 8
2.1.2 Behälter . . . 8
2.1.3 Typen und Punkte . . . 9
2.1.4 Variablen . . . 9
2.1.5 Prädikatkanten . . . 10
2.2 Transformation . . . 10
2.2.1 Regeln . . . 10
2.2.2 Auswertungsalgorithmus . . . 12
2.3 Weitere Konzepte von Dia**plan** . . . 16
2.3.1 Gestaltdefinitionen . . . 17
2.3.2 Fehlbarkeit und Nichtdeterminiertheit von Prädikaten . . 18
2.3.3 Universelle Behälter und Variablen . . . 18
2.3.4 Kapselung bzw. Modularisierung . . . 19
2.3.5 Nebenläufigkeit . . . 19
2.4 Verwandte Arbeiten . . . 19
2.4.1 Andere Modelle zur Graphtransformation . . . 19
2.4.2 Andere Graphsprachen . . . 22

3 Anwendungsbeispiele **34**
3.1 Verkettete Listen . . . 34
3.1.1 Aufbau einer Liste . . . 34
3.1.2 Regeln . . . 35
3.2 Eine Arithmetik auf den Natürlichen Zahlen . . . 35
3.2.1 Daten . . . 37
3.2.2 Regeln . . . 37

4 Die DiaGen-Transformationsmaschine **41**
4.1 Repräsentation von Graphen, Knoten und Kanten . . . 41
4.1.1 Knoten . . . 41
4.1.2 Kanten . . . 43

4.1.3 Graphen . . . 45
4.2 Repräsentation von Regeln . . . 45
4.2.1 `TransformationSet` . . . 45
4.2.2 `ProdPattern` . . . 47
4.2.3 `EdgeSelector` und `SameNode` . . . 47
4.2.4 `Redex` und `PatternAction` . . . 48
4.3 Die Ansatzsuche . . . 48

5 Erweiterungen der DiaGen-Transformationsmaschine 51
5.1 Die hinzugekommenen Graphbestandteile . . . 52
5.1.1 Behälter . . . 52
5.1.2 Typen und Punkte . . . 58
5.1.3 Variablen . . . 59
5.1.4 Prädikatkanten . . . 62
5.2 Der erweiterte Transformationsalgorithmus . . . 67
5.2.1 Die neuen Selektoren . . . 67
5.2.2 Die neuen `TransformationSets` . . . 84
5.2.3 Die neuen `ProdPatterns` . . . 94
5.2.4 Die Klasse `HierarchicalTransformationHelper` . . . 99
5.3 Der „eigentliche“ Interpreter . . . 100
5.3.1 ASTEdge . . . 101
5.3.2 PredicateVariablePatternAction . . . 104

6 Zusammenfassung und Ausblick 107

A Die Eingabegrammatik für den Diaplan-Interpreter 111

Kapitel 1

Einleitung

Ein Algorithmus muss gesehen werden, damit er begreifbar wird, und der beste Weg zu lernen, wie ein Algorithmus funktioniert, ist, ihn auszuprobieren. ([1], Übers. nach [2])

TEX-Erfinder Don Knuth schrieb diesen Satz im ersten Band seines Werks „The Art of Computer Programming“ vor nunmehr über 30 Jahren. Heute setzt man in nahezu allen Teildisziplinen der Informatik Diagramme vor allem als Entwurfs- und Dokumentationshilfsmittel ein: Datenbankentwickler benutzen E-R-Diagramme ([3]), mit Petri-Netzen ([4]) werden verteilte Systeme modelliert, und die bekannte Unified Modeling Language (UML, [5]) ist ganz allgemein aus der Softwareentwicklung nicht mehr weg zu denken.

In der Implementierungsphase dienen diese Diagramme meist als Gedächtnisstütze oder liefern den Rahmen für die konkrete Programmierung. So generiert z.B. das bekannte UML-Werkzeug Rational Rose[1] aus der Struktur eines UML-Klassendiagramms „Klassenskelette“ mit Variablendeklarationen und Methodenköpfen. Die konkrete Implementierung einer Methode erfolgt in der eingestellten Programmiersprache, üblicherweise Java.

1.1 Motivation

Die verschiedenen Diagrammsprachen bzw. ihre Diagramme lassen sich mathematisch durch *Graphen* beschreiben. Ein Graph besteht aus einer Menge von Objekten, *Knoten* genannt, und einer zweiten Menge von anderen Objekten, *Kanten* genannt, die Knoten miteinander verbinden. Je nach Graphmodell können sowohl Knoten als auch Kanten zusätzliche Eigenschaften wie *Markierungen* oder *Attribute* besitzen.

Auf Basis dieses mathematischen Modells lassen sich mit Hilfe der sog. *Graphtransformation* Berechnungen durchführen. Ein hierauf basierendes System besteht aus *Regeln*, die einen Graphen durch Löschen oder Hinzufügen von Objekten in einen neuen überführen. Ausgangs- und Ergebnisgraph einer solchen Operation gehören dabei stets der gleichen *Graphsprache* an.

Trotz der Existenz eines solchen mathematischen Modells und mehrerer dieses Prinzips einsetzender Programmiersprachen und -werkzeuge dominieren in

[1] `http://www-306.ibm.com/software/rational/sw-atoz/indexR.html`

der Programmierpraxis mit Ausnahme spezieller Anwendungsbereiche textbasierte Sprachen. Die Gründe dafür sind vielschichtig und sprengen den Rahmen dieser Arbeit. Einer ist sicher die mangelnde Übersichtlichkeit einer großen Graphstruktur, die ein komplexes Softwaresystem darstellt[2]; ein weiterer dürfte die damit ein wenig einhergehende Tatsache sein, dass auch erfahrene Programmierer sich mangels bekannter Konstrukte erst lange in eine graphbasierte Sprache einarbeiten müssen.

Dia**plan** (DIAgram Programming LANguage) ist eine regelbasierte Sprache zur Programmierung mit Graphen, in deren Entwicklung mehrere wichtige Funktionalitäten moderner (textueller) Programmiersprachen teilweise schon eingeflossen sind, teilweise noch sollen. Zum Zeitpunkt der Abgabe dieser Diplomarbeit liegen Arbeiten zu folgenden Konzepten vor:

- *strukturierte Daten* ähnlich Objekten
- *Abstraktion* von Transformationsregeln ähnlich *Funktionen*
- *Kontrollmechanismen* zur Behebung ungewollten Nichtdeterminismus' während der Transformation ähnlich dem logischen Fehlschlag oder der funktionalen Ausnahme
- *Typisierung* der Daten

Weitere Konzepte wie Kapselung und Nebenläufigkeit sollen ebenfalls Eingang finden. Trotz der graphbasierten Natur ist die Entwicklung von Dia**plan** ans objektorientierte Programmierparadigma angelehnt.

1.2 DiaGen

DiaGen[3] ist ein Werkzeug zur Generierung von Editoren für beliebige Diagrammsprachen. Über eine graphische Schnittstelle kann der Benutzer eine Spezifikation für eine Sprache erstellen, nach der das System einen Editor generiert, der sowohl für freies als auch für *strukturiertes* Editieren, also die Arbeit mit vorgegebenen Sprachelementen geeignet ist.

Mit Hilfe eines von DiaGen generierten Editors lassen sich also zum einen systemunterstützt Diagramme erstellen, die stets der vorgegebenen Diagrammsprache genügen (strukturiertes Editieren), zum anderen aber auch frei erstellte Diagramme auf ihre *syntaktische* Korrektheit gemäß der vorgegebenen Diagrammsprache überprüfen. Eine Weiterverarbeitung der Diagramme, die eine Spezifizierung ihrer *semantischen* Eigenschaften voraussetzte, ist jedoch nicht möglich.

Genau zu diesem Zweck wurde (und wird) Dia**plan** primär entwickelt: mit einem von DiaGen generierten Editor erstellte Diagramme direkt ausführen zu können[4]. Abb. 1.1 auf der nächsten Seite zeigt die geplante Systemarchitektur:

- Der *Interpreter* führt Programme der jeweiligen Diagrammsprache aus, indem er den Eingabegraphen liest, gemäß dem Programm transformiert und den so modifizierten Graphen anzeigt. Dieser Vorgang erfolgt in einer

[2]Ein kompletter textbasierter Java-Klassenbaum ist zwar nicht übersichtlicher, aber die Sprache besitzt einen „historischen" Vorsprung, den eine graphbasierte Sprache durch Verbesserungen wettmachen müsste.

[3]DIAgram editor GENerator, `http://www2-data.informatik.unibw-muenchen.de/DiaGen/`

[4]Wenngleich dies das Primärziel darstellt, eignet sich Dia**plan** genauso als „eigenständige" Programmiersprache; da die hier beschriebenen Werkzeuge jedoch auf Teilen von DiaGen aufsetzen, ist die Entwicklung an letzteres gekoppelt.

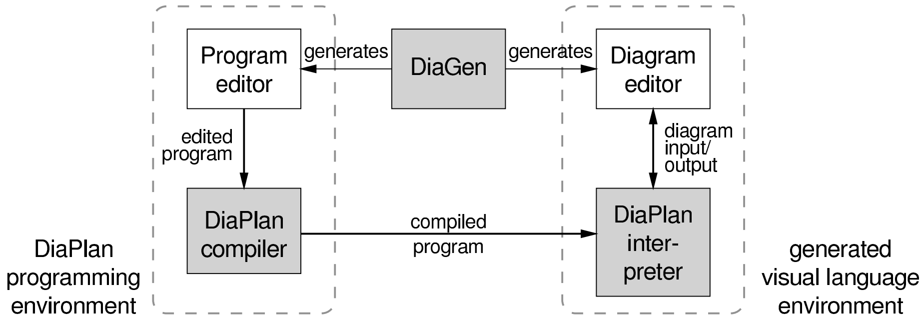


Abbildung 1.1: Die angestrebte DiaGen/**Diaplan**-Systemarchitektur (aus [6])

nutzergesteuerten Schleife. Die dafür eingesetzte *Graphtransformationsmaschine* wird auf der in DiaGen eingebauten basieren.

- Der *Übersetzer* liest Programme und transformiert sie in eine für den Interpreter einfach und effizient ausführbare Form. Gleichzeitig überprüft er, ob das Programm allen lexikalischen, syntaktischen und kontextuellen Regeln der jeweiligen Sprache gehorcht.
- Zur Erzeugung der Eingabedaten für den Interpreter soll selbiger über einen interaktiven visuellen Editor, der naheliegenderweise mit Hilfe von DiaGen erzeugt wird, verfügen.
- Ebenso sollen die Programme selbst durch einen von DiaGen erzeugten Editor erstellt werden.

1.3 Ziel dieser Arbeit

Bis zu dieser Arbeit existiert **Diaplan** „nur“ als eine Reihe von Artikeln, jedoch in keiner Implementierung. Daher wird hier die erste Version eines **Diaplan**-Interpreters vorgestellt, der **Diaplan**-Graphen einliest, verarbeitet und das Ergebnis der Transformation ausgibt. Das Hauptaugenmerk der Implementierung liegt dabei auf den für die Verarbeitung eines solchen Graphen notwendigen Erweiterungen der abstrakten DiaGen-Graphtransformationsmaschine; Ein- und Ausgabe erfolgen in einem sehr einfachen, speziell für diese Arbeit entwickelten textbasierten Format und dienen lediglich zur Überprüfung der internen Funktionen[5]. Der erstellte Interpreter ist somit bestenfalls als Prototyp zu betrachten. Die Implementierung erfolgte in der bekannten objektorientierten Programmiersprache Java[6], in der auch DiaGen selbst geschrieben ist.

1.4 Übersicht

Der Rest dieser Arbeit ist folgendermaßen gegliedert: *Kapitel 2 – Hypergraphen und Hypergraphtransformation* stellt **Diaplan**s theoretische Grundlagen

[5] Ein „Abfallprodukt“ dieser Arbeit ist ein Werkzeug, mit dem sich auch die Funktionalität der DiaGen-Basisgraphtransformationsmaschine testen lässt.

[6] `http://java.sun.com/`

und Sprachelemente vor und erläutert das Berechnungsmodell. Zusätzlich werden Vergleiche zu ausgewählten anderen Arbeiten zur Graphtransformation, sowohl auf theoretischer Ebene als auch über andere Graphtransformationswerkzeuge, angestellt. Zwei Beispiele für bereits erfolgreich mit dem in dieser Arbeit vorgestellten Interpreter getesteten und ausgeführten Programme enthält *Kapitel 3 – Anwendungsbeispiele. Kapitel 4 – Die* DiaGen-*Transformationsmaschine* erläutert die für diese Arbeit wichtigen Bestandteile der als Basis dienenden DiaGen-Graphtransformationsmaschine, an der selbst ebenfalls Änderungen vorgenommen wurden; diese Änderungen stellt *Kapitel 5 – Erweiterungen der* DiaGen-*Transformationsmaschine* vor, ebenso die für die Verarbeitung von Dia**plan**-Daten notwendigen Erweiterungen. Außerdem gibt es einen kurzen Einblick in die Ein-/Ausgabeschnittstelle. Abschließend fasst *Kapitel 6 – Zusammenfassung und Ausblick* die Ergebnisse dieser Arbeit zusammen und gibt Denkanstöße für Schritte zur Realisierung der oben beschriebenen Architektur. Im *Anhang A – Die Eingabegrammatik für den* Dia**plan**-*Interpreter* findet sich die für Ein- und Ausgabe benutzte Sprache in Form von Syntaxdiagrammen.

„Technische“ Anmerkung: Obgleich die entsprechende Reform erst ab dem kommenden Sommer verbindlich wird, wurde diese Diplomarbeit bereits nach den Regeln der sog. „neuen deutschen Rechtschreibung“ erstellt.

Kapitel 2

Hypergraphen und Hypergraphtransformation

Dieses Kapitel beschreibt die **Diaplan** zugrundeliegenden graphentheoretischen Konzepte. Abschnitt 2.1 beschäftigt sich mit den Datenstrukturen, Abschnitt 2.2 mit den verwendeten Algorithmen. Abschnitt 2.3 stellt Konzepte vor, deren Theorie teilweise schon ausgearbeitet ist, die aber nicht Bestandteil des im Rahmen dieser Arbeit implementierten Interpreters sind. Abschnitt 2.4 schildert verwandte Graphtransformationsansätze und beschäftigt sich mit anderen Graphprogrammiersprachen, insbesondere mit deren Unterschieden zu **Diaplan**. Die ersten drei Abschnitte entstammen in wesentlichen Teilen [7], die Quellen für den letzten Abschnitt sind jeweils an den entsprechenden Stellen vermerkt.

2.1 Hypergraphen und ihre Bestandteile

Ein *Hypergraph*[1] ist ein Tripel $G = \langle V, E, a\rangle$ aus zwei disjunkten endlichen Mengen V von *Knoten* und E von *Hyperkanten*[1] sowie einer Funktion $a\colon E \to V^{*}$[2], welche die Assoziation von Kanten zu Knoten beschreibt. Sind $G = \langle V, E, a\rangle$ und $H = \langle V', E' a'\rangle$ zwei Graphen, so ist G ein *Subgraph* von H ($G \subseteq H$) genau dann, wenn gilt: $V \subseteq V'$, $E \subseteq E'$ und $\forall e \in E : a(e) = a'(e)$. Eine Funktion m von $V \cup E$ nach $V' \cup E'$ ist ein *Hypergraphmorphismus* ($m : G \to H$), wenn gilt:

$$\begin{array}{rcll} m(o) \in V' & \Leftrightarrow & o \in V & \text{für alle } o \in V \cup E \\ a'(m(e)) & = & m^*(a(e)) & \text{für alle } e \in E\ ^{3} \end{array}$$

Ein Hypergraphmorphismus wird entsprechend der zugrundeliegenden Funktion als injektiv, surjektiv oder bijektiv bezeichnet. Zwei Graphen G und H heißen isomorph ($G \cong H$), wenn ein bijektiver Hypergraphmorphismus von G nach H existiert. In den folgenden Betrachtungen wird zwischen isomorphen Graphen

[1] In dieser Arbeit werden der Einfachheit halber oft die Begriffe „Graph“ und „Kante“ statt „Hypergraph“ und „Hyperkante“ verwendet.

[2] A^* ist die Menge aller endlichen Folgen, die aus den Elementen einer gegebenen Menge A bestehen, inklusive der leeren Folge ϵ.

[3] Ist $f\colon A \to B$ eine Funktion, so bezeichnet man mit $f^*\colon A^* \to B^*$ die Erweiterung dieser Funktion zu Folgen, die definiert sind als $f^*(a_1 \cdots a_n) = f(a_1)\cdots f(a_n)$ für $a_i \in A$, $1 \leqslant i \leqslant n$, $n \geqslant 0$.

nicht unterschieden, so dass die *Klassen* $\mathbb{G}$ von Graphen strenggenommen aus Isomorphieklassen von Graphen bestehen. In diesem Kapitel vorkommende konkrete Graphen sind also als Repräsentanten ihrer jeweiligen Isomorphieklasse zu betrachten.

2.1.1 (Einfache) Knoten und Kanten

Knoten und Kanten werden als *Objekte* eines Graphen bezeichnet; im DiaGen-Kontext heißen sie auch Graph*komponenten*. Ist eine Kante über die oben erwähnte Funktion a mit einem Knoten verbunden, so spricht man davon, dass sie diesen Knoten *besucht*.

In den Zeichnungen dieser Arbeit werden Knoten als Kreise oder ggf. abgerundete Rechtecke (Ovale) dargestellt, Kanten als Rechtecke mit je einer geraden Linie zu einem Knoten für jedes Auftreten dieses Knotens im Bild der Assoziationsfunktion a für diese Kante. Diese Linien werden Arme oder auch *Tentakel* genannt und sind, soweit nicht anders angegeben, entgegen dem Uhrzeigersinn sortiert, wobei der erste geradeaus nach oben zeigt. Binäre Kanten, also Kanten mit genau zwei Tentakeln, werden bisweilen als Pfeil vom ersten zum zweiten Knoten visualisiert (bzw., falls beide Arme mit demselben Knoten verbunden sind, als Pfeil mit Ausgangs- und Zielpunkt in ebendiesem Knoten), wobei das Rechteck entfällt.

2.1.2 Behälter

In **Diaplan**s theoretischem Modell können sowohl bestimmte Knoten als auch bestimmte Kanten Hypergraphen enthalten, deren Objekte teilweise wiederum Hypergraphen enthalten können usw., so dass ein *hierarchischer* bzw. *geschachtelter* Graph entsteht. Solche Objekte heißen *Behälter* (engl. *Container*); einen Graph ohne Behälter nennt man *flach*. Formal ausgedrückt besteht die Graphklasse $\mathbb{G}^n$ der Schachtelungstiefe n aus Quintupeln $G = \langle V, E, a, C, (G.c)_{c \in C} \rangle$, wobei $\langle V, E, a \rangle$ die bereits erwähnten Bestandteile eines Graphen darstellen, die Menge $C \in V \cup E$ die Behälter bezeichnet und darüberhinaus gilt: Ist $n = 0$, so $C = \emptyset$; anderenfalls $C \subseteq V \cup E$ sowie $\forall c \in C\colon G.c \in \mathbb{G}^{n-1}$. Der Graph $G.c$ heißt *Inhalt* des Behälters c. Graphobjekte, die nicht in C liegen, heißen *atomar*. Für einen in obiger Form gegebenen Graphen G bezeichnet man den *obersten* Graphen $\langle V, E, a \rangle$ mit $\widehat{G}$. Graphobjekte aus $V \cup E$ bezeichnet man als *direkt* im entsprechenden Graphen enthalten; *indirekt* enthalten sind alle, die einem der in den Behältern enthaltenen Graphen angehören.

Als *Behälterpositionen* in einem Graphen G bezeichnet man die Folgen

$$\Gamma_G = \{\epsilon\} \cup \{c\gamma | c \in C, \gamma \in \Gamma_{G.c}\}$$

Hierbei steht die oberste Position ϵ für den gesamten Graphen (also $G.\epsilon = G$), während eine geschachtelte Position $c\gamma \in \Gamma_G$ den Graphen $G.c\gamma = (G.c).\gamma$ bezeichnet. $G(\gamma)$ bezeichnet den obersten Graphen von $G.\gamma$. Die Graphhierarchie entspricht somit einer Baumstruktur, was in den Zeichnungen dadurch ausgedrückt wird, dass Graphen direkt und vollständig in ihren jeweiligen Behälter hinein gezeichnet werden, so dass keine Kante den Behälter verlässt.

Sind $G = \langle V, E, a, C, (G.c)_{c \in C} \rangle$ und $H = \langle V', E', a', C', (H.c)_{c \in C'} \rangle$ zwei Graphen, so ist G ein (geschachtelter) Subgraph von H ($G \Subset H$), wenn H

eine Behälterposition $\gamma \in \Gamma_H$ besitzt, so dass gilt: $\hat{G} \subseteq H(\gamma)$, und $\forall c \in C$: c ist auch in $H(\gamma)$ ein Behälter und $G.c \Subset H.\gamma c$. Ein *Graphmorphismus* m von G nach H ($m : G \to H$) ist ein Paar $\langle \hat{m}, M \rangle$, für das gilt:

- $\hat{m}\colon \hat{G} \to \hat{H}$ ist ein Hypergraphmorphismus gemäß der Definition für flache Graphen und erhält zusätzlich die Behälter: $\forall c \in C : \hat{m}(c) \in C'$.
- $M = (m_c : G.c \to H.\hat{m}(c))_{c \in C_G}$ ist eine Familie von Graphmorphismen.

Ein Graphmorphismus ist injektiv, surjektiv oder bijektiv, wenn alle enthaltenen (flachen) Hypergraphmorphismen die entsprechende Eigenschaft besitzen. Für Isomorphie gelten Eigenschaften analog zu denen der Hypergraphmorphismen.

2.1.3 Typen und Punkte

Ein *Typgraph* $\mathcal{C}(X) = \langle \mathcal{C}_V, \mathcal{C}_E \cup X, a \rangle$ ist ein Graph bestehend aus drei paarweise disjunkten Mengen $\mathcal{C}_V$ von *Knotenkonstantennamen*, $\mathcal{C}_E$ von *Kantenkonstantennamen* und X von *Variablennamen*. Die Funktion a eines Typgraphen definiert die *Stelligkeit* der Kantenkonstanten und Variablen. Ein *getypter punktierter Graph über* $\mathcal{C}(X)$ ist ein Tripel $\langle G, ty, p \rangle$ aus einem Graphen G in bekannter Form, einem Graphmorphismus $ty_G : G \to C(X)$ sowie einer Folge $p_G \in V^*$ von *Punkten*, ausgezeichneten Knoten. Solche Graphen werden nach Anzahl und Typ ihrer Punkte klassifiziert; $a(G) = ty^*_G(p_G)$ ist die *Stelligkeit* eines getypten punktierten Graphen G.

Punkte werden wie einfache Knoten visualisiert, erhalten aber einen dickeren Rand. Kantennamen werden in das zur Kante gehörende Rechteck (Unterabschnitt 2.1.1) hineingeschrieben bzw. bei Binärkanten direkt neben die Kante. Knotennamen kommen aufgrund von DiaGens „Kantenlastigkeit“ (Unterabschnitt 4.1.2) nicht vor.

2.1.4 Variablen

Kanten $e \in E_G$ mit $ty_G(e) \in X$ werden *Variablen* genannt; sie stehen in Regeln (Unterabschnitt 2.2.1) als Platzhalter für beliebige, insbesondere auch für den leeren Graphen. Den Teilgraphen, der im konkreten Fall für eine Variable eingesetzt wird, bezeichnet man als *Substitution* (Unterabschnitt 2.2.2, *Variableninstanziierung*). Unter Einsatz von Gestaltdefinitionen (Unterabschnitt 2.3.1) kann die Menge der Graphen, für die eine Variable stehen kann, eingeschränkt werden. Objekte $o \in V_G \cup E_G$ mit $ty_G(o) \in \mathcal{C}_N \cup \mathcal{C}_E$ heißen *konstante Knoten* bzw. *konstante Kanten*; die Menge $Var(G) \subseteq X$ bezeichnet die in G vorkommenden *Variablennamen*.

Ist G ein getypter punktierter Graph, so sind seine Variablennamen definiert als $Var(G) = \bigcup_{\gamma \in \Gamma_G} Var(G(\gamma))$. Der Subgraph von G, aus dem alle Variablen entfernt wurden, heißt *Skelett* von G ($\underline{G}$). G ist ein *Grundgraph*, falls $Var(G) = \emptyset$ (was gleichbedeutend ist mit $G = \underline{G}$).

In den Beispielen beginnen Variablennamen grundsätzlich mit einem Groß-, alle anderen Kantennamen mit einem Kleinbuchstaben oder einem nicht alphanumerischen Symbol.

2.1.5 Prädikatkanten

Im Gegensatz zu den übrigen erwähnten Graphobjekten repräsentieren *Prädikat*kanten keine Daten, sondern noch zu erledigende Auswertungsschritte eines Programms. Prädikatkanten sind stets atomar. Einen Graphen, der Prädikatkanten enthält, bezeichnet man in Analogie zu textuellen Programmiersprachen als *Ausdruck*; entsprechend der oben eingeführten Terminologie heißt ein variablenfreier Ausdruck *Grundausdruck*. Aufgrund der gleichen Analogie bezeichnet man die Knoten, mit denen eine Prädikatkante verbunden ist, als *Parameter*. Enthält der Knoten einen Graphen, so spricht man von einem *Graph*parameter; enthält dieser Graph eine oder mehrere Prädikatkanten, so bezeichnet man den Behälterknoten als *Prädikat*parameter bzw. *Prädikatbehälter*.

Die zu Prädikatkanten gehörenden Rechtecke (Unterabschnitt 2.1.1) werden in den Beispielen mit doppelten Kanten gezeichnet; die Alternativdarstellung für binäre Kanten entfällt hier.

Prädikattypen. Ein *Prädikatentypgraph* $\mathcal{C}(\mathcal{Q}, X)$ enthält einen Typgraphen $\mathcal{C}(X)$ sowie eine Menge $\mathcal{Q}$ von *Prädikatnamen*. Eine Teilmenge $Y \subseteq X$ enthält *Prädikatvariablennamen*. Kanten, die mit einem Namen aus Y bezeichnet sind, nennt man *Prädikatvariablen*; Kanten, die mit einem Namen aus $X \backslash Y$ bezeichnet sind, heißen *Datenvariablen*. Da Prädikatvariablen, wie der Name andeutet, für Prädikatkanten stehen, umfasst ihre Substitution stets nur höchstens eine Kante.

2.2 Transformation

Dia**plan**s Berechnungsmodell ist *Graphtransformation*. Hierbei werden im zu bearbeitenden (Daten-)Graphen vorgegebene Muster gesucht und gemäß entsprechenden *Regeln* modifiziert.

2.2.1 Regeln

Graphtransformationsregeln bestehen im Allgemeinen aus zwei Teilen: einer *linken Seite*, auch *Muster* genannt, nach der im zu bearbeitenden Graphen gesucht, und einer *rechten Seite*, durch die das gefundene Muster ersetzt wird. Beide Seiten sind Graphen. Bei einer Regel*anwendung* werden Objekte, die nur auf der linken Seite vorkommen, aus dem zu bearbeitenden Graphen gelöscht; Objekte, die links und rechts vorkommen, bilden den sogenannten *Klebegraphen* und werden beibehalten; Objekte, die nur auf der rechten Seite vorkommen, werden neu erzeugt.

Dia**plan** erlaubt *bedingte* Regeln, die aus einer linken Seite, einer oder mehreren *Anwendbarkeitsbedingungen* und einer rechten Seite bestehen. Für die Anwendung einer solchen Regel genügt es im Allgemeinen nicht, die linke Seite im zu bearbeitenden Graphen zu finden, da der Bedingungsteil u.U. nicht erfüllbar ist (Unterabschnitt 2.2.2).

Benutzung von Variablen. Eine Variable (Unterabschnitt 2.1.4) darf in den Anwendbarkeitsbedingungen oder der rechten Seite einer Regel nur dann benutzt werden, wenn wenigstens eine Variable gleichen Namens bereits in der linken Seite vorkommt. Anderenfalls wäre für eine solche Variable eine beliebige Substitution möglich, so dass völlig zusammenhanglos neue (Teil-)Graphen „aus dem Nichts“ erzeugt werden könnten. Der oberste Graph der linken Seite einer Regel darf nicht nur aus einer Variablen und ggf. den angebundenen Knoten bestehen, sondern muss wenigstens eine konstante Kante enthalten; ohne diese Einschränkung wäre eine solche Variable nahezu beliebig instanziierbar, wodurch die Transformation selbst divergierte. Ähnliche Einschränkungen gibt es bei der Nutzung von Variablen in der Termersetzung ([8]).

Eindeutige Substitutionen. Der im Rahmen dieser Arbeit implementierte Interpreter schränkt die Nutzung von Variablen zusätzlich folgendermaßen ein:

- Ist bekannt, dass der zu bearbeitende Graph sowie alle in Behältern evtl. enthaltenen Subgraphen zusammenhängend sind, so darf auf der linken Seite jeder Regel nicht mehr als eine Variable mit einem Knoten verbunden sein. Man bezeichnet die Variablen in diesem Fall als *apart*.

- Ist die Zusammenhangseigenschaft nicht gewährleistet, so darf auf der linken Seite jeder Regel im obersten Graphen keine und in jedem der in Behältern enthaltenen Graphen maximal eine Variable direkt enthalten sein.

Wie in [9] dargelegt, ist ein Regelsystem, das diese Eigenschaften erfüllt, in Bezug auf Variablensubstitutionen stets eindeutig; d.h., sofern bei der Ansatzsuche für die übrigen Komponenten ein Morphismus gefunden wurde, ist für die Variablen nur noch höchstens eine Substitution möglich.

Benutzung von Prädikatkanten. Ein Ausdruck, dessen oberster Graph die einzige Prädikatkante e mit $ty_{G(\epsilon)}(e) = q$ des gesamten Graphen enthält, heißt *q-Muster*. Die linken Seiten von **Diaplan**s bedingten Regeln sind q-Muster[4], die Anwendbarkeitsbedingungen und rechten Seiten können beliebig viele Prädikatkanten enthalten.

Abschlussklauseln. Während der Auswertung eines Programms (Unterabschnitt 2.2.2) können zwei verschiedene Formen von „Sackgassen“ auftreten: Den logischen Sprachen entstammt das Konzept des *Fehlschlags* (engl. *failure*), den funktionalen das der *Ausnahme* (*exception*). Um beide Sachverhalte ausdrücken zu können, stellt **Diaplan** vordefinierte Prädikatkanten $-$ (Fehlschlag), $+$ (*Erfolg* (*success*) als Gegenteil von Fehlschlag) und $\bot_1 \ldots \bot_k$ $(k > 0)$ (entsprechend der Anzahl der möglichen auftretenden Ausnahmen) zur Verfügung. Die Kanten für Erfolg und Fehlschlag sind stets nullstellig, aber Ausnahmen können Parameter besitzen. Graphen, die eine dieser vordefinierten Kanten enthalten, heißen *Abschlussklauseln*. Eine *Prädikatdefinition* für ein Prädikat q besteht aus allen Regeln, deren linke Seiten ein q-Muster bildet, Ausnahmebehandlern für die möglicherweise in den Regeln auftretenden Ausnahmen sowie einer obligatorischen Abschlussklausel, auf die zurückgegriffen wird, wenn keine Regel anwendbar ist; die speziell für Abschlussklauseln definierten Prädikatkanten können

[4] Somit erfüllen **Diaplan**-Regeln bereits „von Haus aus“ stets die oben erwähnte Einschränkung, dass der oberste Graph einer linken Seite nicht nur aus einer Variablen mit ggf. den angebundenen Knoten bestehen darf.

auch in der rechten Seite einer Regel vorkommen, jedoch nie mehr als eine pro Regel.

2.2.2 Auswertungsalgorithmus

Bevor eine Regel angewendet werden kann, muss ein Morphismus eines Mustergraphen P in einen gegeben Graphen K gefunden werden; dieser Morphismus wird als *Ansatz* bezeichnet. Die Anwendung der Regel bezeichnet dann das Ersetzen dieses Ansatzes durch einen Graphen R, nämlich die rechte Seite der entsprechenden Regel. Ähnlich wie bei der Termersetzung ([8]) werden bei Dia**plan** in P und R vorhandene Variablen gemäß einer *Substitution* σ, die Variablen durch Graphen ersetzt, *instanziiert*, wodurch *Graphinstanzen* P^σ und R^σ erzeugt werden. Durch *Einbettung* dieser Graphinstanzen in einen *Kontextgraphen* K entsteht ein *Transformationsschritt* $K[P^\sigma] \Rightarrow K[R^\sigma]$. Sowohl Variableninstanziierung als auch Kontexteinbettung sind definiert als *geschachtelte Kantenersetzung*, eine Erweiterung der in [10] erläuterten *Kantenersetzung*.

Kantenersetzung. Seien G und R zwei Graphen, die sowohl in Knoten als auch in Kanten disjunkt sind; sei e eine Kante in G mit $a(R) = a(ty_G(e))$. Die *Ersetzung* von e in G durch R liefert den Graphen $G[e/R]$ auf folgende Weise: Zunächst entfernt man e aus G, fügt dann die Objekte von R hinzu und identifiziert[5] schließlich jeden Punkt von R mit dem entsprechenden Knoten von e. In $G[e/R]$ bleiben alle Kantenbindungen an Knoten gleich, behalten alle Graphobjekte ihren Typ und gelten die Punkte von G.

Geschachtelte Kantenersetzung. Sei G ein Graph, bei dem $G(\gamma)$ an einer beliebigen Behälterposition $\gamma \in \Gamma_G$ eine atomare Kante e enthält; sei R ein Graph mit $a(R) = a(ty_{G(\gamma)}(e))$. Seien weiterhin sowohl die Knoten- als auch die Kantenmengen von $\hat{R}$ und $G(\gamma)$ disjunkt; seien entweder alle durch e verbundenen Knoten oder alle Punkte in $\hat{R}$ atomar. Dann liefert die *geschachtelte Kantenersetzung* von e in G durch R (geschrieben $G[\gamma \cdot e/R]$) einen Graphen G' mit den Behälterpositionen $\Gamma_{G'} = \Gamma_G \cup \{\gamma\bar{\gamma} | \bar{\gamma} \in \Gamma_R \setminus \{\epsilon\}\}$, der folgende Graphen enthält:

$$G'(\gamma') = \begin{cases} G(\gamma)[e/\hat{R}] & \text{falls } \gamma' = \gamma \\ R(\bar{\gamma}) & \text{falls } \gamma' = \gamma\bar{\gamma} \text{ mit } \bar{\gamma} \neq \varepsilon \text{ ,} \\ G(\gamma') & \text{sonst} \end{cases} \quad \text{für alle } \gamma' \in \Gamma_{G'}$$

Kontexteinbettung. Ein Graph K, der eine einzelne Variable e in einem der Graphen $K(\gamma)(\gamma \in \Gamma_K)$ enthält, ist ein *Kontext*, wenn e mit atomaren Knoten verbunden ist. Die *Einbettung* eines Graphen U in K ($K[U]$) liefert einen isomorphen Graphen zu $K[\gamma \cdot e/U]$.

[5]Die Identifikation zweier Knoten des gleichen Typs in einem Graphen G bewahrt einen der beiden und ersetzt jedes Vorkommen des anderen in der Punktfolge von G sowie in den Anbindungen seiner Kanten durch den bewahrten.

Variableninstanziierung. Eine Funktion $\sigma : X \to \mathbb{G}(X)$ ist eine *Substitution*, wenn für alle $x \in X$ gilt, dass alle obersten Punkte von $\sigma(x)$ atomar sind. Die *Instanz* P^σ eines Graphen P gemäß σ erhält man durch gleichzeitiges Ersetzen aller Vorkommen von x in allen in P direkt und indirekt enthaltenen Graphen durch Kopien des Graphen $\sigma(x)$: Sei $\{(\gamma_1, e_1), \dots, (\gamma_k, e_k)\}$ die Menge aller Paare mit $\gamma_i \in \Gamma_P$ und $e_i \in E_{P(\gamma_i)}$, wobei $ty_{P(\gamma_i)}(e_i) \in X$. Dann gilt:

$$P^\sigma \cong P[\gamma_1 \cdot e_1/\sigma(ty_{G(\gamma_1)}(x_1))] \cdots [\gamma_k \cdot e_k/\sigma(ty_{G(\gamma_k)}(x_k))]$$

Die Reihenfolge der Ersetzung ist unerheblich, da geschachtelte Kantenersetzung (genau wie einfache Kantenersetzung) sowohl kommutativ als auch assoziativ ist.

Transformationsschritt. Eine (bedingungsfreie) Regel $t = P/R$ besteht aus zwei Graphen P und R, wobei das Muster P wenigstens eine konstante Kante enthält und im *Ersatz* R nur Variablennamen aus P vorkommen. Ein Graph G wird mittels t in einen Graphen H *transformiert* ($G \Rightarrow_t H$), wenn die Instanzen von P und R gemäß einer Substitution σ in einen Kontext K eingebettet werden können, so dass $G \cong K[P^\sigma]$ und $H \cong K[R^\sigma]$.

Der Algorithmus für einen solchen *Transformationsschritt* zerfällt in fünf einzelne Schritte:

1. *Finde* einen Subgraphen $\underline{O}$ in G mit $\underline{P} \cong \underline{O} \Subset G$. Sei $\gamma \in \Gamma_G$ die innerste Behälterposition mit $\underline{P} \cong \underline{O} \Subset G.\gamma$. ($\underline{O}$ nennt man das *Skelettauftreten* von P in G.)

2. *Binde* die Variablen in P mit einer Substitution σ, so dass $P^\sigma \cong O \Subset G.\gamma$, wobei $\underline{O} \Subset O$ und für alle Behälter in O gilt: $O.c = G.\gamma c$. (O heißt *Instanzauftreten* von P in G.)

3. *Schneide* O aus G durch Entfernen der Kanten und der Knoten bis auf die Punkte heraus; verbinde dann die Punkte von O mit einer Variablen, um den Kontext K zu erhalten.

4. *Instanziiere* eine neue Kopie von R gemäß der Substitution σ.

5. *Bette* die Instanz R^σ in K *ein*, um H zu erhalten.

Diese Einzelschritte können unabhängig voneinander für jeden (flachen) Hypergraphen in P durchgeführt werden, wobei mit einem Auftreten des Skeletts des obersten Hypergraphen $\hat{P}$ in einem der (flachen) Graphen $G(\gamma)$ von G begonnen wird. Bei diesem Vorgehen muss lediglich sichergestellt werden, dass die Inhalte der Behälter isomorph zueinander sind: $\forall \gamma' \in \Gamma_P : \gamma' \neq \epsilon \Rightarrow P(\gamma')^\sigma \cong G(\gamma\gamma')$. Diese Zusatzbedingung wird in dieser Arbeit als *hierarchische Vollständigkeit* bezeichnet; ein „offizieller“ Name existiert bis jetzt nicht. Die auf Seite 11 unter *Eindeutige Substitutionen* genannten Einschränkungen in der Nutzung von Variablen verhindern eine Mehrdeutigkeit bei Schritt 2. Für jeden Graph G sind jede Transformation $G \Rightarrow H$ und ihr Ergebnis H durch einen *Redex* $\rho = \langle t, m, \sigma \rangle$ eindeutig bestimmt, der sich aus der angewendeten (bedingungsfreien) Regel $t = P/R$, einem injektiven Graphmorphismus m, der das Auftreten des Skeletts von P in $G.\gamma$ markiert, sowie der Substitution σ zusammensetzt.

Ausdrucksauswertung. Wie in Unterabschnitt 2.2.1 erwähnt, gehört zu der Definition eines Prädikatnamens $q \in \mathcal{Q}$ eine endliche Menge $\mathcal{T}_q \neq \emptyset$ aus Graphtransformationsregeln t, deren linke Seiten q-Muster bilden. Ein *Programm* ist die endliche Menge $\mathcal{T} = \bigcup_{q \in \mathcal{Q}} \mathcal{T}_q$ von Prädikatdefinitionen. Mit diesen Vorüberlegungen wird Graphtransformation zu *Ausdrucksauswertung*, wobei das Ersetzungssystem $\langle \mathbb{E}, \Rightarrow_{\mathcal{T}} \rangle$ folgendermaßen definiert ist: Kann man zwei Grundausdrücke G und H durch einen *Auswertungsschritt* mithilfe einer Prädikatregel $t \in \mathcal{T}$ ineinander überführen, so schreibt man $G \Rightarrow_{\mathcal{T}} H$; existiert eine *Auswertungsfolge* aus $n \geqslant 0$ aufeinanderfolgenden Schritten, die dies bewerkstelligt, so schreibt man $G \Rightarrow^*_{\mathcal{T}} H$. Ein Grundausdruck G ist eine *Normalform*, wenn keine Regel in $\mathcal{T}$ auf ihn anwendbar ist. Ohne Betrachtung von *Gestaltdefinitionen* (Unterabschnitt 2.3.1) ist jede Normalform, die keine Prädikatkante enthält, *terminal*. Ein Programm heißt *schwach normalisierend*, wenn es jeden Grundausdruck in mindestens eine Normalform überführt; ein Programm heißt *eindeutig normalisierend*, wenn es jeden Grundausdruck in höchstens eine Normalform überführt. $\mathcal{T}$ heißt *terminierend*, wenn jede Auswertungsfolge nach endlich vielen Schritten eine Normalform liefert.

