Entwicklung und Implementierung eines Parsers für visuelle Sprachen basierend auf kritischer Paaranalyse

Thorsten Schultzke

8. Februar 2001

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Zu Anfang soll ein wenig das Anwendungsgebiet dieser Arbeit abgesteckt werden. Diese Motivation erhebt natürlich keinen Anspruch auf Vollständigkeit.

1.1 Motivation

Als erstes soll gleich die kritische Frage nach dem Sinn dieser Arbeit gestellt werden. Um diese Frage zu beantworten, muß wohl ein Stück weiter ausgeholt werden. Es sollen visuelle Sprachen parsiert werden. Wozu brauchen wir visuelle Sprachen? Es gibt doch textuelle Sprachen. Das ist zwar richtig, doch möchte ich an dieser Stelle behaupten, visuelle Sprachen sind schneller verständlich.

Die Ausprägungen einer visuellen Sprache, also die Sätze, begegnen den Menschen in ihrem altäglichen Leben sehr oft. So ist es möglich Straßenschilder als visuelle Sprache aufzufassen. Nach Paragraph 39 (2) StVO sind Verkehrszeichen Gefahrzeichen, Vorschriftszeichen und Richtzeichen, sowie Zusatzschilder. So ist dort genau definiert, daß Zusatzschilder schwarze Zeichnungen oder Aufschriften auf weißen Grund mit schwarzem Rand sind. Weiterhin steht dort, daß die Zusatzschilder dicht unterhalb der Verkehrszeichen angebracht werden müssen.

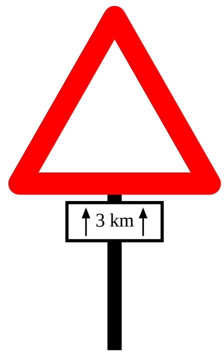

Abbildung 1.1: Gefahrenzeichen mit Zusatzschild

Ein Gefahrenzeichen mit Zusatzschild ist in Abbildung 1.1 dargestellt. Der Entwurf eines neuen Verkehrszeichens ist an Gesetze gebunden. Faßt man nun Verkehrszeichen als visuelle Sprache auf, so kann man wie in [Bar99] vorgestellt einen Editor generieren, der das Zeichnen von Verkehrszeichen gemäß einer sprachspezifischen Beschreibung unterstützt. In [Bar99] werden Zeichnungen zum Beispiel Verkehrszeichen allgemein als visuelle Sätze einer spezifizierten visuellen Sprache eingeführt.

Für die Erzeugung eines Satzes einer visuellen Sprache kann man zwei unterschiedliche Ansätze verfolgen. Zum einen kann man syntaxgesteuert vorgehen, wie in [Bar99] vogestellt wurde. Das heißt, es werden nur Editieroperationen zugelassen, die ein Verkehrszeichen gemäß der Sprachspezifikation erzeugen, da man beim syntaxgesteuerten Editieren an die Sprachspezifikation gebunden ist. Für das Beispiel bedeutet das, ein Entwickler für Verkehrszeichen kann nicht mit einem Zusatzschild anfangen, da ein Zusatzschild dicht unter dem Verkehrszeichen angebracht werden muß. Noch deutlicher wird die Problematik bei einem Austausch von Verkehrszeichen. Ein Austausch bedeutet als ersten Schritt das Entfernen des alten und als zweiten Schritt das Hinzufügen des neuen Schildes. Ist aber der erste Schritt getan, so existiert kein korrektes Schild mehr, wie in Abbildung 1.2 zu sehen ist. Das Zusatzschild muß auch entfernt werden.

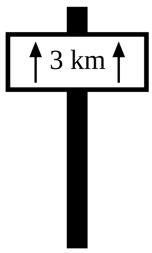

Abbildung 1.2: Nur ein Zusatzschild

Diese Art des Editierens ist sehr benutzerunfreundlich, aber leicht zu realisieren. Auf der anderen Seite gibt es die Möglichkeit, einen Satz freihand zu erstellen. Der Editor verhält sich dann ähnlich eines beliebigen Zeichenprogramms. Für den Designer eines Verkehrszeichens bedeutet das, er könnte beliebig oft und vor allem schnell diverse Schilder mit dem ausgewählten Zusatzschild kombinieren. Das Problem mit dieser Herangehensweise ist aber, daß auch Editierfunktionen zur Verfügung stehen, die ein kurzfristig unsinniges Diagramm erzeugen wie zum Beispiel ein alleinstehendes Zusatzschild. Nach dem Bearbeiten muß aber sichergestellt werden, daß ein korrektes Schild entstanden ist. Genau an der Stelle soll diese Arbeit ansetzen. Nur soll diese Arbeit sich nicht auf Verkehrszeichen beschränken, sondern generell visuelle Sprachen basierend auf Graphtransformation überprüfen können. Diese Überprüfung soll mittels Parsierung geschehen, die einen wesentlichen Anteil dieser Arbeit ausmacht.

Anwendungsgebiete für visuelle Sprachen und somit auch Parsierung lassen sich in vielen Branchen finden. In der Elektrotechnik ist das Erstellen von Schaltplänen notwendig. In der Wirtschaft finden sich immer mehr Arbeitspläne, die als visuelle Sprache zu verstehen sind. Nicht zuletzt sind auch diverse Diagrammarten wie Statecharts, Klassendiagramme oder Sequenzdiagramme in der Informatik zu finden.

1.2 Struktur der Arbeit

Da das Parsieren auf Graphtransformation basiert, wird zuerst eine kleine Einführung in die Graphtransformation in Kapitel 2 gegeben. Um diese Arbeit zu realisieren, sind Werkzeuge zur Modellierung und Umsetzung nötig. Diese werden in 3 behandelt. Die Parsierung arbeitet mit Hilfe von kritischer Paaranalyse. Die Definition der kritischen Paare, sowie deren Design und Umsetzung wird in Kapitel 4 beschrieben. Anzumerken ist, daß die kritische Paaranalyse auch ohne Parsierung benutzt werden kann. Die kritische Paaranalyse ist jedoch wesentlich für den Parsierungsprozeß. Die Beschreibung des Entwurfs und der Implementierung des Parsers ist Gegenstand von Kapitel 5. Die Benutzung der daraus enstandenen Software wird in Kapitel 6 beschrieben. Ein Aspekt dieser Arbeit ist es, die Erweiterung um andere Algorithmen zu ermöglichen. Die Erweiterungsmöglichkeiten werden in Kapitel 7 beschrieben. Das schon vorher erwähnte GenGEd-Umgebung soll in Kapitel 8 Ziel des Interesses sein. Wie schon vorher beschrieben, zeichnet sich GenGEd zur Zeit noch durch syntaxgesteuerte Editoren aus. In diesem Kapitel soll darauf eingegangen werden, was die Anforderungen und mögliche Lösungen für den Einsatz eines Parsers in GenGEd sind. Als letztes Kapitel folgt noch eine Zusammenfassung.

Kapitel 2

Einführung in die attributierte Graphtransformation

Zurückblickend auf das Beispiel aus Kapitel 1 soll der Entwurf neuer Verkehrszeichen freihand geschehen. Wie schon vorgestellt, sollen die Diagramme oder Entwürfe wie zum Beispiel Verkehrszeichen als visuelle Sprache aufgefaßt werden. In [Bar99] wird vorgestellt, wie man dazu Graphtransformation benutzt.

Dieses Kapitel soll eine kurze Einführung in die wichtigsten Begriffe der attributierten algebraischen Graphtransformation nach dem *Single-Pushout-Ansatz* geben. Ausführliche Beschreibungen befinden sich in [Roz97], [Rud97] und [ERT99]. Die Attributierung wurde besonders behandelt in [LKW93] und [Mel98]. Einige Beispiele geben ergänzend einen Eindruck der verwendeten Begriffe.

2.1 Graphen

Ein wesentlicher Bestandteil der Graphtransformation ist der Graph. Ein Graph definiert sich durch seine Knoten und Kanten. Diese Kanten verlaufen von einem Quellknoten zu einem Zielknoten. Sind Quell- und Zielknoten gerade die selben, so bildet eine Kante eine Schlaufe.

Definition 1 (*Graph*)
Ein Graph G ist ein Tupel $G = (G_V, G_E, s_G, t_G, L, m_V^G, m_E^G)$ mit

- einer endlichen Menge G_V von Knoten,
- einer endlichen Menge G_E von Kanten,
- und es gilt: $G_V \cap G_E = \varnothing$,
- zwei totalen Abbildungen $s_G, t_G : G_E \to G_V$, wobei s_g die Quellabbildung (source) und t_g ist die Zielabbildung (target) ist,
- einer Labelmenge L,
- einer Knotentypisierung $m_V^G : G_V \to L$ und
- einer Kantentypisierung $m_E^G : G_E \to L$.

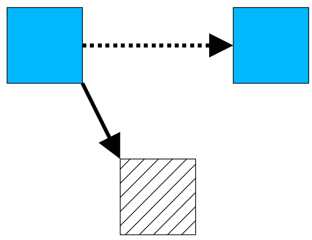

Abbildung 2.1: Beispielgraph

△

Abbildung 2.1 zeigt einen Graphen mit drei Knoten und zwei Kanten. Die Knoten- und Kantentypen werden durch ihr Layout charakterisiert. So existieren in dem Graph zwei Knotentypen: ausgefülltes und schraffiertes Rechteck. Genauso gibt es auch zwei Knotentypen: durchgezogener und gestrichelter Pfeil.

2.2 Morphismen

Zwei Graphen können in Bezug zueinander gesetzt werden. Hierzu gibt es den Begriff Graphmorphismus oder kurz Morphismus. Ein Morphismus bildet Knoten bzw. Kanten aus dem Quellgraph auf entsprechende Knoten bzw. Kanten in den Zielgraphen ab.

Definition 2 (*Morphismus*)
Ein Morphismus $m : G_1 \to G_2$ zwischen zwei Graphen $G_{[1,2]}$ nach Definition 1 ist ein Paar von partiellen Abbildungen $m_V : G_{V1} \to G_{V2}$ und $m_E : G_{E1} \to G_{E2}$ mit

- $\forall e \in G_{E1} : m_V(s_{G_1}(e)) = s_{G_2}(m_E(e))$,
- $\forall e \in G_{E1} : m_V(t_{G_1}(e)) = t_{G_2}(m_E(e))$,
- $\forall e \in G_{E1} : m_E^{G_1}(e) = m_E^{G_2}(m_E(e))$,
- $\forall v \in G_{V1} : m_V^{G_1}(v) = m_V^{G_2}(m_V(v))$.

△

Diese vier Verträglichkeitsbedingungen teilen sich in die ersten beiden Bedingungen, der Strukturverträglichkeit, und den letzten beiden Bedingungen, der Typverträglichkeit.

Abbildung 2.2 zeigt zwei Graphen 1 und 2, zwischen denen ein Morphismus (dünn gestichelte Pfeile) existiert.

2.3 Graphregeln

Um einen Graphen zu verändern sind Regeln nötig. Regeln beschreiben die Transformation von einem Graphen in einen anderen. Die Transformation bewirkt das Hinzufügen, das Löschen und das Erhalten von Knoten und Kanten. Eine Regel besteht aus zwei Graphen, der linken und rechten Regelseite.

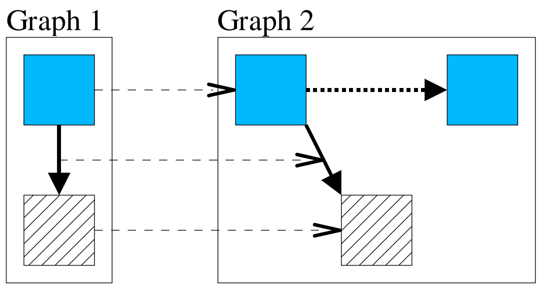

Abbildung 2.2: Beispielmorphismus

Definition 3 (*Regel*)
Eine Regel ist durch einen Morphismus $m : L \rightarrow R$ zwischen zwei Graphen gegeben. L heißt linke, R rechte Regelseite. $\triangle$

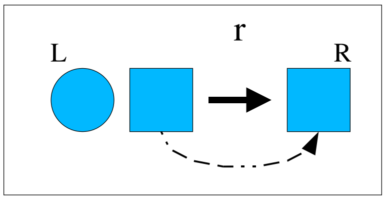

Abbildung 2.3: Beispielregel

Abbildung 2.3 zeigt eine Regel mit der linke und rechten Regelseite. Diese Regel löscht einen Knoten (Kreis). Daß der Knoten (Rechteck) erhalten wird, soll den gestrichelten Pfeil verdeutlichen.

Allgemein beschreibt eine Regel, wie ein Graph[1] verändert werden soll. Es ist jedoch unter Umständen wünschenswert anzugeben, wann eine Regel nicht angewendet werden darf. Man benutzt dazu negative Anwendungsbedingungen.

Definition 4 (*negative Anwendungsbedingungen*)
Sei r eine Regel, dann ist eine negative Anwendungsbedingung zu r ein Morphismus $\overline{l} : L \rightarrow \overline{L}$. $\triangle$

Negative Anwendungsbedingungen werden auch als *negative application condition* bezeichnet und mit **NAC**[2] abgekürzt.

Bemerkung 1 (*Regeln mit NACs*)
Eine Regel r mit NACs ist ein Paar (r,N), wobei r eine Regel und N eine Menge von NACs ist. $\triangle$

Für die Darstellung von NACs gibt es zwei Möglichkeiten. Abbildung 2.4 (a) zeigt die Darstellung, in der der NAC-Morphismus explizit dargestellt wird. In Abbildung 2.4 (b) wird an die linke Regelseite einfach ein Graph mit der Bezeichnung “NAC”angehängt.

[1]auch “Arbeitsgraph genannt”
[2]plural: NACs

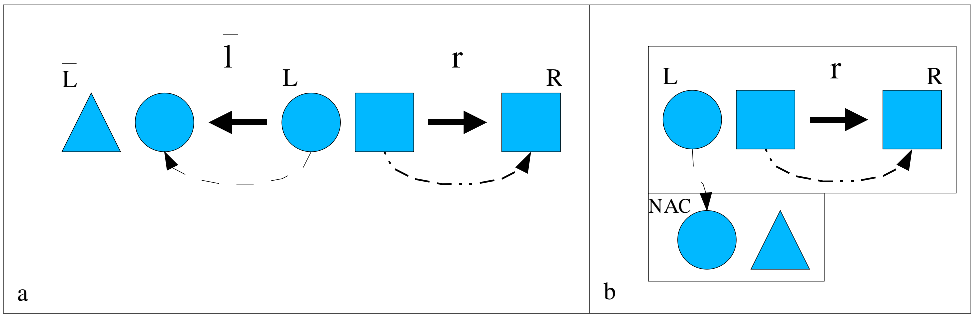

Abbildung 2.4: Beispiel-NAC

2.4 Graphgrammatik

Graphgrammatiken sind eine Verallgemeinerung von textuellen Grammatiken. Analog zu textuellen Sprachen und Grammatiken kann einen Graphgrammatik definiert werden.

Definition 5 (*Graphgrammatik*)
Eine Graphgrammatik ist ein Tuple $GG = (L_V, L_E, S, R)$ mit

- einer endlichen Menge von Knotentypen L_V,
- einer endlichen Menge von Kantentypen L_E (dabei ist aus Definition 1 $L = L_V \cup L_E$),
- einem Startgraph S und
- eine Menge R von Regeln.

$\triangle$

2.5 Ansatz

In den vorherigen Abschnitten finden wir alle Instrumente, um einen Graphen zu verändern. Es muß nur die richtige Stelle im Graphen gefunden werden, die verändert werden soll. Der Ansatz zeigt genau auf diese Stelle.

Definition 6 (*Ansatz*)
Ein gültiger Ansatz einer Regel r in einen Arbeitsgraphen G ist ein totaler Morphismus $m : L \to G$. Sollte r NACs besitzen, so gilt $\forall \overline{l} \in N : \nexists \overline{m} : \overline{L} \to G$, mit $\overline{m} \circ \overline{l} = m$. $\triangle$

Abbildung 2.5 zeigt einen gültigen Ansatz einer Regel in einen Arbeitsgraph. Da kein Morphismus von der NAC in den Arbeitsgraph gefunden wird, ist dieser Ansatz gültig.

2.6 Transformation

Ein Ansatz zeigt an welcher Stelle im Arbeitsgraphen eine Regel ansetzen soll. Dabei stellt sich die Tatsache, daß der Arbeitsgraph sehr viel größer sein kann als die linke Regelseite. Der Teil des Arbeitsgraphen, der nicht im Ansatz liegt, wird als Kontext bezeichnet. Daher

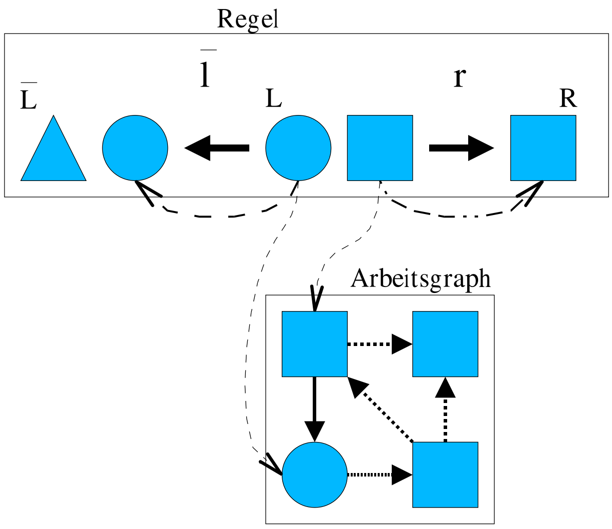

Abbildung 2.5: Beispielansatz

ist es offensichtlich, daß eine Regel mehrere Ansätze haben kann. Dieser Fakt ist auch als Nicht-Determinusmus bekannt. Der letzte Schritt ist nur noch die Regelanwendung.

Definition 7 (*Regelanwendung*)
Das Ergebnis einer Regelanwendung auf einen Graphen G mit einem Ansatz m ist ein neuer Graph H und zwei Morphismen $m^* : R \to H$ und $r^x* : G \to H$. Das Ergebnis ergibt sich durch ein *Pushout*[3]:

$$\begin{array}{ccc} L & \xrightarrow{m} & R \\ {\scriptstyle r}\downarrow & (p.o.) & \downarrow{\scriptstyle m^*} \\ G & \xrightarrow[r^*]{} & H \end{array}$$

△

Eine Regelanwendung wird auch Graphtransformationsschritt genannt.

Die Konstruktion des Ergebnisgraphen H erfolgt in zwei Schritten. Dazu sei uns ein Arbeitsgraph G, eine Regel r und ein Ansatz m gegeben.

1. **Hinzufügen und Verkleben**. Zuerst werden alle die Knoten und Kanten in den Arbeitsgraphen G eingefügt, die nicht durch den Regelmorphismus in der rechten Regelseite identifiziert werden. Durch den Ansatz m hat jedes Graphobjekt der Regel r ein Bild im Arbeitsgraphen G. Jetzt müssen noch die Graphobjekte verklebt werden, die durch den Regelmorphismus identifiziert werden. Es entsteht ein modifizierter Arbeitsgraph G'.

2. **Löschen**. Es müssen alle die Bilder von L in G' gelöscht werden, die kein Bild von L in R haben. Sollten hängende[4] Kanten auftreten, so werden sie auch gelöscht.

[3] siehe auch [Rud97] Abschnitt 6.1.1
[4] s_G oder t_G sind nicht mehr definiert.

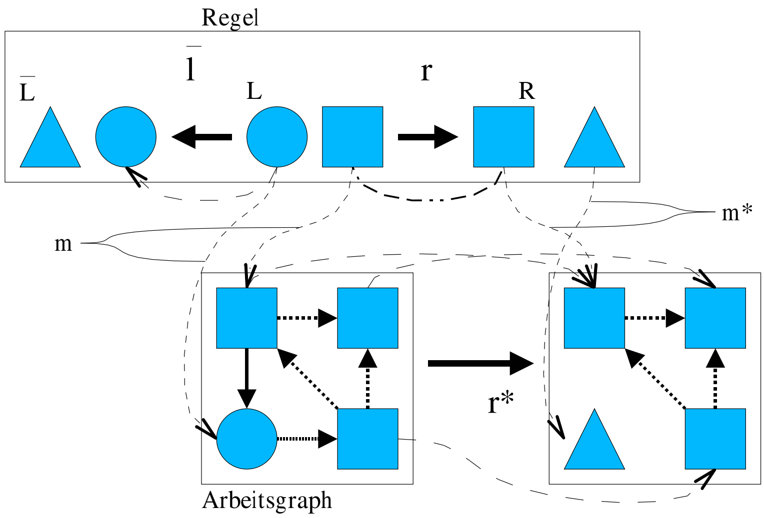

Abbildung 2.6: Beispieltransformation

In Abbildung 2.6 wird eine Transfomation gezeigt, in der der Kreis gelöscht und ein Dreieck erzeugt wird. Da der Kreis gelöscht wird, hängen zwei Kanten, nämlich genau die, die zum Kreis hin bzw. weg führen. Nach der Löschvorschrift müssen diese beiden Kanten ebenfalls entfernt werden. Um die Übersichtlichkeit in Abbildung 2.6 nicht zu verlieren, wurde darauf verzichtet die Morphismen der Kanten vom Arbeitsgraph in den abgeleiteten Graph einzuzeichnen.

Definition 8 (*Graphtransformation*)
Eine Graphtransformation bezüglich einer Graphgrammatik GG bedeutet, beliebig viele Regelanwendungen auszuführen, bis entweder kein Ansatz mehr gefunden werden kann oder bewußt die Regelanwendungen vom Benutzer gestoppt werden. Die erste Regelanwendung wird auf dem Startgraph S ausgeführt. $\triangle$

2.7 Attributierung

In fast allen Graphtransformationsansätzen können Knoten und Kanten markiert werden. Diese Markierungen können teils geändert werden, teils bleiben sie statisch während einer Transformation. Die statischen Markierungen repräsentieren die Knoten- und Kantentypen. Die veränderbaren Markierungen halten Daten in den Knoten und Kanten. Diese veränderbaren Markierungen werden als Attribute bezeichnet. Das bisherige Graphmodel war noch etwas grob und kann nun verfeinert werden. Für die Beschreibung der Attribute wird ein algebraischer Ansatz vorgestellt, der sich an [LKW93] anlehnt.

Die Attributierung wird mit Hilfe von Attributträgern modelliert. In dieser Modellierung wird zum Beispiel einem Knoten ein Knoten mit dem Attributnamen durch eine Kante verbunden. Außerdem wird diesem Knoten auch noch der Wert des Attributs zugeordnet.

Abbildung 2.7 (a) zeigt einen attributierten Graphen mittels Attributträger. In dieser Darstellungform erhält jedes Attribut einen eigenen Knoten und ist damit sehr platzintensiv. Abbildung 2.7 (b) zeigt eine äquivalente platzsparende Darstellungsform.

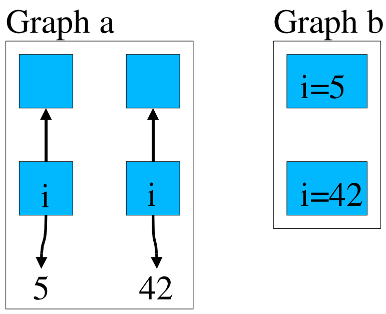

Abbildung 2.7: Attributierter Graph

Definition 9 (*Attributierte Graphsignatur*)
Eine attributierte Graphsignatur $AGS = GS + DS + ATTROP$ besteht aus einer Graphsignatur $GS = (S_{GS}, OP_{GS})$ mit nur einstellige Operationen, einer Datensignatur $DS = (S_{DS}, OP_{DS})$ und einer $S_{GS} \times S_{DS}$ indizierten Familie $ATTROP$ von Attributierungsoperationen. $\triangle$

Definition 10 (*AGS-Algebra*)
Eine Algebra A bezüglich einer attributierten Graphsignatur $AGS = GS{+}DS{+}ATTROP$ ist gegeben durch eine totale GS-Algebra A^{GS}, einer partiellen DS-Algebra A^{DS} und einer Familie von Funktionen $op^A : A_s^{GS} \to A_{s'}^{DS}$ für alle $op : s \to s' \in ATTROP$.

Gegeben sei eine weitere AGS-Algebra B, dann ist ein AGS-Morphismus $h : A \to B$ definiert durch einen partiellen Homomorphismus $h^{GS} : A^{GS} \to B^{GS}$ und einem totalen Homomorphismus $h^{DS} : A^{DS} \to B^{DS}$, so daß $h^{DS}(op^A(x)) = op^B(h^{GS}(x))$ für alle $op : s \to s' \in ATTROP$ und für alle $x \in dom(h^{GS})_s$ gilt. $\triangle$

Definition 11 (*Regel und Ansatz*)
Eine Regel $r : L \to R$ besteht aus einem AGS-Morphismus r, so daß die Algebren L^{DS} und R^{DS} Termalgebren sind und die Komponente r^{DS} ist die Identität. Ein Ansatz $m : L \to G$ ist ein totaler AGS-Morphismus. $\triangle$

Definition 12 (*Attributierte Regelanwendung*)
Gegeben sei eine Regel $r : L \to R$ und ein Ansatz $m : L \to G$. Eine attributierte Regelanwendung $G \overset{r,m}{\Rightarrow} H$ ist definiert durch ein Pushout in der Kategorie AGS [EHK+97]. $\triangle$

Ein Attribut wird durch seinen eindeutigen Namen und einem bestimmten Typ deklariert. Jedem Knoten- und Kantentyp wird nun ein Attributtyp zugeordnet. Ein konkretes Attribut eines Knoten oder einer Kante enthält zusätzlich einen zum Typ passenden Wert. Nun zeigt sich wieder in der Praxis, daß ein Attribut pro Knoten bzw. Kante nicht ausreichend ist. Anstatt eines Attributtyps wird ein Tupel aus Attributtypen gebildet. Dieser Attributtupeltyp wird dann dem Knoten- oder Kantentyp zugeordnet.

Die konkreten Attribute in einem Graph enthalten daher ein Tupel von konstanten Werten. Anders sieht es bei den Regeln aus. Um bei einer Regelanwendung Attribute zu verändern, sind hier zusätzlich Variablen erlaubt. Wie wir wissen, benötigt man für eine Regelanwendung einen Ansatz, der von der linken Regelseite in den Arbeitsgraphen geht. Durch diesen Ansatz werden so auch die Variablen in der Regel mit konkreten Werten aus dem Arbeitsgraph belegt. Da die rechte Regelseite verantwortlich für das Ergebnis einer Regelanwendung ist, können hier sogar komplexe Ausdrücke als Attribute verwendet werden.

Ähnlich wie bei den NACs der Regeln gibt es auch für Attribute Anwendungsbedingungen.

Definition 13 (*Gleichung einer Signatur SIG*)
Eine Gleichung bezüglich einer Signatur $SIG = (S, OP)$ ist ein Paar $(X, l = r)$, wobei $X = (X_s)_{s \in S}$ eine Familie von Mengen von Variablen ist, sind l, r Terme mit Variablen über X. $\triangle$

Die Termalgebra für eine Signatur SIG mit Variablen X wird geschrieben als $T_{SIG}(X)$.

Definition 14 (*Erfüllung von Gleichungen durch Auswertung*)
Sei SIG eine Signatur, X eine Menge von Variablen und $T_{SIG}(X)$ die Termalgebra mit Variablen zur Signatur SIG. Sei weiterhin A eine Algebra zur SIG und $ass : X \to A$ eine gegebene Variablenbelegung, so daß $ass^* : T_{SIG} \to A$ die eindeutige Fortsetzung der Variablenbelegung ist. ass^* ist ein SIG-Homorphismus und dient der Auswertung von Termen in A.

Weiterhin sei $e = (X, l = r)$ eine Gleichung bezüglich SIG. Dann erfüllt A die Gleichung e unter ass^*, geschrieben als $A_{ass^*} \models e$, falls $ass^*(l) = ass^*(r)$. Die Algebra A erfüllt eine Menge von Gleichungen E unter ass^*, geschrieben als $A_{ass^*} \models E$, falls $\forall e \in E : A_{ass^*} \models e$ gilt. $\triangle$

Diese Anwendungsbedingungen sind allerdings positiv. Das heißt, eine Anwendungsbedingung muß erfüllt sein, damit eine Regel angewendet werden darf. Sollte für ein Attribut keine Anwendungsbedingung existieren, so darf immer die Regel mit gültiger Variablenbelegung angewendet werden.

Abbildung 2.8 zeigt zwei attributierte Regeln und einen attributiertn Arbeitsgraph mit einer Beispielregelanwendung.

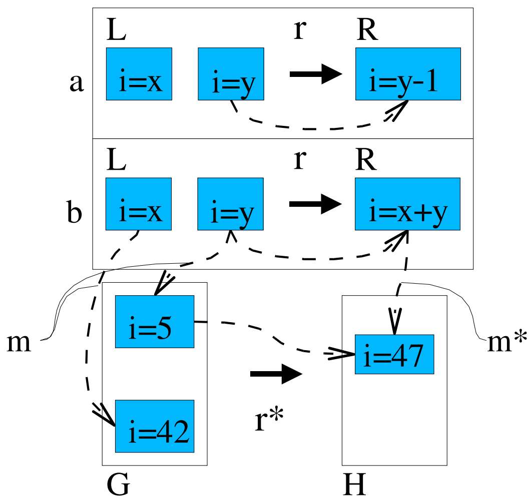

Abbildung 2.8: Attributierte Beispielregelanwendung

2.8 Schichtenmodell

Ohne auf spätere Kapitel vorzugreifen, muß gesagt werden, daß Parsieren sehr aufwendig ist. Zudem ist allgemein auch keine Terminierung garantiert. Es ist also wünschenswert, einen effizienten Parsierungsalgorithmus zu entwickeln. Eine Möglichkeit Graphgrammatiken so zu erweitern, daß diese Eigenschaften erfüllt werden, bietet das sogenannte Layern. Im Wesentlichen handelt es sich dabei, um eine Schichtung in der Menge der Regeln einer Graphgrammatik. Eine Schicht wird als Layer bezeichnet. Dabei werden Regeln in bestimmten Layern zusammengefaßt. Im Extremfall besteht ein Layer nur aus einer Regel, oder im anderen Extrem hat eine Graphgrammatik nur einen Layer. Dieser letzte Fall entspricht einer ungelayerten Graphgrammatik. Eine wesentliche Bedingung wird aber an die Regeln gestellt, sie dürfen nur genau einem Layer zugeordnet werden. Die Änderung zur ungelayerten Graphgrammatik sieht nun folgendermaßen aus: Die Layer werden nummeriert. Eine weitere Konsequenz ergibt sich noch für die Graphgrammatik. Die Auswahl der anzuwendenden Regel darf nicht mehr beliebig ausgeführt werden. Es müssen zuerst die Regeln angewendet werden, die im niedrigsten Layer enthalten sind. Sollte aus einem bestimmten Layer keine Regel mehr angewendet werden können, so wird ein Ansatz für eine Regel aus dem nächst höherem Layer gesucht. Ein Rückschritt zu einem niederen Layer ist nicht mehr möglich.

Zusätzlich können noch weitere beliebige Bedingungen für Layer definiert werden. Eine Möglichkeit befindet sich in [BST00]. In [BST00] handelt es sich um die zusätzliche Überprüfung der Typen und der NACs. In dieser Arbeit wurden zwei Varianten aus [BST00] umgesetzt.

Die erste umfassende Layerfunktion fordert, daß es zwei weitere Layerfunktionen gibt, die jedem Typ einen Layer zuordnen. Die eine Funktion wird als *Creation Layer* bezeichnet, da sie vor allem für die Regeln interessant ist, in denen ein Graphobjekt erzeugt wird. Die andere Funktion heißt *Deletion Layer*, die besonders für zu löschende Objekte interessant ist. Die Layerfunktion fordert nun, daß mindestens ein Graphobjekt in jeder Regel gelöscht wird. Ferner sollen der *Deletion Layer* der zu löschenden Graphobjekte kleiner oder gleich des Regellayer sein. Entgegengesetzt muß der Creation Layer jedes erzeugte Graphobjekt größer als der *Rule Layer* sein.

Definition 15 (*Layerfunktion 1*)
Gegeben sei eine Menge L mit allen Knoten- und Kantentypen, sowie eine Menge R von allen Regeln. Dann existiert die folgenden Layerfunktionen:

- $cl, dl : L \to NAT$ sind totale Funktionen, die jedem Typ einen eindeutigen *Creation* und *Deletion Layer* zuordnen, so daß $\forall l \in L : 0 \leq cl(l) \leq dl(l)$.
- $rl : R \to NAT$ ist eine totale Funktion, die jeder Regel einen eindeutigen Layer zuordnet.

Weiterhin erfülle die Layerfunktion folgende Eigenschaften: $\forall r \in R$ gelte:

- r löscht mindestens einen Knoten oder eine Kante.
- r löscht nur Knoten oder Kanten vom Typ l, so daß $dl(l) \leq rl(r)$.
- r erzeugt nur Knoten oder Kanten vom Typ l, so daß $cl(l) > rl(r)$.

$\triangle$

Die zweite Variante der Layerfunktion entspricht im wesentlichen der ersten. Ergänzend muß jedoch erfüllt sein, daß der Creation Layer der Graphobjekte der NACs kleiner oder gleich dem Regellayer ist.

Definition 16 (*Layerfunktion 2*)
Es gelte Definition 15. Zusätzlich erfülle die Layerfunktion folgende Eigenschaften: $\forall r \in R$ gelte:

- r enthält nur NACs mit Knoten und Kanten vom Typ l, so daß $cl(l) \leq rl(r)$.

$\triangle$

Für das Beispiel in Abbildung 2.8 können folgende Layer gemäß Definition 15 angegeben werden:

Gegeben sei eine Graphgrammatik GG mit:

- Menge der Knotentypen $L_V = \{$“Ausgefülltes Rechteck”$\}$,
- Menge der Kantentypen $L_E = \emptyset$,
- Startgraph $S = G$ aus Abbildung 2.8 und
- Menge der Regeln $R = \{$Regel a, Regel b$\}$ aus Abbildung 2.8.

Dann ist die Layerfunktion wie folgt definiert:

- der *Creation Layer* $cl($“Ausgefülltes Rechteck”$) = 0$
- der *Deletion Layer* $dl($“Ausgefülltes Rechteck”$) = 0$
- der *Rule Layer* $rl($Regel a$) = 1$
- und $rl($Regel b$) = 0$

Kapitel 3

Werkzeuge

Die Realisierung eines Parsers soll in einem Softwarepaket für AGG resultieren. Vorgestellt wird AGG in [ERT99]. Die Entwicklung von Software ist ein komplexer Vorgang, so existieren verschiedenste Theorien diesen Vorgang zu vollziehen. Welche Methode man auch immer verfolgt, so muß ein Modell erstellt werden. Für die Modellierung wurde in dieser Arbeit UML verwendet, die nachfolgend noch weiter erläutert wird. In der rechnergestützte Modellierung kam SoftModeler zum Einsatz.

Für die Implementierung wurde die Programmiersprache Java benutzt. Erläuterungen zu der verwendeten Programmiersprache Java werden ebenfalls später gegeben. Im objektorientierten Design treten immer wieder ähnliche Probleme auf. Eine Sammlung dieser Probleme und ihre Lösung bschreiben Design Pattern. Sie werden kurz als letztes in diesem Kapitel vorgestellt.

3.1 Modellierung mit UML

UML steht für Unified Modeling Language. Es ist eine graphische Sprache zur Visualisierung, Spezifikation, Konstruktion und Dokumentation von Softwaresystemen. Dabei sind die Sprachelemente der verschiedenen Diagramme mit eindeutiger Semantik versehen. Weiterhin ist die Spezifikation in verschiedenen Detaillierungsgraden möglich. Sie ist von sehr abstrakt bis sehr implementierungsnah möglich. So kann ein erstelltes Modell auf verschiedene Programmiersprachen abgebildet werden (Forward Engineering). UML ist zur Zeit State-Of-The-Art in der Modellierung, man kann sagen, UML ist eine standardisierte Modellierungssprache. Es wird jedoch kein bestimmter Softwareentwicklungsprozeß vorgeschrieben. D.h. UML ist keine Methode. Weitere Informationen befinden sich unter [OMG01].