Prädikatauswertung. Da sowohl Daten als auch einzelne Auswertungsschritte in Dia**plan** als Graphen dargestellt werden, liegt es nahe, einen Programmablauf ebenfalls als Graph aufzufassen. Während des Programmablaufs wird ein *Auswertungsbaum*, ebenfalls ein geschachtelter Graph, konstruiert, der folgendermaßen aufgebaut ist: Der oberste Hypergraph $\Delta(\epsilon)$ ist ein Baum von *Auswertungszuständen*, die Grundausdrücke enthalten und durch atomare Binärkanten verbunden werden, welche mit Redizes ρ markiert sind. $\Delta(\epsilon)$ ist unendlich, sofern die Auswertungsrelation $\Rightarrow_{\mathcal{T}}$ nicht terminierend ist. Die Baumwurzel n_0 entspricht dem *Initialzustand* und enthält den eingegebenen Grundausdruck. Enthält $\Delta(\epsilon)$ eine atomare binäre ρ-Kante von einem Zustand n zu einem Zustand n', so gilt $\Delta.n \Rightarrow_t \Delta.n'$ mittels eines Redex' ρ von t. Ein Zustand $n \in \Delta(\epsilon)$ ist *vollständig*, falls eine ρ-Kante von n zu irgend einem Zustand n' existiert, die mit jedem im Ausdruck $\Delta.n$ auftretenden Redex ρ markiert ist. Δ ist vollständig, wenn alle seine Zustände vollständig sind. In diesem Fall enthält jedes Blatt n von Δ einen Ausdruck $\Delta.n$ in Normalform; lässt man Gestaltdefinitionen (Unterabschnitt 2.3.1) außer Acht, so ist $\Delta.n$ stets ein *Ergebnis* von Δ, sofern es sich um einen terminalen Ausdruck handelt.[6]

Die Semantik von Programmen, die ja im Allgemeinen nichtdeterministisch sind, wird über eine *Auswertungsfunktion* $eval_{\mathcal{T}} : \mathbb{E} \to \mathbb{G}^{\infty}$[7] definiert, die eine möglicherweise unendliche Folge von Ergebnissen aufzählt; sie initialisiert den Auswertungsbaum Δ mit dem oben beschriebenen Initialzustand n_0 und fügt für jeden Auswertungsschritt $\Delta.n \Rightarrow_t \Delta.n'$ neue Zustände und Verbindungskanten hinzu, bis jeder Zustand vollständig ist. Ohne Beachtung von Gestaltdefinitionen ist jeder dabei entstehende terminale Graph $\Delta.n$ ein Ergebnis und wird dementsprechend als Teil der Folge zurückgegeben.

Strikte Tiefensuche. Die Auswertungsfunktion $eval_{\mathcal{T}}$ ist in mehreren Punkten nichtdeterministisch:

[6] Wie in [7] angedeutet, konstruiert der im Rahmen dieser Arbeit implementierte Interpreter nicht den kompletten Baum Δ, sondern arbeitet stets auf dem aktuellen Zustand, wobei er Vorgängerzustände bei Bedarf durch Backtracking wiederherstellt.

[7] A^{∞} beschreibt die Menge aller endlichen und unendlichen Folgen über einer Menge A.

1. Auswahl des *aktuellen Zustands* $\hat{n} \in \Delta(\epsilon)$, von dem aus die Auswertung fortgeführt werden soll.

2. Auswahl des *aktuellen Aufrufs*, also der Prädikatkante $\hat{e}$ in $\Delta(\hat{n}\hat{\gamma})$ mit $\hat{\gamma} \in \Gamma_{\Delta.\hat{n}}$, nach deren Auftreten als Nächstes gesucht werden soll.

3. Auswahl der *aktuellen Regel* $\hat{t} \in \mathcal{T}_q$ $(q = ty_{\Delta(\hat{n}\hat{\gamma})}(\hat{e}))$, die auf diese Kante angewendet werden soll.

4. Auswahl des *aktuellen Redex'* $\hat{\rho} = \langle \hat{t}, \hat{m}, \hat{\sigma} \rangle$ der Regel $\hat{t}$, durch den $\Delta.n$ transformiert werden soll.

Die für Dia**plan** vorgesehene Auswertungsstrategie *strikte Tiefensuche* ([7]) eliminiert den Nichtdeterminismus weitgehend:

1. Die Zustände in $\Delta(\epsilon)$ werden nach Alter vollständig sortiert. Der aktuelle Zustand $\hat{n}$ ist der unvollständige Zustand, dessen Einführung am kürzesten zurückliegt. Dies entspricht der bei logischer Programmierung verwendeten *Tiefensuche*.

2. Innerhalb eines Zustands n werden die Prädikatkanten zunächst nach Alter, dann nach Schachtelungstiefe sortiert. Diese Ordnung ist nur partiell, da in einem Transformationsschritt mehrere Prädikatkanten auf der gleichen Schachtelungsebene erzeugt werden können. Der aktuelle Aufruf ist eine der innersten der neuesten Prädikatkanten in $\Delta.\hat{n}$, die außerhalb von Prädikatbehältern auftreten. Dies entspricht der bei funktionaler Programmierung eingesetzten *strikten Auswertung*.

3. Alle Regeln werden gemäß der Programmreihenfolge sortiert. Die aktuelle Regel $\hat{t}$ ist die erste im Programm aufgeführte Regel, für die ein Redex $\rho = \langle \hat{t}, \hat{m}, \hat{\sigma} \rangle$, existiert, wobei das Auftreten $\hat{m}$ den aktuellen Aufruf $\hat{e}$ so enthält, dass keine ρ-Kante bei $\hat{n}$ beginnt.

4. Die aktuelle Regel $\hat{t}$ kann mehrere Redizes besitzen, die $\hat{e}$ enthalten. Diese können verschiedene einander *überlappende* Auftreten $\hat{m}$ sowie miteinander *konkurrierende* Substitutionen $\hat{\sigma}$ besitzen. Weder für verschiedene Auftreten noch verschiedene Substitutionen existieren intuitive Sortierungen; der Nichtdeterminismus bei Substitutionen lässt sich jedoch durch Einschränkung der Regeln wie auf S. 11 unter *Eindeutige Substitutionen* beschrieben beheben. Überlappendes Auftreten kann nicht automatisch ausgeschlossen werden; dies liegt in der Verantwortung des Programmierers beim Regelentwurf.

Auswertung der Abschlussklauseln. Steht keine aktuelle Regel $\hat{t}$ mit einem Auftreten m, das den aktuellen Aufruf $\hat{e}$ enthält, (mehr) zur Verfügung, greift die Abschlussklausel der entsprechenden Prädikatdefinition folgendermaßen:

- Im Erfolgsfall (+) werden ein neuer Zustand n' und eine neue mit + markierte Kante von $\hat{n}$ zu n' hinzugefügt, so dass $\Delta.n'$ $\Delta.n$ ohne $\hat{e}$ enthält, und die Auswertung fortgesetzt.

- Im Fall eines Fehlschlags ($-$) bestimmt Backtracking den Ahnenzustand $\bar{n}$ von $\hat{n}$, in dem $\hat{e}$ eingesetzt wurde. Die Auswertung wird mit dem nächsten Redex für den aktuellen Aufruf von $\bar{n}$ fortgesetzt.

- Beim Auftreten einer Ausnahme ($\perp_k$) bestimmt Interrupt-Handling den nächstliegenden Ahnenzustand $\bar{n}$ von $\hat{n}$, der über eine entsprechende Ausnahmeklausel verfügt. Diese Klausel wird ausgewertet und in Δ mit einer $\perp_k$-Kante von $\bar{n}$ zu einem Zustand n' bezeichnet, von dem aus die Auswertung fortgesetzt wird.

Anwendbarkeitsbedingungen. Die bereits erwähnten Anwendbarkeitsbedingungen dienen dazu, den oben unter Punkt 2 aufgeführten „Rest"-Nichtdeterminismus zu beheben. Eine bedingte Regel für ein Prädikat q schreibt man als $P \parallel A/R$, wobei P ein q-Muster ist, während A und R Ausdrücke sind. Die Anwendung einer solchen Regel auf einen Grundausdruck G erfolgt in zwei Stufen:

1. Die Regel wird versuchsweise wie eine bedingungsfreie Regel $P/A \cup R$ angewendet; hernach werden alle Prädikatkanten aus A vollständig ausgewertet.

2. Schlägt die vollständige Auswertung von A fehl, hat sich die Regel als nicht anwendbar erwiesen; die versuchsweise Regelanwendung wird rückgängig gemacht, woraufhin eine weitere Regel für q angewendet werden kann. Anderenfalls gilt die Regel als anwendbar, worauf die Auswertung fortgesetzt wird. Schlägt nun eines der Prädikate in R fehl, schlägt die gesamte Auswertung für q fehl, ohne dass weitere q-Regeln ausprobiert werden.

Da es stets möglich ist, ein Regelsystem so aufzubauen, dass A und R jeweils höchstens eine Prädikatkante enthalten, genügen Anwendbarkeitsbedingungen, um den bei Schritt 2 der strikten Tiefensuche erwähnten Nichtdeterminismus zu eliminieren.

2.3 Weitere Konzepte von Dia**plan**

Unter den in diesem Abschnitt vorgestellten Konzepten, von denen keines in den im Rahmen dieser Arbeit implementierten Interpreter Eingang gefunden hat, sind die Gestaltdefinitionen (Unterabschnitt 2.3.1) das wichtigste, zu dem bisher auch die Theorie am weitesten ausgearbeitet ist; mit ihrer Hilfe lässt sich eine Typisierung der Daten einführen, was eine notwendige Voraussetzung für die geplante Implementierung eines Compilers darstellt. Die Ideen in Unterabschnitt 2.3.2 erhöhen bei geeigneter Implementierung die Effizienz beim Übersetzungs- oder Interpretationsprozess. Universelle Behälter (Unterabschnitt 2.3.3) erlauben eine intuitive Darstellung für „einfache" Daten wie Zahlen und Namen; Kapselung bzw. Modularisierung (Unterabschnitt 2.3.4) sowie Nebenläufigkeit (Unterabschnitt 2.3.5) sind in modernen objektorientierten Programmiersprachen unverzichtbar.

2.3.1 Gestaltdefinitionen

Gestaltdefinitionen (engl. *shapes*) erlauben es, Graphen und Variablen nach ihrer Struktur in verschiedene Gruppen einzuteilen, was das Äquivalent einer Typisierung in einer textuellen Programmiersprache darstellt.

Zusammengesetzte Gestaltdefinitionen für Graphen. Angenommen, für flache Graphen $\mathbb{H}_{\mathcal{C}}(X)$ existieren *Gestaltregeln* Σ, mit deren Hilfe ebendiese Graphen gemäß einer endlichen Menge S von *Gestaltnamen* klassifiziert werden können. Dann schreibt man $\Sigma \vdash G : s$, wenn sich aus den Gestaltregeln Σ ergibt, dass ein Graph $G \in \mathbb{H}_{\mathcal{C}}(X)$ gemäß eines Gestaltnamens $s \in S$ *wohlgeformt* ist. Die Zuordnung zu einer Gestaltdefinition muss hierbei aus praktischen Überlegungen stets entscheidbar, aber nicht notwendigerweise eindeutig sein. Sei nun jeder flache Graph in den Behältern eines hierarchischen Graphen geformt. Die Gestaltdefinition flacher Graphen in geschachtelten Behälterpositionen wird durch den Typ des Behälters bestimmt. Dazu existiere eine Funktion $sh\colon \mathcal{C}_V \cup \mathcal{C}_E \cup X \to S$, die für jedes Graphobjekt $o \in \mathcal{C}_V \cup \mathcal{C}_E$ die *Inhaltsgestalt* jedes Behälters dieses Typs und für jeden Variablennamen $n \in X$ die *Wertgestalt* der Graphen, an die x gebunden werden kann, definiert. Ein (hierarchischer) Graph $G \in \mathbb{G}_{\mathcal{C}}(X)$ ist gemäß eines Gestaltnamens s *wohlgeformt*, in Kurzform $\Sigma \vdash G : s$, wenn

- sein oberster Graph gemäß s wohlgeformt ist, also $\Sigma \vdash \hat{G} : s$ und
- die flachen Graphen $G(\gamma c)$ an allen geschachtelten Behälterpositionen $\gamma c \in \Gamma_G$ gemäß der Inhaltsgestalt ihrer Behälterobjekte wohlgeformt sind, also $\Sigma \vdash G(\gamma c) : sh(ty_{G(\gamma)}(c))$.

Kontextfreie Gestaltregeln. Sei $\mathcal{C}(S)$ ein *Gestaltgraph* ähnlich dem Typgraphen $\mathcal{C}(X)$ aus Unterabschnitt 2.1.3 mit Gestaltnamen S statt Variablennamen X. Es gelte außerdem $X \cap S = \emptyset$. Die getypten und punktierten flachen Graphen über $\mathcal{C}(S)$ ($\mathbb{H}_{\mathcal{C}}(S)$) heißen *Syntaxgraphen*.

Den *Nonterminalgraphen* eines Gestaltnamens $s \in S$ bezeichnet man mit $\langle s \rangle$. Er besteht aus einer Kante vom Typ s, die an eine ihrer Stelligkeit entsprechende Anzahl von Punkten angebunden ist; also gilt $V_{\langle s \rangle} = \{n_1, \ldots, n_k\}$, $E_{\langle s \rangle} = \{e\}$, $a_{\langle s \rangle}(e) = p_{\langle s \rangle}$ und $a_{\mathcal{C}(S)}(ty_{\langle s \rangle}(e)) = ty^*_{\langle s \rangle}(a_{\langle s \rangle}(e))$. Eine *kontextfreie Gestaltregel* $r : \langle s \rangle ::= R$ besteht aus dem Bezeichnergraphen $\langle s \rangle$ für einen Gestaltnamen s und einem Syntaxgraphen R mit $a(s) = a(R)$. Die Regel *leitet direkt* einen Syntaxgraphen G in einen Syntaxgraphen H *ab*, kurz $G \Rightarrow_r H$, wenn gilt $H \cong G[e/R']$, wobei e eine Kante in G ist, für die gilt $ty_G(e) = s$, und R' isomorph zu R und disjunkt zu G ist.

Sei Σ^{CF} eine endliche Menge kontextfreier Gestaltregeln. Gilt für eine Regel $r \in \Sigma^{\mathrm{CF}}$ $G \Rightarrow_r H$, so schreibt man $G \Rightarrow_{\Sigma^{\mathrm{CF}}} H$; die reflexiv-transitive Hülle von $\Rightarrow_{\Sigma^{\mathrm{CF}}}$ schreibt man als $\Rightarrow^*_{\Sigma^{\mathrm{CF}}}$. Kontextfreie *Gestaltzuordnung* ist definiert als $\Sigma^{\mathrm{CF}} \vdash G : s \;\Leftrightarrow\; \langle s \rangle \Rightarrow^*_{\Sigma^{\mathrm{CF}}} sh(G)$, wobei man die Gestaltdefinition $sh(G) \in \mathbb{H}_{\mathcal{C}}(S)$ eines flachen Graphen $G \in \mathbb{H}_{\mathcal{C}}(X)$ durch das Ersetzen jedes Variablennamens $x \in X$ in G durch dessen Gestaltdefinition $sh(x) \in S$ erhält.

Nach [11] ist die Frage, ob ein Graph einem Gestaltregelsystem Σ^{CF} entspricht, entscheidbar.

Berücksichtigt man Gestaltdefinitionen bei **Diaplan**s Transformationsregeln, so lassen sich für Variablen Gestaltdefinitionen angeben; dann ist für eine Variable keine beliebige Substitution mehr möglich, sondern sie muss der vorgegebenen Gestaltdefinition entsprechen. Ebenso müssen in bedingten Regeln linke Seite, Bedingung und rechte Seite alle die gleiche Gestaltdefinition besitzen. Unter Verwendung von Gestaltdefinitionen ist nicht notwendigerweise jede Normalform im Auswertungsbaum Δ (S. 14, *Prädikatauswertung*) ein Ergebnis; entspricht sie nicht den Gestaltregeln, d.h. handelt es sich um keinen Σ-Graph, so ist sie eine *Sackgasse*.

2.3.2 Fehlbarkeit und Nichtdeterminiertheit von Prädikaten

Wenn wenigstens eine Regel eines Prädikats inkl. der obligatorischen Abschlussklausel eine Fehlschlag-Kante enthält, kann das Prädikat fehlschlagen, anderenfalls nicht. Nicht fehlbare Prädikate heißen *Aktionen*. Aktionen werden nicht notwendigerweise immer erfolgreich ausgewertet, da sowohl Ausnahmen als auch Nichtterminierung möglich sind.

Prädikate können theoretisch mehrere Ergebnisse liefern. Einem Programmierer kann aber ermöglicht werden, selbst zu spezifizieren, ob er bei einem Prädikat Determinismus *beabsichtigt*. In diesem Fall kann die Auswertung abbrechen, sobald das erste Ergebnis gefunden wurde.

Eine vollständige Prädikatdefinition hat dann diese Struktur:

$$
\begin{array}{l}
[\textbf{non}]\textbf{deterministic}\,(\textbf{predicate}|\textbf{action}) \\
\qquad \begin{array}{ccccccc}
q & = & P_1 & \talloblong & A_1 & / & R_1 \\
 & & \vdots & & \vdots & & \vdots \\
 & | & P_m & \talloblong & A_m & / & R_m \\
 & ? & E_1 & \talloblong & H_1 & & \\
 & & \vdots & & \vdots & & \\
 & ? & E_n & \talloblong & H_n & & \\
 & /\!\!/ & F & & & &
\end{array}
\end{array}
$$

Die Definition jedes Prädikats $q \in \mathcal{Q}$ legt fest, ob es deterministisch oder nicht ist und ob es fehlschlagen kann oder eine Aktion sein soll. Es enthält $m \geqslant 1$ bedingte Regeln sowie $n \geqslant 0$ Ausnahmeklauseln, die für zu fangende Ausnahmen E_i „Behandler“ H_i festlegen; eine Abschlussklausel F spezifiziert Erfolg, Fehlschlag oder weckt eine Ausnahme, falls keine Regel angewendet werden kann.

2.3.3 Universelle Behälter und Variablen

Es wäre zwar möglich, Darstellungen für „Basis“-Daten wie Zahlen oder Strings mithilfe von Graphen zu definieren, aber der zu erwartende Aufwand rechtfertigt weder den Nutzen noch die zu erwartende mühsame Handhabbarkeit seitens des Programmierers. Daher soll **Diaplan** zukünftig für solche Daten ebenfalls Behälter und Variablen zur Verfügung stellen; solche Objekte würden durch die Transformationsmaschine nur auf ihre Typzugehörigkeit geprüft; mögliche Operationen müssten unter Zuhilfenahme in Java eingebauter Methoden definiert werden. So könnte beispielsweise eine Kante, die das Zeichen „+“ enthält und zwei Knoten verbindet, die wiederum je eine ganze Zahl enthalten, auf eine noch zu definierende Weise mit der Java-Integeraddition verknüpft sein.

2.3.4 Kapselung bzw. Modularisierung

Um große Programme und Programmpakete zu strukturieren, sind Kapselungs- bzw. Modularisierungskonzepte erforderlich. Die theoretische Basis hierzu ist noch nicht sehr weit entwickelt, aber die Planungen sehen derzeit vor, selbige auf der Grundlage von speziellen *Behältertypen* bzw. -klassen zu definieren, so dass bestimmte Prädikate als Methoden nur bestimmten Behälter(typ)en zugeordnet werden. Für jeden Behältertyp soll eine eigene Gestaltdefinition analog einem Konstruktor existieren.

2.3.5 Nebenläufigkeit

Als objektorientierte Sprache benötigt Dia**plan** ebenfalls ein Konzept zur Nebenläufigkeit. Bisher gibt es zwar keine konkreten Arbeiten in diese Richtung, aber die Planung sieht ein ähnliches klassengebundenes Konzept wie in Java vor.

2.4 Verwandte Arbeiten

Dieser Abschnitt beleuchtet andere Formen der Graphtransformation als die bisher präsentierte. Unterabschnitt 2.4.1 beschreibt die zwei bekanntesten *algebraischen*[8] Modelle[9], *Double Pushout* und *Single Pushout*. Unterabschnitt 2.4.2 stellt ausgewählte Sprachbeispiele vor, die auf Graphtransformation basieren, wobei insbesondere auf die Unterschiede zu Dia**plan** eingegangen wird.

2.4.1 Andere Modelle zur Graphtransformation

Beide hier präsentierten Modelle nutzen den Begriff *Pushout*[10]: Seien A, B und C Graphen, seien $b : A \to B$ sowie $c : A \to C$ Morphismen. Dann ist ein Tripel $\langle D, g : B \to D, f : C \to D\rangle$ ein *Pushout* von $\langle b, c\rangle$, wenn gilt:

- **Kommutativität:** $g \circ b = f \circ c$, und
- **Universaleigenschaft:** Für alle Graphen D' und Morphismen $g' : B \to D'$ sowie $f' : C \to D'$ mit $g' \circ b = f' \circ c$ existiert ein eindeutiger Morphismus $h : D \to D'$, so dass gilt: $h \circ g = g'$ und $h \circ f = f'$.

Diese Eigenschaften veranschaulicht Abb. 2.1. D heißt *Pushout-Objekt* von $\langle b, c\rangle$. Sind zwei Morphismen $b : A \to B$ und $g : B \to D$ gegeben, so ist ein *Pushout-Komplement* von $\langle b, g\rangle$ ein Tripel $\langle C, c : A \to C, f : C \to D\rangle$, wobei $\langle D, g, f\rangle$ ein Pushout von $\langle b, c\rangle$ ist. C bezeichnet man als *Pushout-Komplementärobjekt* von $\langle b, g\rangle$.

[8] Die beiden Modelle heißen „algebraisch“, weil Graphen in diesem Kontext als Algebren betrachtet werden.

[9] Zur Vermeidung von Verwechslungen wird hier nicht wie sonst häufig der Begriff „Ansatz“ verwendet; dieser bleibt dem Bild einer linken Regelseite vorbehalten.

[10] Dieser Begriff entstammt der Kategorientheorie. Die hier vorgestellte Definition ist eine Spezialisierung für den Bereich der Graphtransformation.

$$
\begin{tikzcd}
(A) \arrow[r, "b"] \arrow[d, "c"] & (B) \arrow[d, "g"] \arrow[dr, "g'"] & \\
(C) \arrow[r, "f"] \arrow[rr, "f'"'] & (D) \arrow[r, dotted, "h"] & (D')
\end{tikzcd}
$$

Abbildung 2.1: Die Eigenschaften eines Pushout

Double Pushout (DPO)

Wie in Unterabschnitt 2.2.2 erwähnt geht die geschachtelte Kantenersetzung auf die „einfache“ Kantenersetzung aus [10] zurück; diese kann wiederum als Spezialfall des *Double Pushout*-Modells (DPO, [12]) verstanden werden, das in den frühen 70er Jahren des vergangenen Jahrhunderts an der Technischen Universität in Berlin entwickelt wurde.

Eine DPO-Regel besteht wie jede Graphtransformationsregel (s.a. Unterabschnitt 2.2.1) aus einer linken und einer rechten Seite; allerdings liegt beim DPO-Modell zusätzliches Augenmerk auf dem *Klebegraphen*, der Untergraph der linken und rechten Seite ist. Eine DPO-Regel bzw. DPO-*Produktion* p wird daher häufig in der Form $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ angegeben, wobei L die linke Seite, K den Klebegraphen, R die rechte Seite und l und r injektive Morphismen bezeichnen. Auf die explizite Angabe von K, l und r kann verzichtet werden, wenn diese aus dem Kontext (z.B. durch Nummerierung der Graphobjekte) eindeutig ersichtlich sind.

Kontaktbedingung. Seien $p : L \xleftarrow{l} K \xrightarrow{r} R$ eine DPO-Regel und G ein Graph. Dann erfüllt ein Ansatz $m : L \to G$ die *Kontaktbedingung*, wenn gilt: $\forall e \in (E_G - m_E(E_L)) : \ \forall v \in a(e) : \ v \notin m_V(V_L) - m_V(V_K))$, oder anders ausgedrückt: Wird bei der Anwendung von p ein Knoten gelöscht, so auch alle ihn besuchenden Kanten.

Identifikationsbedingung. Seien p und m definiert wie oben. Dann erfüllt m die *Identifikationsbedingung*, wenn gilt: $\forall o_1, o_2 \in (V_L - V_K) \cup (E_L - E_K) : m(o_1) = m(o_2) \Rightarrow o_1 = o_2$, d.h. m muss in Bezug auf die zu löschenden Objekte injektiv sein.

Kontaktbedingung und Identifikationsbedingung bilden zusammen die *Klebebedingung*. Ist diese nicht erfüllt, kann p mit dem Ansatz m nicht angewendet werden. Da **Diaplan**s Ansatzsuche auf das DPO-Modell zurückgeht, prüft der Interpreter auch die Kontaktbedingung; eine Verletzung der Identifikationsbedingung ist von vorneherein ausgeschlossen, da ein gefundener Ansatz stets isomorph zur linken Seite ist (vgl. Unterabschnitt 2.2.2) und somit zwei Graphobjekte nie miteinander identifiziert werden können[11].

[11] Diese Einschränkung ist vor allem bei Behältern sinnvoll, weil insbesondere unter Berücksichtigung von Gestaltdefinitionen (Unterabschnitt 2.3.1) die Verschmelzung zweier nichtleerer Behälter immense Probleme in sich birgt.

Eine direkte Ableitung mithilfe einer Regel $p : L \xleftarrow{l} K \xrightarrow{r} R$ und eines Ansatzes m wird für gewöhnlich mit einem Diagramm in folgender Form veranschaulicht:

$$\begin{tikzcd}
L \arrow[d, "m"] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*"] \\
G & D \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

Hierbei sind L, K und R die Bestandteile der Regel; G ist der Graph, auf den die Regel angewendet werden soll; D ist der sogenannte *Kontextgraph*, der durch Entfernen aller Graphobjekte aus G entsteht, die zwar Bilder von Ls, aber nicht von Ks Objekten sind; H schließlich ist der resultierende Graph, nachdem die Regel angewendet wurde.

Ein Vergleich mit Abb. 2.1 zeigt, dass $\langle G, m, l^* \rangle$ Pushout von $\langle l, d \rangle$ (1) und $\langle H, m^*, r^* \rangle$ Pushout von $\langle r, d \rangle$ (2) sind; der Name „Double Pushout“ erklärt sich also dadurch, dass sich dieses Diagramm in zwei Pushouts zerlegen lässt.

Single Pushout (SPO)

Das *Single Pushout*-Modell (SPO) zur Graphtransformation ist jünger als das DPO-Modell; die Regeln sind insofern aussagekräftiger als beim DPO, als bestimmte Seiteneffekte auftreten können, die bei letzterem ausgeschlossen sind. SPO-Regeln basieren nicht wie DPO-Regeln auf *totalen*, sondern auf *partiellen Morphismen*: Seien $G = \langle V_G, E_G, a_G \rangle$ und $H = \langle V_H, E_H, a_H \rangle$ (flache) Hypergraphen. Dann ist ein *partieller* Morphismus von G nach H ein totaler Morphismus eines Subgraphen *dom(g)* von G nach H; *dom(g)* heißt *Domäne* von g.

Eine SPO-Regel besitzt wie üblich eine linke und eine rechte Seite, die in diesem Fall durch einen *partiellen* Morphismus r miteinander in Beziehung stehen: $L \xrightarrow{r} R$. Die linke Seite L definiert, welche Objekte ein Graph enthalten *muss*, damit die Regel angewendet werden kann, also alle zu löschenden und den zu bewahrenden *Anwendungskontext*. Die rechte Seite R definiert das Aussehen des entsprechenden Graphteils nach Anwendung der Regel, also alle neu hinzugefügten Objekte zusammen mit dem Anwendungskontext. Die Korrespondenz zwischen Objekten auf der linken und rechten Seite wird dabei durch den Morphismus hergestellt. Objekte der linken Seite, für die der Morphismus undefiniert ist, werden gelöscht; der Rest bildet den Kontext. Dementsprechend werden Objekte der rechten Seite, die unter dem Morphismus kein Urbild in L haben, neu hinzugefügt. Der Anwendungskontext entspricht also in etwa dem DPO-Klebegraphen.

Die „praktischen“ Unterschiede zwischen DPO und SPO bestehen in ihrem Verhalten, wenn ein Ansatz die Kontakt- bzw. Identifikationsbedingung (s.o.) verletzt: Beim DPO wird die entsprechende Regel mit ebendiesem Ansatz nicht angewendet, beim SPO geschieht folgendes:

- Wird ein Knoten des Ansatzes von einer Kante besucht, die nicht zum Ansatz gehört (was im DPO einer Verletzung der Kontaktbedingung entspricht), wird diese gelöscht – obwohl sie nicht zum Bild der entsprechenden linken Regelseite gehört.

- Bildet der Ansatz zwei Knoten der linken Regelseite, von denen einer gelöscht werden soll, der andere jedoch nicht, auf den gleichen Knoten im Graphen ab (was im DPO eine Verletzung der Identifikationsbedingung darstellt), so wird der Löschung des Knoten der Vorzug gewährt.

Das SPO-Modell kennt also in seiner reinen Form keine zusätzlichen Anwendbarkeitsbedingungen, wenngleich in der Praxis oft welche bei Systemen auftauchen, deren theoretische Grundlage das SPO-Modell bildet. Ist ein Ansatz gefunden, kann mit der entsprechenden Regel stets eine direkte Ableitung durchgeführt werden; direkte SPO-Ableitungen bezeichnet man daher als *vollständig*. Das macht eine SPO-Regel zwar ausdrucksstärker als eine entsprechend formulierte DPO-Regel, kann aber bei der Anwendung leichter zu unbeabsichtigten Nebeneffekten führen. Dennoch ist das SPO-Modell bei implementierten Graphtransformationssystemen häufiger anzutreffen – z.B. basiert darauf das erste der drei im nächsten Unterabschnitt vorgestellten Systeme sowie die DiaGen-Transformationsmaschine (Kap. 4).

Aus den Nebeneffekten resultiert u.a. die Tatsache, dass SPO-Ableitungen im Gegensatz zu DPO-Ableitungen im Allgemeinen nicht umkehrbar sind ([13]), d.h. im SPO-Modell existiert im Allgemeinen keine „Undo"-Funktion. Zu jeder DPO-Ableitung $G \overset{p,m}{\Longrightarrow} H$ bestehend aus dem Ansatz m und der Produktion $p : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ existiert jedoch aufgrund der symmetrischen Struktur eines DPO-Ableitungsdiagramms eine inverse Ableitung $H \overset{p^{-1},m^*}{\Longrightarrow} G$ mit der inversen Produktion $p^{-1} : (R \overset{r}{\leftarrow} K \overset{l}{\rightarrow} L$ und dem Co-Ansatz $R \overset{m^*}{\rightarrow} H$ von $G \overset{p,m}{\Longrightarrow} H$ ([14]).

SPO- und DPO-Produktionen lassen sich verhältnismäßig einfach ineinander überführen: Sei $p : (L \overset{s}{\rightarrow} R$ eine SPO-Produktion. Dann beschreibt $\mathcal{D}(p) : (L \overset{l}{\leftarrow} dom(r) \overset{r}{\rightarrow} R)$ ihre Übersetzung in eine DPO-Produktion, wobei l die Inklusion von $dom(s)$ in L und r die Domänenbeschränkung von s auf $dom(s)$ bezeichnen. Sei umgekehrt $p : (L \overset{l}{\rightarrow} K \overset{r}{\rightarrow} R)$ eine DPO-Produktion. Dann beschreibt $\mathcal{S}(p) : (L \overset{s}{\rightarrow} R)$ ihre Übersetzung in eine SPO-Produktion, wobei gilt: $dom(s) = l(K)$ und $s = r \circ l^{-1}$. Wegen der Injektivität von l ist der partielle Morphismus s wohldefiniert. Die Übersetzung der Regeln lässt sich jedoch nicht auf die Ableitungen übertragen; zwar können problemlos alle DPO-Ableitungen ins SPO-Modell abgebildet werden, aber nicht umgekehrt.

2.4.2 Andere Graphsprachen

Das erste der hier vorgestellten Graphtransformationssysteme basiert auf dem Single-Pushout-Verfahren; die anderen beiden verfolgen jeweils ein eigenes Transformationsmodell. Es wird jeweils ein kurzer Überblick über die wesentlichen Bestandteile der zugrundeliegenden Graphen sowie der jeweiligen Transformationsmechanismen gegeben, wobei das Augenmerk auf Gemeinsamkeiten mit und Unterschieden zu Dia**plan** liegt.

AGG-Graphen

Während Dia**plan**s hierarchische Graphen auf dem Hypergraphkonzept basieren, liegt den Graphen des Algebraischen Graphgrammatik-Systems der TU

Berlin[12], kurz `AGG`, das geläufigere Modell des markierten gerichteten binären Graphen zu Grunde. Ein solcher Graph ist ein Tupel $G = \langle V, E, L, s, t, l\rangle$ aus zwei disjunkten endlichen Mengen V von Knoten und E von Kanten, zwei totalen Abbildungen $s, t : E \to V$ (s steht für „source“ bzw. „Quelle“, t für „target“ bzw. „Ziel“), einer Menge von Markierungen („Labels“) L und einer totalen Abbildung $l : V \cup E \to L$. Ein Morphismus $m : G \to H$ zwischen zwei solchen Graphen $G = \langle V_G, E_G, L_G, s_G, t_G, l_G\rangle$ und $H = \langle V_H, E_H, L_H, s_H, t_H, l_H\rangle$ ist ein Paar von partiellen Abbildungen $m = \langle m_V : V_G \to V_H, m_E : E_G \to E_H\rangle$, für das gilt:

1. $\forall v \in dom(m_V) : l_G(v) = l_H(m_V(v))$
2. $\forall e \in dom(m_E) : l_G(e) = l_G(m_E(e))$
3. $\forall e \in dom(m_E)$:
   - $s_G(e) \in dom(m_V) \wedge m_V(s_G(e)) = s_H(m_E(e))$
   - $t_G(e) \in dom(m_V) \wedge m_V(t_G(e)) = t_H(m_E(e))$

Attributierung. Zur Definition der *Attributierung*[13] werden die Begriffe *Signatur* und *Algebra zur Signatur* herangezogen[14]: Eine Signatur $Sig = (S, OP)$ bildet den syntaktischen Rahmen für eine Klasse $Alg(Sig)$ von Algebren zur Signatur. Sie besteht aus einer Menge S von *Sorten*- und einer Menge OP von *Operationssymbolen*. Jede Algebra aus $Alg(Sig)$ „implementiert“ die Sortensymbole der Signatur durch Wertemengen, sog. *Trägermengen*, und die Operationssymbole durch entsprechende Operationen. Ein Sig-Homomorphismus zwischen zwei Sig-Algebren ist eine Familie von operationsverträglichen Abbildungen zwischen den beiden Trägermengen.

Seien nun A eine Algebra zu einer Signatur $AS = (S, OP)$ und G ein markierter gerichteter binärer Graph. Ein Tupel $G_{AS} = \langle V, E, L, s, t, l, A, at, av\rangle$ mit einer totalen Abbildung $at : L \to S$ (at steht für „attribute type“ bzw. „Attributtyp“) und einer totalen Abbildung $av : V \cup E \to AV$ (av steht für „attribute value“ bzw. „Attributwert“), wobei $AV = \biguplus_{s \in S} A_s$ die disjunkte Vereinigung aller Trägermengen von A bildet, heißt *attributierter Graph*, wenn für $al : AV \to S$ mit $\forall v \in A_s : al(v) = s$ folgendes Diagramm kommutiert:

$$\begin{tikzcd}
(L) \arrow[r, "at"] & (S) \\
(V \cup E) \arrow[u, "l"] \arrow[r, "av"] \arrow[ur, phantom, "(=)"] & (AV) \arrow[u, "al"']
\end{tikzcd}$$

Die *Attributierungsfunktion av* ordnet jedem Graphobjekt genau einen Attributwert zu. Benötigt ein Graphobjekt mehrere Attribute, so sind diese in einem Tupel zusammenzufassen.

Diesem Attributierungskonzept entsprechen in **Diaplan** zukünftig die universellen Behälter (Unterabschnitt 2.3.3). Statt wie in bisherigen Veröffentlichungen (z.B. [18]) angedeutet, „nur“ Java-Basistypen wie `int` (bzw. `Integer`) oder

[12] `http://tfs.cs.tu-berlin.de/agg/`

[13] Die hier präsentierte Definition entspricht der aus [15]. Die jüngere in [16] setzt die Attribute direkt als Graphobjekte um, legt aber einen unendlichen Graphen zu Grunde, weswegen von der Betrachtung dieses Modells Abstand genommen wird. Die Erläuterungen zu den übrigen Konzepten sind von diesem Unterschied nicht betroffen.

[14] Beide Begriffe werden hier nur informell definiert. Genaue Definitionen finden sich in [17].

`String` vorzusehen, könnten ebenso gut beliebige durch den Nutzer definierte Klassen zugelassen werden; Prädikate könnten dann über die „einfache" Graphtransformation hinaus Operationen auf diesen Klassen ausführen.

Hierarchiebegriff. Auch das AGG-System kennt einen Hierarchiebegriff für Graphen, den *AGG-Graph* ([19]): Ein AGG-Graph

$$G_{AGG} = \langle O, \perp, P; a, s, t : O \to O, p : O \backslash \{\perp\} \to P, a_P : P \to P \cup \{\perp\} \rangle$$

besteht aus zwei disjunkten, endlichen, nichtleeren Mengen O und P (für „object set"/"Objektmenge" bzw. „plane set"/"Ebenenmenge"), einem ausgezeichneten Objekt $\perp \in O$, genannt *Bottom*, sowie fünf totalen Abbildungen a,s,t,p und a_p (für „abstraction"/"Abstraktion", „source", „target", „plane"/"Ebene" und „plane abstraction"/"Ebenenabstraktion"), so dass folgende Bedingungen erfüllt sind:

1. (Eigenschaften des Bottom-Elements)
   $a(\perp) = s(\perp) = t(\perp) = \perp$
2. (Knoten oder Kante)
   $\forall o \in O : s(o) = \perp \Leftrightarrow t(o) = \perp$
3. (Knoten mit Knoten als Abstraktion)
   $\forall o \in O : s(o) = \perp, a(o) \neq \perp \Rightarrow s(a(o)) = \perp$
4. (Kante mit Knoten als Abstraktion)
   $\forall o \in O : s(o) \neq \perp, a(o) \neq \perp, s(a(o)) = \perp \Rightarrow a(s(o)) = a(t(o)) = a(o)$
5. (Kante mit Kante als Abstraktion)
   $\forall o \in O : s(o) \neq \perp, s(a(o)) \neq \perp \Rightarrow a(s(o)) = s(a(o)) \wedge a(t(o)) = t(a(o))$
6. (Kante, Quelle und Ziel auf gleicher Ebene)
   $\forall o \in O : s(o) \neq \perp \Rightarrow p(o) = p(s(o)) = p(t(o))$
7. (lineare Ebenenstruktur)
   $\forall p_1, p_2 \in P : a_p(p_1) = a_p(p_2) \Rightarrow p_1 = p_2$
8. (Abstraktion nur in die nächsthöhere Ebene)
   $\forall o \in O : a(o) \neq \perp \Rightarrow p(a(o)) = a_p(p(o))$
9. (Zyklenfreiheit der Abbildungen a, s, t, a_p)
   $\neg\exists n \in \mathbb{N} : \exists (o_0, o_1, \ldots, o_n) \in O \backslash \{\perp\} : (o_0, o_1), (o_1, o_2), \ldots, (o_{n-1}, o_n), (o_n, o_0) \in a \cup s \cup t$
   $\neg\exists n \in \mathbb{N} : \exists (p_0, p_1, \ldots, p_n) \in P : (p_0, p_1), (p_1, p_2), \ldots, (p_{n-1}, p_n), (p_n, p_0) \in a_p$

Diese Graphdefinition verwischt die Grenzen zwischen Knoten und Kanten; beide gehören zur Menge der Graphobjekte O und lassen sich nur durch die Betrachtung in Punkt 2 unterscheiden: Knoten werden durch s und t stets auf das Bottom-Element abgebildet, während Kanten auf alle Objekte außer $\perp$ abgebildet werden können. Daraus folgt, dass Kanten nicht nur zwei Knoten, sondern auch zwei Kanten oder einen Knoten und eine Kante verbinden können.

Die Hierarchie bildet genau wie bei Dia**plan** eine Baumstruktur, die jedoch im Gegensatz zu letzterer von den Blättern zur Wurzel, nämlich dem Bottom-Element aufgebaut wird. Zudem können zwar Knoten vollständige Graphen „enthalten"[15], Kanten jedoch nur Kanten. Den Sinn dieser Darstellung verdeutlicht Abb. 2.2: Gegeben sei eine Graphtransformationsregel mit linker Seite L1, L2, L3, k1 und rechter Seite R1, R2, k2. Der Morphismus wird durch die gestrichelten Kanten k3-k5 ausgedrückt. Die nächste Abstraktionsebene kennt nur noch einen Knoten für je eine Seite und eine Kante, die den gesamten Morphismus zwischen diesen beiden darstellt.

[15] Im AGG-Kontext spricht man gewöhnlich nicht von Behältern und enthaltenen Objekten, sondern von *Abstraktionen* und *Verfeinerungen*.