Von den unterschiedlichen UML-Diagrammarten sollen in dieser Arbeit nur zwei benutzt werden: Klassendiagramme und Sequenzdiagramme. Der Einsatz von UML wird inzwischen durch das breite Angebot an Werkzeugen erleichtert. In dieser Arbeit wurde als CASE-Tool der SoftModeler eingesetzt.

3.1.1 Klassendiagramm

Das Klassendiagramm beschreibt die Struktur des Systems bestehend aus Klassen und ihren Beziehungen. In dem Diagramm wird kein Verhalten der Klassen sondern nur die statische Beziehung zwischen den Klassen dargestellt (s. Abbildung 3.1).

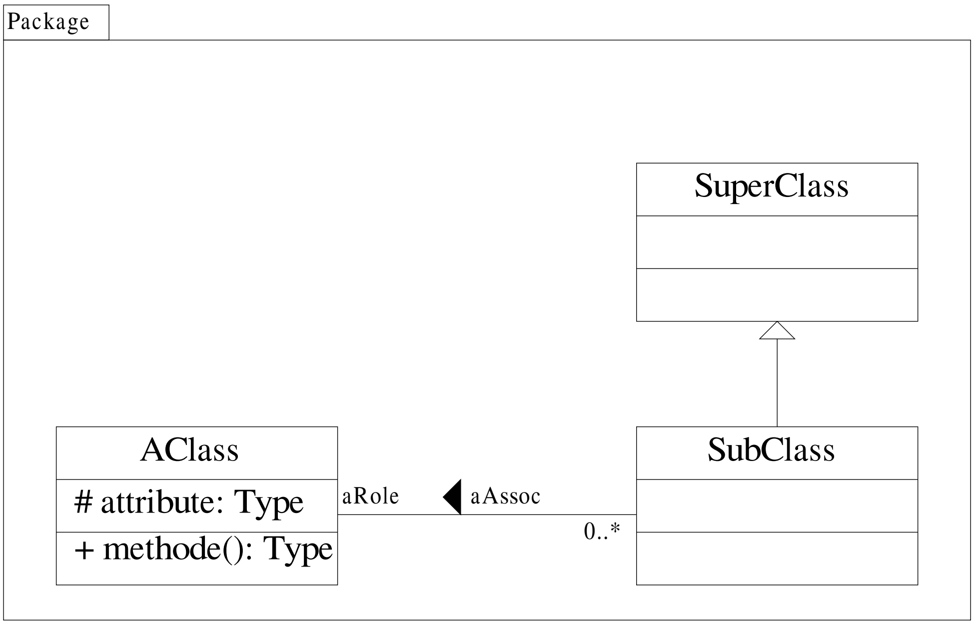

Abbildung 3.1: Beispiel eines Klassendiagramms

Eine Klasse ist eine Menge von Objekten, in der die Eigenschaften (Attribute), Operationen und die Semantik der Objekte definiert werden. Alle Objekte einer Klasse entsprechen dieser Festlegung. Eine Klasse ist eine Zusammenfassung gleichartiger Objekte. Objekte sind die agierenden Grundelemente einer Anwendung. Die Gleichartigkeit bezieht sich auf Eigenschaften (Attribute) und/oder auf Fähigkeiten (Operationen/Methoden) der Objekte einer Klasse. Eine Klasse enthält gewissermaßen die Konstruktionsbeschreibung für Objekte die mit ihr erzeugt werden. Das Verhalten der Objekte wird durch die Möglichkeit eines Objektes, Nachrichten zu empfangen und zu verstehen beschrieben. Dazu benötigt das Objekt bestimmte Operationen. Die Begriffe Operation und Nachricht sollten nicht synonym verwendet werden.

Klassen werden durch Rechtecke dargestellt, die den Namen der Klasse und die Attribute und Operationen der Klasse enthalten. Klassenname, Attribute und Operationen werden durch eine horizontale Linie getrennt. Der Klassenname steht im Singular und beginnt mit einem Großbuchstaben. Attribute können näher beschrieben werden, z.B. durch ihren Typ und einen Initialwert. Sie werden aber mindestens mit ihrem Namen aufgeführt. Operationen können ebenfalls durch Parameter beschrieben werden. Auch Sie werden mindestens mit ihrem Namen aufgeführt.

Ein Attribut ist ein Datenelement, das in jedem Objekt einer Klasse gleichermaßen enthalten ist und von jedem Objekt mit einem individuellen Wert repräsentiert wird. Attribute sind Informationen bzw. Daten die ein Element einer Klasse näher beschreiben. Sie werden mindestens durch ihren Namen beschrieben, können aber zusätzlich einen Initialwert besitzen. Mit Hilfe von Merkmalen können weitere besondere Eigenschaften von Attributen beschrieben werden, z.B. daß ein Attribut nur gelesen werden darf. Eine weitere Ausprägung von Attributen sind Klassenattribute. Sie gehören nicht zu einem einzelnen Objekt, sondern zu einer Klasse. Alle Objekte dieser Klasse können ein solches Klassenattribut benutzen.

Es gibt, je nach Programmiersprache, die Möglichkeit die Sichtbarkeit der Attribute nach außen hin einzuschränken. Dies geschieht mit Hilfe der Sichtbarkeitskennzeichen:

- public: verfügbar für alle Klassen des Systems, abgekürzt +.
- protected: verfügbar für Objekte der eigenen und aller abgeleiteten Klassen, abgekürzt #.
- private: verfügbar nur für Objekte dieser Klasse, abgekürzt -.

Ein weiteres Element in Klassendiagrammen ist das Paket oder auch Package genannt. Pakete sind Ansammlungen von Modellelementen mit denen das Gesamtmodell in kleine überschaubare Einheiten untergliedert wird. Sie definieren keine Modellsemantik. Jedes Modellelement gehört zu genau einem Paket. Pakete können verschiedene Modellelemente, z.B. Klassen. Sie können hierarchisch gegliedert werden, also ihrerseits wieder Pakete enthalten. Pakete werden aufgrund physischer oder logischer Zusammenhänge gebildet. Vorhandene Bibliotheken, Untersysteme und Schnittstellen bilden jeweils eigene Pakete. Wird das eigentliche Modell zu groß, kann es ebenfalls nach logischen Gesichtspunkten in Pakete gegliedert werden. Pakete dienen zur Modellorganisation, definieren aber keine Modellsemantik. Eine Klasse besitzt immer ihr Stammpaket, sie kann jedoch auch in verschiedenen anderen Klassen vorkommen. Dann wird sie allerdings nur in der Schreibweise *Paketname::Klassenname* zitiert. Dadurch entstehen Abhängigkeiten zwischen den Paketen, beispielsweise nutzt eine Klasse Klassen anderer Pakete.

Pakete werden als Aktenregister dargestellt, welches den Namen des Paketes enthält. Oberhalb des Paketnamens können Stereotypen notiert werden. Stereotypen beschreiben dabei noch weiter das Packet und haben dazu den Vorteil, daß sie nicht die Struktur des Diagramms verändern. Werden innerhalb des Symbols Modellelemente angezeigt, steht der Name auf der Registerlasche, andernfalls innerhalb des Rechteckes.

Vererbung ist ein weiteres Element in der objektorientierten Entwicklung. Vererbung ist ein Programmiersprachenkonzept, d.h. ein Umsetzungsmechanismus für die Relation zwischen Oberklasse und Unterklasse, wodurch Attribute und Operationen der Oberklasse auch den Unterklassen zugänglich werden. Mit Hilfe der Vererbung können Klassen hierarchisch strukturiert werden. Dabei werden Eigenschaften einer Oberklasse an die zugehörige Unterklasse weitergegeben.

Bei der Generalisierung bzw. Spezialisierung werden Eigenschaften hierarchisch gegliedert, d.h. Eigenschaften mit allgemeinerer Bedeutung werden Oberklassen zugeordnet und speziellere Eigenschaften werden Unterklassen zugeordnet. Eigenschaften der Oberklasse werden dabei an die zugehörige Unterklasse weitergegeben. Eine Unterklasse verfügt folglich über ihre speziellen Eigenschaften und über die Eigenschaften ihrer Oberklasse(n). Die Unterscheidung in Ober- und Unterklasse erfolgt m.H. eines Unterscheidungsmerkmals, dem sogenannten Diskriminator. Dieser definiert den für die Strukturierung maßgeblichen Aspekt. Er ist nicht von selbst gegeben, sondern das Ergebnis der Modellierungsentscheidung. Beispielsweise könnte man Fahrzeuge aufgrund des Diskriminators Antriebsart (Benzinmotor, Dieselmotor, Elektromotor) gliedern oder aber aufgrund des Fortbewegungsmediums (Luft, Wasser, Straße, Schiene).

Die Vererbung wird mit einem nicht ausgefüllten Pfeil, der von der Unterklasse zur Oberklasse zeigt dargestellt. Die Pfeile von den Unterklassen können zu einer gemeinsamen Linie zusammengefaßt werden oder direkt zur Oberklasse gezogen werden. Bei der Mehrfachvererbung[1] kann eine Unterklasse mehr als eine Oberklasse besitzen.

[1] In Java ist Mehrfachvererbung nur in Interfaces möglich.

Eine Assoziation beschreibt als Relation zwischen Klassen, die gemeinsame Semantik und Struktur einer Menge von Objektverbindungen. Varianten der Assoziation sind die Aggregation und die Komposition. Die Multiplizität einer Assoziation gibt an, mit wievielen Objekten der gegenüberliegenden Klasse ein Objekt assoziiert sein kann. Ist diese Zahl variabel, wird die Bandbreite angegeben, also Minimum und Maximum. Ist das Minimum gleich Null, so ist die Beziehung optional. Auf jeder Seite der Assoziation können Rollennamen vergeben werden. Diese beschreiben, welche Rolle die jeweiligen Objekte in der Beziehung einnehmen. Im allgemeinen sind Assoziationen notwendig damit die Objekte miteinander kommunizieren können. Die konkrete Beziehung zwischen zwei Objekten wird Objektverbindung (engl. Link) genannt. Sie ist damit eine Instanz einer Assoziation.

Assoziationen werden durch eine Linie zwischen den beteiligten Klassen dargestellt. Die Linie wird mit einem Namen versehen (kursiv), der beschreibt, worin und warum diese Beziehung besteht. Neben dem Beziehungsnamen kann ein kleines ausgefülltes Dreieck, dessen Spitze in Leserichtung zeigt, gezeichnet werden. An den Enden der Verbindungslinie kann die Multiplizität der Beziehung angegeben werden. Sie wird als einzelne Zahl oder als Wertebereich auf jeder Seite der Assoziation notiert. Der Wertebereich wird wie folgt notiert:

Angabe des Minimums und des Maximums, getrennt durch zwei Punkte.

- Mit einem * wird der Joker beschrieben, also "viele".
- Mit einem Komma können unterschiedliche Möglichkeiten der Multiplizität aufgezählt werden.

Beispiele für Multiplizitätsangaben:

- 1 genau eins
- 0, 1 null oder eins
- 0..4 zwischen null und vier
- 0..* größer oder gleich null

3.1.2 Sequenzdiagramm

Sequenzdiagramme sind Interaktionsdiagramme, die der Modellierung des Kontrollflusses dienen. Sequenzdiagramme zeigen das dynamische Systemverhalten zeitorientiert ("Was passiert im zeitlichen Ablauf?"). Abbildung 3.2 zeigt ein Beispielsequenzdiagramm. In dem Diagramm wird die Kommunikation der einzelnen Objekte (Instanzen eines Klassendiagramms) untereinander dargestellt, sowie die Interaktionen zu Objekten festgehalten. Für jeden Fall der in dem System eintreten kann, muß ein seperates Sequenzdiagramm erstellt werden, da das Diagramm nur den Ablauf eines Falls in Abhängigkeit von der Zeit darstellt. Am einfachsten zeigt sich diese Tatsache an einer bedingten Verzweigung, auch *IF-THEN-ELSE* genannt. Für diesen Konstrukt sind zwei Sequenzdiagramme nötig. Einmal muß der Ablauf für den *IF*-Fall und andererseits der Ablauf für der *ELSE*-Falls angegeben werden.

Die Objektinstanzen der Sequenzdiagramme werden durch ein Rechteck, das den Typ enthält, dargestellt. Zusätzlich kann noch dem Objekt ein Name zugeordnet werden. Der

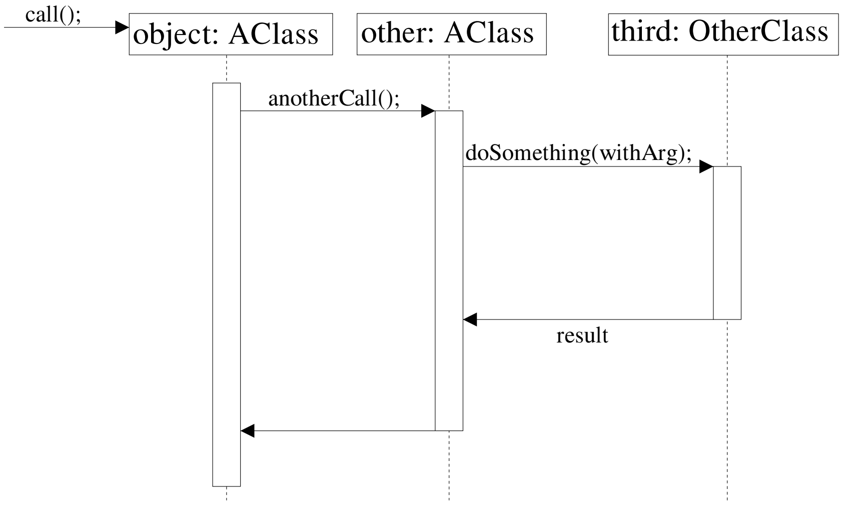

Abbildung 3.2: Beispiel eines UML-Sequenzdiagramms

Name sowie der Typ werden unterstrichen. Entlang der vertikalen Achse des Sequenzdiagramms verläuft die Zeit. Die vertikale Achse gibt jedoch keine quantitative Aussage über den Zeitverlauf. Durch die Höhe des Sequenzdiagramms wird lediglich symbolisiert, daß Zeit vergeht, aber nicht wieviel. An ein Objekt hänkt eine gestrichelte Linie und stellt die Lebensdauer des Objekts dar. Ein Methodenaufruf wird als Pfeil notiert, und die zeitliche Ausdehnung der Ausführung dieses Aufrufs symbolisiert dann ein weißer Balken. Innerhalb eines Methodenaufrufs können weitere Aufrufe auf andere Objekte stattfinden. Terminiert eine Methode, so wird die Rückkehr des Kontrollflusses zum Aufrufer durch einen Pfeil vom terminierenden Objekt zum Aufrufer dargestellt. Dabei kann ein Rückgabewert zusätzlich angegeben werden. Das Ende der Ledensdauer eines Objekts wird durch ein Kreuz am Ende der Lebenslinie gekennzeichnet.

3.2 SoftModeler

SoftModeler ist eine UML-basierte Java-Modellierungsumgebung, die in Benutzerfreundlichkeit und Leistungsfähigkeit übertrifft. SoftModeler erweitert UML um den funktionellen Entwurf verteilter Komponenten in einer eindeutigen Weise, die im wesentlichen Entwicklungszeit verringert. SoftModeler unterstützt Teamwork und hat viele hochentwickelte Eigenschaften wie nahtlose Javacodeerzeugung, der Speicherung von Diagrammen als GIF-Dateien und anderer Möglichkeiten. SoftModeler unterstützt den vollen Javaentwicklungszyklus, während die Komplexität vom UML beseitigt wird. SoftModeler ist plattformunabhängig, da es komplett in Java geschrieben ist. Weitere Informationen befinden sich unter [Sof01].

3.3 Realisierung mit JAVA

Da diese Arbeit das bestehende AGG-System (s. [ERT99]) erweitert, mußte die Programmiersprache verwendet werden, mit der schon AGG implementiert wurde. Weitere

Informationen finden sich in Kapitel 3.3 in [Rud97] und Kapitel 3.3 in [Mel98]. Für die Entwicklung wurde das Java Development Kit 2 verwendet (s. [Sun00]).

Als nächstes sollen nur ein paar Eigenschaften aufgezählt werden, die Java enthält:

Einfach: Java wurde bewußt schlank gehalten, was die fest in die Sprache integrierten Befehle und Möglichkeiten anbetrifft. Vieles ist aus den Sprachen C und C++ übernommen worden, einiges wurde weggelassen. So gibt es beispielsweise keine Pointer, d.h. der Programmierer muß sich nicht um die Speicherverwaltung kümmern.

Objektorientiert: Die Objektorientierung bietet einen entscheidenden Vorteil für Entwickler, die mit dieser Art der Modellierung und Programmierung vertraut sind. Durch sie ist es möglich, auf sehr schnelle Weise Sachverhalte und Theorien als Objekte mit Eigenschaften und Verhaltensweisen zu modellieren und in eine Implementierung umzusetzen. Durch Vererbungsmechanismen können mitunter viele Programmzeilen eingespart werden.

Robust: Die starke Typsisierung von Java sowie das zuverlässige Speichermodell verhindern einige Arten von Programmierfehlern. Der Programmierer muß sich beispielsweise keinerlei Gedanken mehr zur Freigabe von benutztem Speicher machen, da Java über einen Mechanismus namens Garbage Collection verfügt, der automatisch nicht mehr benötigten Speicher freigibt. Durch die Möglichkeit, zur Laufzeit auftretende Fehler und Ausnahmen abzufangen und zu behandeln, wird ein Programm ebenfalls robuster.

Portabilität: Java ist plattformunabhängig. Das heißt: Java-Applikationen laufen auf jedem System, auf dem eine Java Virtual Machine implementiert ist. Dadurch ist es möglich, daß Programme sowohl unter Linux, Solaris, Windows 98, Windows NT als auch auf einem Macintosh laufen. Wird die grafische Benutzeroberfläche mit Hilfe der Swing-Klassen[2] implementiert, kann sogar sichergestellt werden, daß die Arbeitsfläche auf allen Plattformen gleich aussieht.

Multi-Thread-Unterstützung: Java unterstützt die Ausführung mehrerer Threads, die verschiedene Aufgaben wahrnehmen können.

Interpretersprache: Java ist eine Interpretersprache, die zwar in einen plattformunabhängigen Byte-Code kompiliert, aber erst zur Laufzeit in Maschinen-Code übersetzt wird. Das Übersetzen kostet viel Zeit. Vergleiche haben gezeigt, daß ein Java-Programm wesentlich langsamer läuft als ein vergleichbares C++-Programm.

GUI: Die Java Foundation Classes und besonders der Teil Swing bieten dem Entwickler einen leichten Zugang, um grafische Benutzeroberflächen zu erzeugen. Java bietet eine Vielzahl von vorgefertigten Komponenten, die leicht in einem Baukastensystem zusammengesetzt werden können. Eine prototypische Präsentation der Arbeit kann schnell vorgestellt werden.

[2] auch Java Foundation Classes (JFC) genannt

3.4 Design Pattern

Im objektorientierten Entwurf treten immer wieder spezifische Probleme auf, die einfach und elegant mit Design Pattern [GHJV95] gelöst werden können. Christopher Alexander sagte einmal:"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same way twice." [AIS+77]. Obwohl Alexander von Pattern in Gebäuden und Städten spricht, trifft diese Aussage auch genau auf objektorientierte Design Pattern zu. Design Pattern erfassen allgemein Lösungen, die sich im Laufe der Zeit entwickelt haben. Sie sind insofern allgemein, als daß sie von den Eigenschaften einer konkreten Programmiersprache abstrahieren. Design Pattern haben in der Implementierung einen gering höheren Arbeitsaufwand als *ad-hoc* Lösungen, sie bestechen aber in einem hohen Maß an Flexibilität und Wiederverwendbarkeit.

In dieser Arbeit kamen folgende Design Pattern zum Einsatz:

Erbauer: Dieses Pattern trennt die Konstruktion komplexer Objekte von seiner Repräsentierung.

Strategie: Eine Familie von Algorithmen wird so definiert, daß jeder Algorithmus in einer eigenen Klasse gekapselt wird. Dadurch können Algorithmen leicht ausgetauscht werden.

Kapitel 4

Kritische Paare

Denkt man wieder an ein freihand entworfenes Verkehrszeichen, so muß dieses neue Verkehrszeichen auf Korrektheit gemäß der Sprachdefinition überprüft werden. Für das Überprüfen eines Entwurfs soll Parsierung verwendet werden. Das Parsieren ist im allgemeneinen sehr rechenintensiv. Um diese Rechenzeit zu verkürzen, kann ein Parser optimiert werden. Der Parser basiert auf Graphgrammatik, genau an dieser Stelle soll die Optimierung ansetzen.

In einer Graphgrammatik stehen Regeln untereinander in keinem Zusammenhang. Eine Graphgrammatik ist ein Tupel (T_V, T_E, S, P) mit einer endlichen Menge T_V von Knotentypen, einer endlichen Menge T_E von Kantentypen, einem Startgraph S und einer Menge P von Regeln. In einer Menge gibt es keine Reihenfolge oder ähnliches zwischen den Elementen der Menge. Ein einfacher Weg einen Parser zu realisieren, ist ein Backtrackverfahren zu benutzen. Der Backtracking-Algorithmus arbeitet nach dem "trial and error" Prinzip, was so viel bedeutet, wie "Versuchen und Nachprüfen". Man macht dabei immer einen Schritt zum Ziel hin, zeichnet diesen auf und macht den nächsten Schritt. Führen diese Schritte in eine Sackgasse, so löscht man die Aufzeichnungen und probiert die nächste Lösung aus. Ein Parser mit einem primitiven Backtrackverfahren würde jeden Schritt aufzeichnen. Ein Schritt bedeutet in diesem Zusammenhang eine Regelanwendung. An dieser Stelle läßt sich das Backtracking optimieren. Der Rechenaufwand kann reduziert werden, indem Beziehungen zwischen den Regeln hergestellt werden. Diese Beziehung unterscheidet nun die zwei Fälle: Steht eine Regel mit einer anderen Regel im Konflikt oder nicht. Dabei muß sogar der Sonderfall betrachtet werden, daß eine Regel auch mit sich selbst in Konflikt stehen kann. Steht eine Regel mit einer anderen in Konflikt, so ist sie Element der Exclude-Beziehung. Ein wichtiger Bestandteil eines Parsers wird daher auch das Finden aller Regeln sein, die mit einer oder mehreren Regeln in Konflikt bzw. mit keiner Regel in Konflikt steht.

4.1 Exclude

Die Exclude-Beziehung zerfällt in drei Teile, dabei beschreiben zwei Bedingungen die Struktur und die dritte den Attributteil.

Definition 17 (*Excluderelation*)
Gegeben seien zwei Regeln r_1, r_2, die über die Ansätze m_1 und m_2 auf einen gemeinsamen Arbeitsgraph G_0 anwendbar sind (schematisch in Abbildung 4.1 dargestellt). Die

Bedingungen sehen wie folgt aus:

1. (r_1, m_1) löscht Graphobjekte, die (r_2, m_2) zum Ansatz benötigt.

2. (r_1, m_1) erzeugt Graphobjekte, die (r_2, m_2) durch die negative Anwendungsbedingung $\overline{r_2}$ verbietet.

3. (r_1, m_1) ändert Attribute, die (r_2, m_2) zum Ansatz benötigt.

Formal:

$$m_1(L_1 - dom(r_1)) \cap m_2(L_2) \neq \varnothing \tag{4.1.i}$$
$$\exists l_2 : \overline{L_2} \to G_1 \wedge n_1(R_1 - codom(r_1)) \cap l2(\overline{L_2}) \neq \varnothing \tag{4.1.ii}$$

△

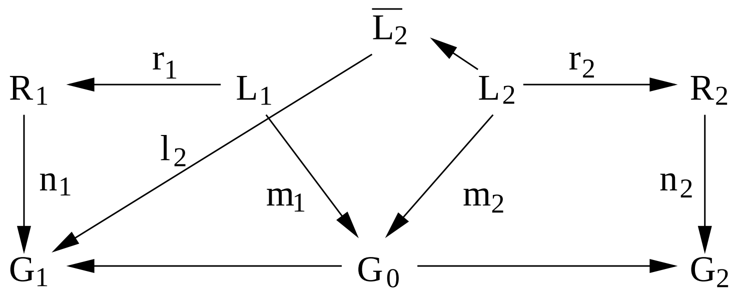

Abbildung 4.1: Excludebeziehung

Diese Excluderelation ist stark situationsabhängig. Es hängt vom Arbeitsgraph G_0 ab, ob die jeweiligen Bedingungen eintreten. Abbildung 4.2 zeigt zwei Regeln r_1, r_2, die einen

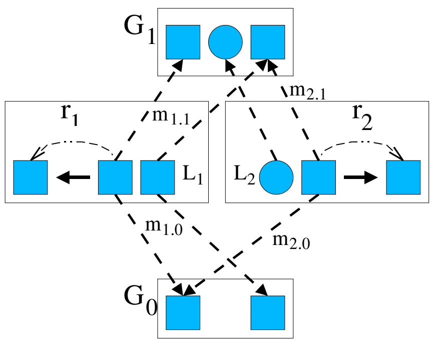

Abbildung 4.2: Beispiel für unterschiedliche Ansätze mit unterschiedlichen Ergebnissen

Ansatz in einen Arbeitsgraphen $G_{[0,1]}$ suchen. Die Regel r_1 löscht dabei einen Knoten (rechtes Quadrat) und erhält den anderen (linkes Quadrat). Ähnlich sieht es in Regel r_2 aus. Hier wird aber der Kreis gelöscht und das Quadrat erhalten. Bei dem Versuch einen Ansatz m von einer Regel in den Arbeitsgraph zu finden, stellt man fest, daß für r_1 ein vollständiger Ansatz (Pfeile $m_{1.0}$) nach G_0 existiert, nicht aber für r_2 (Pfeil $m_{2.0}$). Anders sieht es bei einem Ansatz nach G_1 aus. Nach G_1 können beide Regeln r_1, r_2

einen volländig Ansatz finden (Pfeile $m_{1.1}$, $m_{2.1}$). In Abbildung 4.2 sieht man, daß die Excluderelation vom Arbeitsgraphen G_0 bzw. G_1 abhängt. Obwohl r_1 etwas löscht, das r_2 zum Ansatz braucht, findet r_2 nie einen Ansatz in den Arbeitsgraphen G_0. Diese Situation ist vollkommen unkritisch. Im Gegensatz zu den Matches $m_{[1,2].1}$ nach G_1. Hier können die Regeln r_1, r_2 so ansetzen, daß ein Konflikt verursacht wird.

Definition 18 (*Konflikt*)
Zwei Regeln r_1, r_2 stehen in Konflikt, wenn mindestens eine Bedingung der Excluderelation erfüllt ist. △

Die Attribute gestalten sich allerdings komplizierter, als daß man sie in einem Satz zusammenfassen könnte. Daher wird später noch mal besonders auf sie eingegangen. (4.1.i) und (4.1.ii) zeigen die Bedingungen. Auf den ersten Blick scheint die Bedingung 3 zu fehlen. Sie ist aber in (4.1.i) und (4.1.ii) enthalten. Die Attribute hängen an Attributträgern (s. [Mel98]), die bei Aktionen auf den Attributen gelöscht und erzeugt werden.

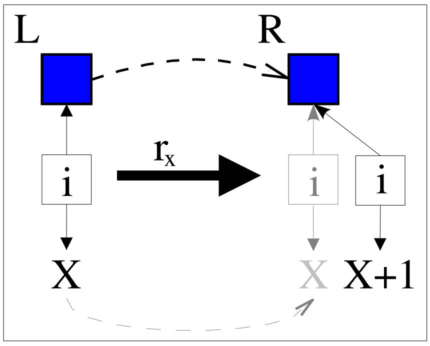

Abbildung 4.3: Regel mit Attributträger

Es gibt im wesentlichen drei Aktionen, die auf Attributen ausgeführt werden können: Erzeugen, Löschen und Ändern.

Die Aktion “Ändern” ist dabei ein gewisser Sonderfall, da diese Aktion als das Löschen und Erzeugen eines Attributträgers aufgefaßt werden kann. Abbildung 4.3 zeigt eine Änderung der Attribute. Da der Attributträger i als Knoten aufgefaßt und gelöscht wird, greift hier (4.1.i). Für (4.1.ii) gilt, daß in einer Regel neue Attributträger erzeugt werden können. Diese Attributträger können allerdings nicht ohne ihre dazugehörige Struktur existieren, so daß (4.1.ii) für die Attribute nicht isoliert auftreten wird. Allerdings muß die Bedingung für die Attribute einerseits eingeschränkt und andererseits erweitert werden. Die Bedingung muß auf Konstanten eingeschränkt werden. Ändert sich ein Attribut in L_1, so hat es nur Konsequenzen, wenn in L_2 eine Konstante steht. Verändert Regel r_1 ein Attribut, so kann Regel r_2 an diesem Attribut nicht mehr ansetzen. Nehmen wir an, L_1 enthält eine Variable und L_2 eine Konstante und r_2 findet einen Ansatz in G. So spielt es für r_1 keine Rolle, auf was sie nach Anwendung von r_2 ansetzt. Dreht man die Reihenfolge um, so daß r_1 zuerst angewendet wird, so kann r_2 nicht mehr an der Stelle ansetzen, da sich das Attribut geändert hat. Zwei Regeln sind unabhängig, wenn notwendigerweise gilt:

$$\begin{tikzcd} G \arrow[r, "{(r_1,m_1)}"] \arrow[d, "{(r_2,m_2)}"'] \arrow[dr, phantom, "="] & H_1 \arrow[d, "{(r_2,m'_2)}"] \\ H_2 \arrow[r, "{(r_1,m'_1)}"'] & H \end{tikzcd}$$

Definition 19 (*Konfluenz unabhängiger Regeln*)
Gegeben seien zwei Regeln r_1, r_2, die über die Ansätze m_1, m_2 auf einen gemeinsamen Arbeitsgraphen G anwendbar sind. Die zwei Regelanwendungen $G \overset{(r_1,m_1)}{\longrightarrow} H_1$ und $G \overset{(r_2,m_2)}{\longrightarrow} H_2$ sind unabhängig, wenn es zwei Regelanwendungen $H_1 \overset{(r_2,m_2')}{\longrightarrow} H$ und $H_2 \overset{(r_1,m_1')}{\longrightarrow} H$ gibt. Das heißt, (r_1, m_1) und (r_2, m_2) sind lokal konfluent. △

Ein weiterer Punkt betrifft die Erweiterung der dritten Bedingung in Definition 17. Durch die Änderung eines Attributes durch r_1 kann es passieren, daß die Attributbedingung in r_2 verletzt wird. In diesem Fall stehen r_1 und r_2 in Konflikt. Die attributierte Graphtransformation erlaubt es, eine implizite Attributbedingung zu definieren. Tritt in einer Regel in der linken Seite eine Variable mehrfach auf, so muß diese Variable beim Ansatz immer auf Attribute mit dem gleichen Wert matchen. Diese Bedingung muß nicht explizit formuliert werden.

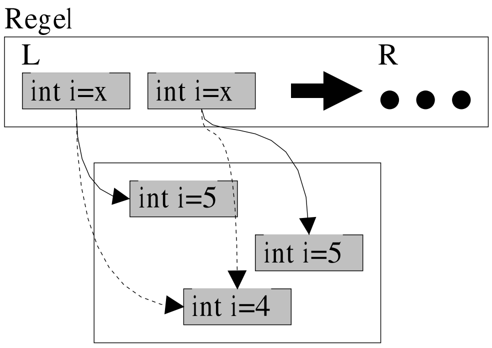

Abbildung 4.4: Ansatz mit gleichen Variablen

Abbildung 4.4 zeigt die Verwendung von zwei gleichen Variablen. Wird ein Ansatz gefunden, so saß ein Knoten der Regel aus Abbildung 4.4 auf einen Knoten mit $i = 5$ ansetzt, so muß der zweite Knoten ebenfalls auf einen Knoten mit $i = 5$ ansetzen. Ein möglicher Ansatz ist durch die durchgezogenen Pfeile dargestellt. Ein weiterer – nicht injektiver – Ansatz ist durch die gestrichelten Pfeile dargestellt.

Definition 20 (*Kritisches Paar*)
Gegeben seien zwei Regeln $r_1 : L_1 \to R_1$ und $r_2 : L_2 \to R_2$, die über die Ansätze $m_1 : L_1 \to G$ und $m_2 : L_2 \to G$ auf einen gemeinsamen Arbeitsgraphen G anwendbar sind. Ein kritisches Paar ist ein Paar abhängiger Regelanwendungen $G \overset{(r_1,m_1)}{\longrightarrow} H_1$ und $G \overset{(r_2,m_2)}{\longrightarrow} H_2$, mit $\exists g \in m_1(L_1) : g \in m_2(L_2)$. △

Die Ansätze kritischer Paare müssen sich also mindestens in einem Graphobjekt überlappen. In Definition 20 steht ein gemeinsamer Arbeitsgraph G zur Verfügung an den Ansätze m_1, m_2 ansetzen können. Für die generische Überprüfung steht jedoch kein Arbeitsgraph zur Verfügung. Es ist aber möglich, alle Arbeitsgraphen zu generieren, an denen zwei Regeln r_1, r_2 gemeinsam ansetzen können.

Definition 21 (*Überlappungsgraph*)
Gegeben seien zwei Regeln $r_1 : L_1 \to R_1$ und $r_2 : L_2 \to R_2$ ohne NACs, die über die

Ansätze $m_1 : L_1 \to G_Ü$ und $m_2 : L_2 \to G_Ü$ auf einen gemeinsamen Arbeitsgraphen $G_Ü$ anwendbar sind. $G_Ü$ ist ein Überlappungsgraph zu r_1 und r_2, genau dann wenn $\exists g \in m_1(L_1) : g \in m_2(L_2)$ und $G_Ü = m_1(L_1) \cup m_2(L_2)$. $\triangle$

Diese Definition der Überlappungsgraphen ist solange ausreichend wie r_2 keine NACs besitzt. Die Exclude-Relation beachtet bei der Überprüfung NACs. Daher muß die Definition des Überlappungsgraphen für Regeln mit NACs angepaßt werden. Ein Beispiel für diese Notwendigkeit befindet sich in Abbildung 4.8 und 4.9.

Definition 22 (*Erweiterter Überlappungsgraph*)
Gegeben seien zwei Regeln $r_1 : L_1 \to R_1$ und $r_2 : L_2 \to R_2$ mit NACs, die über die Ansätze $m_1 : L_1 \to G_Ü$ und $m_2 : L_2 \to G_Ü$ auf einen gemeinsamen Arbeitsgraphen $G_Ü$ anwendbar sind. $G_Ü$ ist ein Überlappungsgraph zu r_1 und r_2, genau dann wenn ein partieller Morphismus $\overline{m_2} : \overline{L_2} \to G_Ü$ existiert und $G_Ü = m_1(L_1) \cup m_2(L_2) \cup \overline{m_2}$. $\triangle$

4.1.1 Layer

Die Berechnung der kritischen Paare für gelayerte Graphgrammatiken enthält einen Vorteil. Da Regeln eines höheren Layer nicht angewendet werden können, solange Regeln eines niedrigeren Layers anwendbar ist, ist es ausreichend, kritische Paare nur innerhalb eines Layers zu berechnen. Diese Vereinfachung bewirkt, daß nicht unbedingt kritische Paare zwischen allen Regeln berechnet werden müssen.

4.2 Design der kritischen Paaranalyse

Das Design der kritischen Paaranalyse hat zwei Aspekte. Zum einen ist die oben beschriebene Theorie zu modellieren. Andererseits soll auch das Ergebnis der Berechnung der kritischen Paare visualisiert werden, so daß eine grafische Benutzeroberfläche nötig ist.

Auf den ersten Blick erscheint ein Modell für die kritischen Paare trivial zu sein, denn im wesentlichen sind kritische Paare nur die Elemente eine Relation zwischen den Regeln und Ansätze einer Graphgrammatik. So wird anscheinend nur eine Klasse benötigt. Nach [GHJV95] bietet es sich hier an, das Strategiemuster anzuwenden und den Algorithmus auszugliedern. So ergeben sich zwei Klassen: CriticalPair und PairContainer. Abbildung 4.5 zeigt die Klassenstruktur der kritischen Paaranalyse. PairContainer entspricht der Menge der kritischen Paare. Da es für den Container keinen Unterschied macht, welche Art von Relation er enthält, verfügt PairContainer nicht nur über die Relation der kritischen Paare. Zusätzlich hält der Container noch alle konfliktfreien Regeln. Die Klasse CriticalPair enthält nur den Algorithmus, der die kritischen Paare berechnet.