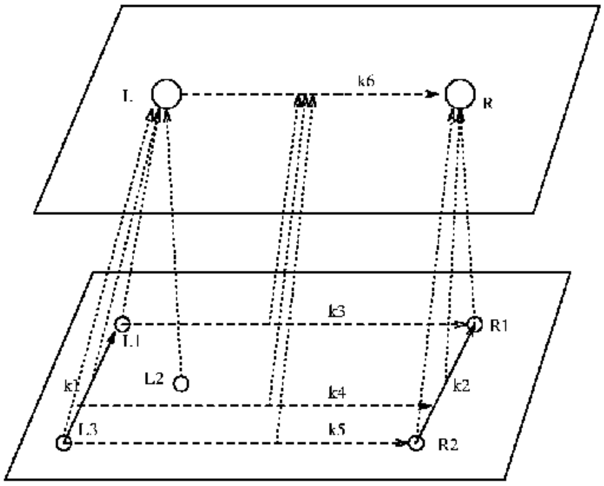


Abbildung 2.2: Graphregel als AGG-Graph (aus [19])

Der wichtigste Unterschied zu **Diaplan** besteht jedoch darin, dass die Graphhierarchie nur systemintern zur Datenstrukturierung und -verwaltung dient. Der Nutzer verwendet das System, als würde es nur auf flachen Graphen arbeiten. Dagegen basieren in **Diaplan** sowohl Daten als auch Operationen auf der Hierarchiestruktur. Zudem ist der Aufbau der Hierarchie von oben nach unten, also vom Allgemeinen ins Spezielle, intuitiver für Programmieraufgaben als entgegengesetzt. Für AGG, das in erster Linie als Werkzeug für Rapid Prototyping gedacht ist, sind diese Eigenschaften unwichtig; **Diaplan** jedoch ist als „vollwertige“ objektorientierte Programmiersprache angelegt, was diesen Abstraktions- und Strukturierungsmechanismus erforderlich wie sinnvoll macht.

PROGRES-Graphschemata und -Ausdrücke

Die Sprache PROGRES (**PRO**grammierte **GR**aph**E**rsetzungs**S**ysteme), ein Projekt der RWTH Aachen[16], dessen vielfältige Möglichkeiten und Sprachbestandteile hier nur im Überblick dargestellt werden können, verwendet das gleiche Basisgraphmodell des gerichteten knoten- und kantenmarkierten binären Graphen, wobei PROGRES im Gegensatz zu AGG keine zwei Kanten des gleichen Typs zwischen denselben Knoten erlaubt.

Graphschemata. Anders als in AGG bzw. **Diaplan** dient in PROGRES kein Typgraph (Unterabschnitt 2.1.3) zur Typisierung der Graphobjekte eines Datengraphen, sondern ein sog. *Graphschema.* In einem Graphschema existieren für Knoten *Knotentypen* und *Knotenklassen.* Dabei gehören jeder Knoten zu genau einem Knotentyp und jeder Knotentyp zu genau einer Knotenklasse, die Vererbungsbeziehungen, mögliche Attribute und Typen von möglichen einen Knoten besuchenden Kanten festlegt. Dieses zweistufige Typkonzept ist erforderlich, um Knotentypen, die als Werte von Variablen, Attributen, Parametern und Ausdrücken auftreten können, wiederum einen Typ zuweisen zu können

[16] `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`

(s.a. [20]). In Knotentypen werden üblicherweise keine neuen Eigenschaften definiert. Sie dienen in der Praxis dazu, Knotenklassen auszuwählen, von denen tatsächlich Knoteninstanzen gebildet werden können ([21]); Knotenklassen, zu denen keine Knotentypen existieren, bezeichnet man als *abstrakte* Klassen. Der PROGRES-Vererbungsmechanismus erlaubt Mehrfachvererbung.

Ebenso wie Knotenklassen definieren, welche Art Kanten die Knoten besuchen dürfen, definieren *Kantentypen*, welcher Knotenklasse Quelle und Ziel einer Kante angehören müssen. Zusätzlich erlauben *Kardinalitäten* eine Festlegung, wie viele Kanten des gleichen Typs denselben Knoten maximal besuchen dürfen bzw. mindestens besuchen müssen[17]. So lässt sich zum einen die Aussagekraft einer Kantentypdeklaration erhöhen, zum zweiten geben die Kardinalitäten Optimierungshinweise für den PROGRES-Interpreter.

Dieses Typkonzept ist gegenüber **Diaplan** insbesondere in Bezug auf die Knoten wesentlich ausgefeilter, kann einen unerfahrenen Anwender jedoch leicht verwirren.

Attribute. PROGRES unterscheidet drei Arten von Attributen:

- *intrinsic*-Attribute beschreiben *lokale Variablen* eines Knotens. In solchen Attributen können in einem Knoten zusätzliche Informationen abgelegt werden, die sich nicht direkt aus der Graphstruktur ergeben (z.B. der Name einer durch die entsprechende Knoteninstanz repräsentierten Person). Die Werte dieser Attribute können im Zuge eines Transformationsschritts gelesen und modifiziert werden. Man kann bereits bei der Deklaration eines solchen Attributs innerhalb der entsprechenden Knotenklasse eine *Initialisierungsvorschrift* angeben; diese kann sowohl aus einem konstanten Wert als auch aus einem arithmetischen Ausdruck oder beliebigen Funktionsaufruf bestehen, solange der Ausdruck bereits zur Kompilierzeit ausgewertet werden kann.

- *meta*-Attribute stellen während des Transformationsvorgangs Konstanten dar. Innerhalb der Vererbungshierarchie ist es jedoch möglich, in einer Kindklasse die Initialisierungsvorschrift eines solchen Attributs neu zu definieren. Dementsprechend ist es bei abstrakten Klassen nicht erforderlich, einem `meta`-Attribut gleich bei der Deklaration einen Wert zuzuweisen.[18] Nur bei Klassen, von denen Knotentypen (und damit auch Knoteninstanzen) direkt gebildet werden können sollen, muss für alle `meta`-Attribute ein zur Kompilierzeit berechenbarer Wert definiert werden. Bei einer Initialisierungsvorschrift für ein `intrinsic`-Attribut kann in einer Klasse *immer* auf alle `meta`-Attribute zurückgegriffen werden, auch wenn deren Werte erst weiter unten in der Vererbungshierarchie gesetzt werden.

- *derived*-Attribute schließlich können in ihren Berechnungsvorschriften beliebig auf Attribute anderer Knoten im Graphen zugreifen. Ähnlich wie bei `meta`-Attributen werden ihre Werte bei einer graphverändernden

[17]Beim PROGRES-Transformationsmodell können ähnlich wie beim SPO-Modell Kanten gelöscht werden, ohne direkt einem Regelansatz anzugehören; daher lassen sich die durch Kardinalitäten festgelegten Bedingungen nur zum Teil statisch überprüfen.

[18]Mit Hilfe dieser Attribute lässt sich also eine Knotenklasse in einem Graphschema als abstrakt sichern, indem man für die Klasse wenigstens ein `meta`-Attribut ohne sofortige Wertzuweisung deklariert.

Operation nie explizit gesetzt, können jedoch im Unterschied zu diesen trotzdem durch eine Regelanwendung beeinflusst werden. Wie der Name „`derived`“ schon andeutet, errechnen sich diese Attribute aus anderen, die in diesem Fall zu einem beliebigen Knoten im Graphen gehören können, solange dieser über eine beliebig lange Kantenfolge erreichbar ist. Da es in PROGRES stets möglich ist, Kanten sowohl in als auch gegen ihre Richtung zu traversieren[19], ist es dabei unerheblich, wie diese Kanten gerichtet sind. Während des Transformationsvorgangs wird der Wert eines `derived`-Attributs beim ersten Lesezugriff berechnet und im entsprechenden Knoten gespeichert. Ändert sich die zugrunde liegende Graphstruktur, werden alle möglicherweise von dieser Änderung betroffenen `derived`-Attribute als „ungültig“ definiert und entsprechend beim nächsten Lesezugriff neu berechnet. Der Einsatz dieser Attribute ist aus Gründen der Effizienz also in erster Linie dort sinnvoll, wo zwischen Transformationsschritten mehrere Lesezugriffe zu erwarten sind.

Wie schon bei AGG erwähnt sollen in Dia**plan** Attribute, die von der Art her den `intrinsic`-Attributen ähneln, obgleich auch Konstanten ähnlich den `meta`-Attributen zumindest denkbar wären, zukünftig in Form der universellen Behälter umgesetzt werden können, wobei Dia**plan** diese Möglichkeit auch für Kanten anbieten wird.

Attributausdrücke. Für die drei fest eingebauten Attributtypen `boolean`, `integer` und `string` stellt PROGRES die üblichen Operationen wie z.B. `and`, `or`, `not`, Vergleichsoperatoren, arithmetische Grundrechenarten, Konkatenation und Konvertierung von Werten in Strings oder umgekehrt zur Verfügung. Zusätzlich kann man in Programmiersprachen wie Modula-2 oder C implementierte Datentypen zusammen mit den benötigten Funktionen importieren.

Des Weiteren erlaubt PROGRES die Definition eigener Funktionen und stellt eine mengenwertige Arithmetik für Berechnungen über mehrere Kanten zur Verfügung, die in ihren Möglichkeiten an die zum UML-Standard gehörende Object Constraint Language (OCL, [5]) heranreicht: Es existieren Operatoren für Durchschnitt, Vereinigung und Differenz von Mengen, Teilmengentest, die beiden prädikatenlogischen Existenz- und Allquantoren und ein Operator `all`, der eine zweistellige Funktion iterativ auf alle Elemente einer Menge anwendet. Diese mengenwertige Arithmetik wird dort benötigt, wo ein Knoten von mehreren Kanten gleichen Typs besucht werden kann; wird nämlich in einer Funktions- bzw. Attributdefinition auf eine Kante dieses Typs zurückgegriffen, so erhält man die Menge aller möglichen erreichbaren Knoten als Antwort.

Ähnliche Operatoren werden in Dia**plan** voraussichtlich nur eingeschränkt eingeführt werden, da sich die Gegenstücke zu den Attributen stark auf Java stützen und dementsprechend die dort bereits vorhandenen Methoden nutzen werden.

Regeln. Die PROGRES-Regeln ähneln den AGG-Regeln; allerdings stellt PROGRES einige Programmierkonstrukte zur Verfügung, mit denen sich verhältnismäßig einfache Regeln zu größeren Einheiten kombinieren lassen, die sich in dieser Form nur schwer auf reiner Graphebene ausdrücken ließen:

[19] Die tatsächliche Kantenrichtung hängt vom intuitiven Modellverständnis ab.

- Mit dem Konkatenationsoperator & lassen sich mehrere Regeln einfach hintereinander schalten. Z.B. wird bei einer Regel `Prod1 & Prod2 & Prod3` erst nach erfolgreicher Anwendung von `Prod1` mit der Ansatzsuche für `Prod2` begonnen und erst nach erfolgreicher Anwendung beider Regeln mit der Ansatzsuche für `Prod3`. Nur wenn alle drei angewendet werden konnten, gilt die gesamte Konstruktion als angewendet; anderenfalls erfährt der Graph durch dieses Konstrukt gar keine Veränderung.

- Ersetzt man den Operator & durch `and`, so müssen ebenfalls alle Einzelregeln angewendet werden können, allerdings in beliebiger Reihenfolge.

- Der Operator `or` dient zur Modellierung einer Fallunterscheidung: Wenn irgendeine der mit `or` verbundenen Regeln erfolgreich angewendet werden kann, ist das gesamte Konstrukt erfolgreich. Scheitert eine der nachgeordneten Regeln, probiert der PROGRES-Interpreter alle anderen Alternativen durch; erst wenn alle scheitern, scheitert auch deren `or`-Verbindung.

- Eine Fallunterscheidung mit Reihenfolge bietet der Befehl `choose`:

  ```
  choose
    Prod1
  else
    when Test2 then Prod2
  else
    Prod3
  end
  ```

  Die Alternativen eines `choose`-Befehls werden in der angegebenen Reihenfolge durchprobiert, bis eine anwendbare gefunden wird. Wie das `or`-Konstrukt scheitert auch eine `choose`-Fallunterscheidung, falls keine Alternative sich als anwendbar herausstellt. Eine mit `when` eingeleitete Alternative bezeichnet man als *bewachte* Alternative, den ersten Teil `when Test` als *Wächter*. Scheitert die Ausführung des Wächters, fährt das gesamte `choose`-Konstrukt wie gehabt mit der nächsten Alternative fort. Gelingt jedoch die Anwendung, gilt die entsprechende Alternative unwiderruflich als ausgewählt, unabhängig vom Erfolg des `then`-Teils; scheitert dessen Anwendung, so gilt der gesamte `choose`-Befehl als gescheitert.

- Eine Schleife kann mit dem `loop`-Befehl erstellt werden:

  ```
  loop
    Prod1
  end
  ```

  Der Schleifenrumpf wird angewendet, solange dies möglich ist (also u.U. gar nicht). Der Anwender muss darauf achten, dass die Anwendung des Rumpfs keinen neuen Ansatz für selbigen produziert, da der PROGRES-Interpreter sonst bereits nach der ersten erfolgreichen Anwendung in eine Endlosschleife übergeht.

- Die einfache Form des `loop`-Konstrukts ist immer erfolgreich, auch wenn der Schleifenrumpf gar nicht angewendet werden kann. Der Schleifenrumpf kann jedoch Bedingungen ähnlich dem `choose`-Befehl enthalten:

```
loop
  Prod1
else
  when Test2 then Prod2
else
  Prod3
end
```

In jedem Schleifendurchlauf werden wie beim `choose`-Befehl alle Alternativen geprüft. Solange eine Alternative sich als erfolgreich herausstellt, setzt die Ausführung der Schleife fort. Analog zum `choose`-Befehl scheitert die Schleife jedoch, wenn der Wächter einer bewachten Alternative erfolgreich angewendet wird, der `then`-Teil jedoch nicht. In diesem Fall gilt die Schleife insgesamt als gescheitert - inklusive aller bisherigen Durchläufe, d.h. deren Auswirkungen müssen ggf. rückgängig gemacht und dort andere Alternativen ausgewählt werden.

Die „bewachten Alternativen" ähneln Dia**plan**s bedingten Regeln; die übrigen (textbasierten) Programmierkonstrukte lassen sich mit Hilfe modifizierter Versionen der in [6] vorgestellten *Kontrollkombinatoren*, einer Menge von speziellen Prädikaten, nachbilden.

Hierarchiekonzept. Obwohl bereits Zündorf ([21]) die Einführung eines Hierarchiekonzepts in PROGRES anregt, arbeitet die Sprache genau wie AGG bis heute mit flachen Graphen, die wie oben erläutert sehr schnell zu unübersichtlichen „Monstern" anwachsen können. So ist genau wie bei AGG für den Anwender ab einer gewissen Größe die Struktur eines Graphen nur noch schwer überschaubar.

GReAT – UML-basierte Graphtransformation

Die Sprache GReAT (**G**raph **Re**writing **A**nd **T**ransformation) vom Institute for Software Integrated Systems (ISIS) der Vanderbilt University[20] entstand im Rahmen des ebenfalls dort entwickelten Softwareentwicklungskonzepts *Model Integrated Computing* ([22]). Die Hauptaufgabe, für die das Werkzeug entwickelt wurde, ist, Modelle, die einem bestimmten Metamodell zu entsprechen, in Modelle eines anderen Metamodells zu überführen. Weil diese Modelle durch Graphen repräsentiert werden, ist GReAT ebenfalls ein Graphtransformationswerkzeug.

Graphbegriff. Da wie erwähnt im GReAT-Kontext die Begriffe „Graph" und „Modell" synonym gebraucht werden, dient als Basis für einen solchen Graphen kein Typgraph (wie in Dia**plan** oder AGG) oder Graphschema (wie in PROGRES), sondern ein als UML-Klassendiagramm ([5]) spezifiziertes *heterogenes Metamodell*, das die beiden Metamodelle des Ausgangs- und Zielgraphen zu einem neuen verbindet, so dass auch Assoziationen zwischen Klassen *beider* Metamodelle zulässig sind ([23]).

GReAT definiert einen *Knoten* v als ein Paar $\langle class, attrs \rangle$ bestehend aus einer UML-Klasse $class$ aus dem heterogenen Metamodell und einer Abbildung $attrs$, die jedem in $class$ deklarierten Attribut einen Wert zuordnet.

[20] `http://www.isis.vanderbilt.edu/Projects/mobies/default.html`

Eine *Kante* e ist in GReAT ein Tripel $\langle etype, src, dst\rangle$, wobei $etype$ die Assoziation ist, zu der die Kante gehört. Eine Assoziationsklasse wird als ein Knoten repräsentiert, der je eine Kante zu Instanzen der beiden an der Assoziation teilnehmenden Klassen erhält. Die Bestandteile src und dst sind die beiden von der Kante besuchten Knoten[21]. Die Klassen dieser beiden Knoten müssen selbstverständlich mit den von $etype$ besuchten übereinstimmen bzw. von diesen abgeleitet sein.

Ein *Graph* G ist dann einfach ein Paar $\langle V_G, E_G\rangle$, wobei V_G die Menge der Knoten und E_G die Menge der Kanten darstellen und folgende Beziehung gilt:

$$\forall e = \langle etype, src, dst\rangle \in E_G : src \in V_G \wedge dst \in V_G.$$

Damit ist auch hier das Graphmodell der attributierte binäre Graph, allerdings in der unmarkierten Variante. Ebenso wie PROGRES verbietet GReAT zwei Kanten des gleichen Typs zwischen denselben Knoten. In UML-Begriffen sind ein Knoten ein Objekt, eine Kante ein Link und ein Graph ein Objektdiagramm.

Musterspezifikation. Für die Ansatzsuche stehen drei Arten von Suchmustern zur Verfügung:

- *Einfache* Suchmuster repräsentieren einen *exakt* zu suchenden Subgraphen, entsprechen also den linken Regelseiten der „klassischen“ Graphtransformation.

- Suchmuster *fester Kardinalität* sind eine vereinfachte Schreibweise für Suchmuster, in denen kleinere Einheiten mehrfach auftreten. Soll z.B. ein Muster aus einem Knoten der Klasse A bestehen, der mit drei Knoten der Klasse B über jeweils eine Kante der Assoziation AB verbunden ist, so genügt als Suchmuster je ein Knoten beider Klassen und eine Kante zwischen den beiden, wobei der Knoten der Klasse B mit der Kardinalität $<< 3 >>$ versehen wird. Eine im Suchmuster mit einer Kardinalität versehene Kante steht für entsprechend viele Kanten zwischen denselben Knoten im Modell. Mehrfach auftretende *Submuster*, die beliebig aus evtl. ebenfalls mit Kardinalitäten versehenen Knoten und Kanten zusammengesetzt sein können, lassen sich ebenfalls als Einheit zusammenfassen und mit einer Kardinalität versehen.

- Suchmuster *variabler Kardinalität* fassen ebenfalls beliebige Untereinheiten eines Suchmusters zusammen, versehen sie aber nicht mit einer fixen Kardinalität, sondern einem *Wertebereich*, der das minimale und maximale Auftreten des Submusters in einem Ansatz angibt. Analog zu UML-Kardinalitäten kann die untere Grenze minimal 0, die obere Grenze maximal $*$ ($\widehat{=}$ „beliebig oft“) sein.

Graphkomponenten in solchen Suchmustern benötigen also zusätzliche Eigenschaften: Ein *Musterknoten* pv ist ein Paar $\langle class, card\rangle$ bestehend aus einer Klasse $class$ aus dem heterogenen Metamodell und einem Paar $card = \langle l, h\rangle$, wobei l und h für die untere bzw. obere Grenze des Wertebereichs für die Kardinalität stehen. Falls $l = h$, ist die Kardinalität konstant. Entsprechend ist

[21] Das Graphmodell unterstützt nur binäre Assoziationen.

eine *Musterkante* pe ein Quadrupel $\langle etype, src, dst, card \rangle$ bestehend aus der Assoziation $etype$, zu der die Kante gehört, zwei Musterknoten src und dst und einem wie beim Musterknoten definierten Paar $card$. Wie bei den „einfachen" Graphkomponenten müssen src und dst Instanzen der an der Assoziation teilnehmenden Klassen bzw. davon abgeleiteter sein. Ein *Mustergraph* PG ist dann ein Paar $\langle PV_{PG}, PE_{PG} \rangle$ aus einer Menge PV_{PG} von Musterknoten und einer Menge PE_{PG} von Musterkanten, für die gilt:

$$\forall pe = \langle etype, src, dst, card \rangle \in PE_{PG} : src \in PV_{PG} \wedge dst \in PV_{PG}$$

Für Eigenschaften, die sich nicht über die Graphstruktur ausdrücken lassen (wie z.B. ein Wertebereich für ein bestimmtes Attribut eines Knotens), können mit Hilfe von OCL sog. *Constraints* definiert werden.

Die Abbildungen dieser Musterkomponenten in einem konkreten Graphen sind mengenwertig: Eine *Knotenbindung* vb ist ein Paar $\langle pv, HV \rangle$ bestehend aus einem Musterknoten pv und einer Menge HV von Knoten im zu transformierenden Graphen. Analog ist eine *Kantenbindung* eb ein Paar $\langle pe, HE \rangle$ bestehend aus einer Musterkante pe und einer Menge HE von Kanten im zu transformierenden Graphen. Ein *Ansatz* m ist dann ein Paar $\langle VB_m, EB_m \rangle$ bestehend aus einer Menge VB_m von Knoten- und einer Menge EB_m von Kantenbindungen, der folgende Bedingungen erfüllt:

$$\begin{aligned} &\forall eb \in EB_m \text{ mit } eb = \langle pe, HE \rangle, pe = \langle pet, psrc, pdst, card \rangle, \\ &\forall he \in HE, he = \langle het, hsrc, hdst \rangle\ \exists vbs \in VB_m \wedge vbd \in VB_m : \\ &\qquad vbs = \langle psrc, hsrc \rangle \wedge vbd = \langle pdst, hdst \rangle. \end{aligned} \tag{2.1}$$

$$\forall eb \in EB_m \text{ mit } eb = \langle pe, HE \rangle, pe = \langle pet, psrc, pdst, \langle l, h \rangle \rangle : \quad l \leq |HE| \leq h. \tag{2.2}$$

$$\forall vb \in VB_m \text{ mit } vb = \langle pv, HV \rangle, pv = \langle pclass, \langle l, h \rangle \rangle : \quad l \leq |HV| \leq h. \tag{2.3}$$

Bedingung (2.1) stellt sicher, dass zu jeder Kantenbindung Knotenbindungen für Ursprung und Ziel der entsprechenden Musterkante existieren; die Bedingungen (2.2) und (2.3) sind die Kardinalitätsbedingungen für Kanten bzw. Knoten.

Suchmuster fester Kardinalität sind letztendlich nichts weiter als Schreiberleichterungen; Suchmuster variabler Kardinalität können in Dia**plan** durch Variablen und geeignet gewählte Gestaltdefinitionen simuliert werden. Zusätzliche Bedingungen wie die sehr mächtigen OCL-Constraints können zwar nicht angegeben werden, dafür ist die hierarchische Graphstruktur aber aussagekräftiger als die eines GReAT-Graphen.

Regeln. Anders als bei so gut wie allen Graphtransformationswerkzeugen bestehen *Regeln* in GReAT nicht aus zwei verschiedenen Graphen für linke und rechte Seite, sondern aus einem einzigen Mustergraph, in dem jedes Objekt über ein *Rollenattribut* verfügt. Dieses Attribut kann einen von drei verschiedenen Werten annehmen:

Binden Das Musterobjekt wird auf ein oder mehrere Graphobjekte abgebildet, die jedoch durch den Transformationsschritt nicht beeinflusst werden.

Löschen Das Musterobjekt wird auf ein oder mehrere Graphobjekte abgebildet, die durch den Transformationsschritt gelöscht werden.

Neu Nachdem der Ansatz gebildet wurde, werden neue Graphobjekte erzeugt, die dem Musterobjekt entsprechen.

Enthält ein Ansatz ein Graphobjekt in der Rolle „Löschen“, für das wenigstens ein anderer ebenfalls eine Bindung enthält, gilt die Regel mit ersterem als nicht anwendbar.

Formal ist eine Regel bzw. Produktion p in GReAT ein Quadrupel $\langle PG, pr : PV_{PG} \cup PE_{PG} \rightarrow Ro, gd, AM \rangle$ bestehend aus

- einem Mustergraphen PG,
- einer Abbildung pr, die jedem Musterobjekt aus PG ein Element der Menge $Ro = \{\text{„Binden“}, \text{„Löschen“}, \text{„Neu“}\}$ zuordnet,
- einem booleschen Ausdruck gd (für „guard“), der sich aus den Attributen der Graphobjekte ergibt, und
- einer Menge von Zuweisungen AM (für „attribute mapping“), deren Elemente den Attributen der Graphobjekte Werte zuweisen.

Ist der Ausdruck gd durch die Attribute der Graphobjekte nicht erfüllt, so wird die Regel nicht angewendet.

Eine nur aus einem Mustergraphen bestehende Regel lässt sich problemlos in eine „klassische“ Graphtransformationsregel mit linker und rechter Seite umwandeln:

1. Objekte mit der Rolle „Binden“ kommen in beiden Regelseiten vor.
2. Objekte mit der Rolle „Löschen“ kommen nur in der linken Regelseite vor.
3. Objekte mit der Rolle „Neu“ kommen nur in der rechten Regelseite vor.

Die Grundform der Regeln lässt sich also auch in Dia**plan** als unbedingte Regel ausdrücken. Für den zusätzlichen „Wächterausdruck“ gd gilt ähnlich wie für oben erwähnte OCL-Constraints, dass er in Dia**plan** nicht direkt abbildbar ist, da Dia**plan**s Regeln nur die Graphstruktur bearbeiten.

Kontrollfluss. Um die Anwendungsreihenfolge der „reinen“ Graphtransformationsregeln zu steuern, bietet GReAT folgende Kontrollflusskonzepte an:

Sequenz Mehrere Regeln können zu einer großen hintereinander geschaltet werden.

Nichtdeterminismus Es ist möglich, Regeln so zu spezifizieren, dass sie parallel ausgeführt werden. „Parallel“ bedeutet hier, dass die Reihenfolge der Ausführung unbestimmt ist.

Hierarchie Sog. `Compound`-Regeln können Regeln enthalten.

Rekursion Eine Regel kann sich selbst rekursiv aufrufen.

Fallunterscheidung Mit Hilfe einer bedingten Verzweigung kann zwischen verschiedenen Kontrollflusspfaden gewählt werden.

Mit Ausnahme des Nichtdeterminismus’, der ja durch die Strategie der strikten Tiefensuche gerade so weit wie möglich ausgeschlossen werden soll, sind diese Konzepte nach Dia**plan** überführbar:

- Die *Hintereinanderausführung* zweier Prädikate wird dadurch erreicht, dass die Anwendung des ersten eine Prädikatkante für das zweite generiert.

- Dia**plan** besitzt zwar keine „hierarchischen“ Regeln, aber die durch dieses Konstrukt in GReAT erzielbaren Effekte lassen sich in der hierarchischen Graphstruktur abbilden.
- Die Anwendung einer Prädikatregel kann beliebige neue Prädikatkanten erzeugen, insbesondere eine, die das gleiche Prädikat repräsentiert, zu dem die gerade angewendete Regel gehört. Somit kann ein Prädikat sich selbst *rekursiv* aufrufen.
- Ein `if`-Konstrukt, das eine *Fallunterscheidung* simuliert, wird in [9] vorgestellt.

Kapitel 3

Anwendungsbeispiele

Hier werden zwei Anwendungsbeispiele vorgestellt, welche zum Testen der verschiedenen Anforderungen an den implementierten Interpreter eingesetzt wurden. Verkettete Listen (Abschnitt 3.1) sind Teil des Basishandwerkszeugs, das jeder Informatikstudierende im Grundstudium erlernt; die in Abschnitt 3.2 vorgestellte Arithmetik auf den Natürlichen Zahlen ist ein Beispiel für Rekursion in Prädikaten. Beide Beispiele nutzen Behälter (Unterabschnitt 2.1.2), Variablen (Unterabschnitt 2.1.4) und bedingte Regeln (Unterabschnitt2.2.1); die Listen machen zudem Gebrauch von Punkten (Unterabschnitt 2.1.3), Prädikatbehältern und -variablen (Unterabschnitt 2.1.5), während die Arithmetik zusätzlich den Einsatz eines Ausnahmebehandlers (S. 11, *Abschlussklauseln*) demonstriert.

3.1 Verkettete Listen

Verkettete Listen gehören zu den ersten komplexeren Strukturen, mit denen sich Informatikstudierende während des Grundstudiums auseinandersetzen müssen. Daher verwundert es nicht, dass sie schon früh als Anwendungsbeispiel für Dia**plan** herangezogen wurden. Die hier präsentierten Definitionen von verketteten Listen und den ihnen zugeordneten Prädikaten entstammen [7].

3.1.1 Aufbau einer Liste

Eine verkettete Liste besteht aus zwei leeren Knoten, die Listenbeginn und -ende markieren, je einem Behälterknoten für ein Listenelement und binären Verbindungskanten, die die Reihenfolge der Elemente ausdrücken. Eine Liste, die n Elemente enthält, besteht somit aus $n + 2$ Knoten und $n + 1$ Binärkanten. Um sie als Parameter an ein Prädikat übergeben zu können, wird sie im allgemeinen in einen Behälterknoten gesteckt (Abb. 3.1).

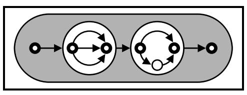


Abbildung 3.1: Eine zweielementige verkettete Liste (aus [7])

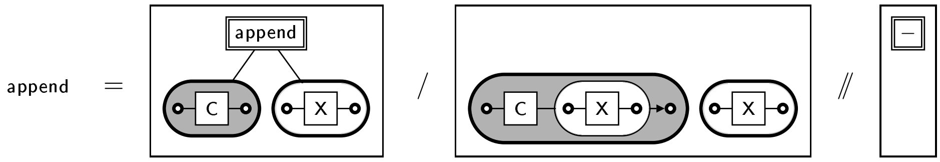


Abbildung 3.2: Das Prädikat append (aus [7])

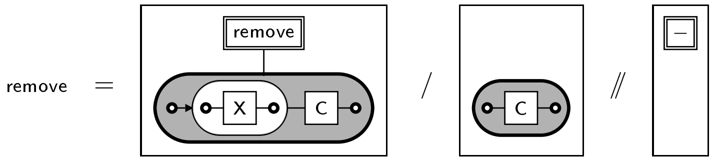


Abbildung 3.3: Das Prädikat remove (aus [7])

3.1.2 Regeln

Als Prädikate werden hier nur die Basisoperationen append (Abb. 3.2) und remove (Abb. 3.3) benutzt; erstere hängt einen übergebenen Parameter als letztes Element an eine Liste an, letztere entfernt das erste Listenelement. Das Prädikat append verdeutlicht die Mächtigkeit der Variablen: Für X kann ein beliebiger Graph eingesetzt werden; C umfasst die gesamte Liste vor Anwendung der Regel unabhängig von der Anzahl der Elemente.

In [7] wird zusätzlich der *Kontrollkombinator* norm (Abb. 3.4) eingeführt, der ein in einem Prädikatbehälter (Unterabschnitt 2.1.5) übergebenes Prädikat so oft wie möglich auf seinen zweiten Parameter anwendet und danach mit „Erfolg“ abbricht.

Abb. 3.5 zeigt den Ablauf einer Berechnung, wenn norm mit remove als Prädikatparameter auf den Listengraphen aus Abb. 3.1 angewendet wird.

3.2 Eine Arithmetik auf den Natürlichen Zahlen

Trotz der geplanten universellen Behälter (Unterabschnitt 2.3.3) existiert ein Modell für eine Arithmetik der Natürlichen Zahlen, die die vier Grundrechenar-

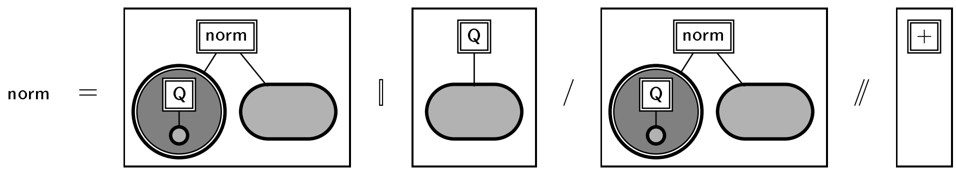


Abbildung 3.4: Der Kontrollkombinator norm (aus [7])

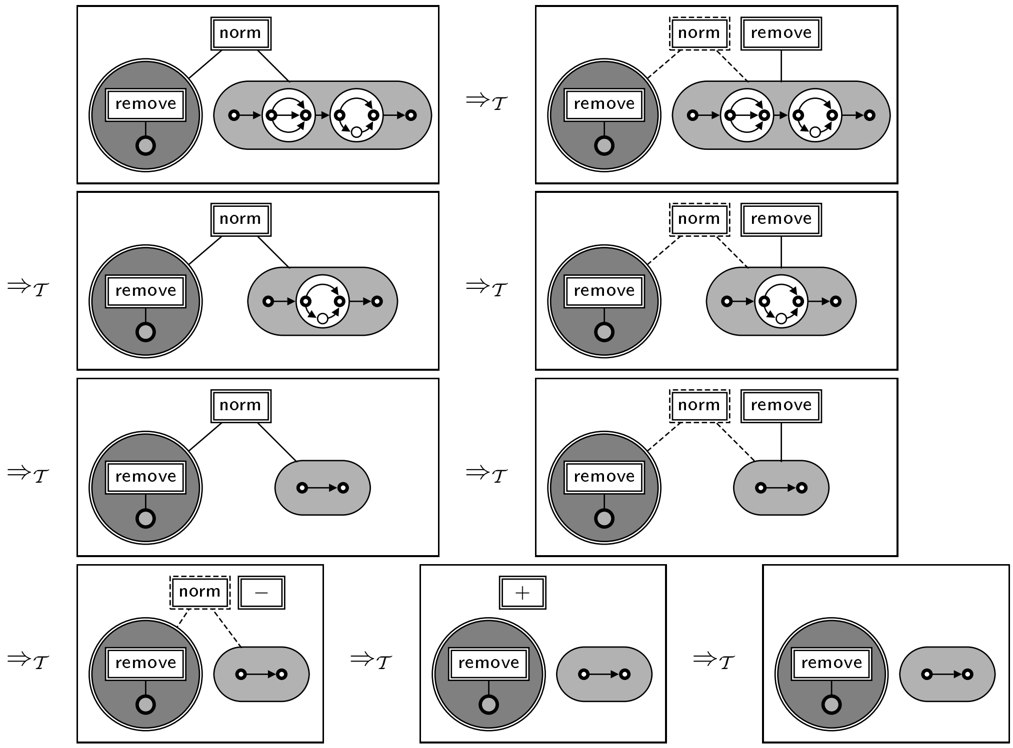


Abbildung 3.5: Eine Berechnung von norm mit remove als Prädikatparameter (aus [7]). Die gestrichelten Linien deuten an, dass die Auswertung des Prädikats norm in diesen Zuständen zurückgestellt wird.

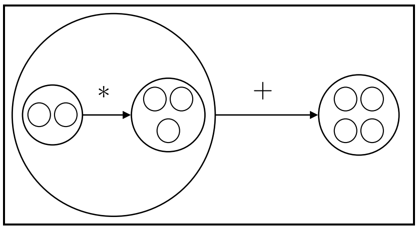
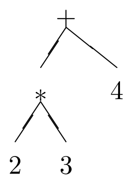


Abbildung 3.6: Darstellung der Formel $2 * 3 + 4$

ten enthält und nur die im implementierten Interpreter umgesetzten Konzepte benötigt. Diese Arithmetik ist ein gutes Testbeispiel für die strikte Tiefensuche (Unterabschnitt 2.2.2), da selbst verhältnismäßig einfache Rechnungen mehrere Hierarchieebenen benötigen.

3.2.1 Daten

Zahlen werden in dieser Arithmetik als Graphen mit einer entsprechenden Anzahl isolierter Knoten dargestellt[1]. In einer zu berechnenden Formel stehen gerichtete binäre Kanten für die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division. Die Knoten am Anfang und Ende einer solchen Kante enthalten hierbei den Wert des ersten bzw. zweiten Operanden. Bei mehr als zwei Operanden wird die Reihenfolge der Operanden in die Hierarchiestruktur umgesetzt (Abb. 3.6). Die Gesamtstruktur erinnert stark an einen Syntaxbaum ([24]); der zur Abbildung äquivalente Baum wäre:



Die Analogie ist wenig überraschend, da **Diaplan**s strenge Graphhierarchie einem Baum entspricht (Unterabschnitt 2.1.2).

3.2.2 Regeln

Die Zahlenarithmetik arbeitet mit fünf Prädikaten: je einem für jede Grundrechenart (Abb. 3.8 - 3.11) und einem namens eval, das eine ähnliche Funktion hat wie das Prädikat norm im Listenbeispiel; es sorgt dafür, dass die Operationen in der richtigen Reihenfolge ausgeführt werden (Abb. 3.7).

[1]Ein leerer Graph entspricht also der Null, ein Graph mit nur einem Knoten der Eins usw.

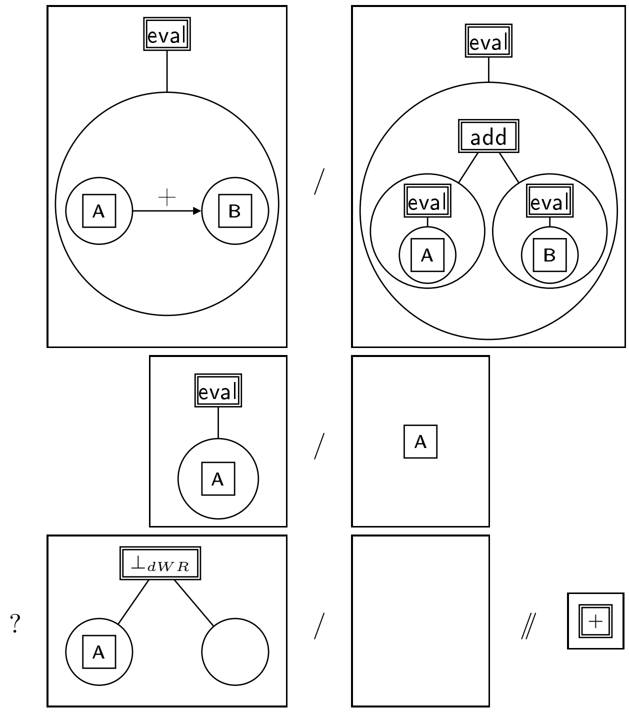


Abbildung 3.7: Das Prädikat eval. Die erste Regel übersetzt die Operatorkante $+$ in das Prädikat add und evaluiert hernach die beiden Operanden; entsprechende Regeln gibt es für $-$/sub, $*$/mult und $\div$/div. Die zweite Regel holt aus allen übrigen Knoten den Inhalt „heraus", damit dieser weiterverarbeitet werden kann (auf Grund der Reihenfolge der Regeln kann es sich nur noch um einen Zahlenknoten handeln, wenn die Ansatzsuche an diese Stelle gelangt). Die Ausnahme $\perp_{dWR}$ („Division with Remainder") zeigt eine Division mit Rest an und wird in diesem Regelsystem einfach zusammen mit ihren Parametern gelöscht (s.a. Abb. 3.11); ein differenzierteres Regelsystem könnte den Divisionsrest jedoch weiterverarbeiten wollen, weswegen die Löschung nicht sofort in der entsprechenden div-Regel vorgenommen wird. Der erfolgreiche Abschluss der Evaluierung wird durch eine Erfolgskante signalisiert.

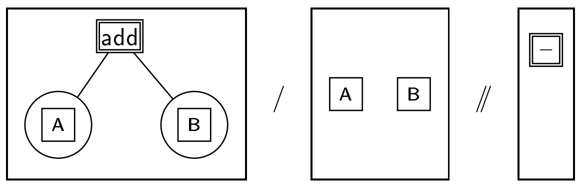


Abbildung 3.8: Das Prädikat add ersetzt seine beiden Parameterknoten sowie sich selbst durch die Inhalte der Parameter. Die Abschlussklausel ist „Fehlschlag", weil sie für einen korrekt strukturierten Graphen nie gerufen werden sollte.

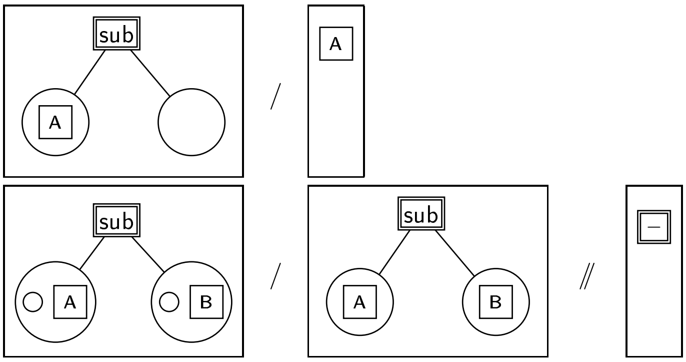


Abbildung 3.9: Das Prädikat sub. Ist der zweite Parameter leer, was der Differenz $A - 0$ entspricht, wird die Prädikatkante mit ihren Parameterknoten durch den Inhalt des ersten Parameters ersetzt. Sind beide Parameter nichtleer, wird je ein Knoten entfernt und sub erneut gerufen. Ist der erste Parameter leer, der zweite jedoch nicht, so greift die Abschlussklausel „Fehlschlag".