Da es denkbar ist, noch mehr Arten von kritischen Paaren zur Verfügung zu stellen, wird bei Fehlbenutzung der Klassen CriticalPair und PairContainer an gegebener Stelle eine InvalidAlgorithmException geworfen. Diese Ausnahme tritt immer dann auf, wenn versucht wird, eine Abfrage abzusetzen, für die kein Algorithmus existiert.

4.2.1 Algorithmus zur kritischen Paaranalyse

Da aus PairContainer der Algorithmus separiert wurde, deligiert PairContainer alle Anfragen an CriticalPair. Um eine Antwort zu beschleunigen, wird das Ergebnis der Berechnung

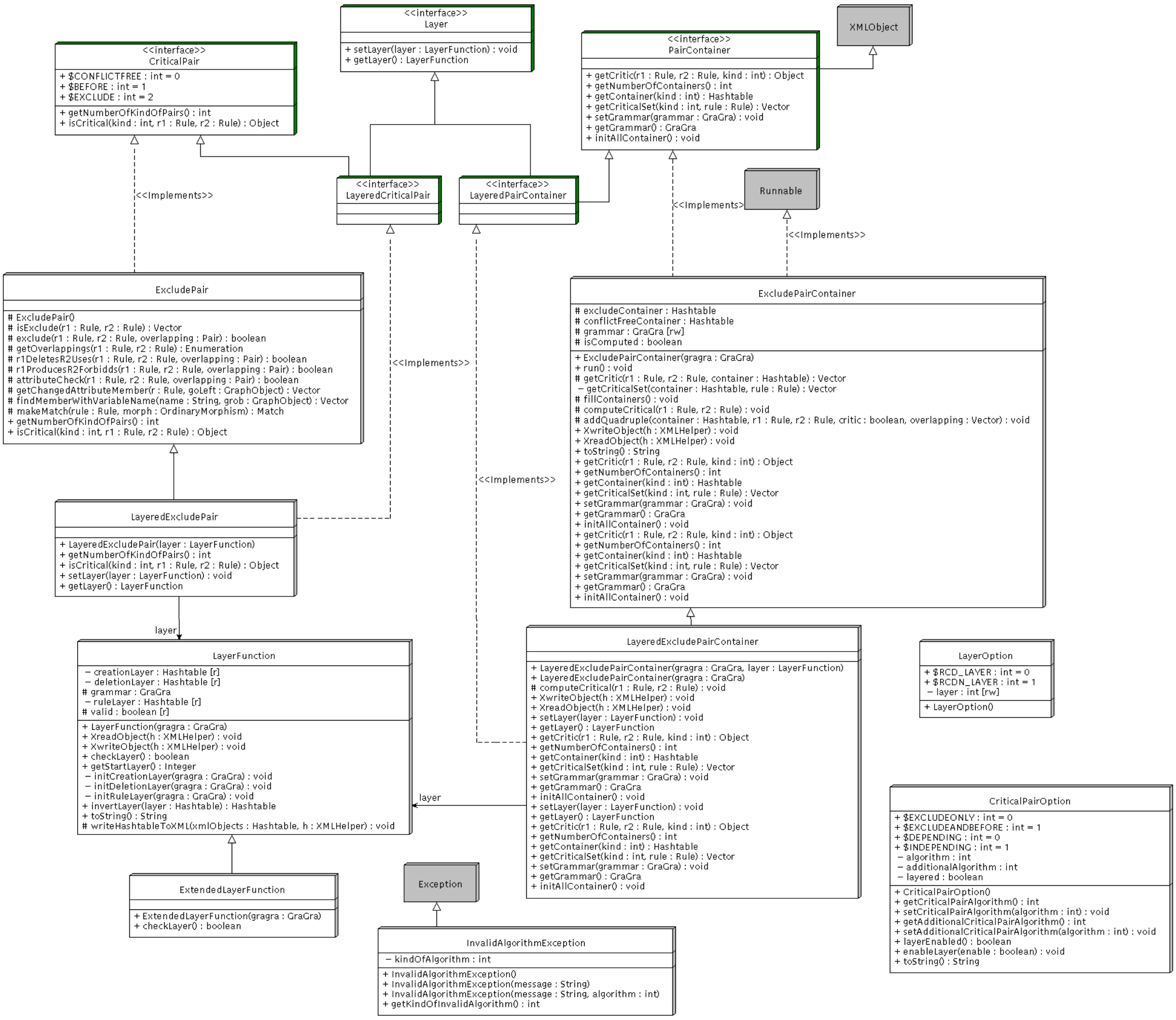

Abbildung 4.5: Kritische Paarklassen, Ausschnitt aus `agg.parser`

in PairContainer gehalten. Die Berechnung muß für jede Regel erfolgen. Dieses Szenario spiegelt auch Abbildung 4.6 wider.

In Kapitel 4.1 wurde definiert, daß die Excluderelation in drei Teile zerfällt. Es gibt die Bedingungen: r_1 löscht Graphobjekte, die r_2 zum Ansatz braucht, bzw. r_1 erzeugt Graphobjekte, die r_2 durch eine negative Anwendungsbedingung (NAC) verbietet. Die letzte Bedingung betraf die Attribute. Jede Bedingung wird damit durch eine eigenständige Berechnung überprüft. Falls eine Bedingung erfüllt wird, so müssen die anderen nicht mehr überprüft werden, da ein Kriterium ausreicht, damit zwei Regeln abhängig sind. Abbildung 4.7 zeigt den Ablauf der kritischen Paarberechnung.

Der erste Fall (r_1 löscht Graphobjekte, die r_2 zum Ansatz braucht) setzt voraus, daß die beiden linken Regelseiten an dem selben Graphen G ansetzen, wie in Abbildung 4.1 dargestellt. Dieser Arbeitsgraph steht allerdings bei der Berechnung nicht zur Verfügung. Ein Trick an dieser Stelle ist alle Möglichkeiten zu generieren, an denen die Regeln ansetzen können. Diese Generierung ist relativ einfach, denn es müssen nur alle möglichen Überlappungen der linken Regelseiten gefunden werden. Anhand dieser Überlappungs-

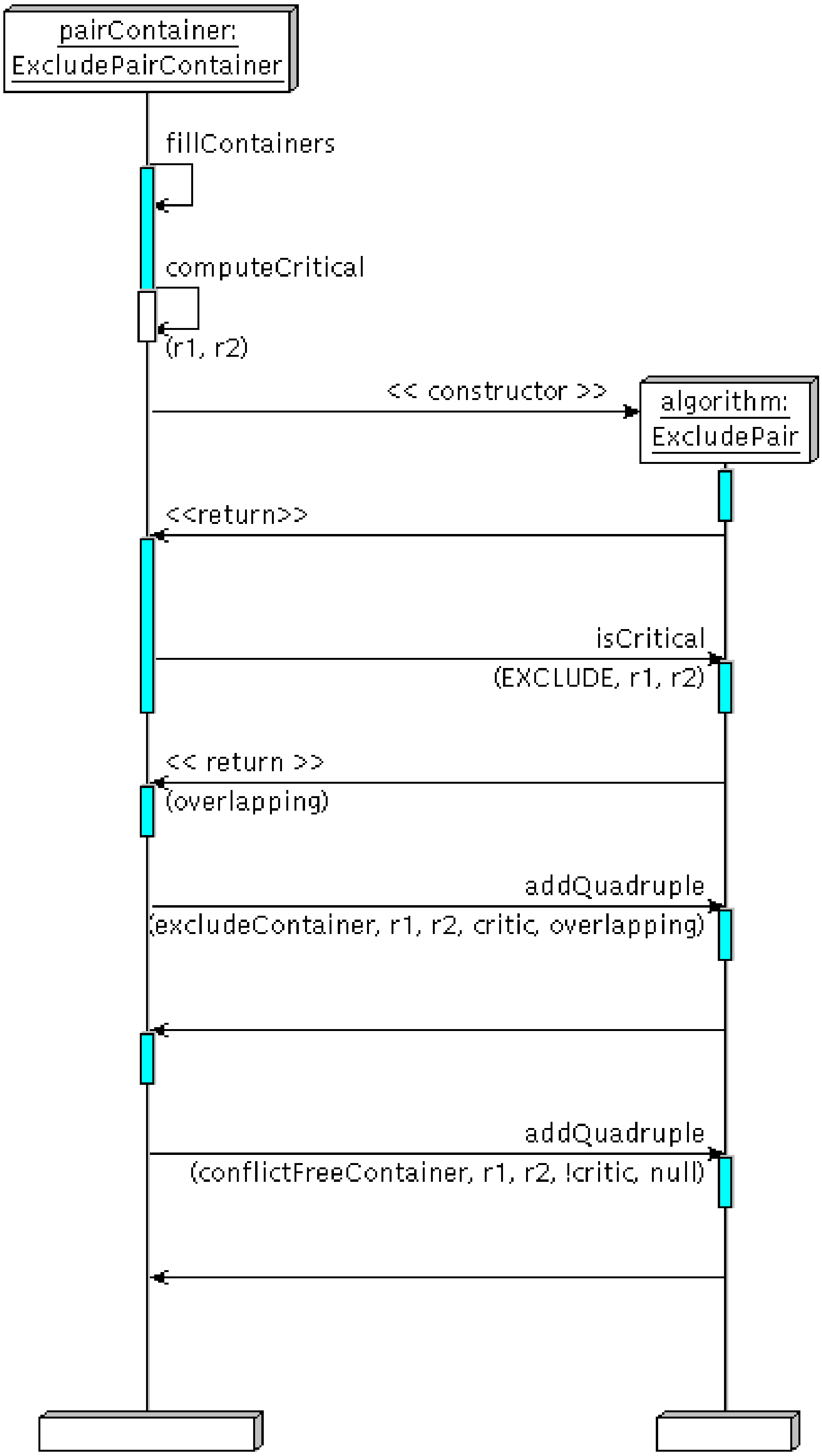

Abbildung 4.6: Sequenzdiagramm für die Generierung von kritischen Paaren

graphen läßt sich entscheiden, ob r_1 in Konflikt zu r_2 steht. Ein kleines Problem stellt die Berechnung der Überlappungsgraphen von Regeln mit NAC dar. Abbildung 4.8 zeigt den einen Überlappungsgraphen für die beiden Regeln. Wie die Bedingung es voraussetzt, wird H berechnet und ein Ansatz für Regel 2 gesucht. Da kein totaler Morphismus von

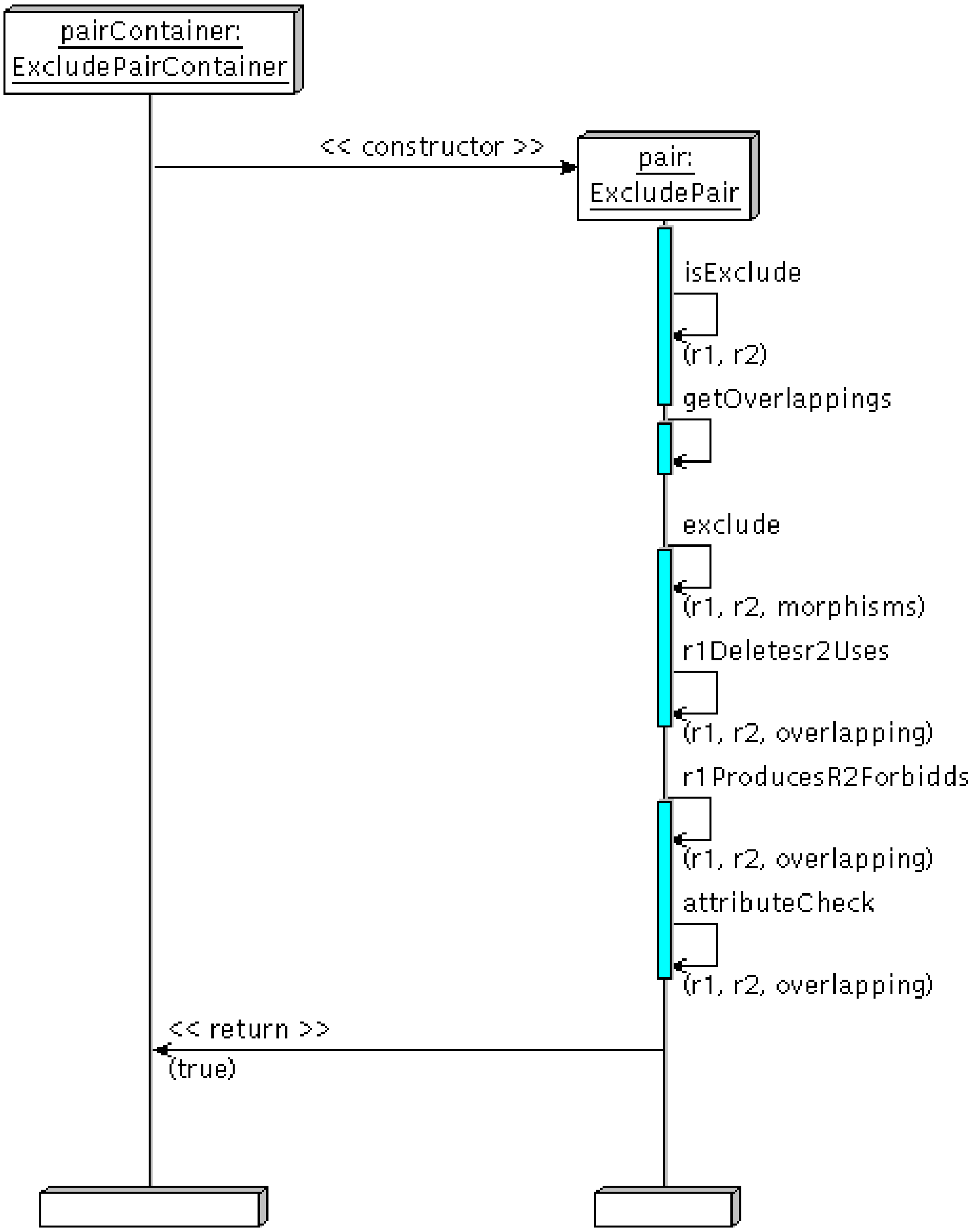

Abbildung 4.7: Sequenzdiagramm für die Berechnung eines kritischen Paares

der NAC in H gefunden werden kann, scheinen diese Regeln nicht in Konflikt zu stehen. Würde ein Teil der NAC schon im späteren Arbeitsgraphen vorhanden sein und der andere Teil durch eine Regel hinzugefügt werden, so fällt dieser Fall nicht in die Exclude-Beziehung. Abhilfe schafft die Änderung der Berechnung der Überlappungsgraphen. Die NAC wird um die Elemente aus der linken Regelseite erweitert, die noch nicht in der NAC vorkommen. Von diesem Graphen werden dann alle echten Teilgraphen gebildet, die einen totalen Morphismus von der linken Regelseite in den Teilgraphen bilden können. Abbildung 4.8 müßte dann so erweitert werden wie in Abbildung 4.9 dargestellt.

So läßt sich mit diesem erweiterten Überlappungsgraphen auch die zweite Bedingung (r_1 erzeugt Graphobjekte, die r_2 durch die negative Anwendungsbedingung $\overline{L_2}$ verbietet) überprüfen.

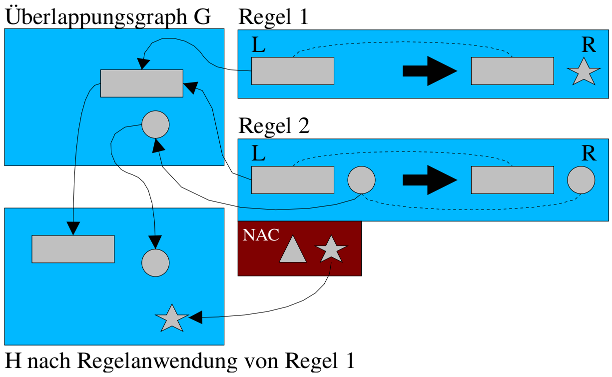

Abbildung 4.8: Unvollständig erzeugter Überlappungsgraph

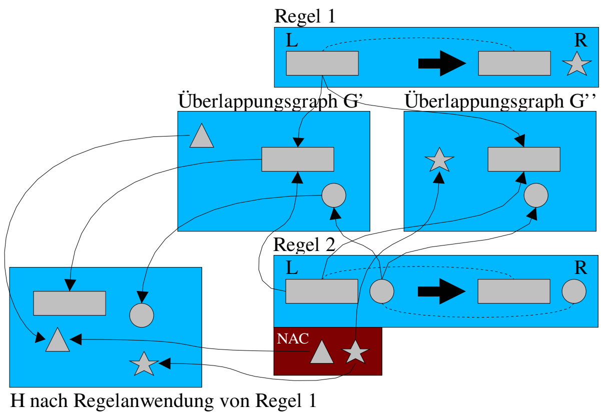

Abbildung 4.9: Erweiterter Überlappungsgraph

Die letzte Bedingung (r_1 ändert Attribute, die r_2 zum Ansatz benötigt) berechnet sich ein wenig anders, da die Attribute im wesentlichen eigenständig sind, allerdings werden hier auch wieder Überlappungsgraphen benötigt. Aus dem Regelmorphismus ist leicht zu erfahren, welche Attribute sich ändern.

In Abbildung 4.10 wird gezeigt, wie man mit Hilfe der Morphismen in den Überlappungsgraphen die Abhängigkeit zwischen Attributen finden kann. Über den Morphismus[1] in den Überlappungsgraphen erhält man auch dort die Attribute.

Der Morphismus von der linken Regelseite der zweiten Regel kann rückwärts gegangen werden (gestrichelte Pfeile in Abbildung 4.10). Man erhält so alle Graphobjekte, die auf das Graphobjekt im Überlappungsgraphen matchen. So können nun auch die Attribute überprüft werden.

Für jeden Überlappungsgraph muß überprüft werden, ob zwei Regeln kritisch sind oder nicht. Nicht jeder Überlappungsgraph liefert eine kritische Situation.

In Abbildung 4.11 wird ein möglicher Überlappungsgraph zweier Regeln gezeigt. Dort ist G_0 zwar ein gültiger Überlappungsgraph, aber mit diesem Ansatz tritt kein kritischer Fall ein. Da der schraffierte Knoten nicht von r_1 gelöscht wird, kann r_2 diesen auch ohne Probleme für den Ansatz benutzen. So werden alle kritischen Fälle gefiltert. Diese kriti-

[1]dicke Pfeile in den Überlappungsgraphen in Abbildung 4.10

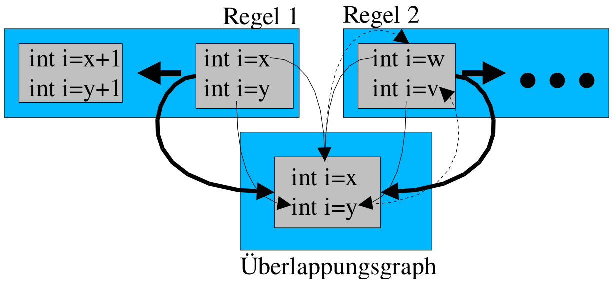

Abbildung 4.10: Ansatz mit Attributen in einen Überlappungsgraphen

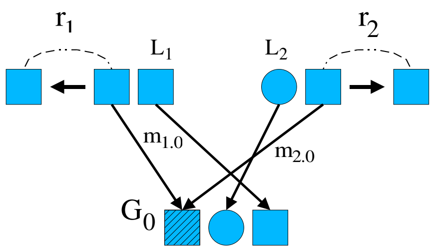

Abbildung 4.11: Ein möglicher Überlappungsgraph

schen Fälle und zwischen welchen Regeln sie auftreten, werden in der Exclude-Relation gehalten.

4.2.2 Layer

Um gelayerte kritische Paare zu berechnen, wissen wir, daß nur Regeln im selben Layer überprüft werden müssen. So kann vorab eine Überprüfung erfolgen, die false zurück gibt, falls zwei Regeln nicht den selben Layer haben. Anderenfalls kann die herkömmliche Methode benutzt werden.

Da der Algorithmus und der Container von einander getrennt sind, sind hier zwei weitere Klassen nötig, nämlich LayeredExcludePair für den Algorithmus und LayeredExcludePairContainer. Sie sind entsprechende Unterklassen der nicht gelayerten Versionen.

4.2.3 Graphische Benutzeroberfläche

Um das Ergebnis der kritischen Paarberechnung zu visualisieren, ist noch eine graphische Benutzeroberfläche nötig. Die kritischen Paare stellen allerdings andere Anforderungen an die Oberfläche als das klassische AGG erfüllen kann. Zum einen ist es nötig zwei Regeln auszuwählen, in AGG ist nur die Auswahl einer Regel möglich. Zum anderen ist die partielle Anzeige der ausgewählten Regeln erforderlich. So ist man bei den kritischen Paaren nur an der linken Regelseite interessiert. Die letzte Anforderung betrifft die Überlappungsgraphen. Für ein kritisches Paar können mehrere Überlappungsgraphen existieren. Diese sollen dann auch gleichzeitig sichtbar sein, um sie vergleichen zu können.

Ein schematischer Entwurf ist in Abbildung 4.12 dargestellt. Abbildung 4.13 zeigt das Klassenmodell für die Oberfläche für kritische Paare, die in Abbildung 4.13 vorgestellt wurde. Dieses Modell ist sehr modular aufgebaut und ermöglicht so einen leichten Aus-

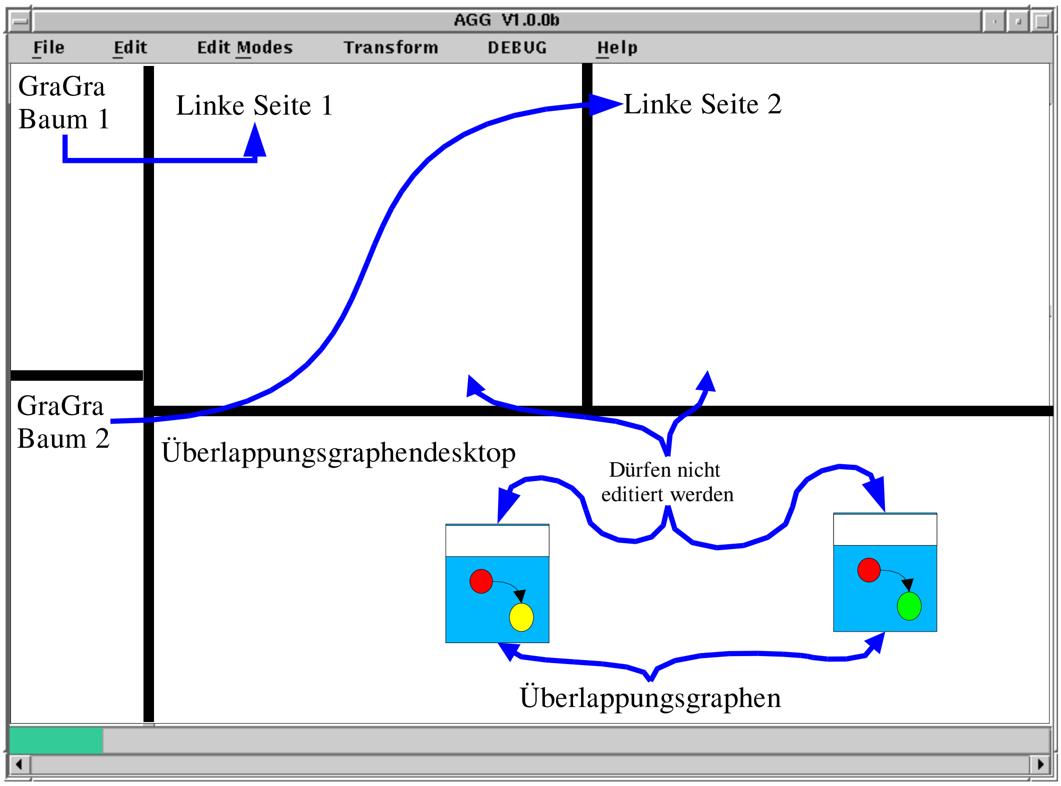

Abbildung 4.12: Oberfläche für kritische Paare

tauch der Komponenten. Die oberste Klasse, die die klassische AGG-Oberfläche ersetzt, stellt die Klasse CriticalPairAnalysisGUI dar. Diese Klasse enthält den weiteren Aufbau der Benutzeroberfläche. Für die Auswahl zweier Regeln stehen zwei Objekte der Klasse RuleTree zur Verfügung. Als nächstes ist eine Anzeige der linken Regelseiten interessant. Diese Umsetzung findet sich in der Klasse GraphBrowser. Die letzte Anforderung, die Überlappungsgraphen parallel zu betrachten, wird in der Klasse GraphDesktop realisiert. Eine detailierte Beschreibung der Methoden befindet sich in der API.

4.3 Implementierung

Wie auch im Design soll der Schwerpunkt auf den kritischen Paaren liegen und nicht auf der graphischen Benutzeroberfläche.

Die Implementierung der PairContainer übernimmt die Klasse ExcludePairContainer. Diese Klasse implementiert zusätzlich das Interface Runnable. Dieses Interface ermöglicht es, Prozesse parallel ausführen zu lassen. Diese Klasse besteht nur aus seinen zwei Containern und der entsprechenden Graphgrammatik, zu der die kritischen Paare berechnet werden. In dieser Version bestehen die Container aus Hashtable. Diese Wahl begründet sich dadurch, daß zu jeder Regel R_x die Informationen[2] gehalten werden müssen, mit welcher anderen Regel R_y sie in Konflikt steht. Dabei steht noch ein boolscher Wert zur Verfügung, der angibt, ob sich Regel R_x und Regel R_y in der gefragten Relation befinden.

[2] $R_x \to Hashtable[R_y \to Pair(Boolean, Vector[Graph_{overlappings}, ...]), ...]$

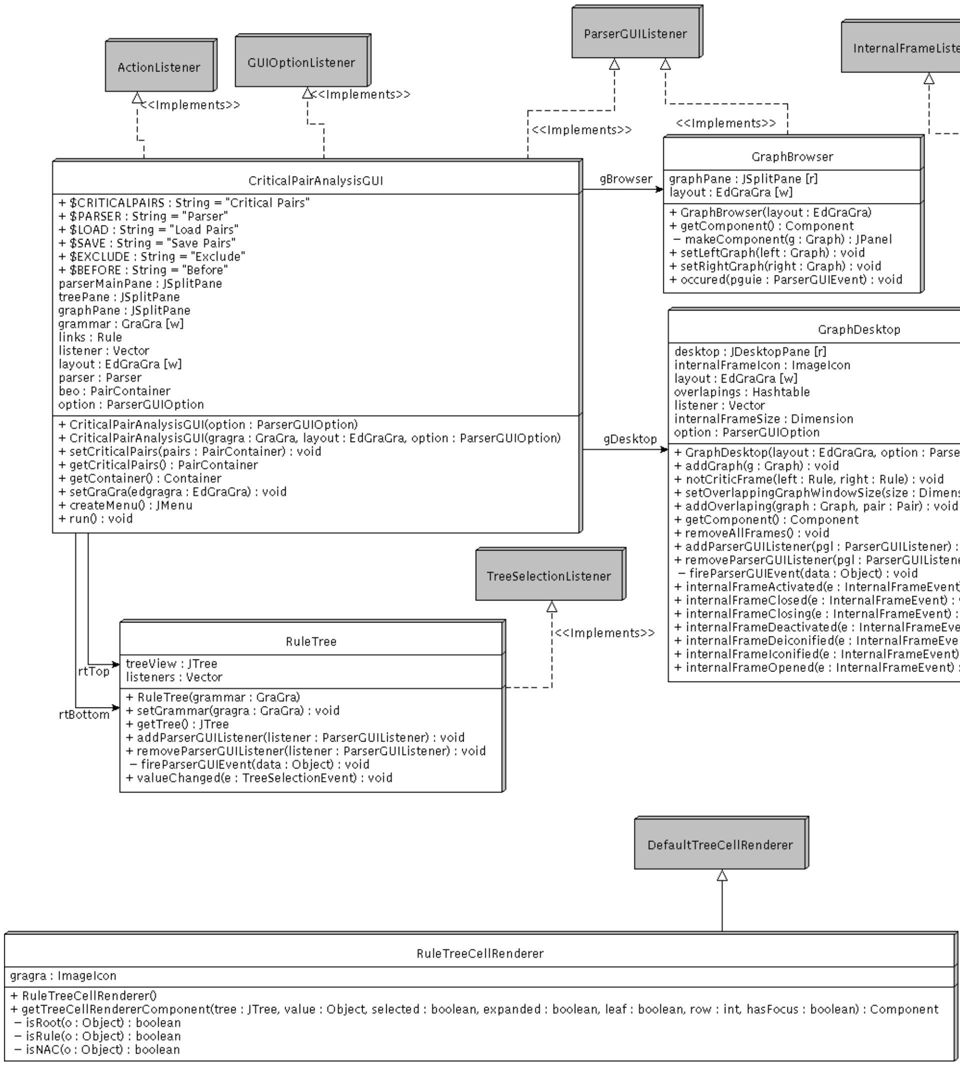

Abbildung 4.13: Benutzeroberfläche aus `agg.gui.parser`

Beispiel: Seien R_1, R_2 Regeln einer Graphgrammatik und R_1, R_2 bilden ein kritisches Paar. Dann sind beide Regeln in beiden Container. Im excludeContainer ist der boolsche Wert auf true im conflictFreeContainer auf false gesetzt (s. Abbildung 4.5). Jeder Container bezeichnet dabei die Art der Beziehung. Die Container können zusätzlich geladen und gespeichert werden. Dadurch kann eine vorhandene Berechnung für die nächste Arbeitssitzung beibehalten werden. So verlangt auch die Erzeugung eines Objekts nach der entsprechenden Graphgrammatik. Der Konstruktor benutzt zusätzlich die Methode initAllContainer, um dazugehörige Containerobjekte zu erzeugen und zu initialisieren.

Eine besondere Fähigkeit des ExcludePairContainer ist es, seine Container parallel zu füllen also zu berechnen. Wird ein Prozess in Java parallel ausgeführt, so wird automatisch die Methode public void run() aufgerufen. Hier macht die Methode nichts weiter als private void fillContainers() aufzurufen. Das heißt, der Benutzer kann schon andere Aktionen ausführen, während die kritischen Paare noch berechnet werden. Da diese Berechnung einige Zeit in Anspruch nehmen kann, gibt es die Möglichkeit ein kritisches Paar direkt zu berechnen. Das heißt, werden zwei Regeln ausgewählt, für die noch kein kritisches Paar berechnet wurde, so findet diese Berechnung ad hoc statt. Um diese Berechnung nicht wiederholen zu müssen, wird das Ergebnis ebenfalls in den entsprechenden Container eingetragen. Ein größeren Überblick verschafft Abbildung 4.14. In dem Sequenzdiagramm ist der Aufruf vom AGG her betrachtet.

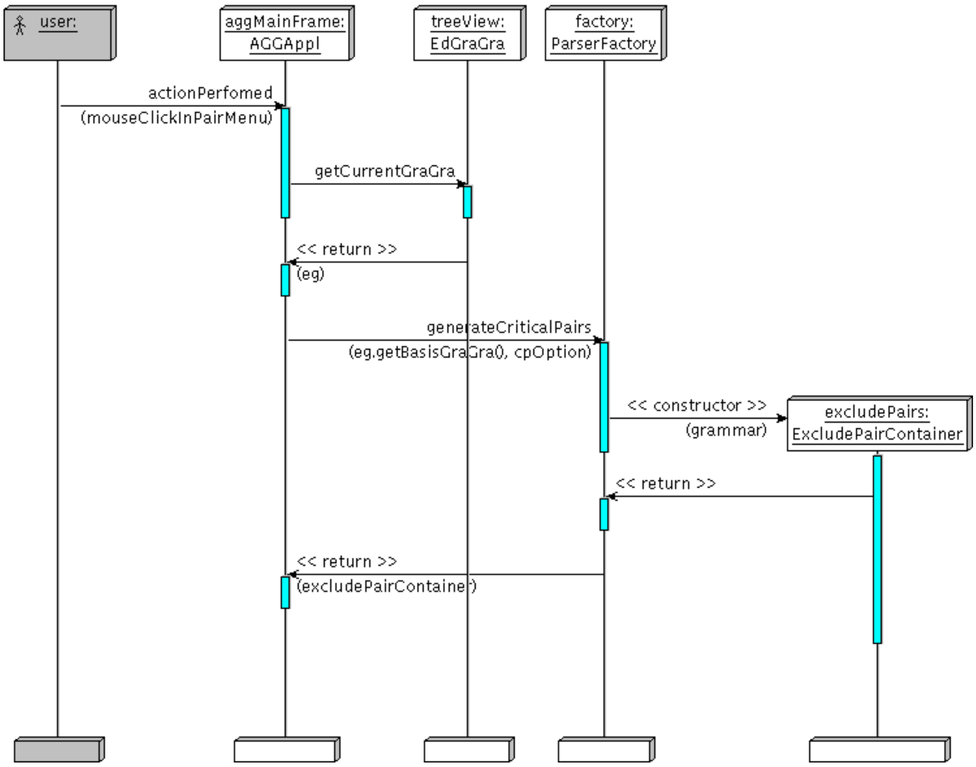

Abbildung 4.14: Sequenzdiagramm für den Aufruf und Füllung des kritischen Paarcontainers

Wie schon im Design in Kapitel 4.2 erwähnt, findet die Berechnung der kritischen Paare in einer gesonderten Klasse statt, der Klasse CriticalPair. Die Implementierung befindet sich in der Klasse ExcludePair.

4.3.1 Algorithmus zur kritischen Paaranalyse

Wie in Kapitel 4.2 gesehen, ist die Berechnung der Überlappungsgraphen ein zentraler Bestandteil. Wie genau ein Überlappungsgraph berechnet wird, soll hier nicht weiter erläutert werden, da dieser Service von der Klasse agg.xt_basis.Graph genutzt werden kann. Schwierigkeiten machte hier die Totalisierung des Morphismus $\overline{l_2} : L_2 \to \overline{L_2}$. Um einen Überlappungsgraphen zu berechnen, sind zwei Graphen nötig, in diesem Fall L_1 und die angereicherten NACs. Um die NACs anzureichern, können sie nicht selbst angereichert werden, da sonst die NACs verändert werden. Kopiert man hier nur einfach die Graphen der NACs, so verliert man die Information von welcher Quelle die Kopie stammt. Also muß außerdem ein Isomorphismus zwischen den beiden Graphen gehalten werden. Man erhält im wesentlichen die Situation, die in Abbildung 4.15 dargestellt ist.

Wenn man mit dieser angereicherten NAC $\overline{L'_2}$ und L_1 Überlappungsgraphen bildet, so erhält man natürlich Graphen, die die NAC enthalten. Diese Problematik ist auch schon

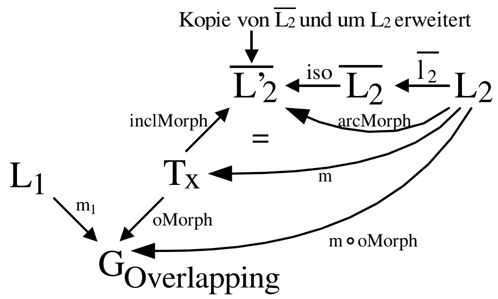

Abbildung 4.15: Bildung eines Überlappungsgraphen von einer Regel mit NAC

in Kapitel 4.2 beschrieben. Um Teilgraphen zu berechnen, ist es möglich, die Dienste aus agg.xt_basis.Graph zu nutzen. Um ein wenig Geschwindigkeit aus dieser Berechnung herauszuholen, ist es nicht nötig, alle echten Teilgraphen zu berechnen. Da für das spätere Parsieren eine Einbettung der linken Regelseite in den Überlappungsgraphen benötigt wird, müssen nur die Teilgraphen berechnet werden, die mindestens so groß wie die linke Regelseite, aber höchstens um ein Graphobjekt kleiner sind als die erweiterte NAC $\overline{L'_2}$. Bei dieser Berechnung werden sämtliche Graphen entsprechender Größe erstellt. Die verschiedenen Teilgraphen T_x müssen nur noch gefiltert werden, damit immer eine Einbettung der linken Regelseite in den Teilgraphen gefunden wird. Für das spätere Parsieren ist es noch interessant, einen Morphismus von den linken Regelseiten in die Überlappungsgraphen zu erhalten. Für L_1 existiert der Morphismus m_1. Leider existiert für L_2 kein direkter Morphismus in die Überlappungsgraphen. Da nur ein Morphismus inclMoph, wie in Abbildung 4.15 zu sehen ist, von T_x nach $\overline{L'_2}$ existiert, ist ein einfaches Komponieren der Morphismen arcMorph, inclMorph, oMorph nicht möglich. Es läßt sich aber ein Morphismus m finden, so daß gilt:

$$L_2 \xrightarrow{m} T_x \xrightarrow{inclMorph} \overline{L'_2} = L_2 \xrightarrow{arcMorph} \overline{L'_2}$$

Der Morphismus m kann dann mit oMorph komponiert werden, um die nötige Verbindung zu bekommen. An dieser Stelle sei angemerkt, daß bei Regeln ohne NAC dieser Auswand komplett entfällt. Eine kleine Überprüfung steigert so die Effizienz.