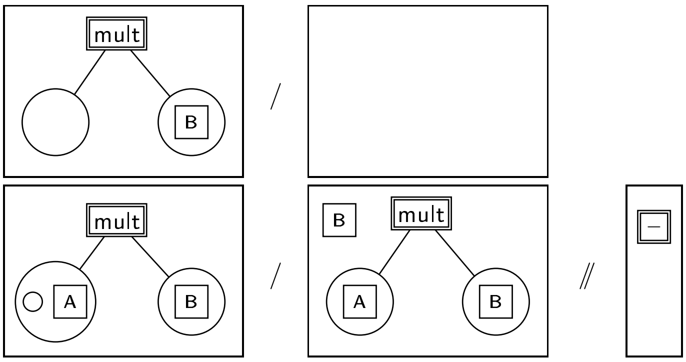


Abbildung 3.10: Das Prädikat mult. Ist wenigstens einer der beiden Parameter leer (also $A*0$ oder $0*B$, wobei hier nur die Regel für den zweiten Fall aufgeführt ist), so ist das Ergebnis der leere Graph. Anderenfalls wird, solange die Möglichkeit besteht, der Inhalt des ersten Parameters um einen Knoten reduziert und der Inhalt des zweiten nach außen kopiert. Sobald der erste Parameter dadurch komplett geleert wurde, greift die erste Regel. Die Abschlussklausel sollte wie bei add in einem korrekt strukturierten Graphen nie aktiv werden.

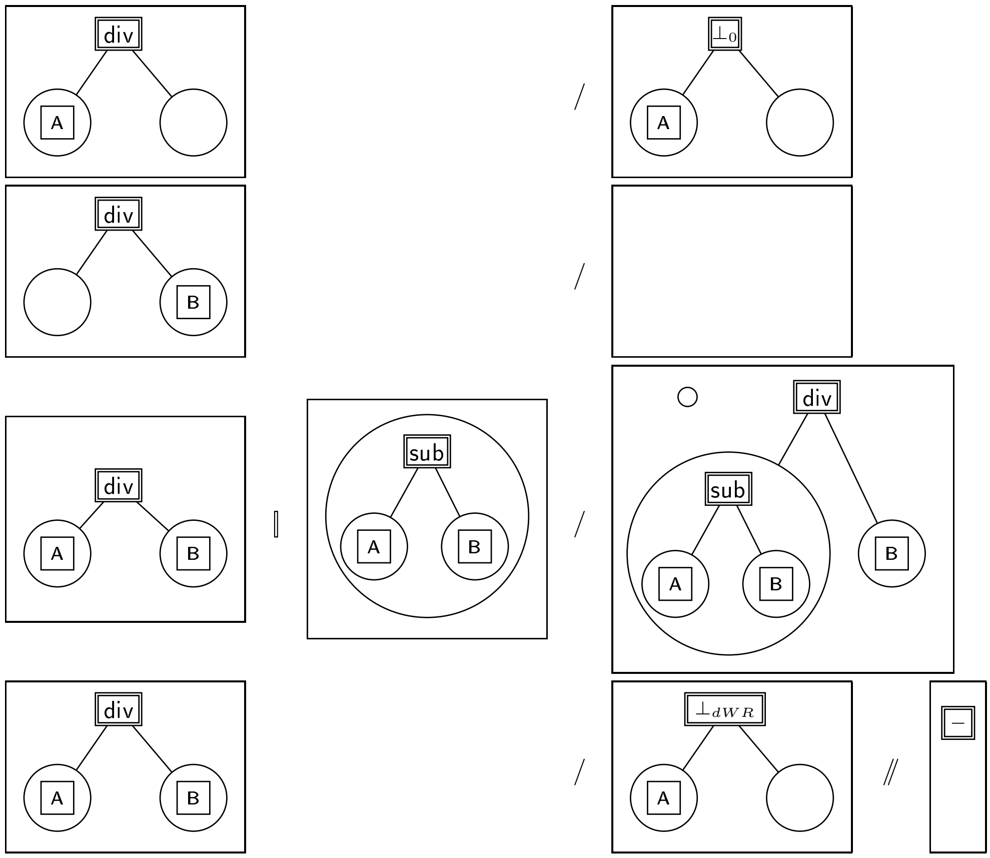


Abbildung 3.11: Das Prädikat div. Ist der zweite Parameter leer (entspricht $A \div 0$), so wird eine Ausnahme geweckt, für die innerhalb dieser Arithmetik keine Behandlung existiert. Ist dagegen nur der erste Parameter leer (was $0 \div B$ mit $B \neq 0$ entspricht), so ist das Ergebnis der leere Graph. Ist beides nicht der Fall, so versucht die einzige „echte“ bedingte Regel dieser Arithmetik, mittels des Prädikats sub, den Divisor vom Dividenden zu subtrahieren; gelingt dies, wird ein Knoten auf Ebene des Prädikats generiert (also das laufende Resultat um 1 inkrementiert) und div erneut gerufen - mit dem reduzierten Dividenden und dem gleichen Divisor. Wird der Dividend im Laufe dieser Rekursion komplett geleert, so greift Regel 2; schlägt die Subtraktion in Regel 3 jedoch fehl, so liegt offensichtlich eine Division mit Rest vor, was durch die Ausnahme in der vierten Regel angezeigt wird. Wie schon bei add und mult wird die Abschlussklausel für einen korrekt strukturierten Eingabegraphen nie angewendet. Man beachte die Reihenfolge der Regeln: Wären z.B. die erste und zweite vertauscht, würde $0 \div 0$ zu 0 berechnet, statt eine Ausnahme zu wecken (s.a. Schritt 3 der strikten Tiefensuche, S. 15.)

Kapitel 4

Die DiaGen-Transformationsmaschine

Dieses Kapitel gibt einen Überblick über die für den im Rahmen dieser Arbeit implementierten Interpreter wichtigen Systemkomponenten von DiaGen. Die Transformationsmaschine bildet nur einen Teil der DiaGen-Architektur (Abb. 4.1) und unterstützt in einem generierten Editor u.a. das sog. *strukturierte Editieren*, welches das Erstellen eines in Bezug auf die entsprechende Sprache syntaktisch korrekten Diagramms wesentlich erleichtert ([25]). Diese Primärnutzung erklärt die für die in dieser Arbeit betrachteten Aspekte bisweilen verwirrende Benennung von Klassen und Methoden. Abschnitt 4.1 beschäftigt sich mit der Repräsentation der Graphen und ihrer Graphobjekte; Abschnitt 4.2 erläutert die baumartige Struktur der Transformationsregeln, während der eigentliche Transformationsvorgang in Abschnitt 4.3 dargelegt wird. Zur Übersicht dienen mit ArgoUML[1] erstellte UML-Klassendiagramme. Kommen Klassen in mehreren Diagrammen vor, so werden ihre Attribute und Operationen nur beim ersten Auftreten aufgeführt.

4.1 Repräsentation von Graphen, Knoten und Kanten

4.1.1 Knoten

Knoten (`Node`, Abb. 4.3) sind in DiaGen so einfach wie möglich gehalten; neben einer potenziell eindeutigen Identifikationsnummer[2] besitzen sie eine zweidimensionale Position im entsprechenden Diagramm[3]. Üblicherweise werden Knoten durch den Aufruf entsprechender Methoden der Klasse `NodeManager` erzeugt. Diese Klasse ist als Einzelstück ([26]) implementiert und definiert eine Schnittstelle `Factory`, die eine Umsetzung des Fabrikentwurfsmusters darstellt; jede

[1] `http://argouml.tigris.org`

[2] Die Eindeutigkeit hängt von der benutzenden Applikation ab; der in Kap. 5 beschriebene Interpreter betrachtet gleich nummerierte Knoten nicht notwendigerweise als gleich.

[3] Auch diese Eigenschaft nutzt der im Rahmen dieser Arbeit implementierte Interpreter nicht, da er „nur“ mit abstrakten graphentheoretischen Konzepten arbeitet, wofür diese Position irrelevant ist.

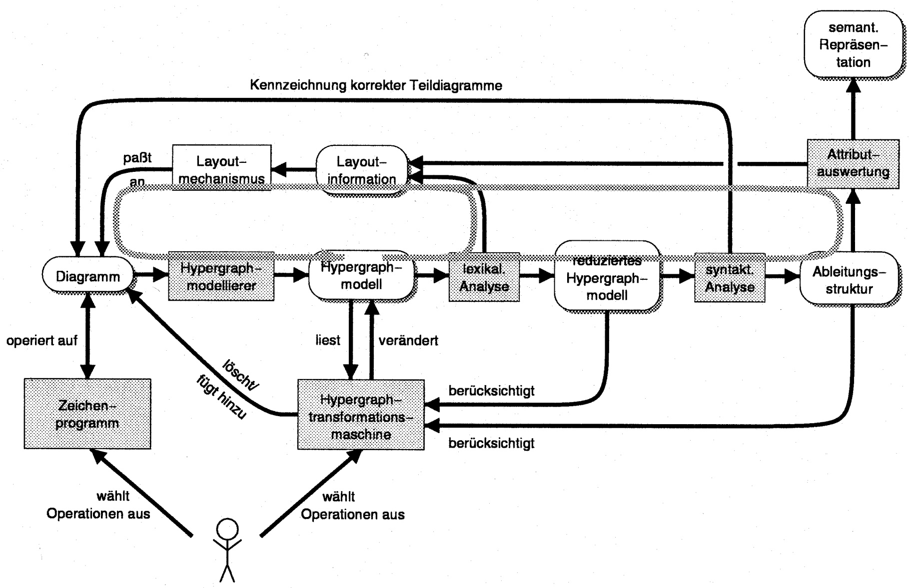


Abbildung 4.1: Die Struktur eines DiaGen-Editors im Überblick. Ovale repräsentieren Datenstrukturen, Rechtecke die funktionalen Bestandteile, Pfeile den Informationsfluss. Die grau hinterlegten Funktionsblöcke sind nur teilweise fest in das Framework integriert; der Programmgenerator passt sie an die Sprachspezifikation an (aus [25]).

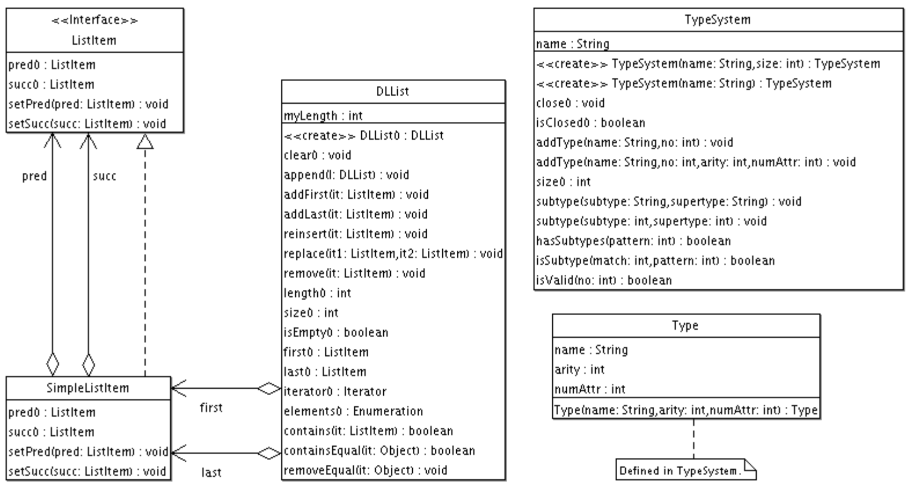


Abbildung 4.2: Wichtige Hilfsklassen von DiaGen

Klasse zur Repräsentation von Knoten definiert ihre eigene Implementierung dieser Schnittstelle.

4.1.2 Kanten

Eine Kante (`Edge`) verwaltet die von ihr besuchten Knoten in einem Array, das der Funktion a aus der allgemeinen Hypergraphdefinition entspricht; ebenso verfährt ein Knoten mit den ihn besuchenden Kanten, wobei diese sich allerdings erst bei ihm anmelden müssen. Eine Kante befindet sich stets in einem der drei Zustände „nicht angemeldet“ (`NOT_ENROLLED`), „angemeldet“ (`ENROLLED`) oder „getrennt“ (`SEGREGATED`); der Zustand bezieht sich auf *alle* besuchten Knoten, was eine „selektive“ Anmeldung ausschließt. Der Anmeldezustand lässt sich von außen nicht direkt beeinflussen - die entsprechenden Methoden sind nicht öffentlich. Fügt man eine Kante zu einem Graphen hinzu, so meldet sie sich von selbst bei den besuchten Knoten an; entfernt man sie, so trennt sie sich ebenfalls von selbst wieder. Nur bei nicht angemeldeten bzw. abgetrennnten Kanten besteht die Möglichkeit, ein oder mehrere Tentakel mit einem anderen Knoten zu verbinden.

Mithilfe der Klasse `TypeSystem` (Abb. 4.2) lassen sich für jeden Kantentyp `Type` Name, eine eindeutige Identifikationsnummer sowie Stelligkeit festlegen; Knoten dagegen sind in DiaGen grundsätzlich ungetypt. Zur Realisierung eines Typgraphen (Unterabschnitt 2.1.3) muss man sich bei den Knoten einstelliger Markierungskanten bedienen. Darüber hinaus bietet die Klasse `TypeSystem` die Möglichkeit, auf den Kantentypen eine Vererbungshierarchie zu definieren.

Die schon in Unterabschnitt 2.1.3 erwähnte „Kantenlastigkeit“ dieser Darstellung resultiert aus der Verwendung der DiaGen-Graphtransformationsmaschine als Werkzeug zur Analyse von Diagrammen. Jede Kante in einem DiaGen -Hypergraphen repräsentiert entweder eine Komponente eines mit einem von DiaGen generierten Editor erstellten Diagramms oder eine für die jeweilige Diagrammsprache relevante räumliche Beziehung[4] zwischen Diagrammkomponenten. Knoten in diesen Hypergraphen repräsentieren dagegen sog. *Konnektoren*, mit deren Hilfe verschiedene Diagrammkomponenten miteinander in Beziehung treten können. Der Typ einer Kante legt die repräsentierte Komponente oder Beziehung fest.

Neben den einfachen Kanten existieren Terminal- und Nichtterminalkanten (`TermEdge` (Abb. 4.3) bzw. `NontermEdge` (nicht in einem Diagramm aufgeführt))[5]. Terminalkanten (als Instanzen der einzigen die Schnittstelle `ModelEdge` implementierenden Klasse) können im Gegensatz zu einfachen Kanten, die nur auf der linken Seite einer Transformationsregel vorkommen können, auch auf der rechten Seite vorkommen; die speziellen Eigenschaften dieser Klasse, die u.a. einer Zurückverfolgung der bis zu ihrer Erzeugung angewendeten Regeln dienen, kommen jedoch bei dem im Rahmen dieser Arbeit implementierten Interpreter nicht zum Einsatz, weswegen sie hier nicht betrachtet werden. Ebenso wenig werden Nichtterminalkanten benutzt.

[4]Solche Beziehungen sind z.B. „berührt“ oder „liegt innerhalb“. Je nach Diagrammsprache sind auch abstraktere Beziehungen möglich.

[5]Die Namen deuten auf den Verwendungszweck, die syntaktische Analyse (Abb. 4.1), hin.

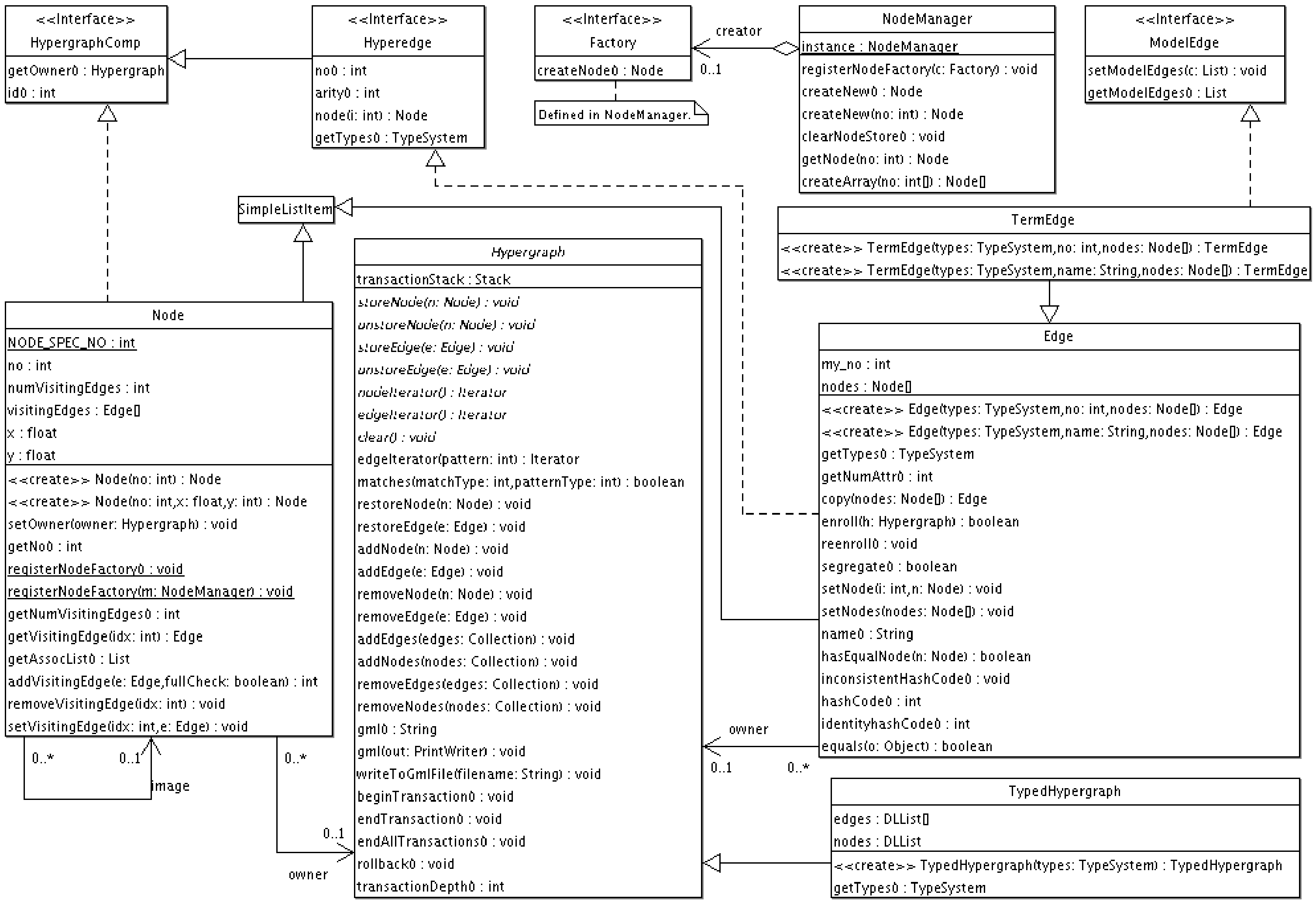


Abbildung 4.3: Die DiaGen-Klassen zur Repräsentation von Graphen und Graphobjekten

4.1.3 Graphen

DiaGens Graphen verwalten ihre Graphobjekte, die im DiaGen-Kontext als Hypergraph*komponenten* bezeichnet werden, in doppelt verketteten Listen (`DLList`, Abb. 4.2). Um dies zu ermöglichen, sind sowohl die Klasse `Node` für Knoten als auch die Klasse `Edge` für Kanten von `SimpleListItem` abgeleitet (Abb. 4.3). Da die Verkettung durch Zeiger in den Objekten selbst realisiert ist, kann keine Graphkomponente (a) mehr als einer Liste bzw. einem Graphen zugeordnet sein und (b) an mehr als einer Stelle innerhalb derselben Liste stehen; somit ist der Definition, dass Graphkomponenten in Mengen organisiert sind (Abschnitt 2.1), Genüge getan.

Die abstrakte Klasse `Hypergraph` verfügt über einen überraschend simplen *Transaktionsmechanismus*, durch den man Änderungen am Graphen (Hinzufügen bzw. Entfernen von Komponenten) im Fall eines Fehlers auf einfache Weise rückgängig machen kann: Der Nutzer kann alle Änderungen, die er als einheitliche *Transaktion* betrachtet (also z.B. eine Regelanwendung), ausführen und im Bedarfsfall „in einem Rutsch“ umkehren.

Eine Instanz der Klasse `TypedHypergraph` (Abb. 4.3) sortiert die ihr zugeordneten Kanten nach deren Typ in ein Array doppelt verketteter Listen, was u.a. die Effizienz der Ansatzsuche (Abschnitt 4.3) erhöht; daher ist auch die für die Transformation vorgesehene Graph-Implementierung `PatternEdgeManager` von dieser Klasse abgeleitet (Abb. 4.5).

4.2 Repräsentation von Regeln

Dieser Abschnitt befasst sich mit der DiaGen-internen Repräsentation von Graphtransformationsregeln und stellt die dafür wichtigsten Klassen und Schnittstellen vor. Wie in Unterabschnitt 2.4.1 erwähnt basiert die DiaGen-Transformationsmaschine auf dem SPO-Modell, d.h. das Finden eines Ansatzes genügt, um die entsprechende Regel anwenden zu können. Die linke Seite einer Regel wird durch einen Weg von der Wurzel zu einem Blatt in einem Baum aus Instanzen der Klasse `ProdPattern` (Unterabschnitt 4.2.2, Abb. 4.4) repräsentiert. Jeder Knoten auf diesem Weg entspricht einer Kante der linken Seite; Knoten der linken Seite erhalten keine explizite Repräsentation im Baum[6]. Im Blatt am Wegende findet sich die entsprechende rechte Seite.

4.2.1 `TransformationSet`

Die Klasse `TransformationSet` bzw. die davon abgeleitete Klasse `ActionTransformationSet` (Abb. 4.5) bildet die Basis für die Transformationsmaschine eines generierten Editors. Der Vektor `roots` besteht aus den Wurzeln der oben angesprochenen Bäume. Regeln werden mittels der beiden öffentlichen `addRule`-Methoden hinzugefügt, wobei linke wie rechte Seite objektweise in je einem Array aufgereiht sind. Das `TransformationSet` arbeitet das Array für die linke Seite von vorne nach hinten ab, indem es für jede Kante ein `ProdPattern` (Unterabschnitt 4.2.2) mit einem entsprechenden `EdgeSelector` (Unterabschnitt 4.2.3) erzeugt, das jeweils als Kind an das

[6] Die zweite Möglichkeit für Bestandteile der linken Seite sind die sog. *Pfadausdrücke* (`PathExpr`), die jedoch im Rahmen dieser Arbeit ohne Belang sind.

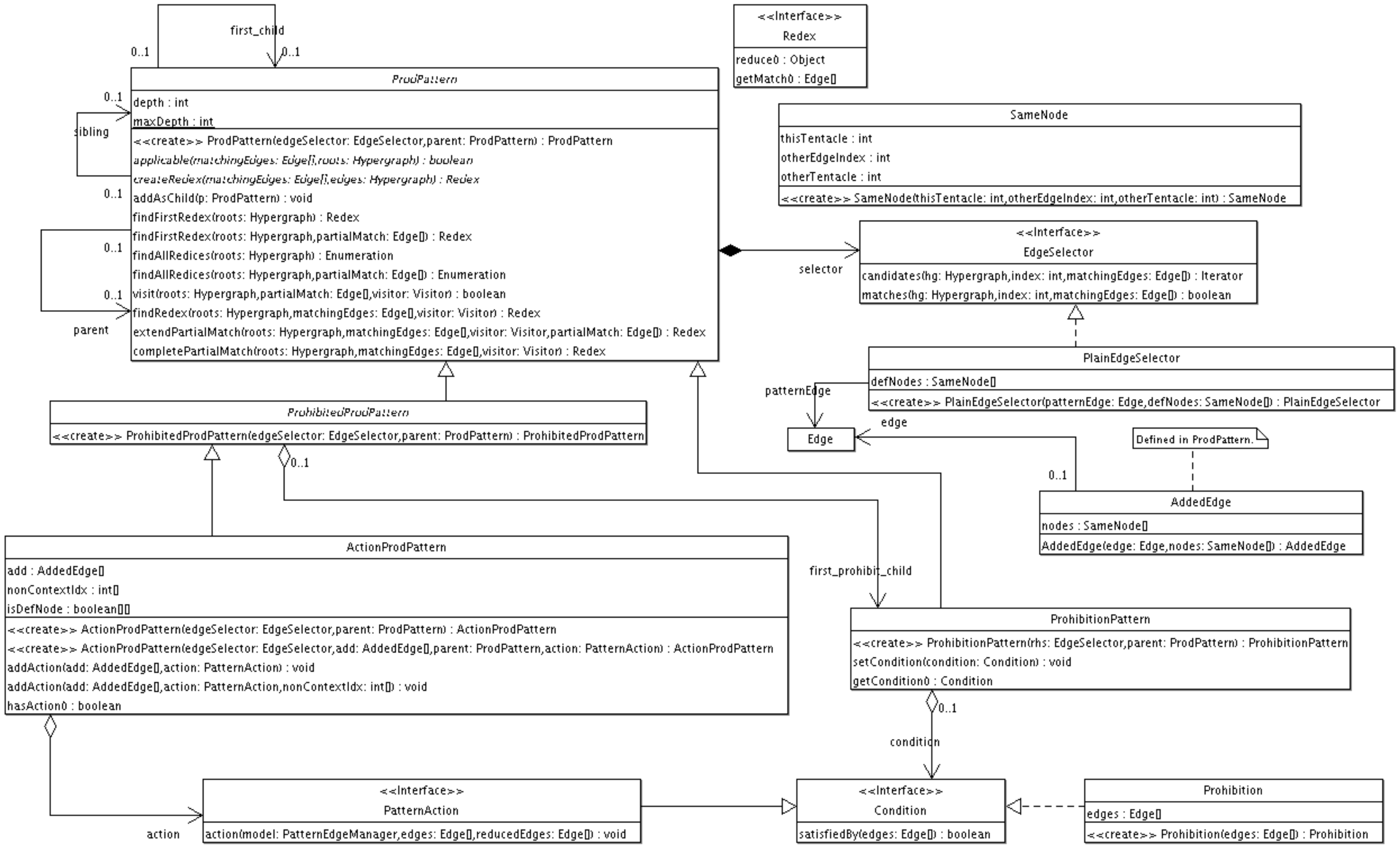


Abbildung 4.4: Die DiaGen-Klassen für die Graphtransformation (Teil 1)

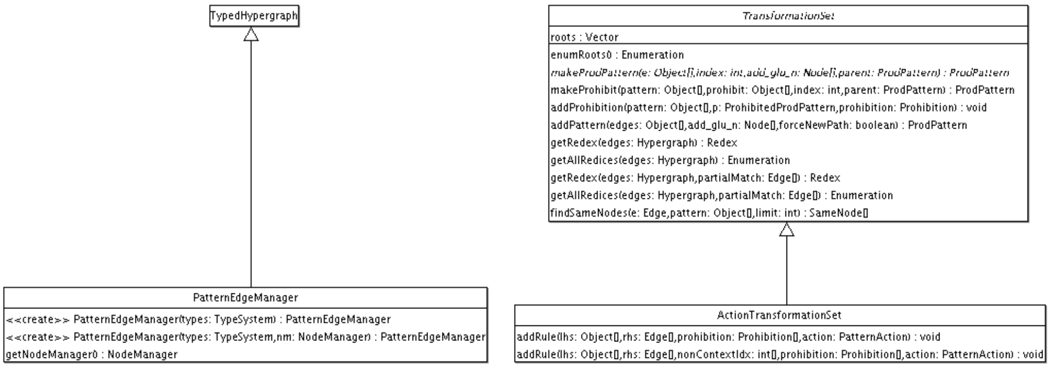


Abbildung 4.5: Die DiaGen-Klassen für die Graphtransformation (Teil 2)

vorherige angehängt wird. Findet das `TransformationSet` einen Baumbeginn der Tiefe $n-1$, der mit den ersten n Kanten im Array übereinstimmt, wird der Baum erst ab Tiefe n an der entsprechenden Stelle erweitert.

4.2.2 `ProdPattern`

Aus Instanzen der abstrakten Klasse `ProdPattern` bzw. ihrer Ableitungen setzen sich die schon beschriebenen Bäume zusammen, die die linken Regelseiten darstellen. Jedes `ProdPattern`-Objekt besitzt einen `EdgeSelector` (Unterabschnitt 4.2.3), in dem die durch dieses Pattern repräsentierte Kante codiert ist.

In einem `ProdPattern`-Baum lassen sich zwar mehrere Regeln unterbringen, sofern deren linke Seiten teilweise übereinstimmen; aber häufig entartet er zu einem einzigen Strang[7].

Negative Kontextbedingungen werden durch `ProhibitionPattern`s repräsentiert, zu suchende Graphkomponenten (der „Positiv-Baum“) durch `ActionProdPattern`s. Das Array `add` jedes Blatts im „Positiv-Baum“ stellt die entsprechende rechte Seite dar; die zusätzlich definierbare `action` (Unterabschnitt 4.2.4) gibt an, wie genau ein gefundener Ansatz in die rechte Seite zu überführen ist.

4.2.3 `EdgeSelector` und `SameNode`

Der einem `ProdPattern` zugeordnete `PlainEdgeSelector`[8] erhält bei seiner Erzeugung (einen Zeiger auf) die entsprechende Kante der linken Regelseite sowie ein Array von `SameNode`-Objekten, die wie in Abb. 4.6 demonstriert die Anbindung dieser Kante an die im Baum höher befindlichen beschreiben: Der Arm `thisTentacle` der repräsentierten Kante besucht den gleichen Knoten wie der Arm `otherTentacle` der Kante in der Tiefe `otherEdgeIndex` im `ProdPattern`-Baum; `otherEdgeIndex` beschreibt dabei stets eine innerhalb des Baums auf demselben Pfad höher befindliche Kante, so dass der `EdgeSelector` für die durch die Baumwurzel repräsentierte Kante kein `SameNode`-Objekt erhält. Da

[7] Dieser unschöne Umstand rührt daher, dass bei gleichen Komponenten zweier linker Regelseiten Objektgleichheit erforderlich ist, um diesen dasselbe Pattern zuzuweisen.

[8] Für Pfadausdrücke existiert eine Klasse `PathExprSelector`, die hier mangels Relevanz nicht betrachtet wird.

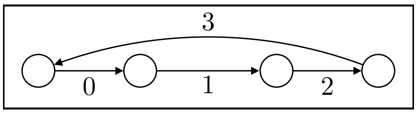


Abbildung 4.6: Demonstration des `SameNode`-Konzepts. Sei dieser Graph die linke Seite einer Regel. Der Einfachheit halber bezeichnen hier die Kantennummern die Reihenfolge (= Tiefe) im `ProdPattern`-Baum. Angenommen, es existierten bereits Repräsentanten für die Kanten 0, 1 und 2. Der `EdgeSelector` für die Kante 3 erhält dann bei seiner Erzeugung ein `SameNode`-Objekt mit den Werten 0, 2 und 1 in den drei Parametern `thisTentacle`, `otherEdgeIndex` und `otherTentacle` sowie eines mit den Werten 1, 0 und 0 in dieser Reihenfolge.

die Ansatzsuche (Abschnitt 4.3) den Baum von oben nach unten „abarbeitet", wird somit in jedem Zustand nur auf bereits zugeordnete Kanten zurück gegriffen.

4.2.4 `Redex` und `PatternAction`

Die Schnittstelle `Redex` beschreibt einen gefundenen Ansatz in einem Graphen. Die Methode `reduce()`[9] führt den Transformationsschritt durch; im Standardfall ruft sie lediglich die `action`-Methode einer dem `Redex` zugewiesenen `PatternAction`, die die „eigentliche" Arbeit übernimmt. Die `PatternAction`-Implementierungen sind editorspezifisch, während die Standardimplementierung der Schnittstelle `Redex` bereits in `ActionProdPattern` festgelegt ist. Allen (generierten) `PatternAction`-Implementierungen ist jedoch gemein, dass sie zwar mit den Daten des Basis-Hypergraphmodells arbeiten, jedoch nur das reduzierte modifizieren - die Klasse `PatternEdgeManager` verwaltet dabei sowohl das Basis- als auch das reduzierte Hypergraphmodell[10]. Zusätzlich zur „eigentlichen" Graphtransformation, also dem in diesem Fall sogar nur Hinzufügen von Kanten zum reduzierten Modell, können sprachspezifische *Attribute* definiert werden, um Informationen zu speichern, die bei der reinen Reduktion verloren gingen (wie z.B. Namen von Diagrammkomponenten).

4.3 Die Ansatzsuche

Die Suche nach einem Ansatz (einer Instanz einer `Redex`-Implementierung) startet durch den Aufruf einer der `getRedex`- bzw. im Fall einer Mehrfachsuche durch Aufruf einer der `getAllRedices`-Methoden der Klasse `TransformationSet`. Die nutzende Klasse kann im Parameter `partialMatch` bereits eine Vorauswahl von Kanten angeben, die im gesuchten Ansatz auftreten müssen; die Reihenfolge muss den Positionen im der Regel zugehörigen

[9] Der Methodenname rührt vom „traditionellen" Zweck der DiaGen -Transformationsmaschine her: Aus einem Hypergraphmodell, das die reinen optischen Elemente eines Diagramms enthält, ein *reduziertes* Hypergraphmodell zu erstellen, in dem sprachabhängig ebendiese reinen optischen Elemente zu größeren Einheiten zusammengefügt sind, was wiederum das Modell vereinfacht.

[10] Da der im Rahmen dieser Arbeit implementierte Interpreter diese Eigenschaft nicht nutzt, sind die entsprechenden Methoden in Abb. 4.5 nicht aufgeführt.

`ProdPattern`-Baumpfad entsprechen, „Lücken" (also `null`-Einträge) im Array sind nicht zulässig.

Suchalgorithmus. In jedem Knoten des `ProdPattern`-Baums erfolgt die Suche nach folgendem Schema:

1. Suche alle Kanten im Graphen, die vom gleichen Typ wie die entsprechende Modellkante (die dem entsprechenden `PlainEdgeSelector` zugeordnete `patternEdge`) sind.

2. Suche aus diesen Kanten alle, die gemäß der Vorgabe an die bereits gefundenen Kanten angebunden sind (die den Vorgaben des Attributs `defNodes` entsprechen).

3. Der sogenannte *Injektivitätstest*: Filtere alle Kanten heraus, auf die obige Bedingungen zutreffen, die aber bereits an anderer Stelle dem Ansatz zugeordnet wurden.

4. Wähle aus den verbleibenden eine beliebige Kante aus und fahre auf der nächsttieferen Baumstufe mit Schritt 1 fort.

Schritt 1 verdeutlicht einen Zweck des Typsystems und insbesondere der Klasse `TypedHypergraph`, von der die Transformationsgraphklasse `PatternEdgeManager` abgeleitet ist (Abschnitt 4.1): Die Kanten sind genau die Elemente der Liste, die im Array `edges` unter der entsprechenden Typnummer zu finden ist. Somit benötigt dieser Schritt konstante Zeit.

Je größer und komplexer die entsprechende linke Seite, desto länger dauert Schritt 2, aber desto wahrscheinlicher findet der Algorithmus nur Kanten, die auch den restlichen Bedingungen, insbesondere denen auf den tiefer liegenden Ebenen entsprechen.

Für Schritt 1-3 ist die Methode `candidates()` des entsprechenden `PlainEdgeSelector` zuständig, Schritt 4 wird durch den gegenseitig rekursiven Aufruf der `ProdPattern`-Methoden `findRedex()` und `completePartialMatch()` realisiert.

Steht für Schritt 4 keine Kante zur Auswahl, so wird die Ansatzsuche auf derselben Ebene mit dem nächsten Geschwister-`ProdPattern`, also der nächsten Regel, fortgesetzt. Ist kein Geschwisterpattern vorhanden, überprüft die Methode `applicable()`, ob die bisher gefundenen Kanten weitere Vorgaben wie z.B. den eventuellen negativen Kontext erfüllen; ist dies der Fall, wird ein entsprechender Redex generiert, anderenfalls setzt die Suche auf der nächsthöheren Ebene mit dem nächsten Geschwisterpattern fort usw.[11] Die gleiche Prüfung findet statt, wenn ein Pattern keine Kinder hat, also ein Blatt und somit die letzte Komponente einer linken Seite darstellt.

Mehrfachsuche. In vielen Fällen genügt die einfache Ansatzsuche, aber bisweilen ist der erste gefundene Ansatz nicht der tatsächlich oder einzig gewünschte. Lässt sich dies nicht ausschließen, sollte man mithilfe der Methode

[11] In diesem Kontext kann man die Baumwurzeln als „Geschwister" betrachten.

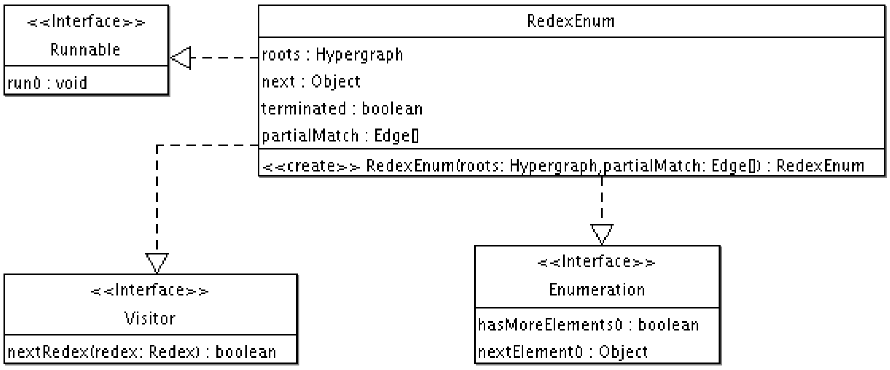


Abbildung 4.7: Die Klasse `RedexEnum` (innere Klasse von `ProdPattern`)

`getAllRedices()` aus der Klasse `TransformationSet` eine *Mehrfachsuche* starten.[12] Statt eines einzelnen `Redex` gibt die Methode eine Aufzählung von Redizes zurück, hinter der sich eine Instanz der Klasse `RedexEnum` (Abb. 4.7) verbirgt. Diese Klasse implementiert die beiden Standardschnittstellen `Enumeration` und `Runnable` sowie die in `ProdPattern` definierte Schnittstelle `Visitor`, die gemeinsam mit der `ProdPattern`-Methode `visit` eine Umsetzung des *Besucher*-Entwurfsmusters ([26]) darstellt.

Bei der Mehrfachsuche werden nicht, was eine nahe liegende Vermutung ist, alle Redizes auf einmal gefunden, sondern die Suche nach dem nächsten startet nur, wenn der entsprechende `Visitor` „Bedarf" anmeldet, d.h. wenn die Methode `nextRedex`, der der zuletzt gefundene `Redex` übergeben wird, den Wert `true` liefert. Eine `RedexEnum` meldet, da sie stets in einem eigenen Thread arbeitet, „Bedarf" an, sobald sie den letzten an sie übergebenen `Redex` über die Methode `nextElement` an das sie nutzende Objekt weitergegeben hat. Solange dieser `Redex` nicht abgerufen wird, verharrt das `RedexEnum`-Objekt in Wartestellung. So kann die entsprechende `PatternAction` für jeden gefundenen `Redex` sofort durchgeführt werden, ohne dass erst die gesamte Liste gefunden werden muss. Auf dieser Basis kann dann an anderer Stelle, sobald der erste Ansatz gefunden ist, weiter gearbeitet werden, was aufgrund des Nichtdeterminismus' in der Ansatzsuche einen merklichen Effizienzgewinn bedeutet. Im Falle des im nächsten Kapitel beschriebenen Interpreters wird durch das Ausführen einer solchen Aktion die Suche nach weiteren Ansätzen oftmals sogar hinfällig.

[12] Neuere DiaGen-Versionen nutzen den hier beschriebenen Mechanismus nicht mehr, arbeiten jedoch mit dem gleichen Entwurfsmuster. Der im Rahmen dieser Arbeit implementierte Interpreter setzt aber auf die hier beschriebene Variante, die nach wie vor unterstützt wird, auf.

Kapitel 5

Erweiterungen der DiaGen-Transformationsmaschine

Dieses Kapitel beschreibt die Implementierung des prototypischen Dia**plan** -Interpreters. Auf eine detaillierte Anforderungsdefinition wird verzichtet, da diese sich kanonisch aus den Kapiteln 2 und 4 ergibt: Der Interpreter soll alle in Abschnitt 2.1 und 2.2 präsentierten Konzepte beinhalten, wobei die Implementierung auf den in Kapitel 4 vorgestellten Teilen der DiaGen -Graphtransformationsmaschine aufsetzt. Die im vorangegangenen Kapitel erläuterten Klassen wurden teilweise auch selbst modifiziert. Die Änderungen betrafen zwar auch Köpfe und Rümpfe nach außen hin sichtbarer Methoden, jedoch nicht deren äußere Effekte oder Nutzbarkeit, damit die Kompatibilität zu existierenden Applikationen, welche die im vorangegangenen Kapitel vorgestellten Klassen nutzen, gewahrt bleibt. Dieses Kapitel ist folgendermaßen gegliedert: Abschnitt 5.1 erläutert die Klassen zur Datenrepräsentation, also die Umsetzung der Konzepte aus Abschnitt 2.1, soweit diese nicht schon im vorhergehenden Kapitel besprochen wurden, in entsprechender Reihenfolge. Abschnitt 5.2 geht auf die algorithmischen Bestandteile, also die Umsetzung der in Abschnitt 2.2 beschriebenen Konzepte ein. Abschnitt 5.3 schließlich stellt den „eigentlichen“ Interpreter vor, also die Klassen für Ein- und Ausgabe von Dia**plan**-Daten und -Programmen. Wie schon im vorangegangenen Kapitel werden UML-Klassendiagramme als Erklärungshilfen eingesetzt. Für jeden Unterabschnitt werden die neu implementierten Klassen detailliert vorgestellt und die wichtigsten Modifikationen an den in Kap. 4 beschriebenen Klassen informell erläutert. Die neu definierten Schnittstellen werden nicht detailliert erklärt, da sich ihr Zweck jeweils aus dem Kontext ergibt.

Die Beschreibungen der neuen Klassen folgen dabei stets diesem Schema:

Klasse	*Name der Klasse*
Basisklasse	*Name der Basisklasse*
Schnittstellen	*Namen implementierter Schnittstellen*
Innere Klassen	*Namen innerer Klassen*
Aufgabe	*Aufgabenbeschreibung der Klasse*

Eigenschaften	*Übersicht über alle Eigenschaften, also Attribute, Konstruktoren und Methoden*
	`Typ Attributname;`
	`Typ Methodenname (Parameter);`

Attribute

`Typ Name;`

Beschreibung des Attributs

Konstruktoren

`Typ Name(Parameter);`

Beschreibung des Konstruktors. Definiert eine Klasse mehrere Konstruktoren, so werden nur die mit den meisten Parametern erläutert.

Methoden

`Typ Name(Parameter);`

Beschreibung der Methode. Definiert eine Klasse mehrere gleichnamige Methoden, so werden nur die mit den meisten Parametern erläutert.

5.1 Die hinzugekommenen Graphbestandteile

Dieser Abschnitt beschreibt die Umsetzung der in Abschnitt 2.1 präsentierten Konzepte im prototypischen Dia**plan**-Interpreter. Einen Überblick über die neuen Klassen zeigt Abb. 5.1 auf der nächsten Seite. Zu jeder neu definierten `Edge`-Klasse existiert im Paket `diagen.hypergraph.parser` eine entsprechende `TermEdge`-Klasse; auf eine detaillierte Beschreibung dieser `TermEdge`-Klassen wird hier verzichtet, da ihre Funktionalität dem jeweiligen `Edge`-Pendant entspricht (s.a. Unterbaschnitt 4.1.2) und die Vererbungsbeziehungen identisch mit denen innerhalb der Nicht-`TermEdge`s sind (z.B. erbt `FrameTermEdge` von `TermEdge` und implementiert `HypergraphContainer`); stattdessen enthalten die Klassenbeschreibungen der Nicht-`TermEdge`s einen Hinweis auf die Funktionsweise der jeweiligen `TermEdge`. Die Repräsentation von Graphen sowie „einfachen“ Knoten und Kanten (Unterabschnitt 2.1.1) wurde im vorhergehenden Kapitel dargelegt, so dass hier nur die „spezielleren“ Konzepte besprochen werden. Soweit nicht explizit anders erwähnt, entstammen die im Folgenden vorgestellten Klassen und Schnittstellen dem Paket `diagen.hypergraph`.

5.1.1 Behälter

5.1.1.1 FrameNode

Klasse	FrameNode
Basisklasse	Node
Schnittstellen	HypergraphContainer
Innere Klassen	StandardFactory, PredicateContainerFactory
Aufgabe	Knoten, der einen Hypergraphen enthalten kann.

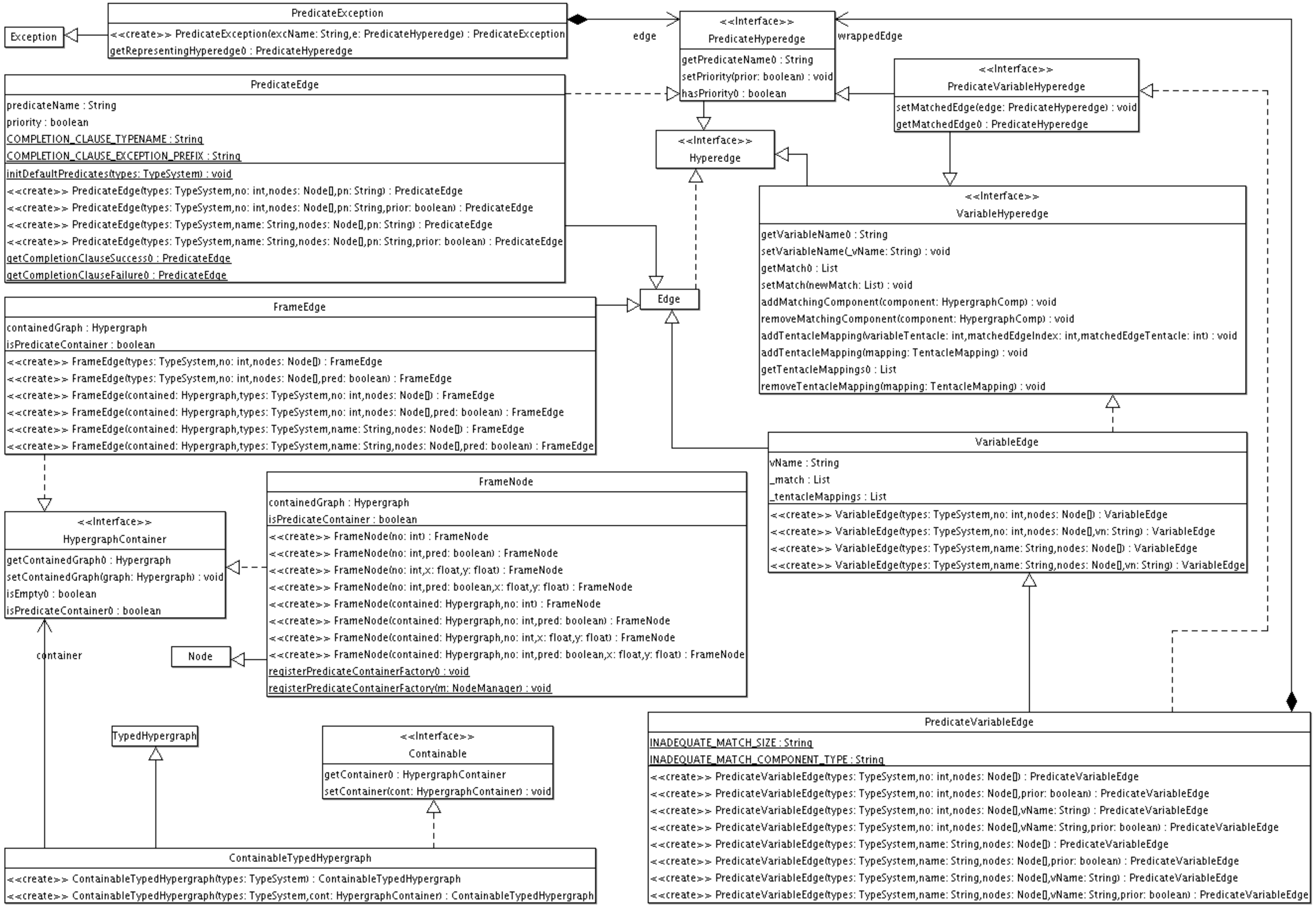


Abbildung 5.1: Die neuen Graphkomponenten

Eigenschaften

```
      private Hypergraph containedGraph;
   private final boolean isContainer;

                 public FrameNode       (int no);
                 public FrameNode       (int no, boolean pred);
                 public FrameNode       (int no, float x, float y);
                 public FrameNode       (int no, boolean pred,
                                         float x, float y);
                 public FrameNode       (Hypergraph contained, int no);
                 public FrameNode       (Hypergraph contained, int no,
                                         boolean pred);
                 public FrameNode       (Hypergraph contained, int no,
                                         float x, float y);
                 public FrameNode       (Hypergraph contained, int no,
                                         boolean pred,
                                         float x, float y);

      public static void registerNodeFactory();
      public static void registerNodeFactory(NodeManager m);
      public static void registerPredicateContainerFactory();
      public static void registerPredicateContainerFactory
                                 (NodeManager m);
       public Hypergraph getContainedGraph();
public synchronized void setContainedGraph(Hypergraph graph);
          public boolean isEmpty         ();
          public boolean isPredicateContainer();
```

Attribute

`private Hypergraph containedGraph;`
Der enthaltene `Hypergraph`.

`private boolean isPredicateContainer;`
Gibt an, ob es sich bei diesem `FrameNode` um einen Prädikatbehälter handelt.

Konstruktoren

```
public FrameNode(Hypergraph contained, int no,
                 boolean pred, float x, float y);
```

Erzeugt einen `FrameNode`.
Parameter:
`contained` Der enthaltene Hypergraph.
`no` Die Nummer dieses `FrameNode`.
`pred` Gibt an, ob dieser `FrameNode` ein Prädikatbehälter sein soll.
`x` Die x-Koordinate dieses `FrameNode` im zweidimensionalen Diagramm.
`y` Die y-Koordinate dieses `FrameNode` im zweidimensionalen Diagramm.

Methoden

`public static void registerNodeFactory();`
Meldet eine Instanz der inneren Klasse `StandardFactory` beim Standard-`NodeManager` an.

`public static void registerNodeFactory(NodeManager m);`
Meldet eine Instanz der inneren Klasse `StandardFactory` bei `m` an.

`public static void registerPredicateContainerFactory();`
Meldet eine Instanz der inneren Klasse `PredicateContainerFactory` beim Standard-`NodeManager` an.

`public static void registerPredicateContainerFactory(NodeManager m);`
Meldet eine Instanz der inneren Klasse `PredicateContainerFactory` bei `m` an.

`public Hypergraph getContainedGraph();`
Gibt den enthaltenen `Hypergraph` zurück.

`public synchronized void setContainedGraph(Hypergraph graph);`
Setzt den enthaltenen `Hypergraph` auf den angegebenen Wert.

`public boolean isEmpty();`
Gibt `true` zurück, falls dieser `FrameNode` keinen `Hypergraph` enthält bzw. falls der enthaltene `Hypergraph` keine Knoten oder Kanten enthält. Anderenfalls ist der Rückgabewert `false`.

`public boolean isPredicateContainer();`
Gibt an, ob dieser `FrameNode` ein Prädikatbehälter ist.

5.1.1.2 FrameEdge

Klasse	FrameEdge
Basisklasse	Edge
Schnittstellen	HypergraphContainer
Aufgabe	Kante, die einen Hypergraphen enthalten kann. Die Methoden der Klasse `FrameTermEdge` sind bis auf `TermEdge`-spezifische Eigenheiten identisch mit denen dieser Klasse.[1]

Eigenschaften

```
private Hypergraph containedGraph;
private boolean isPredicateContainer;
```

```
public FrameEdge(TypeSystem types, int no,
                 Node[] nodes);
public FrameEdge(TypeSystem types, int no,
                 Node[] nodes, boolean pred);
public FrameEdge(TypeSystem types, String name,
                 Node[] nodes);
public FrameEdge(TypeSystem types, String name,
                 Node[] nodes, boolean pred);
public FrameEdge(Hypergraph contained, TypeSystem types,
                 int no, Node[] nodes);
public FrameEdge(Hypergraph contained, TypeSystem types,
                 int no, Node[] nodes, boolean pred);
public FrameEdge(Hypergraph contained, TypeSystem types,
                 String name, Node[] nodes);
```

[1] Das bei den anderen Klassen eingesetzte Adapter-Entwurfsmuster ([26]) ist hier ungeeignet, da der enthaltene Graph möglicherweise über einen Zeiger auf seinen Behälter verfügt.

```
                public FrameEdge(Hypergraph contained,
                                 TypeSystem types,
                                 String name, Node[] nodes,
                                 boolean pred);

       public Hypergraph getContainedGraph();
public synchronized void setContainedGraph(Hypergraph graph);
          public boolean isEmpty  ();
          public boolean isPredicateContainer();
```

Attribute

`private Hypergraph containedGraph;`
Der enthaltene `Hypergraph`.

`private boolean isPredicateContainer;`
Gibt an, ob es sich bei dieser `FrameEdge` um einen Prädikatbehälter handelt.

Konstruktoren

```
public FrameEdge (Hypergraph contained, TypeSystem types,
                  int no, Node[] nodes, boolean pred);
```
Erzeugt eine `FrameEdge` anhand ihrer Typnummer.
Parameter:
`contained` Der enthaltene `Hypergraph`.
`types` Das benutzte `TypeSystem`.
`no` Die Nummer des Typs dieser Kante in `types`.
`nodes` Die von dieser Kante besuchten `Nodes`.
`pred` Gibt an, ob diese Kante ein Prädikatbehälter sein soll.

```
public FrameEdge (Hypergraph contained, TypeSystem types,
                  String name, Node[] nodes, boolean pred);
```
Erzeugt eine `FrameEdge` anhand ihres Typnamens.
Parameter:
`name` Der Name des Typs dieser Kante in `types`.
Rest wie oben.

Methoden

`public Hypergraph getContainedGraph();`
Gibt den enthaltenen `Hypergraph` zurück.

`public synchronized void setContainedGraph(Hypergraph graph);`
Setzt den enthaltenen `Hypergraph` auf den angegebenen Wert.

`public boolean isEmpty();`
Gibt `true` zurück, falls diese `FrameEdge` keinen `Hypergraph` enthält bzw. falls der enthaltene `Hypergraph` keine Knoten oder Kanten enthält. Anderenfalls ist der Rückgabewert `false`.

`public boolean isPredicateContainer();`
Gibt an, ob diese `FrameEdge` ein Prädikatbehälter ist.

5.1.1.3 ContainableTypedHypergraph

Klasse	ContainableTypedHypergraph
Basisklasse	TypedHypergraph
Schnittstellen	Containable
Aufgabe	Hypergraph, der über einen Verweis auf seinen Behälter verfügt. Stellt die neue Basisklasse von `PatternEdgeManager` dar.

Eigenschaften

```
private HypergraphContainer container;

                    public ContainableTypedHypergraph(TypeSystem types);
                    public ContainableTypedHypergraph(TypeSystem types,
                                                  HypergraphContainer cont);

 public HypergraphContainer getContainer             ();
   public synchronized void setContainer             (HypergraphContainer cont);
```

Attribute

`private HypergraphContainer container;`
Der Behälter, in dem dieser Hypergraph sich befindet.

Konstruktoren

```
public ContainableTypedHypergraph(TypeSystem types,
                                  HypergraphContainer cont);
```
Erzeugt einen `ContainableTypedHypergraph`.
Parameter:
`types` Das `TypeSystem`, mit dessen Hilfe die Kanten dieses Hypergraphen getypt werden.
`cont` Der `HypergraphContainer`, der diesen Hypergraphen enthält.

Methoden

`public HypergraphContainer getContainer();`
Gibt den `HypergraphContainer` zurück, in dem sich dieser Hypergraph befindet.

`public synchronized void setContainer(HypergraphContainer cont);`
Setzt den Verweis auf den `HypergraphContainer`, in dem sich dieser Hypergraph befindet, auf den angegebenen Wert.

5.1.1.4 Wichtige Änderungen an den existierenden Klassen

Der in Unterabschnitt 4.1.3 beschriebene Transaktionsmechanismus wurde so ausgedehnt, dass alle entsprechenden Methodenaufrufe (Beginn, Abschluss oder Abbruch bzw. Annullierung einer Transaktion) an alle eventuell in den Komponenten eines Graphen enthaltenen Graphen weitergegeben werden. Zudem wurde die Klasse `Hypergraph` um Methoden erweitert, die verschiedene Anfragen erleichtern:

`public final Iterator componentIterator();`
Zählt alle Knoten und Kanten auf, die direkt diesem `Hypergraph` angehören (bisher war die Aufzählung von Knoten und Kanten nur separat möglich).

`public final Iterator containerIterator();`
Zählt alle `HypergraphContainer` auf, die direkt diesem `Hypergraph` angehören.

`public Iterator fullComponentIterator();`
Zählt alle Graphkomponenten auf, die direkt oder indirekt diesem `Hypergraph` angehören.

`public Iterator fullEdgeIterator();`
Zählt alle Kanten auf, die direkt oder indirekt diesem `Hypergraph` angehören.

`public Iterator fullNodeIterator();`
Zählt alle Knoten auf, die direkt oder indirekt diesem `Hypergraph` angehören.

`public Iterator fullContainerIterator();`
Zählt alle Behälter auf, die direkt oder indirekt diesem `Hypergraph` angehören.

`public int getRelativeNestingDepth(HypergraphComp component);`
Gibt die Schachtelungstiefe der angegebenen Komponente in Bezug auf diesen Graphen zurück. Der zurückgelieferte Wert ist
0, wenn `component` direkt in diesem Graphen enthalten ist,
$n > 0$, wenn `component` n Hierarchieebenen tiefer in diesem Graphen enthalten ist, und
-1, wenn `component` gar nicht in diesem Graphen enthalten ist.

5.1.2 Typen und Punkte

Die Klasse `diagen.util.TypeSystem` wurde konzeptionell nicht verändert; sie verfügt jetzt lediglich über zwei zusätzliche Methoden, welche es bei der Deklaration von Subtypen erlauben, Typnamen und -nummern beliebig miteinander zu mischen, sowie einen konstanten String `POINT_MARKER_BASE_TYPE`, der als Name für den Basistypen aller *Punktmarkierungskanten* vorgesehen ist: Jeder punktierte Graph verfügt über eine solche Kante, die seine Punkte in der entsprechenden Reihenfolge besucht. Nur anhand dieser Kante sind Punkte von Nichtpunkten unterscheidbar, was insofern eine unbefriedigende Umsetzung der Theorie ist, als bei der Erstellung einer Regel das Wissen über die Gesamtanzahl der Punkte im transformierten Graphen erforderlich ist. Bis auf die Stringkonstante ist die Umsetzung des Punkt-Konzepts jedoch bisher noch kein Bestandteil der Transformationsmaschine, sondern befindet sich im prototypischen Interpreter (s.a. Abschnitt 5.3). Zur Realisierung der Konzepte aus Unterabschnitt 2.1.3 wurden keine zusätzlichen Schnittstellen oder Klassen implementiert.

Es gibt wie bisher keine Möglichkeit, Knoten „direkt“ zu typisieren; somit ist die Implementierung gegenüber der Theorie an dieser Stelle eingeschränkt, da im Typgraphen $\mathcal{C}(X) = \langle \mathcal{C}_V, \mathcal{C}_E \cup X, a \rangle$ die Menge C_V hier nur stets ein Element besitzt.

5.1.3 Variablen

5.1.3.1 diagen.util.VariableTypeSystem

Klasse diagen.util.VariableTypeSystem
Basisklasse diagen.util.TypeSystem
Aufgabe Typsystem, das bestimmte Typen als Variablentypen markiert.

Eigenschaften

```
private SortedSet variableTypes;

          public VariableTypeSystem(String name, int size);
          public VariableTypeSystem(String name);

     public void addType           (String name, int no);
     public void addType           (String name, int no, int arity, int numAttr);
     public void addType           (String name, int no, int arity, int numAttr,
                                    boolean isVariable);
  public boolean isVariableType    (String name);
  public boolean isVariableType    (int no);
```

Attribute

`private SortedSet variableTypes;`
Verwaltet die Variablentypen als `Integer`.

Konstruktoren

`public VariableTypeSystem(String name, int size);`
Erzeugt ein `VariableTypeSystem`. Beide Parameter sind durch die Konstruktoren der Basisklasse vorgegeben.
Parameter:
`name` Der Name dieses `VariableTypeSystems`.
`size` Die Initialgröße (= Anzahl der Typen) dieses `VariableTypeSystems`.

Methoden

```
public void addType(String name, int no, int arity,
                    int numAttr, boolean isVariable);
```
Fügt einen neuen Typen zu diesem `VariableTypeSystem` hinzu.
Parameter:
`name` Der Name des neuen Typs.
`no` Die Nummer des neuen Typs.
`arity` Die Stelligkeit, die Kanten des neuen Typs besitzen.
`numAttr` Die Anzahl möglicher Parser-Attribute, die eine Kante dieses Typs verwalten kann (für Dia**plan** ohne Belang, aber von der Basisklasse vorgegeben).
`isVariable` Gibt an, ob der neue Typ ein Variablentyp sein soll.

`public boolean isVariableType(String name);`
Zeigt an, ob der Typ mit dem übergebenen Namen ein Variablentyp ist.

`public boolean isVariableType(int no);`
Zeigt an, ob der Typ mit der übergebenen Nummer ein Variablentyp ist.

5.1.3.2 VariableEdge

Klasse	VariableEdge
Basisklasse	Edge
Schnittstellen	VariableHyperedge
Aufgabe	Dient zur Repräsentation einer Variablenkante. Jede `VariableEdge` verwaltet während eines Transformationsschritts die ihr zugeordnete Substitution. Die Klasse `VariableTermEdge` delegiert intern sämtliche variablenspezifischen Methodenaufrufe an diese Klasse weiter und stellt somit einen Adapter ([26]) dar.

Eigenschaften

```
private String vName;
  private List _match;
  private List _tentacleMappings;

        public VariableEdge           (TypeSystem types, int no,
                                       Node[] nodes);
        public VariableEdge           (TypeSystem types, int no,
                                       Node[] nodes, String vn);
        public VariableEdge           (TypeSystem types, String name,
                                       Node[] nodes);
        public VariableEdge           (TypeSystem types, String name,
                                       Node[] nodes, String vn);

protected void clearMatchAndMappings  ();
 public String getVariableName        ();
   public void setVariableName        (String _vName);
   public List getMatch               ();
   public void setMatch               (List newMatch)
   public void addMatchingComponent   (HypergraphComp component);
   public void removeMatchingComponent(HypergraphComp component);
   public void addTentacleMapping     (int variableTentacle,
                 int matchedEdgeIndex, int matchedEdgeTentacle);
   public void addTentacleMapping     (TentacleMapping mapping);
   public List getTentacleMappings    ();
   public void removeTentacleMapping  (TentacleMapping mapping);
```

Attribute

`private String vName;`
Der Name der durch diese Kante repräsentierten Variable.

`private List _match;`
Die Substitution dieser Variablenkante.

`private List _tentacleMappings;`
Die dieser Variablenkante zugewiesenen `TentacleMappings` (s. Klassenbeschreibung 5.2.1.7).

Konstruktoren

`public VariableEdge(TypeSystem types, int no, Node[] nodes, String vn);`
Erzeugt eine `VariableEdge` anhand der Typnummer.

Parameter:
`types` Das benutzte Typsystem.
`no` Die Typnummer dieser Kante in `types`.
`nodes` Die von dieser Variablenkante besuchten Knoten.
`vn` Der Name der durch diese Variablenkante repräsentierten Dia**plan**-Variable.

`public VariableEdge (TypeSystem types, String name, Node[] nodes, String vn);`
Erzeugt eine `VariableEdge` anhand des Typnamens.
Parameter:
`name` Der Name des Typs dieser Kante in `types`.
Rest wie oben.

Methoden

`protected void clearMatchAndMappings();`
Löscht die Listen für Substitution und Tentakelbilder.

`public String getVariableName();`
Gibt den Namen der durch diese `VariableEdge` repräsentierten Variable zurück.

`public void setVariableName (String _vName);`
Setzt den Namen der durch diese `VariableEdge` repräsentierten Dia**plan**-Variable auf den angegebenen Wert.

`public List getMatch();`
Gibt eine unmodifizierbare Kopie der dieser Kante zugewiesenen Substitution zurück.

`public void setMatch(List newMatch);`
Ersetzt die Substitution dieser Kante durch die in `newMatch` enthaltenen Graphkomponenten.

`public void addMatchingComponent(HypergraphComp component);`
Fügt die angegebenen Graphkomponente zur Substitution dieser Kante hinzu.

`public void removeMatchingComponent(HypergraphComp component);`
Entfernt `component` aus der Substitution dieser Kante.

```
public void addTentacleMapping(int variableTentacle, int matchedEdgeIndex,
                               int matchedEdgeTentacle);
```
Fügt ein neues `TentacleMapping` mit den angegebenen Werten zu dieser `VariableEdge` hinzu.

`public List getTentacleMappings();`
Gibt eine unmodifizierbare Kopie des Attributs `_tentacleMappings` zurück.

`public void removeTentacleMapping(TentacleMapping mapping);`
Entfernt `mapping` aus der entsprechenden Liste.

5.1.4 Prädikatkanten

5.1.4.1 PredicateEdge

Klasse PredicateEdge
Basisklasse Edge
Schnittstellen PredicateHyperedge
Aufgabe Ein Objekt dieser Klasse repräsentiert eine Prädikatkante. Die Klasse stellt zudem eine Reihe statischer Methoden zur Arbeit mit Abschlussklauseln (Unterabschnitt 2.2.1) zur Verfügung. Die Klasse `PredicateTermEdge` delegiert intern sämtliche prädikatspezifischen Methodenaufrufe an diese Klasse weiter und stellt somit einen *Adapter* ([26]) dar.

Eigenschaften

```
             private String predicateName;
            private boolean priority;
 private static final String COMPLETION_SUCCESS_NAME;
 private static final String COMPLETION_FAILURE_NAME;
private static PredicateEdge COMPLETION_CLAUSE_SUCCESS;
private static PredicateEdge COMPLETION_CLAUSE_FAILURE;
  public static final String COMPLETION_CLAUSE_TYPENAME;
  public static final String COMPLETION_CLAUSE_EXCEPTION_TYPENAME;

                      public PredicateEdge(TypeSystem types, int no,
                                           Node[] nodes, String pn);
                      public PredicateEdge(TypeSystem types, int no,
                                           Node[] nodes, String pn,
                                           boolean prior);
                      public PredicateEdge(TypeSystem types, String name,
                                           Node[] nodes, String pn);
                      public PredicateEdge(TypeSystem types, String name,
                                           Node[] nodes, String pn,
                                           boolean prior);

    public static final void initDefaultPredicates      (TypeSystem types);
 public static PredicateEdge getCompletionClauseSuccess ();
 public static PredicateEdge getCompletionClauseFailure ();
              public String getPredicateName            ();
                public void setPriority                (boolean prior);
             public boolean hasPriority                ();
```

Attribute

`private String predicateName;`
Der Name des durch diese Kante repräsentierten Prädikats.

`private boolean priority;`
Zeigt an, ob diese `PredicateEdge` Priorität besitzt, d.h. ob sie zum Bedingungsteil einer Regel bzw. dessen Bild gehört.

`private static final String COMPLETION_SUCCESS_NAME;`
`private static final String COMPLETION_FAILURE_NAME;`
Die vordefinierten Prädikatnamen für die Kanten „Erfolg“ bzw. „Fehlschlag“.

`private static PredicateEdge COMPLETION_CLAUSE_SUCCESS;`
`private static PredicateEdge COMPLETION_CLAUSE_FAILURE;`
Die vorgegebenen Prädikatkanten für „Erfolg“ bzw. „Fehlschlag“. Die Initialisierungen erfolgen in `initDefaultPredicates()`.

`public static final String COMPLETION_CLAUSE_TYPENAME;`
Der vordefinierte Name des Basiskantentyps für Abschlussklauseln. Dient als Basistyp für Ausnahmekanten.

`public static final String COMPLETION_CLAUSE_EXCEPTION_TYPENAME;`
Dient als Name des Basistyps für alle Ausnahmekantentypen, der selber in `initDefaultPredicates()` als vom Typ zu `COMPLETION_CLAUSE_TYPENAME` abgeleitet definiert wird. Die verschiedenen Ausnahmetypen von diesem Basistyp abzuleiten, obliegt der nutzenden Applikation, also in diesem Fall dem prototypischen Interpreter.

Konstruktoren

```
public PredicateEdge(TypeSystem types, int no, Node[] nodes,
                     String pn, boolean prior);
```

Erzeugt eine neue Prädikatkante anhand der Nummer im Typsystem.
Parameter:
`types` Das benutzte Typsystem.
`no` Die Typnummer dieser Kante in `types`.
`nodes` Die von dieser Prädikatkante besuchten Knoten.
`pn` Der Name des durch diese Prädikatkante repräsentierten Dia**plan**-Prädikats.
`prior` Gibt an, ob diese Prädikatkante zum Bedingungsteil einer Regel bzw. dessen Bild gehört.

```
public PredicateEdge(TypeSystem types, String name, Node[] nodes,
                     String pn, boolean prior);
```

Erzeugt eine neue Prädikatkante anhand des Namens im Typsystem.
Parameter:
`name` Der Typname dieser Kante in `types`.
Rest wie oben.

Methoden

`public static final void initDefaultPredicates(TypeSystem types);`
Fügt je einen Kantentypen namens `COMPLETION_CLAUSE_TYPENAME` und `COMPLETION_CLAUSE_EXCEPTION_TYPENAME` zu `types` hinzu, schließt das Typsystem für weitere Typdefinitionen und deklariert die entsprechende Ableitungsrelation. Initialisiert zudem die beiden Standardprädikatkanten für „Erfolg“ und „Fehlschlag“.

`public static PredicateEdge getCompletionClauseSuccess();`
Gibt die vordefinierte Kante für „Erfolg“ zurück.

`public static PredicateEdge getCompletionClauseFailure();`
Gibt die vordefinierte Kante für „Fehlschlag“ zurück.

`public String getPredicatName();`
Gibt den Namen des durch diese Kante repräsentierten Dia**plan**-Prädikats zurück.

`public void setPriority(boolean prior);`
Setzt das Attribut `priority` auf den angegebenen Wert.

`public boolean hasPriority();`
Gibt an, ob diese Prädikatkante Priorität besitzt (s.a. `priority`).

5.1.4.2 PredicateVariableEdge

Klasse	PredicateVariableEdge
Basisklasse	VariableEdge
Schnittstellen	PredicateVariableHyperedge
Aufgabe	Eine Instanz dieser Klasse repräsentiert eine Prädikatvariable. Die prädikatspezifischen Methoden werden intern an eine `PredicateEdge` delegiert, so dass diese Klasse technisch einen Adapter ([26]) zwischen `VariableEdge` und `PredicateHyperedge` (bzw. `PredicateVariableHyperedge`) darstellt. So ist die gleiche Wirkung ebendieser Methoden in `PredicateEdge` und dieser Klasse sicher gestellt. Eine Instanz der Klasse `PredicateVariableTermEdge` benutzt für die variablenspezifischen Methoden das gleiche Objekt wie die Basisklasse `VariableTermEdge` und delegiert genau wie die hier beschriebene Klasse `PredicateVariableEdge` die prädikatspezifischen Methodenaufrufe an eine `PredicateEdge`-Instanz.

Eigenschaften

```
private PredicateHyperedge wrappedPredicateEdge;
public static final String INADEQUATE_MATCH_SIZE;
public static final String INADEQUATE_MATCH_COMPONENT_TYPE;
```

```
public PredicateVariableEdge    (TypeSystem types, int no,
                                 Node[] nodes);
public PredicateVariableEdge    (TypeSystem types, int no,
                                 Node[] nodes, boolean prior);
public PredicateVariableEdge    (TypeSystem types, int no,
                                 Node[] nodes, String vName);
public PredicateVariableEdge    (TypeSystem types, int no,
                                 Node[] nodes, String vName,
                                 boolean prior);
public PredicateVariableEdge    (TypeSystem types, String name,
                                 Node[] nodes);
public PredicateVariableEdge    (TypeSystem types, String name,
                                 Node[] nodes, boolean prior);
public PredicateVariableEdge    (TypeSystem types, String name,
                                 Node[] nodes, String vName);
public PredicateVariableEdge    (TypeSystem types, String name,
                                 Node[] nodes, String vName,
                                 boolean prior);
```

```
 public String getPredicateName         ();
   public void setPriority             (boolean prior);
public boolean hasPriority             ();
   public void setMatch                (List newMatch);
   public void setMatchedEdge          (PredicateHyperedge edge);
public PredicateHyperedge getMatchedEdge ();
   public void addMatchingComponent    (HypergraphComp component);
```

Attribute

`private PredicateHyperedge wrappedPredicateEdge;`
Die Prädikatkante, an die intern alle prädikatspezifischen Methoden delegiert werden.

`public static final String INADEQUATE_MATCH_SIZE;`
`public static final String INADEQUATE_MATCH_COMPONENT_TYPE;`
Standardnachrichten für mögliche Ausnahmen, die durch die Methoden `setMatch()` und `addMatchingComponent()` geweckt werden können.

Konstruktoren

```
public PredicateVariableEdge (TypeSystem types, int no, Node[] nodes,
                              String vName, boolean prior)
```

Erzeugt eine `PredicateVariableEdge` anhand der Nummer im Typsystem.
Parameter:
`types` Das benutzte Typsystem.
`no` Die Typnummer dieser Kante in `types`.
`nodes` Die von dieser Prädikatvariablenkante besuchten Knoten.
`vName` Der Name der durch diese Prädikatvariablenkante repräsentierten Dia**plan**-Prädikatvariablen. Dieser Name wird als Variablenname an die Basisklasse übergeben.
`prior` Gibt an, ob diese Prädikatvariablenkante zum Bedingungsteil einer Regel bzw. dessen Bild gehört.

```
public PredicateVariableEdge (TypeSystem types, String name, Node[] nodes,
                              String vName, boolean prior)
```

Erzeugt eine neue Prädikatvariablenkante anhand des Namens im Typsystem.
Parameter:
`name` Der Typname dieser Kante in `types`.
Rest wie oben.

Methoden

`public String getPredicateName();`
`public void setPriority(boolean prior);`
`public boolean hasPriority();`
Rufen die entsprechenden Methoden der gekapselten Prädikatkante.

`public void setMatch(List newMatch);`
Überschreibt die gleiche Methode in `VariableEdge`. Die übergebene Liste wird genau dann akzeptiert, wenn sie

1. nicht mehr als ein Element enthält (s.a. S. 10, *Prädikattypen*), und
2. falls sie nicht leer ist, das eine enthaltene Element die Schnittstelle `PredicateHyperedge` implementiert.

Ist Bedingung 1 verletzt, wird eine DiaGen-Standardausnahme mit der Nachricht `INADEQUATE_MATCH_SIZE` geweckt; ist Bedingung 2 verletzt, wird eine DiaGen-Standardausnahme mit der Nachricht `INADEQUATE_MATCH_COMPONENT_TYPE` geweckt.

`public void setMatchedEdge(PredicateHyperedge edge);`
Setzt die Substitutionskante auf den angegebenen Wert.

`public PredicateHyperedge getMatchedEdge();`
Gibt die Substitutionskante zurück.

`public void addMatchingComponent(HypergraphComp component);`
Überschreibt die gleiche Methode in `VariableEdge`. Die übergebene Komponente wird genau dann akzeptiert, wenn

1. bisher keine Substitution vorhanden ist, und
2. die übergebene Graphkomponente die Schnittstelle `PredicateHyperedge` implementiert.

Anderenfalls werden Ausnahmen wie bei `setMatch()` geweckt.

5.1.4.3 PredicateException

Klasse	PredicateException
Basisklasse	java.lang.Exception
Aufgabe	Wird benutzt, um eine Ausnahme im Sinne der Abschlussklauseln anzuzeigen.

Eigenschaften

`private PredicateHyperedge edge;`

```
public PredicateException     (String excName,
                               PredicateHyperedge e);
```

`public PredicateHyperedge getRepresentingHyperedge();`

Attribute

`private PredicateHyperedge edge;`
Die dieser `PredicateException` entsprechende Kante im Transformationsgraphen.

Konstruktoren

```
public PredicateException(String excName,
                          PredicateHyperedge e);
```

Erzeugt eine `PredicateException`.
Parameter:
`excName` Der Name dieser `PredicateException`. Fungiert in der JVM als Nachricht.
`e` Die dieser `PredicateException` entsprechende Kante im Transformationsgraphen.

Methoden

`public PredicateHyperedge getRepresentingHyperedge();`
Gibt die diese `PredicateException` repräsentierende Kante zurück.

5.2 Der erweiterte Transformationsalgorithmus

Dieser Abschnitt beschreibt die Implementierung der in Abschnitt 2.2 geschilderten Konzepte als Erweiterung der in Abschnitt 4.2 vorgestellten Klassen. Die Reihenfolge entspricht der aus Abschnitt 4.2. Soweit nicht explizit anders erwähnt, befinden sich die hier vorgestellten Klassen und Schnittstellen im Paket `diagen.hypergraph.transformer`.

5.2.1 Die neuen Selektoren

Wie aus Abschnitt 4.3 hervorgeht, bilden die Schnittstelle `EdgeSelector` sowie die Klassen `PlainEdgeSelector` und `SameNode` das Herzstück der Ansatzsuche in DiaGen. Die für den **Diaplan**-Interpreter angepasste Ansatzsuche arbeitet nach demselben Prinzip, allerdings erfordern die neuen Konzepte andere Selektoren. So sind jetzt z.B. nicht mehr nur Kanten, sondern auch Knoten relevante Bestandteile der linken wie rechten Regelseiten; daher wurden die Schnittstelle `EdgeSelector` durch eine neue Schnittstelle `ComponentSelector` und die Klasse `SameNode` durch die neuen Klassen `SameComponent` und die davon abgeleitete `SameEdge` ersetzt. Einen Überblick über alle Selektoren bietet Abb. 5.2 auf der nächsten Seite.

5.2.1.1 SameComponent

Klasse	SameComponent
Basisklasse	java.lang.Object
Aufgabe	Ersetzt die Klasse `SameNode` (Unterabschnitt 4.2.3). Die Funktionsweise bleibt gleich; nur die Attributnamen wurden geändert, da jetzt auch auf gleiche Kanten an unterschiedlichen Knoten getestet wird.

Eigenschaften

```
public static final int UNUSED;
final int thisConnector;
final int otherComponentIndex;
final int otherConnector;

   public SameComponent       (int thisConnector, int otherComponentIndex);
   public SameComponent       (int thisConnector, int otherComponentIndex,
                               int otherConnector);
```

Attribute

`public static final int UNUSED;`
Hat eines der drei anderen Attribute diesen Wert, so ist sie für die durch dieses `SameComponent`-Objekt repräsentierte Beziehung bedeutungslos.

`final int thisConnector;`
Zeigt an, mit welchem Konnektor die Musterkomponente an der Beziehung teilnimmt.

`final int otherComponentIndex;`
Der Index im Array bereits gefundener Komponenten der Komponente jenseits der durch diese `SameComponent` repräsentierten Beziehung.

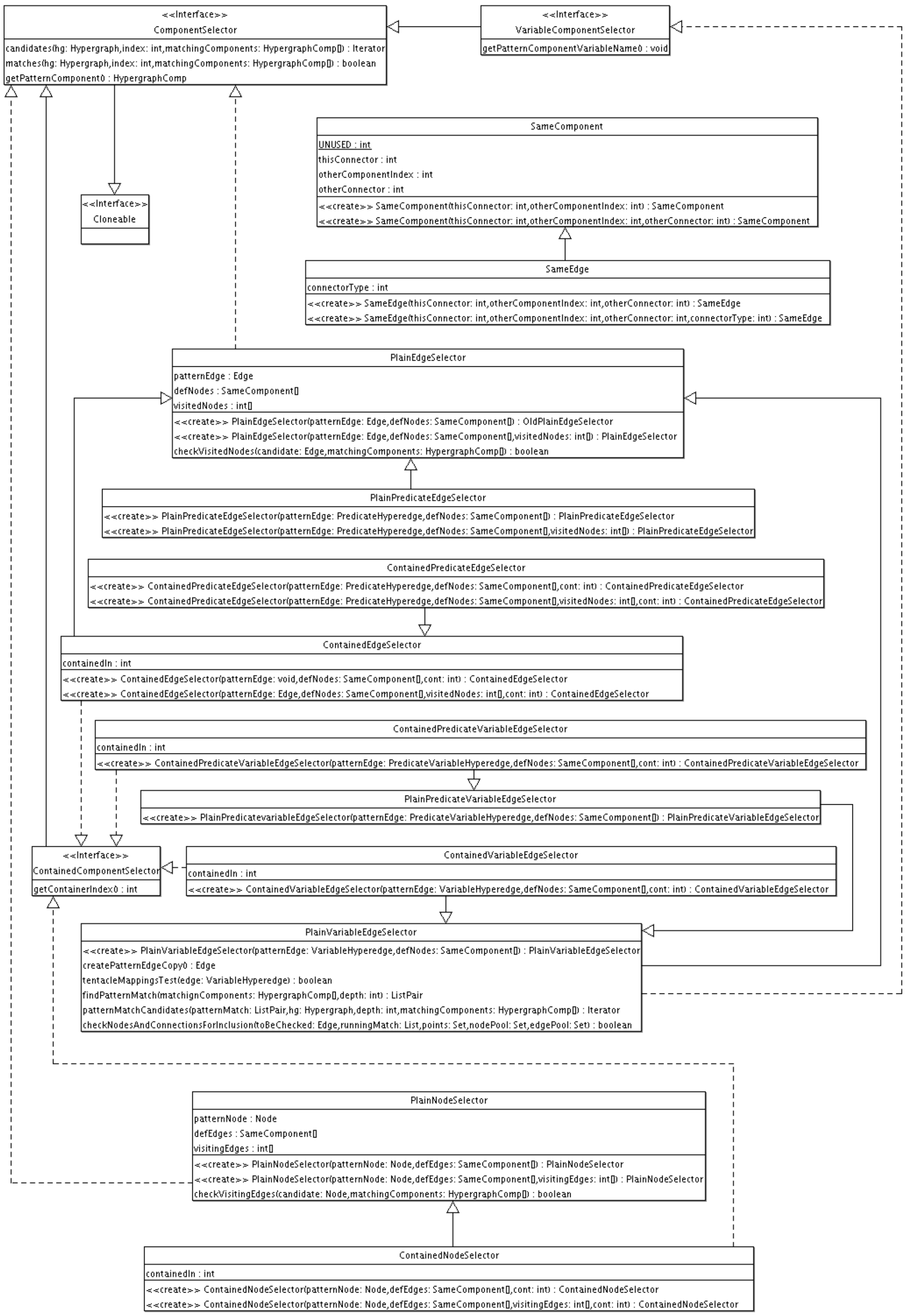


Abbildung 5.2: Die neuen Selektoren

`final int otherConnector;`
Zeigt an, mit welchem Konnektor die Komponente jenseits der durch diese `SameComponent` repräsentierten Beziehung an ebendieser Beziehung teilnimmt. Bei der Verbindung zweier Knoten durch eine Kante wird dieser Wert auf `UNUSED` gesetzt, sofern es sich nicht um eine `SameEdge` (Klassenbeschreibung 5.2.1.2) handelt.