Zum Aufwand dieser Implementierung muß aber gesagt werden, daß durch den Vergleich aller Regeln untereinander ein Aufwand $O(n^2)$ existiert. Hinzu kommt allerdings der Aufwand für die Regelanwendung während der Berechnung der Überlappungsgraphen und deren Ableitungen beim Prüfen der Excluderelation. Durch die exponientelle Ansatzsuche verschlechtert sich der Gesamtaufwand. Weitere Details sind in [Rud97] nachzulesen.

4.3.2 Graphische Benutzeroberfläche

Das Design der graphischen Oberfläche konnte ohne weiteres umgesetzt werden, so wie es in Kapitel 4.2.3 vorgestellt wurde. Da diese Oberfläche "nur" eine Visualisierung der kritischen Paare ist, wird auch keine Funktionalität zum Editieren bereitgestellt. Nur das Layout kann geändert werden. Hier können die Graphobjekte, wie man es vom klassischen AGG gewöhnt ist, verschoben und gezogen werden.

Die Oberfläche wird durch ein Klick im Menü Parser aktiviert. Der Eintrag Critical Pairs zeigt durch ein Checkbutton, ob die kritische Paaranalyse aktiviert ist.

Kapitel 5

Der Parser

In den vorherigen Kapiteln wurden die Grundlagen wie Graphtransformation sowie Optimierung mittels kritischer Paaranalyse vorgestellt. Betrachtet man also wieder den Designer der Verkehrsschilder, so soll nun ein visueller Satz erkannt werden. Diese Thematik ist auch allgemein unter dem Namen "Wortproblem" bekannt. Das Wortproblem lösen heißt festzustellen, ob für eine formale textuelle Sprache ein beliebiges Wort über dem gegebenen Alphabet zur Sprache gehört oder nicht. Bezogen auf Graphgrammatiken bedeutet das Wortproblem festzustellen, ob ein Graph zu einer gegebenen Graphgrammatik gehört oder nicht. Die Prüfung, ob ein Wort zu einer Sprache gehört, heißt, eine Sequenz von Regelanwendungen zu finden, die das Wort erzeugen. Eine andere Möglichkeit besteht auch darin, in einer Grammatik die Regeln rückwärts anzuwenden und das Wort auf das Startsymbol zu reduzieren. Diese Herangehensweise ist bei Algorithmen, die sich mit dem Wortproblem beschäftigen, weit verbreitet. So arbeiten die meisten Parser für Programmiersprachen auf diese Weise.

Es ist also nötig einen Algorithmus zu finden, der Schritt für Schritt eine Sequenz aufbaut, die ein Wort erkennt. Um ein Problem Schritt für Schritt zu lösen, fällt jedem Entwickler das Prinzip des Backtracking ein.

Ein Backtracking Algorithmus versucht, ein Problem Schritt für Schritt zu lösen, während der Algorithmus alle Schritte aufzeichnet, und eventuell rückgängig macht. Das Programm probiert dabei alle Schritte aus, bis es merkt, daß ein Schritt nicht zur Lösung führen kann, oder zur Lösung führt. Ist das Problem nicht gelöst, geht es anhand seiner Aufzeichnungen zurück zu einem Punkt, an dem es noch Lösungsmöglichkeiten gibt, die es noch nicht ausprobiert hat. Wird es nicht unterbrochen, so probiert es alle Möglichkeiten aus, und findet so auch alle möglichen Lösungen.

Für Graphen und Graphgrammatiken muß also jede Regelanwendung aufgezeichnet werden, bis der Graph auf den Startgraph reduziert werden konnte.

Definition 23 (*Stopgraph*)
Der Startgraph einer Graphgrammatik wird beim Parsieren Stopgraph genannt. △

5.1 Algorithmus des Parsers

Der Parser könnte nach den obigen Überlegungen relativ schnell implementiert werden. Allerdings lassen sich noch ein paar Optimierungen einbringen. In Kapitel 4 wurde gezeigt, daß es für die Regelauswahl zwei Arten von Regeln gibt: die konfliktfreien und die unabhängigen. Für beide Arten zusammen müssen nun drei Möglichkeiten unterschieden werden:

1. Es findet sich ein Ansatz für eine konfliktfreie Regel. Sie wird angewendet.

2. Es findet sich kein Ansatz für eine konfliktfreie Regel. Dann muß ein Ansatz für eine abhängige Regel so gefunden werden, daß der Ansatz gerade nicht kritisch ist. Die Regel kann dann ohne weiteres angewendet werden.

3. Es kann nur ein konfliktbehafteter Ansatz für eine abhängige Regel gefunden werden. Diese Situation kann Backtracking auslösen. Daher müssen alle wichtigen Daten protokolliert werden. Dann kann die Regel angewendet werden.

Ein optimierter Parser läuft nun in mehreren Schritten ab: Erstens müssen Regelanwendungen, die mit keiner anderen Regel in Konflikt stehen, nicht protokolliert werden. Zweitens, da es leichter ist in einem Graphene mit wenig Graphobjekten einen Ansatz einer Regel zu finden als in einem großen Graphen, sollten diese konfliktfreien Regeln so früh und lange wie möglich angewendet werden, um den Graphen so stark wie möglich zu reduzieren. Der Parser versucht also zuerst, konfliktfreie Regeln anzuwenden.

Aus Kapitel 4 erhält man außerdem auch die Menge der abhängigen Regeln. Sollte also keine konfliktfreie Regel mehr anwendbar sein, so muß ein Ansatz für eine dieser Regeln gesucht werden. Hier profitiert der Parser wieder von der Vorarbeit in der kritischen Paaranalyse.

Gegeben seien zwei Regeln r_1, r_2, und r_1 stehe mit r_2 in Konflikt. Der erste Schritt ist es, einen Ansatz für die Regel r_1 zu finden. Um in einem Konfliktfall nicht immer wieder den gleichen Ansatz zu benutzen, muß jeder gefundene Ansatz in einer Liste überprüft werden, ob dieser Ansatz schon einmal ausprobiert wurde. Ist ein Ansatz für r_1 gefunden, so gibt es zwei Möglichkeiten: r_2 findet auch einen Ansatz oder nicht.

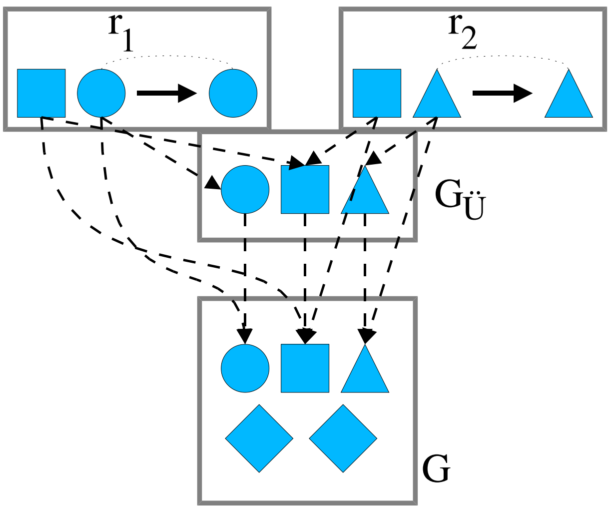

Abbildung 5.1: Kritischer Ansatz einer Regel

In Abbildung 5.1 wird der Fall gezeigt, daß beide Regeln einen Ansatz finden. Für die beiden Regeln r_1, r_2 in Abbildung 5.1 und Abbildung 5.2 gibt es genau nur einen interessanten Überlappungsgraphen $G_{\text{Ü}}$. Es ist genau der, wenn sich die beiden Quadrate

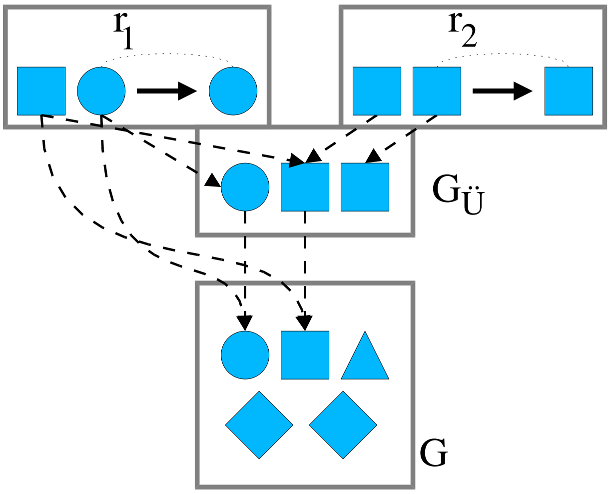

Abbildung 5.2: Unkritischer Ansatz einer Regel

überlappen. Für die Regel r_1 und den Überlappungsgraphen $G_{Ü}$ gilt also: r_1 findet einen Ansatz in den Arbeitsgraphen G, und es gibt eine Einbettung von $G_{Ü}$ in G. Diese beiden Bedingungen lassen sich relativ einfach prüfen. Die erste Bedingung ist trivial, es muß

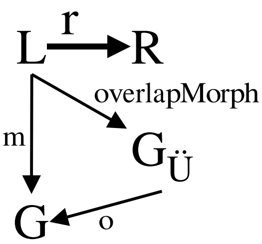

Abbildung 5.3: Konzept der Überprüfung eines kritischen Regelansatzes

nur ein Ansatz für die Regel gefunden werden. Die Prüfung der zweiten Bedingung erklärt sich leichter durch Abbildung 5.3. Der Morphismus m ist durch die erste Bedingung gegeben. Genauso steht auch der Morphismus overlapMorph durch die Generierung der Überlappungsgraphen fest. Es muß nun der Morphismus o gefunden werden, so daß gilt:

$$L \xrightarrow{m} G = L \xrightarrow{overlapMorph} G_{Ü} \xrightarrow{o} G$$

Sind nach dieser Prüfung die Morphismen m, o gefunden, steht fest, daß ein Regelansatz gefunden wurde, der Backtracking auslösen könnte. Ein besserer Regelansatz wäre ein Ansatz wie in Abbildung 5.2. Hier findet sich keine Einbettung des Überlappungsgraphen in den Arbeitsgraphen. An dieser Stelle wird deutlich, daß eine Regel, die mit einer anderen in Konflikt steht, in einem bestimmten Arbeitsgraph keinen Konflikt erzeugen muß. Nur diese Konflikte können aber Backtracking auslösen. Für den Algorithmus heißt das, Konflikte zu vermeiden. Wird also ein Ansatz mit einem Konflikt wie in Abbildung 5.1 gefunden, so ist das ein ungünstiger aber möglicher Ansatz. Dieser Ansatz kann angewendet werden, auch wenn er Backtracking verursacht. Der Ansatz wird gespeichert für den

Fall, daß kein besserer Ansatz gefunden wird. Es muß für alle anderen Regeln sichergestellt werden, daß kein konfliktfreier Ansatz wie in Abbildung 5.1 existiert. Sollte doch ein konfliktfreienrAnsatz gefunden werden, so kann ohne weiteres die Regel angewendet und der konfliktbehaftete Ansatz gelöscht werden.

Wenn nun doch ein konfliktbehafteter Ansatz angewendet werden soll, so müssen die momentanen Daten aufgezeichnet werden. Im wesentlichen sind das zwei Daten. Der erste wesentliche Teil ist der zu parsierende Graph. Im Falle eines Rückschritts muß natürlich der aktuelle Graph wieder hergestellt werden. Zum anderen muß die Liste der angewendeten konfliktbehafteten Ansätze um den aktuellen Ansatz erweitert und mit aufgezeichnet werden. Aufzeichnen heißt in einem Backtrackalgorithmus im wesentlichen, daß die Daten in einem Stack gespeichert werden. Diese Technik soll hier auch verwendet werden.

Nach jeder Regelanwendung muß geprüft werden, ob der Arbeitsgraph isomorph zu dem Stopgraphen ist. In dem Fall wurde der Graph erkannt und gehört zu der Sprache, die die Grammatik angibt. Ansonsten muß wieder mit der Ansatzsuche fortgefahren werden.

Läßt sich kein Ansatz mehr finden, so ist es nötig, das Backtracking einzuleiten. Dazu ist es wichtig festzustellen, ob Daten wieder hergestellt werden können. Sollte der Stack leer sein, und es können die Daten nicht hergestellt werden, so ist der Graph nicht Teil der Sprache, die die Grammatik angibt. Es muß eine Fehlermeldung generiert werden. Andererseits besteht das Restaurieren der Daten lediglich darin, den obersten Eintrag im Stack herunterzunehmen und die Daten zu setzten. Die schematische Darstellung der Arbeitsweise des Parsers wird nochmal in Abbildung 5.4 dargestellt.

5.1.1 Layer

Für das gelayerte Parsieren kann die Idee des Parser, so wie er vorgestellt wurde, ohne weiteres übernommen werden. Es gibt nur zwei Kleinigkeiten zu beachten.

1. Beim Parsieren muß mit Regeln aus dem niedrigsten Layer begonnen werden. Die Auswahl der Regeln muß also wie folgt aussehen:
   (a) Auswahl einer konfliktfreien Regel aus dem niedrigsten Layer, bis keine dieser Regeln mehr anwendbar ist.
   (b) Auswahl einer abhängigen Regel aus dem selben Layer, bis keine dieser Regeln mehr anwendbar ist.
   (c) Da weder konfliktfreie noch abhängigen Regeln aus diesem Layer anwendbar sind, muß der nächst höhere Layer genommen werden.
   (d) Wieder fängt die Auswahl mit den konfliktfreien Regeln aus diesem Layer an.
2. Ein anderer Punkt betrifft das Backtracking. Sollte der Parser sich an einer Stelle mit einem niedrigen Layer falsch entschieden haben, so kann sich das erst in einem viel höherem Layer bemerkbar machen. Daher muß der momentane Layer mit auf den Stack gelegt werden, um so einen korrekten Schritt rückwärts machen zu können.

5.2 Parser Design

Wie schon bei dem Design der kritischen Paaren in 4.2 bietet sich für einen flexiblen Austausch des Parseralgorithmus auch hier das Strategiemuster aus [GHJV95] an. Ebenfalls

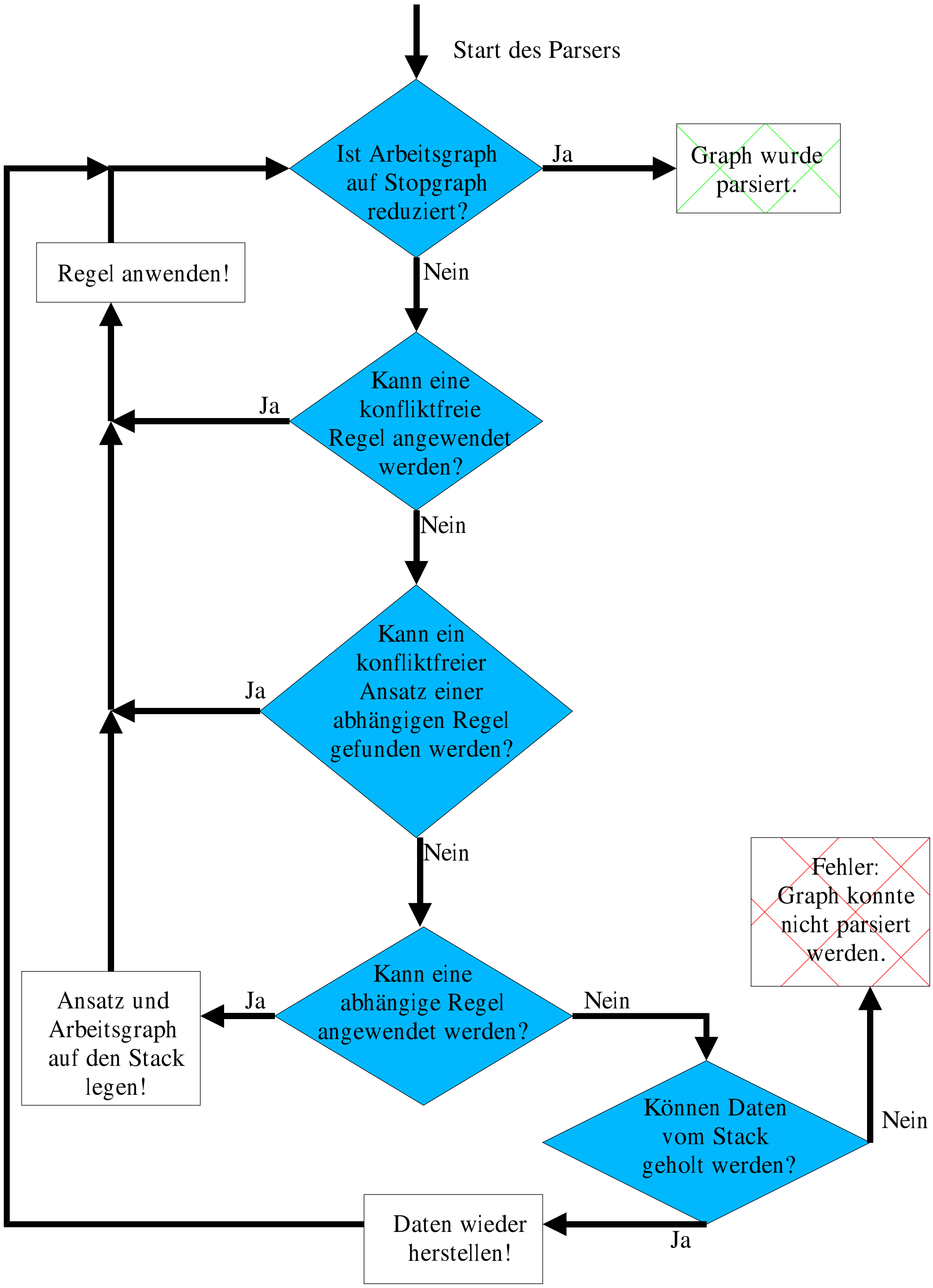

Abbildung 5.4: Schematische Arbeitsweise des Parsers

analog zu den der kritischen Paaren ist auch eine Visualisierung der Parsierung nötig, so daß auch eine graphische Benutzeroberfläche nötig ist.