Konstruktoren

```
public SameComponent(int thisConnector, int otherComponentIndex,
                     int otherConnector);
```

Erzeugt eine `SameComponent` mit den angegebenen Werten.

5.2.1.2 SameEdge

Klasse SameEdge
Basisklasse SameComponent
Aufgabe Repräsentiert die Verbindung zweier Knoten durch eine Kante.
Eigenschaften

```
final int connectorType;

public SameEdge      (int thisConnector, int otherComponentIndex,
                      int otherConnector);
public SameEdge      (int thisConnector, int otherComponentIndex,
                      int otherConnector, int connectorType);
```

Attribute

`final int connectorType;`
Der Typ der die beiden durch diese `SameEdge` miteinander in Beziehung stehenden Knoten verbindenden Kante.

Konstruktoren

```
public SameEdge(int thisConnector, int otherComponentIndex,
                int otherConnector, int connectorType);
```

Erzeugt eine `SameEdge` mit den angegebenen Werten.

5.2.1.3 PlainNodeSelector

Klasse PlainNodeSelector
Basisklasse java.lang.Object
Schnittstellen ComponentSelector
Aufgabe Repräsentiert einen nicht in einem Behälter enthaltenen Knoten der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.
Eigenschaften

```
     protected final Node patternNode;
protected SameComponent[] defEdges;
    protected final int[] visitingEdges;

               public PlainNodeSelector(Node patternNode,
                                        SameComponent[] defEdges);
               public PlainNodeSelector(Node patternNode,
                                        SameComponent[] defEdges,
                                        int[] visitingEdges);
```

```
      protected boolean checkVisitingEdges(Node candidate,
                                   HypergraphComp[] matchingComponents);
        public Iterator candidates(Hypergraph hg, int depth,
                                   HypergraphComp[] matchingComponents);
  public HypergraphComp getPatternComponent();
         public boolean matches   (Hypergraph hg, int depth,
                                   HypergraphComp[] matchingComponents);
```

Attribute

`protected final Node patternNode;`
Der durch diesen `PlainNodeSelector` repräsentierte Knoten der linken Regelseite.

`protected final SameComponent[] defEdges;`
Die Verbindungen zu im `ProdPattern`-Baum höher befindlichen Knoten der linken Regelseite.

`protected final int[] visitingEdges;`
Die Indizes den durch diesen `PlainNodeSelector` repräsentierten Knoten besuchender im `ProdPattern`-Baum höher befindlicher Kanten der linken Regelseite. Diese sind besonders interessant bei Kanten der Stelligkeit 1, da solche naturgemäß nicht bei den `defEdges` berücksichtigt werden.

Konstruktoren

```
public PlainNodeSelector (Node patternNode, SameComponent[] defEdges,
                          int[] visitingEdges);
```
Erzeugt einen `PlainNodeSelector` mit den angegebenen Werten.

Methoden

```
protected boolean checkVisitingEdges(Node candidate,
                                 HypergraphComp[] matchingComponents);
```
Prüft, ob alle Komponenten in `matchingComponents` an den mit Elementen aus `visitingEdges` indizierten Stellen Kanten sind und den Knoten `candidate` besuchen.

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           final HypergraphComp[] matchingComponents);
```
Zählt die als Bild des `patternNode` in Frage kommenden Kandidaten aus `hg` auf. Als Kandidaten gelten alle Knoten aus `hg`, die

1. mit anderen Knoten in `matchingComponents` durch die im Attribut `defEdges` beschriebenen Kanten verbunden sind,
2. von den durch das Attribut `visitingEdges` beschriebenen Kanten in `matchingComponents` besucht werden,
3. in ihren Behältereigenschaften `patternNode` entsprechen, und
4. nicht bereits in `matchingComponents` vorhanden sind (s.a. S.49, Punkt 3).

Punkt 3 bedeutet:

- Ist `patternNode` ein Behälter, so muss das auch für die Kandidaten gelten.
- Ist `patternNode` ein leerer Behälter (also ohne bzw. mit leerem inneren Graph), so muss dies ebenfalls für die Kandidaten gelten.

`public HypergraphComp getPatternComponent()`
Gibt den als Vorlage dienenden Knoten zurück.

```
public boolean matches(Hypergraph hg, int depth,
                        HypergraphComp[] matchingComponents);
```
Prüft, ob die Komponente an Stelle `depth` in `matchingComponents` zu den Vorgaben dieses `PlainNodeSelector` passt. Die Tests sind die gleichen wie für die Kandidaten in der Methode `candidates()`.

5.2.1.4 ContainedNodeSelector

Klasse	ContainedNodeSelector
Basisklasse	PlainNodeSelector
Schnittstellen	ContainedComponentSelector
Aufgabe	Repräsentiert einen in einem Behälter enthaltenen Knoten der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
private final int containedIn;
          public ContainedNodeSelector(Node patternNode, SameComponent[] defEdges,
                                       int cont);
          public ContainedNodeSelector(Node patternNode, SameComponent[] defEdges,
                                       int[] visitingEdges, int cont);

  public Iterator candidates          (Hypergraph hg, int depth,
                                       HypergraphComp[] matchingComponents);
   public boolean matches             (Hypergraph hg, int depth,
                                       HypergraphComp[] matchingComponents);
       public int getContainerIndex   ();
```

Attribute

`private final int containedIn;`
Der Index des Behälters im Abbildungsarray der linken Seite, in dem der durch diesen Selektor beschriebene Knoten enthalten ist.

Konstruktoren

```
public ContainedNodeSelector(Node patternNode, SameComponent[] defEdges,
                             int[] visitingEdges, int cont);
```
Erzeugt einen `ContainedNodeSelector`.

Parameter:

`cont` Der Index des Behälters des durch diesen Selektor beschriebenen Knotens.

Rest siehe Attributbeschreibungen von `PlainNodeSelector`.

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           HypergraphComp[] matchingComponents);
```
Sucht aus `matchingComponents` die Komponente an Stelle `containedIn` und ruft die `candidates()`-Methode der Basisklasse mit dem darin enthaltenen Graphen als erstem Parameter. Ist die entsprechende Komponente kein Behälter oder leer, wird eine Ausnahme geweckt, da dem Behälter eine falsche Komponente zugeordnet wurde.

`public boolean matches(Hypergraph hg, int depth,`
`HypergraphComp[] matchingComponents);`

Prüft zusätzlich zu den Tests in der gleichen Methode der Basisklasse, ob der Graph, dem die Komponente an der Stelle `depth` in `matchingComponents` angehört, in der Komponente an der mit `containedIn` indizierten Stelle enthalten ist.

`public int getContainerIndex();`

Gibt den Wert des Attributs `containedIn` zurück.

5.2.1.5 PlainEdgeSelector

Klasse	PlainEdgeSelector
Basisklasse	java.lang.Object
Schnittstellen	ComponentSelector
Aufgabe	Repräsentiert eine nicht in einem Behälter enthaltene Kante der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
     protected final Edge patternEdge;
protected SameComponent[] defNodes;
    protected final int[] visitedNodes;

                   public PlainEdgeSelector(Edge patternEdge,
                                            SameComponent[] defNodes);
                   public PlainEdgeSelector(Edge patternEdge,
                                            SameComponent[] defNodes,
                                            int[] visitedNodes)

        protected boolean checkVisitedNodes(Edge candidate,
                                   HypergraphComp[] matchingComponents);
          public Iterator candidates       (Hypergraph hg, int depth,
                                   HypergraphComp[] matchingComponents);
    public HypergraphComp getPatternComponent();
           public boolean matches          (Hypergraph hg, int depth,
                                   HypergraphComp[] matchingComponents);
```

Attribute

`protected final Edge patternEdge;`

Die durch diesen `PlainEdgeSelector` repräsentierte Kante der linken Regelseite.

`protected final SameComponent[] defNodes;`

Die mit im `ProdPattern`-Baum höher befindlichen Kanten der linken Regelseite gemeinsamen Knoten.

`protected final int[] visitedNodes;`

Die Indizes der von der durch diesen `PlainEdgeSelector` repräsentierten Kante besuchten im `ProdPattern`-Baum höher befindlichen Knoten der linken Regelseite. Diese sind besonders interessant bei einer Kante der Stelligkeit 1, da solche naturgemäß keine `defNodes` besitzen.

Konstruktoren

```
public PlainEdgeSelector (Edge patternEdge, SameComponent[] defNodes,
                          int[] visitedNodes);
```

Erzeugt einen `PlainEdgeSelector` mit den angegebenen Werten.

Methoden

```
protected boolean checkVisitingEdges(Node candidate,
                                     HypergraphComp[] matchingComponents);
```

Prüft, ob alle Komponenten in `matchingComponents` an den mit Elementen aus `visitingEdges` indizierten Stellen Kanten sind und den Knoten `candidate` besuchen.

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           final HypergraphComp[] matchingComponents);
```

Zählt die als Bild der `patternEdge` in Frage kommenden Kandidaten aus `hg` auf. Als Kandidaten gelten alle Kanten aus `hg`, die

1. vom gleichen Typ wie `patternEdge` oder einem davon abgeleiteten sind,
2. die gleichen Knoten in `matchingComponents` wie die im Attribut `defNodes` beschriebenen Kanten besuchen,
3. die durch das Attribut `visitedNodes` beschriebenen Knoten in `matchingComponents` besuchen,
4. in ihren Behältereigenschaften `patternEdge` entsprechen, und
5. nicht bereits in `matchingComponents` vorhanden sind (s.a. S.49, Punkt 3).

Zu Punkt 4 siehe die entsprechende Anmerkung bei der Beschreibung der gleichnamigen Methode in Klassenbeschreibung 5.2.1.3.

```
public HypergraphComp getPatternComponent()
```

Gibt die als Vorlage dienende Kante zurück.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Prüft, ob die Komponente an Stelle `depth` in `matchingComponents` zu den Vorgaben dieses `PlainEdgeSelectors` passt. Die Tests sind die gleichen wie für die Kandidaten in der Methode `candidates()`.

5.2.1.6 ContainedEdgeSelector

Klasse	ContainedEdgeSelector
Basisklasse	PlainEdgeSelector
Schnittstellen	ContainedComponentSelector
Aufgabe	Repräsentiert eine in einem Behälter enthaltene Kante der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
protected int containedIn;
```

```
public ContainedEdgeSelector(Edge patternEdge, SameComponent[] defNodes,
                             int cont);
public ContainedEdgeSelector(Edge patternEdge, SameComponent[] defNodes,
                             int[] visitedNodes, int cont);
```

```
public Iterator candidates       (Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
public boolean matches           (Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
public int getContainerIndex();
```

Attribute

`private final int containedIn;`

Der Index des Behälters im Abbildungsarray der linken Seite, in dem die durch diesen Selektor beschriebene Kante enthalten ist.

Konstruktoren

```
public ContainedEdgeSelector(Edge patternEdge,
       SameComponent[] defNodes, int[] visitedNodes, int cont);
```

Erzeugt einen `ContainedEdgeSelector`.

Parameter:

`cont` Der Index des Behälters der durch diesen Selektor beschriebenen Kante.

Rest siehe Attributbeschreibungen von `PlainEdgeSelector`.

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           HypergraphComp[] matchingComponents);
```

Sucht aus `matchingComponents` die Komponente an Stelle `containedIn` und ruft die `candidates()`-Methode der Basisklasse mit dem darin enthaltenen Graphen als erstem Parameter. Ist die entsprechende Komponente kein Behälter oder leer, wird eine Ausnahme geweckt, da dem Behälter eine falsche Komponente zugeordnet wurde.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Prüft zusätzlich zu den Tests in der gleichen Methode der Basisklasse, ob der Graph, dem die Komponente an der Stelle `depth` in `matchingComponents` angehört, in der Komponente an der mit `containedIn` indizierten Stelle enthalten ist.

```
public int getContainerIndex();
```

Gibt den Wert des Attributs `containedIn` zurück.

5.2.1.7 TentacleMapping

Klasse	TentacleMapping
Basisklasse	java.lang.Object
Aufgabe	Gibt an, wie die Tentakel einer Variable ihrer Substitution zugeordnet sind. Ohne Einsatz von Gestalten könnte jedes Tentakel einer Substitutionskante, das keinem der zur Substitution gehörenden Knoten zugeordnet ist, im Bild der rechten Seite jedem Tentakel der Variable zugeordnet werden. Die Klasse `TentacleMapping` macht diese Abbildung eindeutig, indem sie für jedes „Außenententakel“ jeder Substitutionskante die entsprechende Tentakelnummer der Variable speichert. Idee und Klassenstruktur ähneln sehr der Klasse `SameComponent`.

Eigenschaften

```
protected int variableTentacle;
protected int matchedEdgeIndex;
protected int matchedEdgeTentacle;

       public TentacleMapping       (int vTentacle, int mEIndex,
                                     int mETentacle);

   public int getVariableTentacle   ();
   public int getMatchedEdgeIndex   ();
   public int getMatchedEdgeTentacle();
```

Attribute

`protected int variableTentacle;`
Der Arm der Variablenkante, der das durch dieses `TentacleMapping` repräsentierte Bild darstellt.

`protected int matchedEdgeIndex;`
Die Substitutionskante, deren Arm durch dieses `TentacleMapping` abgebildet wird.

`protected int matchedEdgeTentacle;`
Der durch dieses `TentacleMapping` abgebildete Arm der Substitutionskante.

Konstruktoren

`public TentacleMapping(int vTentacle, int mEIndex, int mETentacle);`
Erzeugt ein `TentacleMapping` mit den entsprechenden Werten.

Methoden

`public int getVariableTentacle();`
Gibt den Wert des Attributs `variableTentacle` aus.

`public int getMatchedEdgeIndex();`
Gibt den Wert des Attributs `matchedEdgeIndex` aus.

`public int getMatchedEdgeTentacle();`
Gibt den Wert des Attributs `matchedEdgeTentacle` aus.

5.2.1.8 PlainVariableEdgeSelector

Klasse	PlainVariableEdgeSelector
Basisklasse	PlainEdgeSelector
Schnittstellen	VariableComponentSelector
Aufgabe	Repräsentiert eine nicht in einem Behälter enthaltene Variable der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
           public PlainVariableEdgeSelector(VariableHyperedge patternEdge,
                                            SameComponent[] defNodes);

    protected Edge createPatternEdgeCopy();
 protected boolean tentacleMappingsTest (VariableHyperedge edge);
protected ListPair findPatternMatch(HypergraphComp[] match,
                                    int depth);
```

```
protected Iterator patternMatchCandidates (ListPair patternMatch,
                                  Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
   private boolean checkNodesAndConnectionsForInclusion
                    (Edge toBeChecked, List runningMatch,
                     Set points, Set nodePool, List edgePool);
   public Iterator candidates      (Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
    public boolean matches         (Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
     public String getPatternComponentVariableName();
```

Konstruktoren

```
public PlainVariableEdgeSelector (VariableHyperedge patternEdge,
                                  SameComponent[] defNodes);
```

Erzeugt einen `PlainVariableEdgeSelector` mit den angegebenen Werten (s.a. Klassenbeschreibung 5.2.1.5).

Methoden

```
protected Edge createPatternEdgeCopy();
```

Erzeugt eine Kopie der Musterkante, die während der Ansatzsuche als Bild dient.

```
protected final boolean tentacleMappingsTest (VariableHyperedge edge);
```

Prüft, ob alle Knoten, die von Kanten in der `edge` zugeordneten Substitution besucht werden, aber nicht dazugehören, in den `edge` zugeordneten `TentacleMappings` berücksichtigt werden. Ist dem so, ist der Rückgabewert `true`; sonst `false`.

```
protected ListPair findPatternMatch(HypergraphComp[] match, int depth);
```

Prüft, ob in `match` vor dem Index `depth` bereits eine Variable des gleichen Namens wie die Musterkante enthalten ist. Gibt es eine solche Variable, setzt sich das zurückgegebene `ListPair`[2] aus der Substitution und den `TentacleMappings` dieser Variable zusammen; existiert vor dem angegebenen Index keine entsprechende Variable in `match`, ist der Rückgabewert `null`.

```
protected Iterator patternMatchCandidates
                   (ListPair patternMatch, Hypergraph hg,
                    int depth, HypergraphComp[] matchingComponents);
```

Wird im Falle einer nicht linkslinearen Regel[3] gerufen. Hat die Methode `findPatternMatch()` eine passende Variable gefunden, sucht diese Methode im Graphen `hg` nach anderen Komponenten, die eine äquivalente Substitution ergeben. Die Substitution der ersten gefundenen Variablen fungiert also als Modell für alle weiteren Auftreten dieser Variablen innerhalb der linken Regelseite. Für die Suche nach der neuen Variablensubstitution generiert sich diese Methode ein eigenes `TransformationSet`, das nur aus einer „Hilfsregel“ besteht, welche die Substitution der ersten Variable auf sich selbst abbildet. Für jede gefundene Substitution, die den Injektivitätstest (S. 49, Punkt 3; Klassenbeschreibung 5.2.4) besteht, wird dem `Iterator` eine entsprechende Kandidatenvariable angehängt.

[2] `ListPair` ist eine neue Hilfsklasse zur Repräsentation eines geordneten Paares zweier Listen, die zu Vergleichszwecken eine eigene `hashCode()`-Methode implementiert.

```
private boolean checkNodesAndConnectionsForInclusion
   (Edge toBeChecked, List runningMatch, Set points,
    Set nodePool, List edgePool);
```

Füllt durch Aufbau eines Pfads rekursiv eine Liste `runningMatch`, welche die Substitution einer Variable darstellt. Die in `points` enthaltenen Knoten werden von der Variable besucht und dienen somit als Endpunkte der möglichen Pfade. Der Algorithmus arbeitet folgendermaßen:

1. Ist die Kante `toBeChecked` bereits in `runningMatch` enthalten, gib `true` zurück.
2. Ist `toBeChecked` nicht in `edgePool` enthalten (was üblicherweise bedeutet, dass die Kante bereits dem Bild einer anderen Komponente zugeordnet wurde), gib `false` zurück.
3. Füge `toBeChecked` zu `runningMatch` hinzu. Prüfe für jeden von dieser Kante besuchten Knoten, ob dieser sich in `nodePool` befindet. Ist dem nicht so, darf der Knoten nicht der Substitution zugeordnet werden, weswegen die Methode wiederum `false` zurückgibt.
4. Gehört der Knoten jedoch zu `nodePool`, prüfe auf Zugehörigkeit zu `points` bzw. `runningMatch`. Im positiven Fall fahre mit dem nächsten besuchten Knoten fort; im negativen füge diesen Knoten `runningMatch` hinzu, iteriere über alle diesen Knoten besuchenden Kanten und starte rekursiv mit der jeweiligen Kante als `toBeChecked` bei Schritt 1. Ist der Rückgabewert des rekursiven Aufrufs `false`, so gib ebenfalls `false` zurück.
5. Ist die Iteration über alle von `toBeChecked` besuchten Knoten abgeschlossen, antworte `true`.

Der Aufruf von außen erfolgt stets aus der Methode `candidates()` (s.u.) mit einer bisher nicht zugeordneten Kante, die einen der auch von der Variable besuchten Knoten besucht. In `edgePool` sind alle nicht anderweitig zugeordneten Kanten, in `nodePool` die von der Variable besuchten sowie ebenfalls alle nicht anderweitig zugeordneten Knoten des jeweiligen Graphen enthalten. Somit enthält die Liste `runningMatch` am Schluss einen oder mehrere Pfade, deren Enden entweder innerhalb oder „direkt an“ der Variablensubstitution liegen. Jetzt zeigt sich, warum auf Seite 11 unter *Eindeutige Substitutionen* für den Fall eines zusammenhängenen Graphen gefordert wird, dass keine zwei Variablen einen oder mehrere Knoten gemeinsam haben: Der Algorithmus könnte zwar den kompletten Substitutionspfad problemlos erstellen, aber im Allgemeinen ließe dieser sich nicht eindeutig einer der beiden Variablen zuordnen.

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           final HypergraphComp[] matchingComponents);
```

Ist vor dem aktuellen Index `depth` bereits eine andere Variable gleichen Namens vorhanden (s.a. `findPatternMatch()`), gibt diese Methode das Resultat von `patternMatchCandidates()` zurück.

[3] Eine Regel heißt linkslinear, wenn in ihrer linken Seite jede Variable höchstens einmal vorkommt.

Gibt es keine solche Variable, erfolgt ein Test, ob über das Attribut `defNodes` Verbindungen zu anderen Kanten existieren; ist der Test negativ, weckt diese Methode eine Ausnahme, da die Variable offenbar in der linken Seite alleine steht (s.a. Unterabschnitt 2.2.1).[4]
Sind jedoch Verbindungen zu anderen Kanten vorhanden, erzeugt folgender Algorithmus die Kandidaten mit passenden Substitutionen:

1. Sammle alle durch das Attribut `defNodes` beschriebenen Knoten in einer Menge `points`. Es ist unbedingt erforderlich, dass alle diese Knoten bereits in `matchingComponents` enthalten sind (s.a. Klassenbeschreibung 5.2.2.3).
2. Bilde die Menge `nodePool` der für die Substitution zur Verfügung stehenden Knoten aus `points` und den noch nicht in `matchingComponents` enthaltenen Knoten von `hg`.
3. Bilde mit Hilfe von `checkNodesAndConnectionsForInclusion()` die maximale zusammenhängende Substitution. Dazu wird über alle noch nicht zugeordneten Kanten iteriert, die wenigstens einen Knoten aus `points` besuchen. Gibt die Hilfsmethode an irgendeiner Stelle `false` zurück, ist das Resultat von `candidates()` ein leerer `Iterator`. Lautet jedoch für alle Kanten der Rückgabewert `true`, enthält die Liste `runningMatch` die maximal mögliche Substitution.
4. Erzeuge eine Kopie der Musterkante mit `createPatternEdgeCopy()` und setze `runningMatch` als ihre Substitution. Erstelle die fehlenden `TentacleMappings` unter Zuhilfenahme des Attributs `defNodes`. Dazu ist es unabdingbar, dass jeder von einer Variable besuchte Knoten in der linken Regelseite von mindestens einer konstanten Kante besucht wird. Für nicht in einem Behälter enthaltene Kanten ist diese Forderung auch inhaltlich sinnvoll, um die möglichen Substitutionen einzugrenzen (s.a. Unterabschnitt 2.2.1, „Benutzung von Variablen").
5. Werden alle „offenen Enden" der zur Substitution gehörenden Kanten durch die `TentacleMappings` abgedeckt, ist die erzeugte Variable, sofern sie auch den Injektivitätstest besteht (Klassenbeschreibung 5.2.4), das erste vom zurückgegebenen `Iterator` aufgezählte Objekt.
6. Das zweite Objekt ist eine Variable, der als Substitution *alle* nicht zugeordneten Objekte aus `hg` zugewiesen werden, ggf. mit angepassten `TentacleMappings`.
7. Egal ob das erste und zweite Objekt die entsprechenden Tests bzgl. Injektivität oder offener Enden bestehen; als letztes wird eine Variable mit leerer Substitution generiert, da der leere Graph immer erlaubt ist.

```
public boolean matches (Hypergraph hg, int depth,
                        HypergraphComp[] matchingComponents);
```

Prüft zusätzlich zu den Tests der Basisklasse, ob

1. es sich bei der entsprechenden Komponente um eine Variable handelt,
2. der Variablenname dem der Musterkante entspricht, und
3. alle offenen Enden der Substitutionskanten durch die zugewiesenen `TentacleMappings` abgedeckt werden.

[4]Dieser Test könnte zwar schon bei der Generierung des Selektors erfolgen bzw. selbige verhindern, aber die entsprechende Methode in `ContainedVariableEdgeSelector` (Klassenbeschreibung 5.2.1.9) basiert auf diesem Effekt.

`public String getPatternComponentVariableName()`
Gibt den Variablennamen der durch diesen Selektor codierten Musterkante zurück.

5.2.1.9 ContainedVariableEdgeSelector

Klasse	ContainedVariableEdgeSelector
Basisklasse	PlainVariableEdgeSelector
Schnittstellen	ContainedComponentSelector
Aufgabe	Repräsentiert eine in einem Behälter enthaltene Variable der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
   private int containedIn;
        public ContainedVariableEdgeSelector(VariableHyperedge patternEdge,
                                        SameComponent[] defNodes, int cont);

public Iterator candidates       (Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
 public boolean matches          (Hypergraph hg, int depth,
                                  HypergraphComp[] matchingComponents);
     public int getContainerIndex();
```

Attribute

`private final int containedIn;`
Der Index des Behälters im Abbildungsarray der linken Seite, in dem die durch diesen Selektor beschriebene Kante enthalten ist.

Konstruktoren

```
public ContainedVariableEdgeSelector(VariableHyperedge patternEdge,
                                     SameComponent[] defNodes, int cont);
```

Erzeugt einen `ContainedVariableEdgeSelector` mit den angegebenen Werten (s.a. Klassenbeschreibung 5.2.1.6).

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           HypergraphComp[] matchingComponents);
```

Ruft ähnlich wie `ContainedEdgeSelector.candidates()` (Klassenbeschreibung 5.2.1.6) die überschriebene Methode der Basisklasse mit dem in der als Behälter identifizierten Komponente enthaltenen Graphen als Parameter `hg`. Für eine enthaltene Variable können zwei Spezialfälle auftreten:

1. Der identifizierte Behälter ist leer. Dann zählt der von dieser Methode zurückgegebene `Iterator` eine Variable mit leerer Substitution auf, sofern für den Variablennamen nicht bereits eine andere Substitution vorliegt (s.a. Klassenbeschreibung 5.2.1.8, `findPatternMatch()`). In letzterem Fall wird eine Ausnahme geweckt, da offenbar die Behälterkomponente nicht richtig zugeordnet wurde.

2. Die Methode der Basisklasse weckt eine Ausnahme, weil die Variable alleine im Graphen steht (s.a. S. 78, `candidates()`). Dann erstellt diese Methode eine Variable mit allen noch nicht zugeordneten Komponenten des in der als Behälter identifizierten Komponente enthaltenen Graphen als Substitution. Hat diese keine offenen Enden (mangels vorhandener `defNodes` können keine `TentacleMappings` erstellt werden), stellt sie das erste vom zurückgegebenen `Iterator` aufgezählte Objekt dar.[5]
3. Unabhängig von der „Vollsubstitution“ zählt der `Iterator` immer noch eine Variable mit leerer Substitution auf.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Testet zusätzlich zu den Tests in der Basisklassenmethode die Behälterzugehörigkeit der Komponente an der Stelle `depth` in `matchingComponents`.

```
public int getContainerIndex();
```

Gibt den Wert des Attributs `containedIn` zurück.

5.2.1.10 PlainPredicateEdgeSelector

Klasse PlainPredicateEdgeSelector
Basisklasse PlainEdgeSelector
Aufgabe Repräsentiert eine nicht in einem Behälter enthaltene Prädikatkante der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
        public PlainPredicateEdgeSelector
              (PredicateHyperedge patternEdge, SameComponent[] defNodes);
        public PlainPredicateEdgeSelector
              (PredicateHyperedge patternEdge, SameComponent[] defNodes,
               int[] visitedNodes);

public Iterator candidates(Hypergraph hg, int depth,
                           HypergraphComp[] matchingComponents);
 public boolean matches   (Hypergraph hg, int depth,
                           HypergraphComp[] matchingComponents);
```

Konstruktoren

```
public PlainPredicateEdgeSelector(PredicateHyperedge patternEdge,
                      SameComponent[] defNodes, int[] visitedNodes);
```

Erzeugt einen `PlainPredicateEdgeSelector` mit den angegebenen Werten (s.a. Klassenbeschreibung 5.2.1.5).

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           final HypergraphComp[] matchingComponents);
```

Zählt die gleichen Objekte auf wie die überschriebene Methode der Basisklasse, sofern es sich bei diesen Objekten um Instanzen einer die Schnittstelle `PredicateHyperedge` implementierenden Klasse handelt.

[5] Diese Substitution muss nicht mit einer möglicherweise bereits in `matchingComponents` enthaltenen Variable verglichen werden, weil eine solche nicht existieren kann: Der entsprechende Test erfolgt in der Basisklassenmethode *vor* dem auf Anbindungen an konstante Kanten.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Testet zusätzlich zu den Tests in der Basisklassenmethode, ob es sich bei der Komponente an Stelle `depth` in `matchingComponents` um eine Instanz einer die Schnittstelle `PredicateHyperedge` implementierenden Klasse handelt.

5.2.1.11 ContainedPredicateEdgeSelector

Klasse	ContainedPredicateEdgeSelector
Basisklasse	ContainedEdgeSelector
Aufgabe	Repräsentiert eine in einem Behälter enthaltene Prädikatkante der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
public ContainedPredicateEdgeSelector(PredicateHyperedge patternEdge,
                SameComponent[] defNodes, int cont);
public ContainedPredicateEdgeSelector(PredicateHyperedge patternEdge,
                SameComponent[] defNodes, int[] visitedNodes, int cont);

public Iterator candidates(Hypergraph hg, int depth,
                           HypergraphComp[] matchingComponents);
 public boolean matches   (Hypergraph hg, int depth,
                           HypergraphComp[] matchingComponents);
```

Konstruktoren

```
public ContainedPredicateEdgeSelector(PredicateHyperedge patternEdge,
                SameComponent[] defNodes, int[] visitedNodes, int cont);
```

Erzeugt einen `ContainedPredicateEdgeSelector` mit den angegebenen Werten (s.a. Klassenbeschreibung 5.2.1.6).

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           final HypergraphComp[] matchingComponents);
```

Zählt die gleichen Objekte auf wie die überschriebene Methode der Basisklasse, sofern es sich bei diesen Objekten um Instanzen einer die Schnittstelle `PredicateHyperedge` implementierenden Klasse handelt.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Testet zusätzlich zu den Tests in der Basisklassenmethode, ob es sich bei der Komponente an Stelle `depth` in `matchingComponents` um eine Instanz einer die Schnittstelle `PredicateHyperedge` implementierenden Klasse handelt.

5.2.1.12 PlainPredicateVariableEdgeSelector

Klasse	PlainPredicateVariableEdgeSelector
Basisklasse	PlainVariableEdgeSelector
Aufgabe	Repräsentiert eine nicht in einem Behälter enthaltene Prädikatvariable der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
        public PlainPredicateVariableEdgeSelector
                     (PredicateVariableHyperedge patternEdge,
                      SameComponent[] defNodes);
public Iterator candidates(Hypergraph hg, int depth,
                           HypergraphComp[] matchingComponents);
 public boolean matches   (Hypergraph hg, int depth,
                           HypergraphComp[] matchingComponents);
```

Konstruktoren

```
public PlainPredicateVariableEdgeSelector
       (PredicateVariableHyperedge patternEdge, SameComponent[] defNodes);
```

Erzeugt einen `PlainPredicateVariableEdgeSelector` mit den angegebenen Werten (s.a. Klassenbeschreibung 5.2.1.5).

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           final HypergraphComp[] matchingComponents);
```

Da die Substitution einer Prädikatvariablen nur aus genau einer (Prädikat-)Kante bestehen kann, gestaltet sich die Kandidatensuche bedeutend einfacher als bei allgemeinen Variablen:

Fall 1. Es ist bereits eine Prädikatvariable gleichen Namens vor dem aktuellen Index `depth` in `matchingCompoennts` vorhanden. Dann sucht diese Methode aus allen nicht zugeordneten Prädikatkanten von `hg` diejenigen aus, die in Typ und Prädikatnamen mit der Substitution der existenten Prädikatvariable übereinstimmen. Jede dieser Kanten dient als Substitution einer durch den zurückgegebenen `Iterator` aufgezählten Prädikatvariablen. Die `TentacleMappings` werden hierbei so gesetzt, dass jeder Arm der Substitutionskante auf den gleichen Arm der Prädikatvariable abgebildet wird.

Fall 2. Es gibt keine solche Prädikatvariable in `matchingComponents` vor dem Index `depth`, und diese Prädikatvariable hat keine durch das Attribut `defNodes` festgelegten Anbindungen an andere (konstante) Kanten. Dann weckt diese Methode analog zum gleichen Fall in der Basisklasse (Klassenbeschreibung 5.2.1.8 eine Ausnahme, da die Prädikatvariable alleine steht.

Fall 3. Es gibt keine gleichnamige Variable in `matchingComponents` vor dem Index `depth`, aber das Attribut `defNodes` ist definiert. Dann zählt der von dieser Methode zurückgegebene `Iterator` analog zu Fall 1 für jede noch nicht zugeordnete Prädikatkante aus `hg`, die in ihrer Stelligkeit der Musterkante dieses Selektors entspricht, eine Prädikatvariable mit ebendieser Prädikatkante als Substitution auf. Die `tentacleMappings` bilden dabei wieder jeden Arm der Substitutionskante auf den gleichen Arm der Variablen ab.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Testet zusätzlich zu der überschriebenen Basisklassenmethode, ob es sich bei der Komponente an Stelle `depth` in `matchingComponents` um eine Instanz einer die Schnittstelle `PredicateVariableHyperedge` implementierenden Klasse handelt und diese mit der Musterkante in Prädikatname und Stelligkeit übereinstimmt.

5.2.1.13 ContainedPredicateVariableEdgeSelector

Klasse	ContainedPredicateVariableEdgeSelector
Basisklasse	PlainPredicateVariableEdgeSelector
Schnittstellen	ContainedComponentSelector
Aufgabe	Repräsentiert eine in einem Behälter enthaltene Prädikatvariable der linken Seite und sucht während der Ansatzsuche lokal dazu passende Kandidaten.

Eigenschaften

```
private final int containedIn;

          public ContainedPredicateVariableEdgeSelector
                     (PredicateVariableHyperedge patternEdge,
                      SameComponent[] defNodes, int cont);

  public Iterator candidates       (Hypergraph hg, int depth,
                                    HypergraphComp[] matchingComponents);
   public boolean matches          (Hypergraph hg, int depth,
                                    HypergraphComp[] matchingComponents);
       public int getContainerIndex();
```

Attribute

```
private final int containedIn;
```

Der Index des Behälters im Abbildungsarray der linken Seite, in dem die durch diesen Selektor beschriebene Prädikatvariable enthalten ist.

Konstruktoren

```
public ContainedPredicateVariableEdgeSelector
                 (PredicateVariableHyperedge patternEdge,
                  SameComponent[] defNodes, int cont);
```

Erzeugt einen `ContainedPredicateVariableEdgeSelector` mit den angegebenen Werten (s.a. Klassenbeschreibung 5.2.1.6).

Methoden

```
public Iterator candidates(final Hypergraph hg, final int depth,
                           HypergraphComp[] matchingComponents);
```

Passt der Behälter grundsätzlich zum Attribut `containedIn` (sonst wird wie schon bei den anderen `ContainedComponent`-Selektoren eine Ausnahme geweckt), versucht diese Methode, das Resultat der überschriebenen Methode der Basisklasse mit dem im identifizierten Behälter enthaltenen Graphen als `hg` zurückzugeben. Weckt die Basisklassenmethode die Ausnahme der „einsamen Variable“, fängt diese Methode sie ab und erzeugt für jede bisher nicht zugeordnete Prädikatkante aus dem im Behälter enthaltenen Graphen mit passender Stelligkeit eine Prädikatvariable mit ebendieser Prädikatkante als Substitution. Die `TentacleMappings` bilden dabei wieder jeden Arm der Substitutionskante auf den gleich nummerierten der Prädikatvariable ab.

```
public boolean matches(Hypergraph hg, int depth,
                       HypergraphComp[] matchingComponents);
```

Prüft zusätzlich zu den Tests der überschriebenen Basisklassenmethode, ob die Komponente an der Stelle `depth` in `matchingComponents` im richtigen Behälter enthalten ist.

`public int getContainerIndex();`
Gibt den Wert des Attributs `containedIn` zurück.

5.2.2 Die neuen `TransformationSets`

Wie in Unterabschnitt 4.2.1 erläutert, bilden die Klassen `TransformationSet` und `ActionTransformationSet` die Basis für die Transformationsmaschine eines generierten Editors. Die in diesem Unterabschnitt vorgestellten Klassen sind Spezialisierungen dieser Klasse, die für den jeweils zentralen Aspekt (Hierarchie, Variablen, Prädikate) teilweise neue Methoden zur Verfügung stellen, hauptsächlich aber die bisherigen anpassen.

5.2.2.1 Änderungen an den existierenden Klassen

Bisher war die Erzeugung des zur gerade bearbeiteten Komponente passenden Selektors Bestandteil der in `TransformationSet` deklarierten abstrakten Methode `makeProdPattern()`. Um die Selektorerzeugung von der Patternerzeugung zu entkoppeln, wurde diese Funktionalität jedoch in eine eigene (abstrakte) Methode

```
protected abstract ComponentSelector makeSelector(Object[] o, int index);
```

ausgelagert. Die weiteren Änderungen an `TransformationSet` betrafen die Erweiterung der Regelbäume von Kanten auf allgemeine Hypergraphkomponenten sowie die neue Methode

```
protected int[] findVisitComponents(HypergraphComp comp, Object[] pattern,
                                    int limit);
```

Ähnlich wie die Methode `findSameComponents()` (bisher `findSameNodes()`) für einen Selektor die Verbindungen zu anderen gleichartigen Komponenten erstellt, erstellt diese die Referenzen auf die im Baum enthaltenen Kanten, die den gerade betrachteten Knoten besuchen bzw. auf die im Baum enthaltenen Knoten, die von der gerade betrachteten Kante besucht werden.

Weitere Modifikationen an dieser und der abgeleiteten Klasse `ActionTransformationSet`, die hier nicht detailliert beschrieben werden, betreffen die Erweiterung der bisherigen Mechanismen zur Regelspeicherung von Kanten auf beide Arten von Hypergraphkomponenten.

Zu jeder neuen `TransformationSet`-Klasse existiert eine passende `ProdPattern`-Klasse; beide zusammen bilden jeweils die Umsetzung eines der Konzepte *Behälter*, *Variablen* und *Prädikate* für die Transformation in dieser Reihenfolge.

5.2.2.2 HierarchicalActionTransformationSet

Klasse	HierarchicalActionTransformationSet
Basisklasse	ActionTransformationSet
Aufgabe	Verwaltet für die hierarchische Transformation geeignete `ProdPattern`-Bäume.

Eigenschaften

```
 protected static final int NO_CONTAINER_FOUND;

        protected final int findContainer  (Containable hostGraph, Object[] o,
                                             int index);
protected ComponentSelector makeSelector   (Object[] o, int index);
      protected ProdPattern makeProdPattern(Object[] o, int index,
                               Node[] add_glu_n, ProdPattern parent);
                public void addRule(Object[] lhs, HypergraphComp[] rhs,
                                int[] nonContextIdx, Prohibition[] prohibition,
                                PatternAction action);
```

Attribute

`protected static final int NO_CONTAINER_FOUND;`

Rückgabewert der Methode `findContainer()`, wenn kein passender Behälter gefunden wurde.

Methoden

```
protected final int findContainer(Containable hostGraph, Object[] o,
                                  int index);
```

Sucht in `o` nach dem Behälter des Graphen `hostGraph` und gibt den entsprechenden Index zurück. Ist der Behälter nicht vorhanden, ist der Rückgabewert das Attribut `NO_CONTAINER_FOUND`.

Parameter:

`hostGraph` Der Graph, dessen Behälter zu suchen ist.

`o` Die linke Seite der gerade betrachteten Regel.

`index` Der Index der aktuell betrachteten Komponente.

`protected ComponentSelector makeSelector(Object[] o, int index);`

Erstellt den passenden Selektor für die Komponente an der Stelle `index` im Array `o`. Findet die Methode `findContainer()` keinen passenden Behälter für diese Komponente, wird die Aufgabe der Selektorerzeugung an die Basisklasse weitergereicht.

Parameter:

`o` Die linke Seite der gerade betrachteten Regel.

`index` Der Index der aktuell betrachteten Komponente.

```
protected ProdPattern makeProdPattern(Object[] o, int index,
                              Node[] add_glu_n, ProdPattern parent);
```

Erzeugt ein `HierarchicalActionProdPattern` (Klassenbeschreibung 5.2.3.2) mit passendem Selektor und sonstigen Parametern.

Parameter:

`o` Die linke Seite der gerade betrachteten Regel.

`index` Der Index der aktuell betrachteten Komponente.

`add_glu_n` Zusätzliche Verklebungsknoten. Ein für diese Arbeit irrelevanter DiaGen-Pflichtparameter.

`parent` Der Elternknoten im `ProdPattern`-Baum für die linke Regelseite.

```
public void addRule(Object[] lhs, HypergraphComp[] rhs, int[] nonContextIdx,
                    Prohibition[] prohibition, PatternAction action);
```

Fügt eine Regel zu diesem `TransformationSet` hinzu.

Vor der Erstellung des `ProdPattern`-Baums „entfaltet“ diese Methode den Array `lhs` (s.a. Klassenbeschreibung 5.2.4), damit auch alle in Behältern enthaltenen Komponenten bei der Ansatzsuche berücksichtigt werden.

Parameter:

`lhs` Die linke Regelseite.

`rhs` Die rechte Regelseite.

`nonContextIdx` Die Indizes der „wichtigen“ Komponenten (wird von der DiaGen-Transformationsmaschine im Zusammenhang mit `ModelEdges` eingesetzt (Unterabschnitt 4.1.2)).

`prohibition` Der negative Kontext der hinzuzufügenden Regel.

`action` Die bei einem Transformationsschritt aufgrund der hinzuzufügenden Regel anzuwendende `PatternAction`, die jedem erzeugten `Redex` zugewiesen wird.

5.2.2.3 VariableHierarchicalActionTransformationSet

Klasse	VariableHierarchicalActionTransformationSet
Basisklasse	HierarchicalActionTransformationSet
Aufgabe	Verwaltet für die hierarchische Transformation mit Variablen geeignete `ProdPattern`-Bäume.

Eigenschaften

```
  protected SameComponent[] findSameComponents(HypergraphComp comp,
                                               Object[] pattern, int limit);
   protected static boolean testSameNodes  (SameComponent[] sameNodes,
                                               Edge edge);
      protected ProdPattern addPattern     (Object[] edges,
                                               Node[] add_glu_n,
                                               boolean forceNewPath);
protected ComponentSelector makeSelector   (Object[] o, int index);
      protected ProdPattern makeProdPattern(Object[] o, int index,
                                               Node[] add_glu_n,
                                               ProdPattern parent);
```

Methoden

```
protected SameComponent[] findSameComponents(HypergraphComp comp,
                                             Object[] pattern, int limit);
```

Ist `comp` ein Knoten, filtert diese Methode aus dem von der überschriebenen Methode aus `TransformationSet` gelieferten Array die über Variablen hergestellten Beziehungen zu anderen Knoten heraus; da Variablen kein Bestandteil eines Datengraphen sind, könnten solche Verbindungen durch die Ansatzsuche nie gefunden werden. Anderenfalls liefert diese Methode dasselbe Resultat wie die in `TransformationSet`.

Parameter:

`comp` Die gerade betrachtete Komponente, für die Beziehungen zu anderen Komponenten der linken Seite zu suchen sind.

`pattern` Die linke Regelseite.

`limit` Der maximale Index, bis zu dem die Suche in `pattern` erfolgen soll.

```
protected static boolean testSameNodes(SameComponent[] sameNodes,
                                       Edge edge);
```

Unterzieht die für `edge` vorgesehenen `sameNodes` einer zusätzlichen Prüfung. Ist `edge` eine Variable, so darf keine `SameComponent` eine Verbindung zu einer weiteren Variable darstellen (s.a. Unterabschnitt 2.2.1).

Parameter:
`sameNodes` Die zu testenden `SameComponents`.
`edge` Die Kante, für die der Array `sameNodes` vorgesehen ist.

```
protected ProdPattern addPattern(Object[] edges, Node[] add_glu_n,
                                 boolean forceNewPath);
```

Sortiert alle Variablen der linken Seite hinter die konstanten Komponenten und fügt ein entsprechenden `ProdPattern` zu diesem `TransformationSet` hinzu. So werden durch die Ansatzsuche immer erst die Bilder der konstanten Bestandteile der linken Regelseite gefunden, die anderenfalls bereits von einer Variablensubstitution „verschluckt" werden könnten.
Parameter:
`edges` Die linke Regelseite.
`add_glu_n` Zusätzliche Verklebungsknoten. Ein für diese Arbeit irrelevanter DiaGen-Pflichtparameter.
`forceNewPath` Gibt an, ob in jedem Fall ein komplett neuer Weg im Patternbaum erzeugt werden soll oder nur soweit noch keine zur linken Seite passenden vorhanden sind.

```
protected ComponentSelector makeSelector(Object[] o, int index);
```

Erzeugt einen passenden `ComponentSelector` für das Objekt an der Stelle `index` in `o`. Ist das Objekt eine Variable, so ist der Selektor je nachdem, ob das Objekt zum obersten Graph der linken Seite gehört oder nicht , ein `PlainVariableEdgeSelector` bzw. ein `ContainedVariableEdgeSelector`. Ist das Objekt keine Variable, so liefert diese Methode den von der Basisklassenmethode zurückgegebenen Selektor.

```
protected ProdPattern makeProdPattern(Object[] o, int index,
                          Node[] add_glu_n, ProdPattern parent);
```

Erzeugt ein `VariableHierarchicalActionProdPattern` (Klassenbeschreibung 5.2.3.3) mit den passenden Werten.
Parameter:
siehe Klassenbeschreibung 5.2.2.2

5.2.2.4 PredicateVariableHierarchicalActionTransformationSet

Klasse	PredicateVariableHierarchicalActionTransformationSet
Basisklasse	VariableHierarchicalActiontransformationSet
Innere Klassen	PredicateCompletionRedex (Klassenbeschreibung 5.2.2.5)
Aufgabe	Verwaltet für die hierarchische Transformation mit Prädikaten und Variablen geeignete `ProdPattern`-Bäume.

Eigenschaften

```
public static final String CONDITION_EVALUATION_FAILED;
public static final String RHS_EVALUATION_FAILED;
private static final String SUBSIDIARY_RHS_EVALUATION_FAILED;
private static SortedMap completionClauseMap;
private static SortedMap exceptionHandlerMap;
```

```
protected HypergraphComp[] getCompletionClause(String predicateName);
```

```
protected ComponentSelector makeSelector      (Object[] o, int index);
protected ProdPattern makeProdPattern(Object[] o, int index,
                        Node[] add_glu_n, ProdPattern parent);
    public void addRule(Object[] lhs, HypergraphComp[] rhs,
                   int[] nonContextIdx, Prohibition[] prohibition,
                   PatternAction action);
    public void addExceptionHandler(String predicateName,
                   Object[] lhs, HypergraphComp[] rhs,
                   Prohibition[] prohibition, PatternAction action);
    public void addExceptionHandler(String predicateName, Object[] lhs,
                   HypergraphComp[] rhs, int[] nonContextIdx,
                   Prohibition[] prohibition, PatternAction action);
    public void addConditionalRule(Object[] lhs, HypergraphComp[] cond,
                   HypergraphComp[] rhs, Prohibition[] prohibition,
                   PatternAction action);
    public void addConditionalRule(Object[] lhs, HypergraphComp[] cond,
                   HypergraphComp[] rhs, int[] nonContextIdx,
                   Prohibition[] prohibition, PatternAction action);
    public void addConditionalExceptionHandler(String predicateName,
                   Object[] lhs, HypergraphComp[] cond,
                   HypergraphComp[] rhs, Prohibition[] prohibition,
                   PatternAction action);
    public void addConditionalExceptionHandler(String predicateName,
                   Object[] lhs, HypergraphComp[] cond,
                   HypergraphComp[] rhs, int[] nonContextIdx,
                   Prohibition[] prohibition, PatternAction action);
 public boolean addCompletionClause(String predicateName,
                                 HypergraphComp[] clause);
    public void setCompletionClause(String predicateName,
                                 HypergraphComp[] clause);
 public boolean applyPredicate(PredicateHyperedge edge,
                             HypergraphComp[] rhs);
private boolean handleRedex(Redex redex, PredicateHyperedge parentEdge,
                          PredicateHyperedge redexSource);
  private Redex makePredicateCompletionRedex(HypergraphComp[] match);
   public Redex getRedex(Hypergraph graph,
                      HypergraphComp[] partialMatch);
public Enumeration getAllRedices(Hypergraph graph,
                              HypergraphComp[] partialMatch);
```

Attribute

`public static final String CONDITION_EVALUATION_FAILED;`
Zeigt als Nachricht einer Ausnahme an, dass die Auswertung des Bedingungsteils einer Prädikatregel fehlgeschlagen ist.

`public static final String RHS_EVALUATION_FAILED;`
Zeigt als Nachricht einer Ausnahme an, dass die Auswertung der rechten Seite einer Prädikatregel fehlgeschlagen ist.

`private static final String SUBSIDIARY_RHS_EVALUATION_FAILED;`
Zeigt als Nachricht einer Ausnahme an, dass die Auswertung der rechten Seite einer Regel fehlgeschlagen ist, deren Prädikat dem aktuell betrachteten nach den Regeln der strikten Tiefensuche untergeordnet ist.

`private static final SortedMap completionClauseMap;`
Bildet Prädikatnamen auf deren Abschlussklausel ab.

`private static final SortedMap exceptionHandlerMap;`
Bildet Prädikatnamen auf deren Ausnahmebehandler ab.

Methoden

`protected HypergraphComp[] getCompletionClause(String predicateName);`
Gibt die dem angegebenen Prädikatnamen zugeordnete Abschlussklausel zurück.

`protected ComponentSelector makeSelector(Object[] o, int index);`
Erzeugt einen passenden `ComponentSelector` für das Objekt an der Stelle `index` in `o`. Ist das Objekt eine Prädikatkante, ist der Selektor je nachdem, ob das Objekt zum obersten Graph der linken Seite gehört oder nicht, ein `PlainPredicateEdgeSelector` bzw. ein `ContainedPredicateEdgeSelector`.

Ist das Objekt eine Prädikatvariable, ist der Rückgabewert ebenfalls der entsprechende Selektor. Ist das Objekt weder Prädikatkante noch Prädikatvariable, liefert diese Methode den von der Basisklassenmethode zurückgegebenen Selektor.

```
protected ProdPattern makeProdPattern(Object[] o, int index,
                          Node[] add_glu_n, ProdPattern parent);
```
Erzeugt ein `PredicateVariableHierarchicalActionProdPattern` (Klassenbeschreibung 5.2.3.5) mit den passenden Werten.
Parameter:
siehe Klassenbeschreibung 5.2.2.2

```
public void addRule(Object[] lhs, HypergraphComp[] rhs, int[] nonContextIdx,
                    Prohibition[] prohibition, PatternAction action);
```
Setzt die Prädikatkante des obersten Graphen der linken Seite (nach Voraussetzung gibt es genau eine) an die Spitze des Arrays `lhs` und ruft die überschriebene übergeordnete Methode. Für die Durchführung des Transformationsschritts ist es erforderlich, dass sich die entsprechende Prädikatkante an der Sitze des Arrays befindet.
Parameter:
siehe Klassenbeschreibung 5.2.2.2

```
public void addExceptionHandler(String predicateName, Object[] lhs,
                         HypergraphComp[] rhs, int[] nonContextIdx,
                         Prohibition[] prohibition, PatternAction action);
```
Fügt dem angegebenen Prädikat eine Transformationsregel als Ausnahmebehandler hinzu.
Parameter:
`predicateName` Der Name des Prädikats, dem dieser Ausnahmebehandler hinzugefügt wird.
Rest wie ***`addRule()`***

```
public void addConditionalRule(Object[] lhs, HypergraphComp[] cond,
                    HypergraphComp[] rhs, int[] nonContextIdx,
                    Prohibition[] prohibition, PatternAction action);
```

Fügt eine bedingte Regel zu diesem `TransformationSet` hinzu.
Parameter:
`cond` Der Bedingungsteil der hinzuzufügenden Regel.
Rest wie ***`addRule()`***

```
public void addConditionalExceptionHandler(String predicateName,
            Object[] lhs, HypergraphComp[] cond, HypergraphComp[] rhs,
            int[] nonContextIdx, Prohibition[] prohibition,
            PatternAction action);
```

Fügt dem angegebenen Prädikat eine bedingte Transformationsregel als Ausnahmebehandler hinzu.
Parameter:
`predicateName` Der Name des Prädikats, dem dieser Ausnahmebehandler hinzugefügt wird.
Rest wie ***`addConditionalRule()`***

```
public boolean addCompletionClause(String predicateName,
                                   HypergraphComp[] clause);
```

Setzt die Abschlussklausel des angegebenen Prädikats auf die angegebenen Komponenten, sofern dieses Prädikat nicht bereits eine Abschlussklausel besitzt. Der Rückgabewert `true` zeigt an, dass die Abschlussklausel entsprechend gesetzt wurde.
Parameter:
`predicateName` Der Name des Prädikats, dessen Abschlussklausel gesetzt werden soll.
`clause` Die Komponenten, welche die zu setzende Abschlussklausel bilden.

```
public void setCompletionClause(String predicateName,
                                HypergraphComp[] clause);
```

Setzt die Abschlussklausel des angegebenen Prädikats unabhängig davon, ob dieses Prädikat bereits eine Abschlussklausel besitzt, auf die angegebenen Komponenten.
Parameter:
wie bei ***`addCompletionClause()`***

```
public boolean applyPredicate(PredicateHyperedge edge,
                              HypergraphComp[] rhs);
```

Wendet das zur angegebenen Kante gehörende Prädikat mit der angegebenen rechten Seite (inkl. Bedingungsteil) an. Diese Methode wird innerhalb der `action()`-Methode einer `PatternAction` gerufen, nachdem diese den vorgeschriebenen Transformationsschritt unter Vorbehalt durchgeführt hat (s.a. S. 16, *Anwendbarkeitsbedingungen*), und arbeitet nach folgendem Algorithmus:

1. Sei $\Delta.n_0$ der aktuelle Zustand des Transformationsgraphen. Sortiere alle Prädikatkanten in `rhs` zunächst nach dem Wert des `priority`-Flags (`true` kommt hier vor `false`), dann nach Schachtelungstiefe.

2. Iteriere über die Sortierung aus Schritt 1. Sei `currentPredicate` die aktuelle Prädikatkante; sei `owner` der Graph, dem diese Kante (direkt) angehört. Iteriere über alle Redizes aus `owner`, die einer Regel des Prädikats `currentPredicate` entsprechen.
3. Sei `redex` der aktuelle Redex. Rufe `handleRedex()` mit den Parametern `redex`, `edge` und `currentPredicate` in dieser Reihenfolge. Je nach Rückgabewert handle folgendermaßen:

   `true` Ignoriere evtl. vorhandene weitere Redizes, setze die Priorität von `currentPredicate` (unabhängig vom aktuellen Wert) auf `false`, und fahre mit der nächsten Prädikatkante bei Schritt 2 fort.

   `false` Versetze `owner` in den Zustand vor dem Ruf von `handleRedex()`. Versuche den nächsten Redex. Gibt es keinen mehr oder konnte keiner gefunden werden, setze die Priorität von `currentPredicate` auf `false`, versetze den Transformationsgraphen zurück in $\Delta.n_0$, und gib `false` zurück.

   `TransformationException` Die Methode `handleRedex()` kann mittelbar wie unmittelbar das Wecken von Ausnahmen mit unterschiedlichen Nachrichten auslösen:
   - Die Kontaktbedingung wurde verletzt. Dann hat sich `redex` im Nachhinein als nicht anwendbar herausgestellt; folglich verfahre, als hätte die Methode `false` geliefert.
   - Die Auswertung ist fehlgeschlagen, aber `currentPredicate` hat keine Priorität. Versetze `owner` in den Zustand vor dem Aufruf von `handleRedex()` und reiche die Ausnahme weiter.
   - Obige Ausnahme trat bereits innerhalb von `handleRedex()` auf. Dann ist die Auswertung der rechten Seite eines nachgeordneten Prädikats fehlgeschlagen. Versetze `owner` in den Zustand vor Aufruf von `handleRedex()`. Rufe `handleRedex()` mit der Abschlussklausel von `currentPredicate`. Liefert dieser Ruf `false`, versetze `owner` wieder in den Zustand vor dem Aufruf und gib `false` zurück; weckt der Aufruf die Ausnahme aus dem vorigen Punkt, versetze `owner` zurück und reiche die Ausnahme weiter; liefert der Aufruf `true`, fahre in Schritt 2 mit der nächsten Prädikatkante fort.

   Andere Ausnahmen sind nicht vorgesehen und werden daher im Falle eines Auftretens direkt weitergereicht, ohne dass Änderungen am Graphen erfolgen.
4. Gib `true` zurück.

```
private boolean handleRedex(Redex redex, PredicateHyperedge parentEdge,
        PredicateHyperedge redexSource) throws PredicateException;
```

Neben `applyPredicate()` die zweite zentrale Methode dieser Klasse zur Umsetzung der *strikten Tiefensuche* (S. 14f.). Ruft `redex.reduce()` und verhält sich je nach geliefertem Objekt bzw. geweckter Ausnahme folgendermaßen:

`null` Der Rückgabewert von `redex.reduce()`, wenn während der Auswertung keine Abschlussklauseln auftreten. Der Rückgabewert dieser Methode ist in diesem Fall `true`.

`Boolean.TRUE` Der Rückgabewert von `redex.reduce()`, wenn während der Auswertung die Abschlussklausel „Erfolg" aufgetreten ist. Auch in diesem Fall gibt diese Methode `true` zurück.

`Boolean.FALSE` Der Rückgabewert von `redex.reduce()`, wenn während der Auswertung die Abschlussklausel „Fehlschlag" aufgetreten ist. Gehört `redex` zum Bedingungsteil der angewendeten Regel, was genau dann der Fall ist, wenn bei `redexSource` das `priority`-Flag gesetzt ist, liefert diese Methode den Rückgabewert `false`; anderenfalls weckt sie eine Ausnahme, die anzeigt, dass die Auswertung der rechten Seite fehlgeschlagen ist.

`PredicateException` Ist der Rückgabewert eine Dia**plan** -Prädikatausnahme, sucht diese Methode in den dem durch `parentEdge` repräsentierten Prädikat zugeordneten Ausnahmebehandlern nach einem Ansatz. Ist die Suche erfolgreich, liefert die Methode den Rückgabewert eines rekursiven Aufrufs von sich selbst mit ebendiesem `Redex`, der gleichen `parentEdge` und der der Ausnahme zugeordneten Kante als `redexSource`.

Geweckte Ausnahmen Das Rufen von `redex.reduce()` kann zwei Ausnahmen wecken, die innerhalb dieser Methode behandelt werden:

- die im Fall einer Fehlschlagsklausel geweckte Ausnahme, die anzeigt, dass das entsprechende Prädikat nicht zum Bedingungsteil der Regel gehört. Diese stammt aus der Anwendung eines im Auswertungsbaum Δ nachgeordneten Prädikats (also eines rekursiven Aufrufs dieser Methode innerhalb von `redex.reduce()`). Dann weckt diese Methode eine neue Ausnahme, die den Fehlschlag einer nachgeordneten rechten Seite anzeigt[6].
- eine Ausnahme, die anzeigt, dass die Auswertung des Bedingungsteils fehlgeschlagen ist[7]. Diese Ausnahme wird durch die Methode abgefangen, und der Rückgabewert lautet `false`.

```
private Redex makePredicateCompletionRedex(HypergraphComp[] match);
```

Erzeugt einen zu `match` passenden `PredicateCompletionRedex` anhand der in `completionClauseMap` gespeicherten Abschlussklauseln.

```
public Redex getRedex(Hypergraph graph, HypergraphComp[] partialMatch);
```

Enthält `partialMatch` eine einzelne Prädikatkante als einziges Element und findet die überschriebene Methode in `TransformationSet` keinen `Redex`, gibt diese Methode das Resultat von `makePredicateCompletionRedex()` zurück. Anderenfalls ist der Rückgabewert mit dem der Basismethode identisch.

```
public Enumeration getAllRedices(final Hypergraph graph,
                                 final HypergraphComp[] partialMatch);
```

Enthält `partialMatch` eine einzelne Prädikatkante als einziges Element, hängt diese Methode an die von der Basismethode gelieferte `Enumeration` das Resultat von `makePredicateCompletionRedex()` an.

[6] Diese neue Ausnahme wird wiederum in `applyPredicate()` abgefangen und sorgt dort für die Anwendung der Abschlussklausel.

[7] Diese Ausnahme wird in der `action()`-Methode der dem `Redex` zugeordneten `PatternAction` geweckt, wenn der Aufruf von `applyPredicate()` den Wert `false` liefert (s.a. Klassenbeschreibung 5.3.2).

In dieser Klasse finden sich die Schritte 1 und 2 der strikten Tiefensuche (S. 14f.) wieder:

1. *Sortierung der Zustände nach Alter.* Jeder zu einer Prädikatregel erstellte `Redex` ruft zur Durchführung des Transformationsschritts in seiner `reduce()`-Methode die `action()`-Methode der ihm zugewiesenen `PatternAction`; diese ruft `applyPredicate()`, diese wiederum, sofern der Transformationsschritt neue Prädikatkanten erzeugt, `handleRedex()` und diese schließlich wieder `Redex.reduce()`. So entsteht eine Rekursion, in der stets der zuletzt eingeführte Zustand betrachtet wird. Der Auswertungsbaum Δ wird also im Allgemeinen nicht vollständig konstruiert, sondern nur soweit erforderlich (s.a. S. 14).

2. *Sortierung der neuesten Prädikatkanten nach Schachtelungstiefe.* Diese findet im ersten Arbeitsschritt der Methode `applyPredicate()` statt.

Die Umsetzung von Schritt 3 ergibt sich aus der geänderten Erstellung des `ProdPattern`-Baums (Unterabschnitt 5.2.3): Jedes mit `makeProdPattern()` erstellte Pattern wird nicht wie bisher als erstes, sondern als letztes Kind seines Elternknotens eingefügt.

5.2.2.5 PredicateCompletionRedex

Klasse	PredicateCompletionRedex
Basisklasse	java.lang.Object
Schnittstellen	Redex
Aufgabe	Innere Klasse von `PredicateVariableHierarchical-ActionTransformationSet`, welche die Anwendung einer Abschlussklausel umsetzt.

Eigenschaften

```
HypergraphComp[] matchingComponents;
HypergraphComp[] addedComponents;

            PredicateCompletionRedex (HypergraphComp[] matchingComponents,
                                      HypergraphComp[] addedComponents);

public HypergraphComp[] getMatch          ();
         public Object reduce             ();
```

Attribute

`HypergraphComp[] matchingComponents;`
Der Ansatz im Transformationsgraphen, der bei diesem `Redex` nur aus der das Prädikat repräsentierenden Kante besteht.

`HypergraphComp[] addedComponents;`
Die einzusetzende Abschlussklausel.

Konstruktoren

```
PredicateCompletionRedex(HypergraphComp[] matchingComponents,
                         HypergraphComp[] addedComponents);
```

Erzeugt einen `PredicateCompletionRedex` mit den angegebenen Attributen.

Methoden

`public HypergraphComp[] getMatch();`
Gibt die Prädikatkante im Transformationsgraphen zurück, deren Abschlussklausel durch diesen `Redex` umgesetzt wird.

`public Object reduce();`
Ersetzt die Prädikatkante im Transformationsgraphen durch ihre Abschlussklausel. Der Rückgabewert wird durch die Klausel bestimmt:

- Handelt es sich um `PredicateEdge.COMPLETION_CLAUSE_SUCCESS`, also die Standardkante für „Erfolg", ist der Rückgabewert `Boolean.TRUE`.
- Handelt es sich um `PredicateEdge.COMPLETION_CLAUSE_FAILURE`, also die Standardkante für „Fehlschlag", ist der Rückgabewert `Boolean.FALSE`.
- Anderenfalls ist der Rückgabewert eine neu erzeugte `PredicateException`-Instanz, der die Ausnahmekante als Parameter übergeben wird. Da die `reduce()`-Methode von der DiaGen-Schnittstelle `Redex` ohne `throws`-Klausel definiert ist, handelt es sich hier tatsächlich um einen Rückgabewert und nicht um eine Ausnahme im Java-Sinn.

5.2.3 Die neuen `ProdPatterns`

Wie im vorhergehenden Abschnitt erwähnt, existiert zu jeder neuen `TransformationSet`- eine passende neue `ProdPattern`-Klasse, deren Instanzen jeweils durch die `makeProdPattern()`-Methode des zugehörigen `TransformationSet` erzeugt werden. Die in Abschnitt 4.2 und 4.3 geschilderten Mechanismen zur Repräsentation von Regeln sowie zur Ansatzsuche sind gleich geblieben; die im folgenden geschilderten Änderungen stellen wie bei den `TransformationSets` größtenteils Auslagerungen bestehender Funktionalitäten in neue Methoden sowie die Umstellung von Kanten auf allgemeine Graphkomponenten als Bestandteile eines Ansatzes dar.

5.2.3.1 Änderungen an den existierenden Klassen

ProdPattern Die Basisschnittstelle für Selektoren ist jetzt nicht mehr `EdgeSelector`, sondern `ComponentSelector` (s.a. Abb. 5.2); daher wurden das entsprechende Attribut sowie die Konstruktorparameter typangepasst. Ebenso besteht der Array der gefundenen Graphobjekte als Methodenparameter nicht mehr „nur" aus `Edges`, sondern aus `HypergraphComps`.

Neue Methoden:

`protected void addAsChild(ProdPattern p, boolean firstChild);`
Fügt dieses `ProdPattern` als Kind von `p` in den Patternbaum ein. Der Parameter `firstChild` gibt an, ob dieses Pattern als erstes (`true`) oder letztes (`false`) Kind einzufügen ist. Der Standardfall, der auch in der einfacheren, bisher schon existierenden Version ohne den zweiten Parameter umgesetzt ist, ist ersterer. Will man wie z.B. in Schritt 3 auf S. 15 erläutert jedoch die Reihenfolge der Regeln entsprechend der im Programm festgelegten sicherstellen, ist `firstChild` auf `false` zu setzen.

```
protected Iterator findAllCandidates(Hypergraph roots,
                                      HypergraphComp[] matchingComponents);
```

Sucht alle zur durch dieses `ProdPattern` repräsentierten Graphkomponente passenden Kandidaten mithilfe des zugewiesenen `ComponentSelector`. Diese Funktionalität beschränkte sich bisher auf einen Ruf der `candidates()`-Methode des Selektors; bei der hierarchischen Transformation erfolgt dieser Aufruf jedoch u.U. mehrfach (Klassenbeschreibung 5.2.3.2).

ActionProdPattern Die Klasse `ActionProdPattern` verfügt über eine neue innere Klasse `StandardRedex`, die bisher schon implizit als Rückgabetyp der Methode `createRedex()` existierte. Wie schon mehrfach erwähnt, ruft die `reduce()`-Methode dieser Klasse nur die `action()`-Methode der dem `Redex` zugewiesenen `PatternAction` (in diesem Fall wird dem `Redex` kein solches Objekt übergeben, sondern er greift direkt auf das Attribut `action` des umfassenden `ActionProdPattern` zu). Die Erzeugung des `Redex` wurde in eine neue Methode `createStandardRedex()` ausgelagert, die in zwei der abgeleiteten Klassen, die eigene `Redex`-Klassen definieren (Klassenbeschreibungen 5.2.3.3 und 5.2.3.5), überschrieben wird.

5.2.3.2 HierarchicalActionProdPattern

Klasse	HierarchicalActionProdPattern
Basisklasse	ActionProdPattern
Aufgabe	Repräsentiert eine Komponente einer linken Seite im Regelbaum für hierarchische Graphtransformation.

Eigenschaften

```
public HierarchicalActionProdPattern
          (ComponentSelector compSelector, ProdPattern parent);
public HierarchicalActionProdPattern(ComponentSelector compSelector,
          ProdPattern parent, boolean firstChild);
public HierarchicalActionProdPattern(ComponentSelector compSelector,
          AddedComponent[] add, ProdPattern parent,
          PatternAction action);
public HierarchicalActionProdPattern(ComponentSelector compSelector,
          AddedComponent[] add,ProdPattern parent,
          PatternAction action, boolean firstChild);
```

```
 protected boolean applicable       (HypergraphComp[] matchingComponents,
                                     Hypergraph roots);
protected Iterator findAllCandidates(Hypergraph roots,
                                     HypergraphComp[] matchingComponents);
   protected Redex createRedex      (HypergraphComp[] matches, Hypergraph em);
```

Konstruktoren

```
public HierarchicalActionProdPattern
          (ComponentSelector compSelector, AddedComponent[] add,
           ProdPattern parent, PatternAction action, boolean firstChild);
```

Erzeugt ein `HierarchicalActionProdPattern` mit den angegebenen Attributen.

Parameter:
`compSelector` Der zu diesem `ProdPattern` gehörende Selektor.
`add` Die rechte Seite der entsprechenden Transformationsregel. Die Klasse `AddedComponent` ist in `ActionProdPattern` definiert und speichert zusätzlich zu der hinzuzufügenden Komponente deren Verbindungen zu Komponenten der linken Seite[8].
`parent` Der Vorgängerknoten im Regelbaum.
`action` Die `PatternAction`, deren `action()`-Methode ein erzeugter `Redex` zur Durchführung des Transformationsschritts ruft.
`firstChild` Gibt an, ob dieses Pattern als erstes Kind von `parent` in den Baum eingefügt werden soll.

Methoden

```
protected boolean applicable(HypergraphComp[] matchingComponents,
                             Hypergraph roots);
```

Testet die Anwendbarkeit der durch dieses Pattern repräsentierten Regel mit den angegebenen Parametern. Die überschriebenen `applicable()`-Methoden prüfen, ob

- evtl. negative Kontextbedingungen nicht erfüllt sind,
- diesem Pattern eine Transformation (in Form einer `PatternAction` oder im Transformationsschritt hinzuzufügender Komponenten) zugewiesen wurde, und
- ob `matchingComponents` weitere Bedingungen der evtl. diesem Pattern zugewiesenen `PatternAction` erfüllt.

Diese Methode prüft zusätzlich, ob

- dieses Pattern im Baum ein Blatt ist, also eine vollständige linke Regelseite repräsentiert, und
- der Array `matchingComponents` hierarchisch vollständig ist (s.a. S. 13, *Transformationsschritt*, sowie Klassenbeschreibung 5.2.4).

Parameter:
`matchingComponents` Der gefundene Ansatz.
`roots` Der Transformationsgraph.

```
protected Iterator findAllCandidates(Hypergraph roots,
                           HypergraphComp[] matchingComponents);
```

Ist dieses Pattern die Baumwurzel, erfolgt die Ansatzsuche durch Aufruf der überschriebenen Methode nicht nur im Transformationsgraphen `roots`, sondern auch in allen seinen hierarchischen Subgraphen, deren hierarchischen Subgraphen usw.
Anderenfalls wird einfach die Basisklassenmethode gerufen, wobei der Parameter `roots` durch den Graphen ersetzt wird, dem die erste gefundene Komponente angehört (was der gleiche Graph sein kann), da alle Komponenten, die dem Bild des obersten Graphen der linken Seite einer Regel angehören, zwar nicht direkt in `roots`, aber im selben Graphen enthalten sein müssen.

[8]Während der DiaGen-Transformation können Knoten auf andere abgebildet werden, so dass Tentakel von Kanten der rechten Seite auf falsche Knoten zeigen können. Dieser Effekt wird durch die Speicherung der Anbindungen (ebenfalls im `SameComponent`-Format) kompensiert.

5.2.3.3 VariableHierarchicalActionProdPattern

Klasse VariableHierarchicalActionProdPattern
Basisklasse HierarchicalActionProdPattern
Innere Klassen VariableRedex
Aufgabe Repräsentiert eine Komponente einer linken Seite im Regelbaum für hierarchische Graphtransformation mit Variablen. Die hier nicht aufgeführten Konstruktoren sind funktionell identisch mit denen der Basisklasse.

Eigenschaften

```
protected Redex createStandardRedex(PatternEdgeManager roots,
                                    HypergraphComp[] matchingComponents,
                                    HypergraphComp[] addedComponents);
protected Redex createRedex        (HypergraphComp[] matchingComponents,
                                    Hypergraph em);
```

Methoden

```
protected Redex createStandardRedex(PatternEdgeManager roots,
                                    HypergraphComp[] matchingComponents,
                                    HypergraphComp[] addedComponents);
```

Erzeugt einen `VariableRedex` mit den entsprechenden Parametern.

```
protected Redex createRedex(HypergraphComp[] matchingComponents,
                            Hypergraph em);
```

Erzeugt unter Verwendung von `createStandardRedex()` einen `Redex`, wenn die Ansatzsuche erfolgreich verlaufen ist. Diese Methode ordnet allen auf der rechten Seite[9] vorhandenen Variablenkanten Substitutionen und `TentacleMappings` zu, die denen der gleichnamigen Variablenkanten im Ansatz entsprechen. Hernach erzeugt die überschriebene Methode in `ActionProdPattern` eine Kopie der Seite und ruft `createStandardRedex()` mit den Parametern `em`, `matchingComponents` und der erzeugten Kopie der rechten Seite in dieser Reihenfolge.

5.2.3.4 VariableRedex

Klasse VariableRedex
Basisklasse ActionProdPattern.StandardRedex
Aufgabe Innere Klasse von `VariableHierarchicalActionProdPattern`, die eine für die hierarchische Transformation mit Variablen geeignete `Redex`-Implementierung darstellt.

Eigenschaften

```
                   protected VariableRedex(PatternEdgeManager roots,
                                           HypergraphComp[] matchingComponents,
                                           HypergraphComp[] addedComponents);
public HypergraphComp[] getMatch     ();
```

[9] Die rechte Seite ist im geschützten Attribut `add` der Klasse `ActionProdPattern` gespeichert.

Konstruktoren

```
protected VariableRedex(PatternEdgeManager roots,
                        HypergraphComp[] matchingComponents,
                        HypergraphComp[] addedComponents);
```

Erzeugt einen `VariableRedex`. Die Parameter werden in der Basisklasse unter gleichem Namen als Attribute gespeichert.
Parameter:
`roots` Der Transformationsgraph.
`matchingComponents` Das Bild der linken Regelseite.
`addedComponents` Die rechte Regelseite.

Methoden

```
public HypergraphComp[] getMatch();
```

Gibt das Bild der linken Seite zurück. Variablenkanten im Attribut `matchingComponents` werden hierbei durch ihre jeweiligen Substitutionen ersetzt.

5.2.3.5 PredicateVariableHierarchicalActionProdPattern

Klasse	PredicateVariableHierarchicalActionProdPattern
Basisklasse	VariableHierarchicalActionProdPattern
Innere Klassen	PredicateVariableRedex
Aufgabe	Repräsentiert eine Komponente einer linken Seite im Regelbaum für hierarchische Graphtransformation mit Variablen und Prädikaten. Die hier nicht aufgeführten Konstruktoren sind funktionell identisch mit denen von `HierarchicalActionProdPattern`.

Eigenschaften

```
protected Redex createStandardRedex(PatternEdgeManager roots,
                                    HypergraphComp[] matchingComponents,
                                    HypergraphComp[] addedComponents);
```

Methoden

```
protected Redex createStandardRedex(PatternEdgeManager roots,
                                    HypergraphComp[] matchingComponents,
                                    HypergraphComp[] addedComponents);
```

Erzeugt einen `PredicateVariableRedex` mit den entsprechenden Parametern.

5.2.3.6 PredicateVariableRedex

Klasse	PredicateVariableRedex
Basisklasse	VariableHierarchicalActionProdPattern.VariableRedex
Aufgabe	Innere Klasse von `PredicateVariableHierarchicalActionProdPattern`, die eine für die hierarchische Transformation mit Variablen und Prädikaten geeignete `Redex`-Implementierung darstellt. Der hier nicht aufgeführte Konstruktor ist funktionell identisch mit dem der Basisklasse.

Eigenschaften

```
public Object reduce();
```

Methoden

`public Object reduce();`

Führt den Transformationsschritt durch Aufruf der überschriebenen Methode in `ActionProdPattern.StandardRedex` aus. Befindet sich unter den Komponenten der rechten Seite eine Abschlussklausel, ist das zurückgegebene Objekt nicht wie üblich `null`, sondern je nach Klausel:

Erfolg `Boolean.TRUE`

Fehlschlag `Boolean.FALSE`

Ausnahme Eine `PredicateException` mit der entsprechenden Kante als `edge`-Attribut (Klassenbeschreibung 5.1.4.3, Abb. 5.1). Diese `PredicateException` wird nicht geweckt, da die Schnittstelle `Redex` die Methode `reduce()` ohne `throws`-Klausel deklariert.

5.2.4 Die Klasse `HierarchicalTransformationHelper`

Die neue Klasse `HierarchicalTransformationHelper` ist eine Zusammenfassung verschiedener für die **Diaplan**-Transformation nützlicher Methoden, die von den oben beschriebenen Klassen an diversen Stellen genutzt werden.

Klasse	HierarchicalTransformationHelper
Basisklasse	java.lang.Object
Aufgabe	Stellt verschiedene statische Methoden zur Verfügung, die während des Transformationsvorgang genutzt werden.

Eigenschaften

```
            protected static void trimList       (ArrayList list);
       protected static ArrayList unfoldContainerGraph
                                     (HypergraphContainer container,
                                      Object[] components);
         protected static boolean checkHierarchicalCompleteness
                                     (HypergraphComp[] match, int index);
         protected static boolean checkHierarchicalCompleteness
                                     (HypergraphComp[] match);
         protected static boolean injectivityTest(HypergraphComp c, int depth,
                                                 HypergraphComp[] match);
            protected static List unmatchedNodes (Hypergraph hg,
                                                 HypergraphComp[] match);
            protected static List unmatchedNodes (Hypergraph hg,
                                                 HypergraphComp[] match,
                                                 int maxIndex);
            protected static List unmatchedEdges (Hypergraph hg,
                                                 HypergraphComp[] match);
            protected static List unmatchedEdges (Hypergraph hg,
                                                 HypergraphComp[] match,
                                                 int maxIndex);
            protected static List unmatchedComponents(Hypergraph hg,
                                                     HypergraphComp[] match);
            protected static List unmatchedComponents(Hypergraph hg,
                                                     HypergraphComp[] match,
                                                     int maxIndex);
protected static HypergraphComp[] union          (HypergraphComp[] first,
                                                 HypergraphComp[] second);
```

Methoden

`protected static void trimList(ArrayList list);`
Entfernt alle `null`-Einträge aus `list` und passt die Listenkapazität entsprechend an.

```
protected static ArrayList unfoldContainerGraph
          HypergraphContainer container, Object[] components);
```
„Entfaltet" den in `container` enthaltenen Graphen, so dass die zurückgegebene Liste aus den Elementen von `components` sowie den (direkten) Komponenten dieses Graphen besteht.

```
protected static boolean checkHierarchicalCompleteness
                         (HypergraphComp[] match, int index);
```
Prüft, ob alle (direkten) Komponenten des im mit `index` indizierten `HypergraphContainer` enthaltenen Graphen als Elemente in `match` vorkommen. Befindet sich an der Stelle `index` in `match` kein `HypergraphContainer`, ist der Rückgabewert `true`. Die Variante ohne den Parameter `index` prüft alle in `match` enthaltenen Behälter auf diese Eigenschaft.

```
protected static boolean injectivityTest
          (HypergraphComp c, int depth, HypergraphComp[] match);
```
Prüft, ob `c` vor dem Index `depth` als Element oder Teil einer Variablensubstitution in `match` enthalten ist. Ist `c` eine Variable, testet diese Methode jede zur Substitution gehörende Komponente. Der Rückgabewert ist `true`, wenn `c` vor `depth` nicht gefunden wird; anderenfalls lautet er `false`.

```
protected static List unmatchedNodes
          (Hypergraph hg, HypergraphComp[] match, int maxIndex);
```

Gibt eine Liste aller Knoten aus `hg` zurück, die in `match` vor dem Index `maxIndex` weder als Element noch als Teil einer Variablensubstitution enthalten sind. Die Variante ohne `maxIndex` vergleicht die Knoten aus `hg` mit dem Array auf seiner gesamten Länge. Die Methoden `unmatchedEdges()` geben entsprechend die nicht in `match` enthaltenen Kanten und die Methoden `unmatchedComponents()` die nicht enthaltenen Knoten *und* Kanten zurück.

```
protected static HypergraphComp[] union (HypergraphComp[] first,
                                         HypergraphComp[] second);
```
Der zurückgegebene Array ist die mengentheoretische Vereinigung der beiden Parameter (`first` $\cup$ `second`).

5.3 Der „eigentliche" Interpreter

Dieser Abschnitt beschreibt die für Ein- und Ausgabe von Graphen und Regeln verwendeten Klassen des im Rahmen dieser Arbeit implementierten **Diaplan**-Interpreters. Dieser arbeitet rein textbasiert und hat im momentanen Stadium eher den Status eines Testwerkzeugs für die in den vorherigen Abschnitten vorgestellten Klassen; die hier präsentierten Programmteile befinden sich folgerichtig im Java-Paket `diagen.hypergraph.test`. Mit kleinen Modifikationen lässt sich die Hauptklasse `TestTool`, die hier nicht detailliert vorgestellt wird, in ein Werkzeug zum Testen nur einzelner Aspekte der Transformationsmaschine umwandeln. Scanner und Parser für die **Diaplan**-Textvariante wurden mithilfe des frei verfügbaren Sprachwerkzeugs ANTLR[10] erstellt. Die entsprechende Grammatik ist in Anhang A in Form von Syntaxdiagrammen aufgeführt. Jede Eingabedatei stellt ein vollständiges Test„szenario“, d.h. einen zu transformierenden Graphen und alle zur Verfügung stehenden Regeln bzw. Prädikate dar. Der Interpreter verarbeitet die Daten und erzeugt eine Ausgabedatei, die das Typsystem und den transformierten Graphen in der gleichen Notation wiedergibt.

Während der Verarbeitung der Eingabedatei erzeugt der Parser für jedes Element der Grammatik eine Instanz einer passenden `AST`-Klasse[11]. Exemplarisch wird im Folgenden die Klasse `ASTEdge` (Klassenbeschreibung 5.3.1) detailliert vorgestellt; die anderen arbeiten analog. Ebenfalls erläutert wird die Klasse `PredicateVariablePatternAction` (Klassenbeschreibung 5.3.2), die zwar kein integraler Bestandteil der Transformationsmaschine ist, jedoch eine Standardimplementierung für eine `PatternAction` darstellt, deren Instanzen sich in verschiedenen Situationen auf eine bestimmte Weise verhalten, die für den korrekten Fortgang der Transformation erforderlich ist.

5.3.1 ASTEdge

Klasse ASTEdge
Basisklasse AST
Aufgabe Repräsentiert während des Parse-Vorgangs eine Kante. Der Parser „füttert“ eine von ihm erzeugte Instanz mit allen in der Eingabedatei enthaltenen Daten; nach Abschluss des Parsevorgangs erzeugt jede Instanz ihr passendes Gegenstück als Instanz der Klasse `Edge`.

Eigenschaften

```
   private Token name;
   private Token type;
    private List tentacles;
private ASTGraph containedGraph;
 private boolean isPredContainer;
static final int ANY_ATTACHMENT;

                 ASTEdge             ();
                 ASTEdge             (Token n);
                 ASTEdge             (Token n, Token t);
                 ASTEdge             (Token n, Token t, List tents,
                                      ASTGraph g, boolean pred);
            void setName             (Token n);
           Token getName             ();
            void setType             (Token t);
           Token getType             ();
            void addTentacle         (Token t);
            List getTentacles        ();
            void setContainedGraph   (ASTGraph c);
```

[10] ANother Tool for Language Recognition, `http://www.antlr.org`
[11] `AST` ist kurz für „Abstract Syntax Tree“ $\widehat{=}$ „Abstrakter Syntaxbaum“.

```
ASTGraph getContainedGraph    ();
 boolean isEmpty              ();
 boolean isPredicateContainer ();
    void setPredicateContainer(boolean pred);
    Edge generate(VariableTypeSystem ts, Map nodeMap);
    Edge generate(VariableTypeSystem ts, Map nodeMap,
                  Map graphComponentMap);
    Edge generate(VariableTypeSystem ts, Map nodeMap,
                  Map graphComponentMap, boolean parserEdges);
    Edge generate(VariableTypeSystem ts, Map nodeMap,
                  Map graphComponentMap, boolean parserEdges);
                  Map conditionPointMap);
    Edge generate(VariableTypeSystem ts, Map nodeMap,
                  Map graphComponentMap, boolean parserEdges);
                  Map conditionPointMap, boolean identicalPoints);
    void printInfo            (Writer out, int offset);
```

Attribute

`private Token name;`
Der Name dieser Kante innerhalb der Eingabedatei. Innerhalb einer Regel muss jede Graphkomponente unabhängig von der Hierarchieebene eindeutig benannt werden. Eine Instanz der ANTLR-Klasse `Token` speichert zusätzlich zum ihr zugewiesenen Text Zeile und Spalte innerhalb der Eingabedatei, so dass evtl. Fehlermeldungen beim Parsen möglichst sinnvolle Informationen liefern können.

`private Token type;`
Der Name des Typs dieser Kante.

`private List tentacles;`
Enthält die Namen der von dieser Kante besuchten Knoten als `Tokens`. Jeder Knoten erhält innerhalb des Graphen, dem er angehört, einen eindeutigen Identifier.

`private ASTGraph containedGraph;`
Sofern die durch diese `ASTEdge` repräsentierte Kante ein Behälter sein soll, ist dies die den enthaltenen Graphen repräsentierende `ASTGraph`-Instanz.

`private boolean isPredContainer;`
Zeigt an, ob die repräsentierte Behälterkante ein Prädikatbehälter sein soll. Bei Nichtbehälterkanten hat dieses Attribut stets den Wert `false`.

`static final int ANY_ATTACHMENT;`
Die systeminterne Nummer für Knoten, die in Regeln nur von links nach rechts „hinüber gerettet“ werden sollen, ohne dass an ihnen bzw. an dem möglicherweise enthaltenen Graphen Modifikationen vorgenommen werden. Mit dieser Nummer werden die von der Punktmarkierungskante eines Graphen besuchten Knoten initialisiert, die für die Regel keine Bedeutung besitzen (s.a. Unterabschnitt 5.1.2).

Konstruktoren

`ASTEdge(Token n, Token t, List tents, ASTGraph g, boolean pred);`

Erzeugt eine `ASTEdge` mit den angegebenen Parametern für die Attribute.
Parameter:
`n` Die ID dieser Kante innerhalb der Eingabedatei.
`t` Der Name des Typs dieser Kante im verwendeten Typsystem.
`tents` Die Namen der von dieser Kante besuchten Knoten.
`g` Der in dieser Kante enthaltene Graph, falls die Kante ein Behälter ist.
`pred` Gibt an, ob diese Kante ein Prädikatbehälter sein soll.

Methoden

`void setName(Token t);`
Setzt das Attribut `name` auf den angegebenen Wert.

`Token getName();`
Gibt das Attribut `name` zurück.

`void setType(Token t);`
Setzt das Attribut `type` auf den angegebenen Wert.

`Token getType();`
Gibt den Typnamen dieser Kante zurück.

`void addTentacle(Token t);`
Fügt den angegebenen Knotennamen zur Liste der von dieser Kante besuchten Knoten hinzu.

`List getTentacles();`
Gibt eine unmodifizierbare Kopie der Liste der Namen der von dieser Kante besuchten Knoten zurück.

`void setContainedGraph(ASTGraph c);`
Setzt den in dieser Kante enthaltenen Graphen auf den angegebenen Wert.

`ASTGraph getContainedGraph();`
Gibt den in dieser Kante enthaltenen Graphen zurück (bzw. `null`, wenn diese Kante kein Behälter ist).

`boolean isEmpty();`
Zeigt an, ob diese Kante einen Graphen enthält.

`boolean isPredicateContainer();`
Zeigt an, ob diese Kante ein Prädikatbehälter ist.

`void setPredicateContainer(boolean pred);`
Setzt das Flag `isPredContainer` auf den angegebenen Wert.

```
Edge generate (VariableTypeSystem ts, Map nodeMap, Map graphComponentMap,
               boolean parserEdges, Map conditionPointMap,
               boolean identicalPoints);
```

Erzeugt eine zu dieser `ASTEdge` passende `Edge`-Instanz.

Parameter:

`ts` Das benutzte Typsystem.

`nodeMap` Bildet die Knotennamen im diese `ASTEdge` umfassenden `ASTGraph` auf die passenden `Node`-Instanzen ab.

`graphComponentMap` Bildet innerhalb einer Regel Namen eindeutig auf Graphkomponenten ab. Kommt z.B. eine Graphkomponente sowohl im Bedingungsteil als auch in der rechten Seite derselben Regel vor, verhindert der Eintrag in dieser `Map`, dass der Interpreter fälschlicherweise zwei verschiedene Instanzen für dasselbe Graphobjekt erzeugt.

`parserEdges` Gibt an, ob die erzeugte Kante eine `TermEdge` sein soll (s.a. Unterabschnitt 4.1.2). Üblicherweise hat dieser Parameter genau dann den Wert `true`, wenn diese `ASTEdge` zum Bedingungsteil oder der rechten Seite einer Regel gehört.

`conditionPointMap` Bildet alle Punktnamen auf die entsprechenden Knoten ab. Erfüllt einen ähnlichen Zweck wie `graphComponentMap`.

`identicalPoints` Gibt an, ob Punkte falls möglich aus `conditionPointMap` zu übernehmen sind. Für den Bedingungsteil einer Regel wird dieser Wert auf `false` gesetzt, für die rechte Seite einer Prädikatregel auf `true`.

5.3.2 PredicateVariablePatternAction

Klasse	PredicateVariablePatternAction
Basisklasse	java.lang.Object
Schnittstellen	PatternAction
Aufgabe	Standardimplementierung einer `PatternAction`, die für einen Transformationsschritt mit Behältern, Variablen und Prädikaten geeignet ist. Diese Klasse ist eine innere Klasse der hier nicht detailliert erläuterten Klasse `TestScenario`, die einen kompletten Test (zu transformierender Graph und benutzte Regeln) repräsentiert.

Eigenschaften

```
  private void performExceptionCheck   (PredicateHyperedge edge);
  private void replaceVariableWithMatch(VariableHyperedge var);
public boolean satisfiedBy             (HypergraphComp[] edges);
  private void removeEdgeRememberNode  (HypergraphComp component,
                                        Hypergraph hg, List nodes);
   public void action                  (PatternEdgeManager model,
                                        HypergraphComp[] graph,
                                        HypergraphComp[] reducedGraph);
```

Methoden

`private void performExceptionCheck(PredicateHyperedge edge);`
Weckt eine `RuntimeException` mit einer `PredicateException` als `cause`, sofern es sich bei `edge` um eine Ausnahmekante handelt.

`private void replaceVariableWithMatch(VariableHyperedge var);`
Ersetzt die Variable `var` durch ihre Substitution.

`public boolean satisfiedBy(HypergraphComp[] edges);`
Diese `PatternAction` stellt keine zusätzlichen Anforderungen an den Ansatz `edges`; daher gibt diese Methode stets `true` zurück.

```
private void removeEdgeRememberNode(HypergraphComp component,
                                    Hypergraph hg, List nodes);
```

Entfernt `component` aus `hg`, sofern es sich um eine Kante handelt; ansonsten wird `component` an `nodes` angehängt.

```
public void action(PatternEdgeManager model, HypergraphComp[] graph,
                   HypergraphComp[] reducedGraph);
```

Führt, wie schon in Unterabschnitt 4.2.4 angedeutet, den Transformationsschritt durch, wobei folgende Reihenfolge eingehalten wird:

1. *Punkte:* Suche links und rechts die Punktmarkierungskanten (s.a. Unterabschnitt 5.1.2) des obersten Graphen. Setze die nicht explizit in der rechten Regelseite erwähnten Knoten der rechten Punktmarkierungskante (also die mit der Nummer `ASTEdge.ANY_ATTACHMENT` (Klassenbeschreibung 5.3.1)) auf ihre von der linken besuchten Gegenstücke um.[12]
2. *Kanten entfernen:* Entferne alle direkt im bearbeiteten Graphen `model` enthaltenen Kanten, und sammle die Knoten zwecks schnelleren Zugriffs in einer separaten Liste. Verfahre ebenso mit etwaigen Variablensubstitutionen auf oberster Ebene.
3. *Knoten entfernen:* Ersetze alle auf oberster Ebene vorhandenen Punkte der rechten Seite durch die der linken, wobei ggf. die darin enthaltenen Graphen anzupassen sind.[13] Entferne alle sonstigen Knoten der linken Seite, sofern diese von keiner Kante besucht werden. Wird einer noch von einer Kante besucht, so ist die Kontaktbedingung verletzt; in diesem Fall wecke eine entsprechende Ausnahme.
4. *Knoten hinzufügen:* Füge alle Knoten aus `reducedGraph` hinzu, sofern es sich nicht um mit `ASTEdge.ANY_ATTACHMENT` nummerierte Platzhalter für die Punktmarkierungskanten handelt. Ersetze gleichzeitig alle Prädikatvariablen in `reducedGraph` durch ihre jeweilige Substitution, sammle alle Kanten in einer separaten Liste, und ersetze alle in Behältern vorkommenden Variablen durch ihre Substitutionen.
5. *Kanten hinzufügen:* Füge alle Kanten aus `reducedGraph` zu `model` hinzu. Ersetze Variablen auf oberster Ebene durch die Substitutionen.
6. *Mögliche Folgeregeln auswerten:* Handelt es sich bei der angewendeten Regel um eine Prädikatregel, rufe `applyPredicate()` (Klassenbeschreibung 5.2.2.4) mit der entsprechenden Prädikatkante als `edge` und `reducedGraph` als `rhs`. Lautet der Rückgabewert `false`, mach die vorgenommenen Modifikationen an `model` und ggf. in Komponenten enthaltenen Graphen rückgängig und wecke eine Ausnahme, die das Fehlschlagen der Anwendbarkeitsbedingung anzeigt. Wird eine Ausnahme gefangen, die eine Verletzung der Kontaktbedingung oder das Fehlschlagen einer rechten Seite anzeigt, so mach ebenfalls die Modifikationen an `model` rückgängig und reiche die Ausnahme weiter. Weckt `applyPredicate()` schließlich eine Prädikatausnahme (Klassenbeschreibung 5.1.4.3), so wecke eine neue Ausnahme mit dieser als `cause`.

[12] Ist im Array der linken Seite eine Punktmarkierungskante vorhanden, im rechten jedoch nicht, so weckt diese Methode eine Ausnahme.

[13] Somit generiert im Falle von Punkten der Transformationsschritt keine Kopien, sondern behält die Objekte bei.

7. *Ausnahmen auf der rechten Seite:* Enthält `reducedGraph` eine Ausnahmekante, wecke ebenfalls eine neue Ausnahme mit einer entsprechenden Prädikatausnahme als `cause`.
8. *„Säubern“:* Entferne alle in `model` oder ggf. in seinen Komponenten enthaltenen Graphen auftretenden „Erfolgs“-Kanten.

Parameter:
`model` Der Graph, in dem der Ansatz zur aktuellen Regel gefunden wurde.
`graph` Der Ansatz zur aktuellen Regel.
`reducedGraph` Die rechte Seite der aktuellen Regel (inkl. Bedingungsteil).

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurden die theoretische Basis der sowie die erste Implementierung eines Interpreters für die regelbasierte Sprache Dia**plan** vorgestellt. Wie bereits in der Einleitung angedeutet, dominierte dabei die Umsetzung der graphentheoretischen Konzepte, während die Benutzungsschnittstelle nach heutigen Maßstäben insbesondere unter Berücksichtigung der einfachen visuellen Repräsentationsmöglichkeiten von Graphen kaum diesen Namen verdient.

Auch die Entwicklung der Sprache selbst ist auf theoretischer Ebene in einigen Punkten weit von einem Abschluss entfernt, von denen einige bereits in Abschnitt 2.3 erwähnt wurden. In der Entwicklung von Sprache und System mit dem Ziel der in Abb. 6.1 erneut aufgezeigten Systemarchitektur sollten folgende Punkte als Nächstes angegangen werden:

- Die in Unterabschnitt 2.3.1 beschriebenen Gestaltdefinitionen könnten wie schon mehrfach angedeutet die Effizienz der Ansatzsuche merklich steigern, was folgendes Beispiel verdeutlicht: Angenommen, eine Komponente der linken Seite sei ein Knoten, der vier Knoten enthält; dieser Knoten würde durch den entsprechenden Selektor auf einen abgebildet, der fünf enthält. Dann bemerkten die Selektoren für die enthaltenen Knoten keine Fehler, sondern wählten jeweils einen der noch nicht zugeordneten enthal-

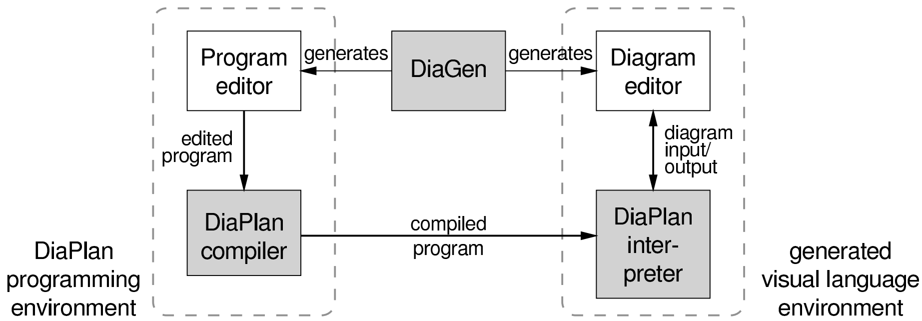


Abbildung 6.1: Angestrebte Systemarchitektur (Wdh. von Abb. 1.1)

tenen Knoten aus; der Fehler fiele schlimmstenfalls erst nach Beendigung der „eigentlichen“ Ansatzsuche (sofern alle anderen Komponenten der linken Seite ein passendes Bild im Transformationsgraphen zugewiesen bekämen) auf, nämlich beim Test auf hierarchische Vollständigkeit. Des Weiteren würde dieser Vorgang insgesamt ${}_5P_4 = 5! = 120$-mal durchgeführt, bevor für den Behälter ein weiterer Knoten ausprobiert würde – was noch nicht notwendigerweise der richtige wäre. Ein differenziertes Gestaltsystem könnte jedoch dem Knoten der linken Seite eine andere Gestaltdefinition zuordnen als dem mit den fünf enthaltenen Knoten. Die Selektoren würden bei der Auswahl der möglichen Behälter zuerst die Gestaltdefinition der Kandidaten abfragen, wodurch derjenige mit fünf enthaltenen Knoten bereits aus der Liste gestrichen wäre. Die Einteilung in Gestaltdefinitionen könnte ähnlich der bereits vorhandenen Einteilung von Kanten in ein Typsystem arbeiten; die Festlegung geeigneter Gestaltdefinitionen und Einteilungskriterien obliegt dabei natürlich dem Nutzer.

- Die erweiterte Transformationsmaschine geht von einem den auf S. 11 unter *Eindeutige Substitutionen* dargelegten Einschränkungen gehorchenden Regelsystem aus, so dass dessen Determiniertheit sichergestellt ist. Wenngleich die Beispiele in Kap. 3 zeigen, dass auch unter diesen Bedingungen bereits gut programmiert werden kann, sollten diese Beschränkungen zukünftig fallen, um dem Nutzer mehr Freiheiten bei der Regelgestaltung zu lassen. Für die Transformationsmaschine entstünde dadurch das neue Problem, *welcher* Variablen ein Graphobjekt als Substitution zuzuordnen wäre. Gemäß einem gegebenen Gestaltsystem geformte Variablen könnten den dadurch entstehenden Nichtdeterminismus beschränken.

- In Unterabschnitt 5.1.2 klingt bereits an, dass die Umsetzung der Punkte nur unzureichend erfolgt ist, da die Anzahl für die Anwendung der Regel nicht erforderlicher Punkte eines Graphen in ebendieser Regel stets zu berücksichtigen ist. Bereits angestellte Überlegungen zur Behebung dieses Mankos konnten mangels vorhandener Zeit noch nicht umgesetzt bzw. ausreichend getestet werden.

- In der derzeitigen Implementierung können im Einklang mit der Theorie keine zwei Behälter denselben Graphen enthalten. Ebenfalls hauptsächlich aus Effizienzgründen könnte jedoch zukünftig sowohl in der Theorie als auch in der Implementierung die strikte Baumstruktur der Hierarchie zu einer DAG-Struktur („directed acyclic graph“/“gerichteter azyklischer Graph“) erweitert werden, so dass, wenn ein Behälter einen Graphen an einen anderen übergibt, nicht der gesamte Graph inkl. der enthaltenen Komponenten kopiert, sondern nur die Referenz weitergereicht würde.

- Bei den Vergleichen mit anderen existierenden Graphtransformationssprachen (Unterabschnitt 2.4.2) taucht stets der Ruf nach einer Attributierungsmöglichkeit auf, die über die in Unterabschnitt 2.3.3 kurz dargestellten universellen Behälter realisiert werden wird. Zur Implementierung bieten sich die in Java 5 eingeführten generischen Datentypen (*Generics* [27]) an.

- Dia**plan**s bedingte Regeln sollen nach jüngeren Planungen zukünftig weiter differenzierbar sein. Die bisherige Form $P \parallel A/R$ wird vermutlich ersetzt durch Regeln der Form

$$P \xrightarrow{+} R_1 \xrightarrow{+} \ldots \xrightarrow{+} R_{k-1} \xrightarrow{|} R_k \xrightarrow{+} R_{k+1} \xrightarrow{+} \ldots \xrightarrow{+} R_n.$$

Hierbei besteht P wie bisher aus einem Ausdruck, der auf oberster Ebene nur eine Prädikatkante enthält, während die R_i beliebige **Diaplan**-Graphen sein können. Die $\xrightarrow{+}$-Beziehungen drücken aus, dass die erfolgreiche Auswertung von R_i Voraussetzung für die Anwendung von R_{i+1} ist. Schlägt die Auswertung eines R_i für $i < k$ fehl, so wird verfahren wie bisher bei einer fehlgeschlagenen Anwendbarkeitsbedingung: Sämtliche Änderungen am Transformationsgraphen werden rückgängig gemacht, wonach die Anwendung der nächsten Regel des gleichen Prädikats versucht wird (bzw. die Abschlussklausel des Prädikats ausgelöst wird, falls keine weitere Regel existiert). Schlägt die Auswertung eines R_i mit $i >= k$ fehl, so schlägt die gesamte Auswertung des Prädikats fehl, ohne dass weitere Regeln probiert werden. Die $\xrightarrow{|}$-Kante trennt also die verschiedenen Graphen in zwei Teile, die dem bisherigen Bedingungsteil und der „eigentlichen" rechten Seite entsprechen, und beschneidet somit ähnlich dem aus Prolog bekannten `Cut`-Operator den Suchbaum. Die bisherigen bedingten Regeln lassen sich in dieser Notation als $P \xrightarrow{+} A \xrightarrow{|} R$ schreiben. Diese Regeln können im bisherigen Konzept überraschend einfach umgesetzt werden, indem man jeder Prädikatkante zusätzlich zum `priority`-Flag (s.a. Klassenbeschreibung 5.1.4.1) noch eine Zahl zuweist; das Flag drückt dann aus, ob die entsprechende Kante sich innerhalb der Regel vor oder hinter der $\xrightarrow{|}$-Kante befindet, und die Zahl entspricht dem Index i.

- Es soll ermöglicht werden, die Parameter einer Prädikatkante nach Ein-, Aus- bzw. Ein-/Ausgang zu klassifizieren. Die Prädikatdefinitionen in [18] machen davon bereits Gebrauch.

- Die Erstellung von Regeln und Eingabegraphen in textueller Form ist selbst für die verhältnismäßig kleinen Beispiele aus Kap. 3 sehr mühevoll und fehleranfällig. Daher ist die Implementierung eines visuellen Editors auch für die jetzige Phase von **Diaplan** erstrebsam. Abb. 6.2 auf der nächsten Seite zeigt, wie ein solcher Editor aussehen könnte.

- Damit einher geht eine mögliche Umstellung des Dateiformats für Graphen und Regeln auf eine adaptierfähigere Variante; wünschenswert wäre die Nutzung von XML für diesen Zweck. Darüber hinaus müssen bei einer „marktfähigen" Anwendung Daten bzw. Eingabegraphen und Programme bzw. Regeln unabhängig voneinander gespeichert werden können.

- Wünschenswert wäre ebenfalls eine Art „Funktionsbibliothek" aus vielseitig verwendbaren Prädikatregeln. Sowohl C++ als auch Java haben ihren Erfolg zu einem großen Teil genau dem Vorhandensein einer solchen Bibliothek zu verdanken.

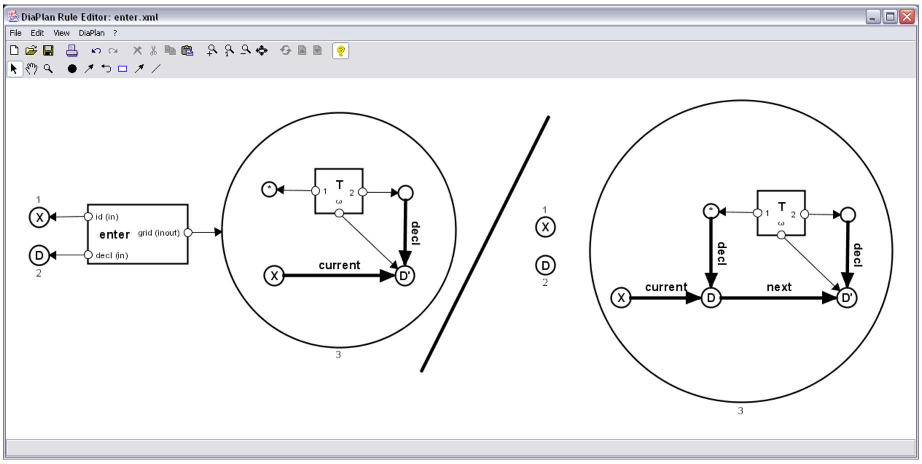


Abbildung 6.2: So könnte ein visueller **Diaplan**-Regeleditor aussehen (aus [18])

Fazit

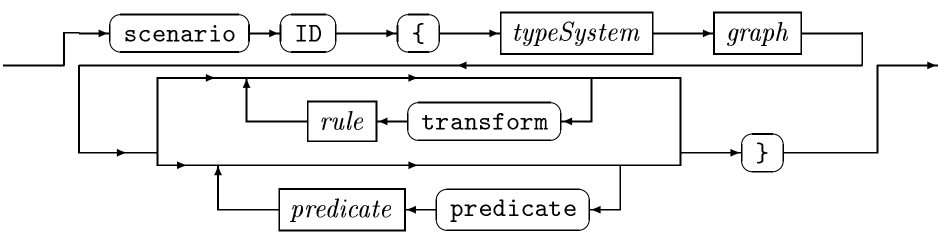
Die implementierten Erweiterungen der DiaGen -Graphtransformationsmaschine verarbeiten hierarchische Graphen mit Variablen und Prädikaten und erlauben somit das Programmieren mit Hilfe ebendieser Konzepte. Obwohl für ein als „ernsthaftes“ Programmierwerkzeug akzeptables Produkt noch einige Arbeit erforderlich ist, zeigen die in Kap. 3 vorgestellten und mit dem implementierten Tool erfolgreich getesteten Beispiele, dass bereits mit den vorhandenen Hilfsmitteln Standardalgorithmen implementiert werden können.

Anhang A

Die Eingabegrammatik für den Diaplan-Interpreter

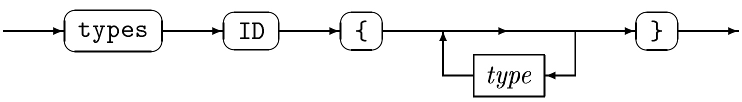
Die Grammatik des Parsers für die Eingabesprache zur Spezifikation von Testszenarien für den im Rahmen dieser Arbeit implementierten **Diaplan**-Interpreter wird hier in der Form von *Syntaxdiagrammen*[1] dargestellt. Dabei gelten folgende Konventionen: Terminalsymbole werden im `Typewriter`-Font geschrieben und durch ein Rechteck mit abgerundeten Ecken dargestellt; Nichtterminale werden *kursiv* geschrieben und befinden sich in einem „normalen" Rechteck; ein Identifier (ID) ist eine beliebige Kombination von Buchstaben, Ziffern und '`_`', wobei er nicht mit einer Ziffer beginnen oder ein Schlüsselwort sein darf.

scenario:



Das Startsymbol ist *scenario*. Jedes Szenario erhält einen Namen und besteht aus einem Typsystem, einem (Eingabe-)Graphen und einer beliebigen Menge „einfacher" Regeln oder Prädikate.

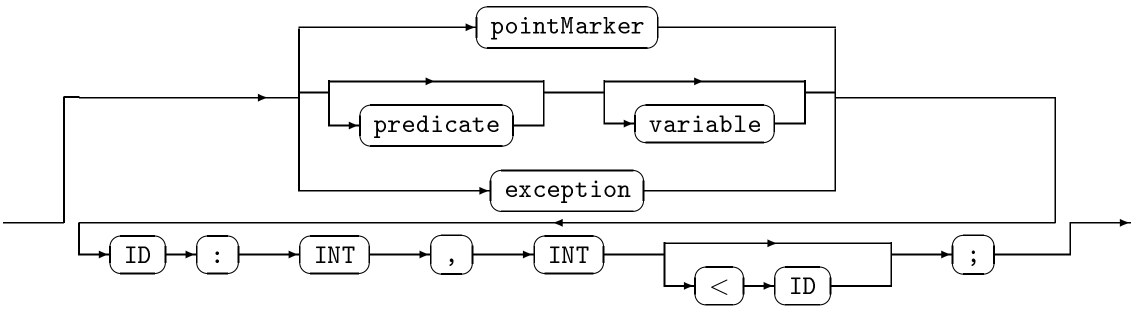
typeSystem:



Jedes Typsystem erhält ebenfalls einen Namen und besteht aus beliebig vielen (Kanten-)Typen.

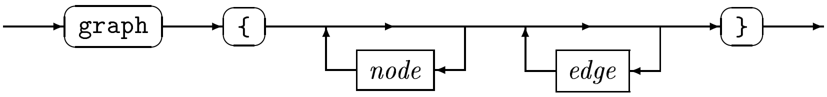
[1] `http://www.informatik.fh-muenchen.de/~schieder/programmieren-1-ws96-97/sprachen.html`

type:



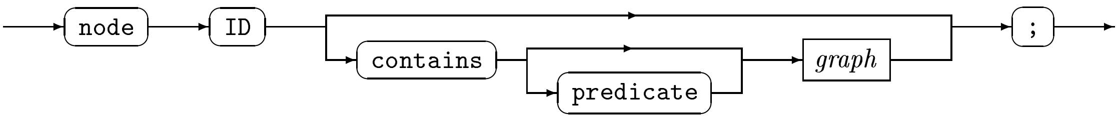
Ein (Kanten-)Typ kann ein Punktmarkierungs-, Prädikat-, Variablen-, Prädikatvariablen-, Ausnahmetyp oder nichts von alledem, also einfacher Kantentyp sein. Jeder Kantentyp erhält einen Namen, eine Stelligkeit, eine Anzahl möglicher Zusatzattribute[2] sowie ggf. einen Basistyp.

graph:



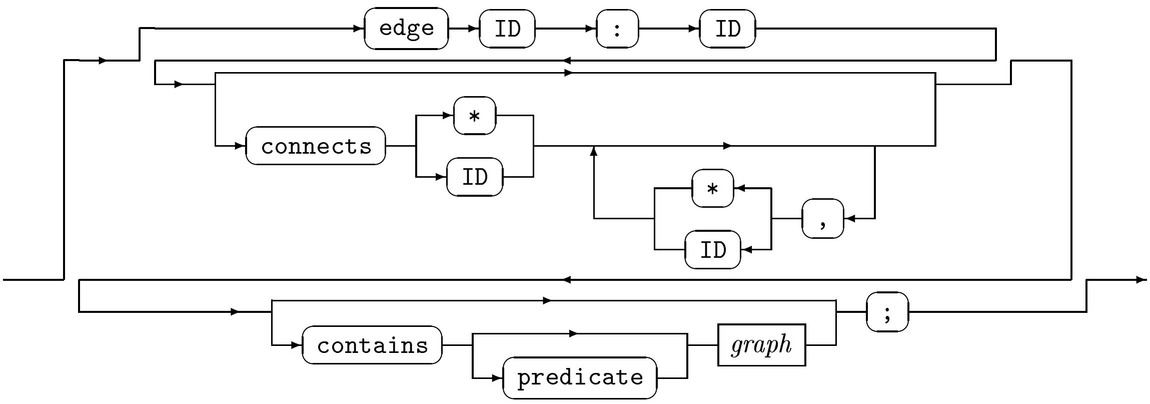
Jeder Graph besteht aus beliebig vielen Knoten und Kanten.

node:



Jeder Knoten erhält einen eindeutigen Identifier und kann einen Graphen enthalten. Enthält der Knoten einen Graphen, so deutet das Schlüsselwort `predicate` an, dass es sich um einen Prädikatbehälter handelt.

edge:

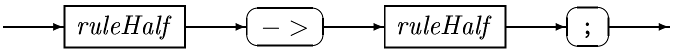


Jede kante erhält ebenfalls einen eindeutigen Identifier; der zweite Identifier ist der ihres Typs. Sie verbindet beliebig viele Knoten (der Test, ob die Stelligkeit

[2]Eine DiaGen-Spezialität, die der Interpreter bisher ignoriert.

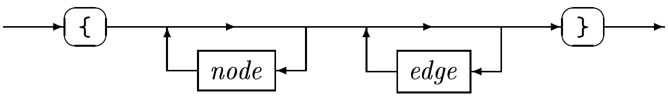
dem Typ entspricht, erfolgt nicht auf syntaktischer Ebene), wobei ein Identifier für den gleichnamigen Knoten steht und das Symbol '`*`' ein Platzhalter für eine Punktmarkierungskante ist (s.a. 5.1.2). Wie ein Knoten kann auch eine Kante einen Graphen enthalten; in diesem Fall wird die Eigenschaft, Prädikatbehälter zu sein, ebenfalls durch das Schlüsselwort `predicate` bestimmt.

rule:



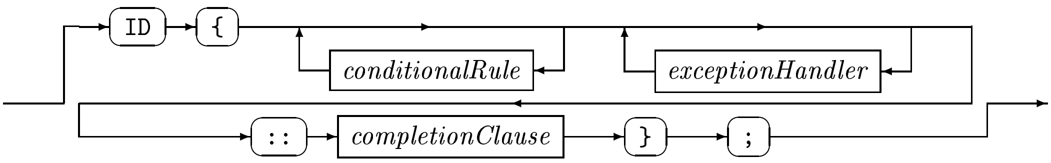
Eine „einfache“ Regel besteht aus einer linken und einer rechten Seite. Mit solchen Regeln können die Eigenschaften der DiaGen-Basisgraphtransformationsmaschine getestet werden.

ruleHalf:



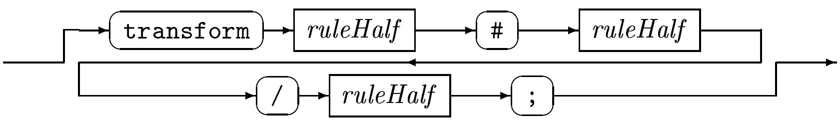
Eine *ruleHalf* stellt jeweils einen Bestandteil einer Regel dar (im Fall einer „einfachen“ Regel linke bzw. rechte Seite, im Fall einer bedingten Regel auch den Bedingungsteil).

predicate:



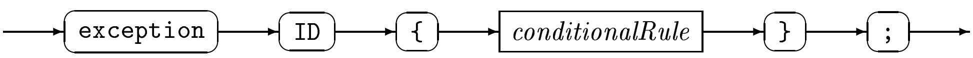
Jedes Prädikat erhält einen (zurzeit bedeutungslosen) Identifier und besteht aus beliebig vielen bedingten Regeln, Ausnahmebehandlern und einer Abschlussklausel. Statt durch das Symbol // aus der Prädikatdefinition auf S. 18, das in dieser Sprache wie in Java einen einzeiligen Kommentar markiert, wird die Abschlussklausel durch einen zweifachen Doppelpunkt gekennzeichnet.

conditionalRule:



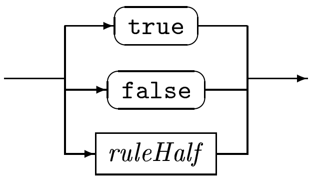
Eine bedingte Regel besteht aus drei einzelnen Regelteilen; das auf einer Tastatur nicht vorhandene Symbol ‖ aus der Prädikatdefinition wird durch das Zeichen # simuliert.

exceptionHandler:



Ein Ausnahmebehandler erhält einen zurzeit bedeutungslosen Identifier und besteht aus einer bedingten Regel.

completionClause:



Die Abschlussklausel `true` steht für „Erfolg“, die Klausel `false` für „Fehlschlag“; für eine Ausnahmeklausel ist die Ausnahmekante zusammen mit ihren Parametern in der Form eines Regelteils aufzuführen.

Literaturverzeichnis

[1] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 1973.

[2] Stefan Schiffer. *Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten*. Addison-Wesley-Longman, 1998.

[3] Andreas Heuer and Gunter Saake. *Datenbanken: Konzepte und Sprachen*. MITP-Verlag, 2000.

[4] Bernd Baumgarten. *Petri-Netze: Grundlagen und Anwendungen*. BI-Wissenschaftsverlag, 1990.

[5] Object Modeling Group. *Unified Modeling Language Specification, version 1.5*, March 2003. OMG document formal/03-03-01. `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.ps`.

[6] Berthold Hoffmann and Mark Minas. Towards rule-based visual programming of generic visual systems. In Nachum Dershowitz and Claude Kirchner, editors, *Proc. Workshop on Rule-Based Languages*, 2000.

[7] Berthold Hoffmann. Abstraction and control for shapely nested graph transformation. *Fundamenta Informaticae*, 58(1):39–65, 2003.

[8] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford Unity Press, 1992.

[9] Frank Drewes, Berthold Hoffmann, and Mark Minas. Constructing shapely nested graph transformations. In Peter Knirsch and Hans-Jörg Kreowski, editors, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)*. Universite de Grenoble, 2002.

[10] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. *Hyperedge Replacement Graph Grammars*, chapter 2, pages 95–162. Volume 1 of Rozenberg [29], 1997.

[11] Frank Drewes, Berthold Hoffmann, and Mark Minas. Context-exploiting shapes for diagram transformation. *Machine Graphics and Vision*, 12(1):117–132, 2003.

[12] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. *Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach*, chapter 3, pages 163–245. Volume 1 of Rozenberg [29], 1997.

[13] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. *Algebraic Approaches to Graph Transformation Part II: Single Pushout Approach and Comparison with Double Pushout Approach*, chapter 4, pages 247–312. Volume 1 of Rozenberg [29], 1997.

[14] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In Claus et al. [38], pages 1–69. International Workshop, Bad Honnef, 30.10. - 03.11.1978.

[15] Michael Rudolf. *Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation*. Diplomarbeit, Technische Universität Berlin, 1997.

[16] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In Andrea Corradini, Hartmut Ehrig, and Hans-Jörg Kreowski, editors, *1st Int'l Conference on Graph Transformation (ICGT'02), Barcelona*, number 2505 in Lecture Notes in Computer Science. Springer-Verlag; D-69121 Heidelberg, Germany; http://www.springer.de, 2002.

[17] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics*, volume 6 of *EACTS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[18] Frank Drewes, Berthold Hoffmann, Raimund Klein, and Mark Minas. Rule-based programming with dia**plan**. To appear in Electronic Notes in Theoretical Computer Science, 2004.

[19] Mirko Conrad, Magdalena Gajewsky, Rainer Holl-Biniasz, Michael Rudolf, Jochen Demuth, Stephan Weber, Reiko Heckel, Jürgen Müller, Gabriele Taentzer, and Annika Wagner. Graphische spezifikation ausgewählter teile von agg – einem algebraischen graphgrammatiksystem. Projektbericht zur Lehrveranstaltung „Spezifizieren mit Graphgrammatiken und Petrinetzen", TU Berlin, WS 1993/94, 1995.

[20] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991. Wiesbaden, Deutscher Universitäts-Verlag.

[21] Albert Zündorf. *PROgrammierte GRaphErsetzungsSysteme: Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, 1996. Deutscher Universitäts-Verlag.

[22] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4):110–112, Apr. 1997.

[23] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, November 2003.

[24] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, world student series edition edition, 1986. “The Dragon Book”.

[25] Mark Minas. *Spezifikation und Generierung graphischer Diagrammeditoren.* Habilitationsschrift, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2001. Shaker Verlag.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* Addison Wesley, 1996.

[27] Gilad Bracha. Generics in the java programming language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[28] Mark Minas. The EBNF grammar for editor specifications. http://www2.informatik.uni-erlangen.de/DiaGen/Doc/SpecGrammar.html.

[29] Grzegorz Rozenberg, editor. *Handbook of Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing Company, Singapore, 1997.

[30] Giorgio Busatto and Berthold Hoffmann. Comparing notions of hierarchical graph transformation. *Electronic Notes in Theoretical Computer Science*, 50(3), 2001.

[31] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Science*, 64(2):249–283, 2002.

[32] Berthold Hoffmann and Mark Minas. A generic model for diagram syntax and semantics. In J. D. P. Polim et al., editors, *Proceedings of the Satellite Workshops of the 27th International Colloquium on Automata, Languages, and Programming*, number 8 in Proceedings in Informatics, pages 443–450. Carleton Scientific, 2000.

[33] Berthold Hoffmann and Mark Minas. Transformation of shaped nested graphs and diagrams. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.

[34] Berthold Hoffmann. From graph transformation to rule-based programming with diagrams. In Nagl et al. [37], pages 165–180.

[35] Berthold Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.

[36] Hans-Jörg Kreowski. Formale Sprachen: Graphtransformation. Vorlesungsskript Universität Bremen Studiengang Informatik, Sommersemester 2004.

[37] Manfred Nagl, Andreas Schürr, and Manfred Münch, editors. *Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *Lecture Notes in Computer Science.* Springer Verlag, 1999.

[38] Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*. Springer Verlag, 1979. International Workshop, Bad Honnef, 30.10. - 03.11.1978.

Danksagung

Mein besonderer Dank geht an Berthold Hoffmann, der überhaupt erst mein Interesse an graphbasierten Sprachen geweckt hat und auch beim hundertsten Mal nicht müde wurde, mir die theoretischen Grundlagen von Dia**plan** zu erläutern, bis ich tatsächlich selbst davon überzeugt war, sie verstanden zu haben. Mark Minas danke ich für seine Erklärungen zu DiaGen, ohne die ich diese Arbeit nicht einmal ansatzweise zu einem erfolgreichen Abschluss hätte bringen können, und für die tolle Betreuung während meines Kurzaufenthalts im Bergland. Ebenso möchte ich mich bei Hans-Jörg Kreowski für die Begutachtung meiner Diplomarbeit bedanken.

Großer Dank gebührt auch Olli, der mich davon überzeugt hat, dass sich die Ausübung eines Ehrenamts mit dem Schreiben einer Diplomarbeit überhaupt nicht verträgt, nur um dann selbst für mich in die Bresche zu springen. Ich danke Hansen, Nadine, Jörn, Martin und Andre, die mich genau wie meine Verwandtschaft durch ständiges Lästern und nervige Fragen nach meinem Studienende nun doch zu selbigem getrieben haben.

Last, but certainly not least, danke ich auch meinen Eltern, ohne deren Unterstützung ich es nie bis hierher gebracht hätte.

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche gekennzeichnet.

Bremen, den 25. November 2004