Die Klassenstruktur des Parser wird in Abbildung 5.5, 5.6, 5.7 und 5.8 dargestellt.

Die Umsetzung des Algorithmus befindet sich in der Klasse ExcludeParser. Diese Klasse muß durch das Stategiemuster das Interface Parser implementieren. Allerdings erfolgt diese

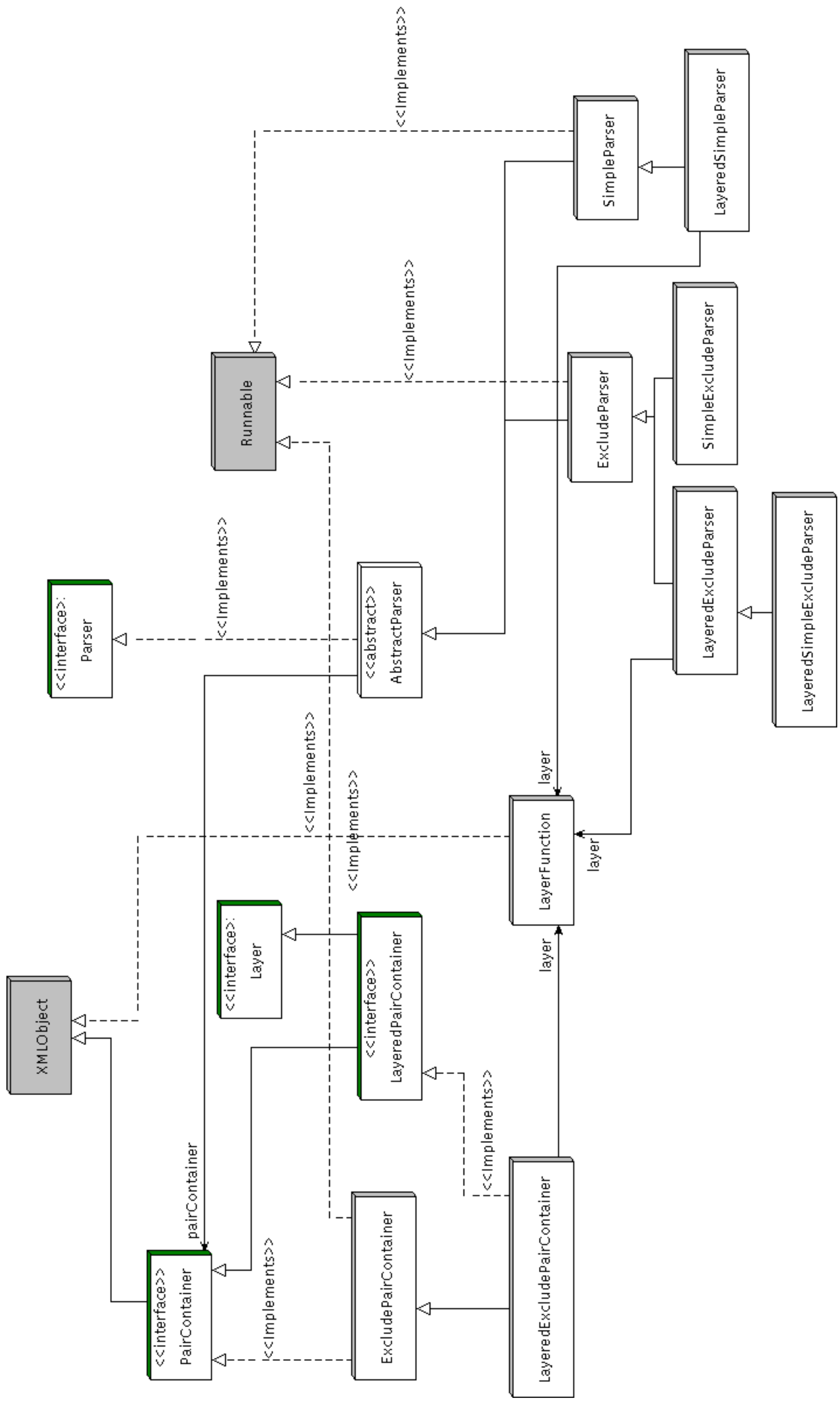

Abbildung 5.5: Klassendiagramm des Parsers Teil I

Implementierung des Interface über die abstrakte Klasse AbstractParser.

Da der Parser die kritische Paaranalyse benutzt, benötigt er einen Container, der die kritischen Paare aufbewahrt. Ein solcher Container existiert bereits in PairContai-

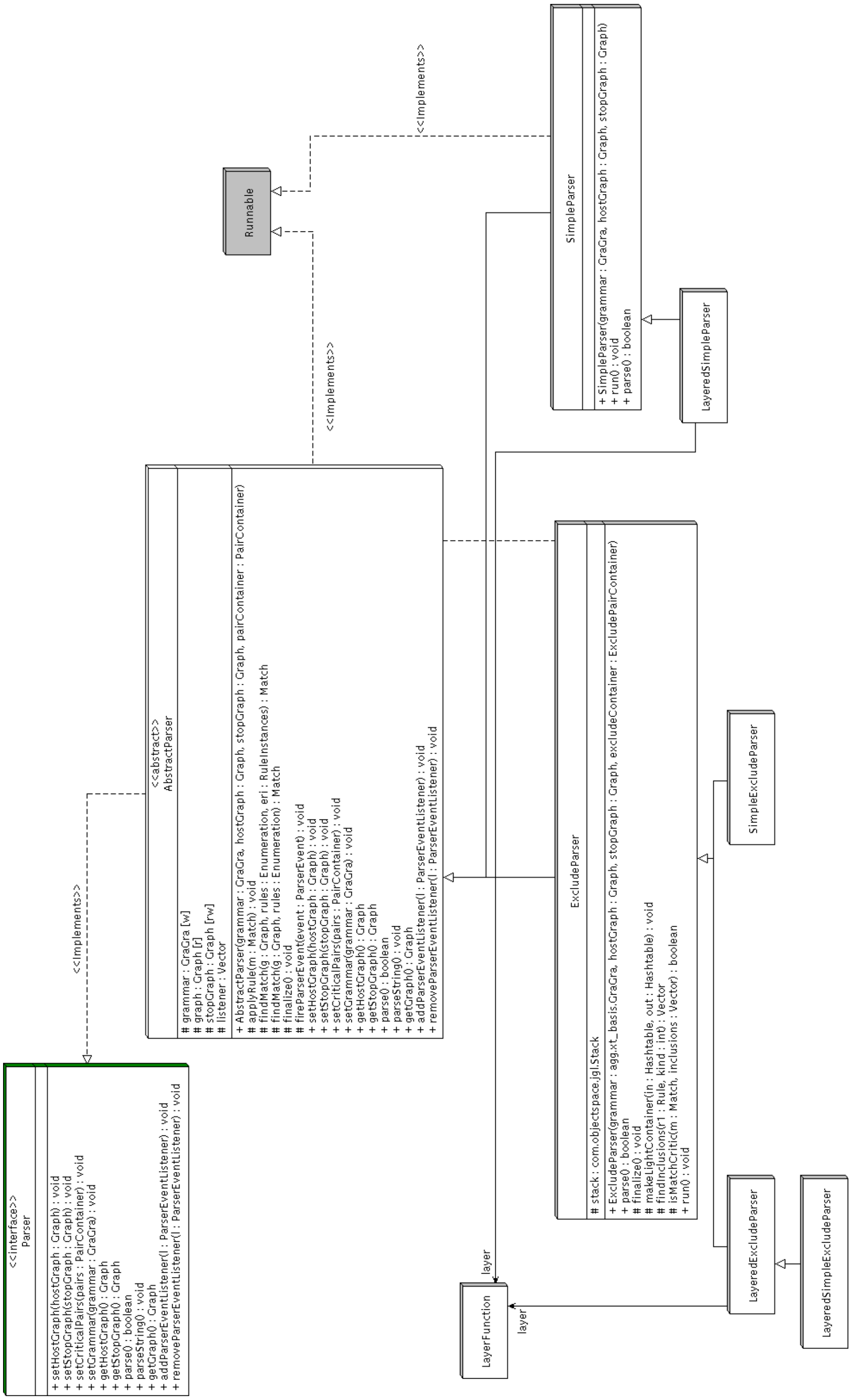

Abbildung 5.6: Klassendiagramm des Parsers Teil II

ner. Diese Klasse wurde bereits in 4.2 beschrieben. Die Klasse AbstractParser enthält

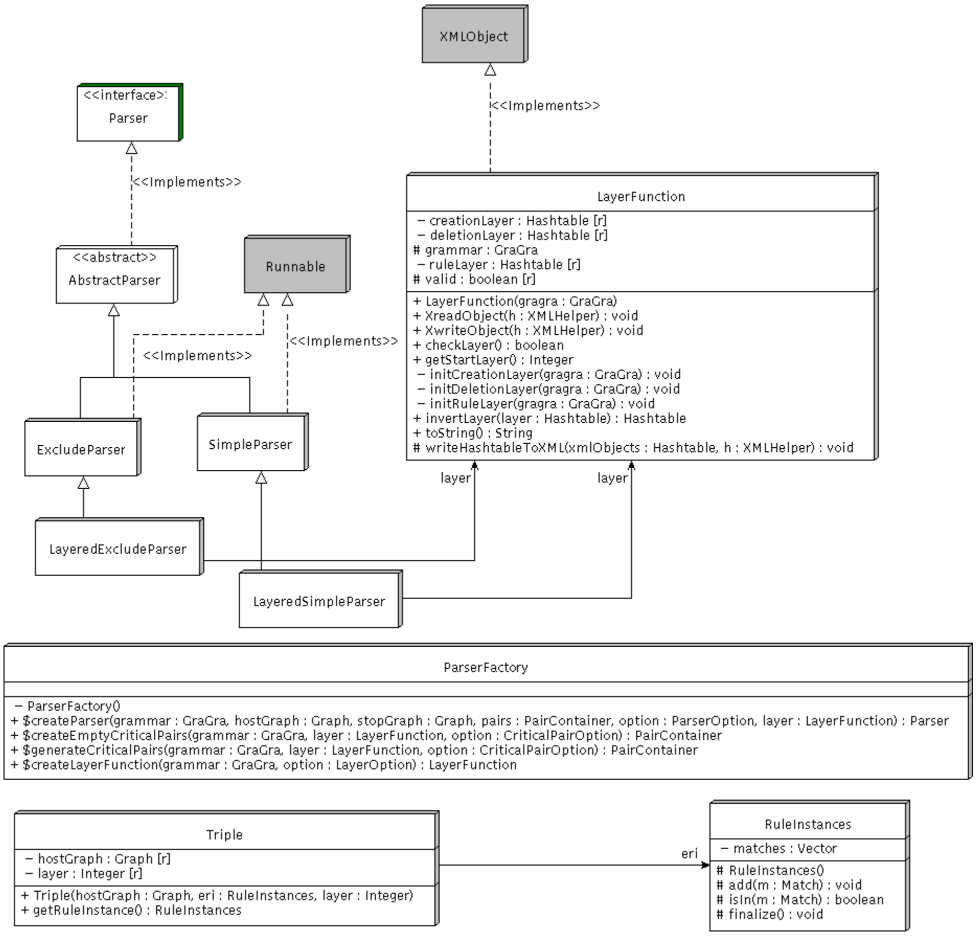

Abbildung 5.7: Klassendiagramm des Parsers Teil III

drei Kategorien von Methoden. Zum einen gibt es die Methoden, die den Zugriff auf die Daten ermöglichen, die zum Parsieren nötig sind. Dazu gehören: setHostGraph(Graph hostGraph), setStopGraph(Graph stopGraph), setCriticalPairs(PairContainer pairs), setGrammar(GraGra grammar), Graph getHostGraph(), Graph getGraph(), Graph getStopGraph(). Die zwei Methoden addParserEventListener(ParserEventListener l) und removeParserEventListener(ParserEventListener l) der zweiten Kategorie dienen dazu, daß sich Objekte registrieren lassen können, wenn sie an Nachrichten vom Parser interessiert sind. Diese Art der Kommunikation entspricht dem Beobachtermuster aus [GHJV95]. Dabei entspricht das Listenerobjekt dem Beobachter. Als letztes bleibt dann nur noch die Methode, die den Parser startet. Sie nennt sich parse(). In AbstractParser ist natürlich genau diese Methode abstrakt, da sie von den jeweiligen konkreten Parsern implementiert werden muß. Zusätzlich hält AbstractParser auch die Informationen wie Arbeits-, Stopgraph, Grammatik, kritische Paare und die Menge der Beobachter. Diese Daten stecken in den Attributen graph, stopGraph, grammar, excludeContainer sowie listener. Der konkrete Algorithmus verbirgt sich in der Klasse ExcludeParser. Die Erweiterungen dieser Klasse gegenüber AbstractParser sind minimal. Trotzdem sollen sie erläutert werden. Durch die Vererbung aus AbstractParser kann schon auf die Methoden dort zugegriffen werden. Da ExcludeParser mit einem Stack arbeitet, ist das Attribut stack das einzige, das hinzukommt. Dieses Attribut liefert genau die Funktionalität, die der Stack des Parserargorithmus fordert. Eine weitere Neuerung

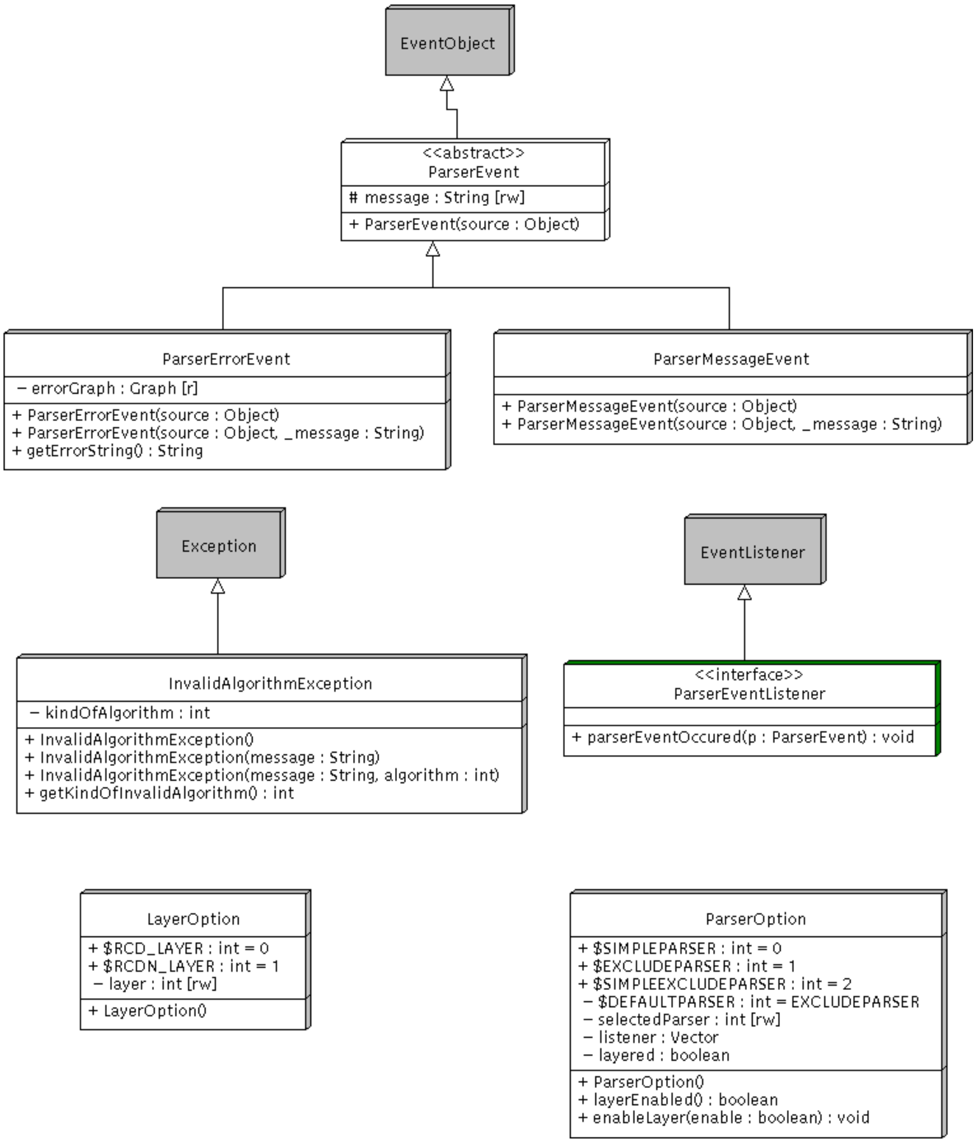

Abbildung 5.8: Klassendiagramm des Parsers Teil IV

ist die Implementierung des Interface Runnable und ihrer Methode run(). Dadurch ist es möglich, daß das Parsieren auch parallel zu anderen Arbeiten stattfinden kann. Ein typisches Szenario hierfür ist die Bearbeitung einer anderen Grammatik während ein Graph parsiert wird.

Zum Vergleich der Performanz existieren noch zwei weitere Parser. Der Parser in der Klasse SimpleParser realisiert ein Backtrackingalgorithmus ohne jegliche Optimierung. Die kritischen Paare werden hier komplett ignoriert. Es werden so auch keine Resourcen für die Berechnung verbraucht. So braucht diese Klasse nur die Methode parse() zu implementieren. Sämtliche andere Funktionalität übernimmt AbstractParser.

Der dritte Parser ist eine Mischform aus SimpleParser und ExcludeParser. Während in ExcludeParser versucht wird, immer einen konfliktfreien Ansatz zu finden, nutzt der Parser aus der Mischform nur die Unterscheidung zwischen konfliktfreien und abhängigen Regeln aus. Dieser Parser ist in der Klasse SimpleExcludeParser realisiert. Im Gegensatz zu den anderen beiden Parsern ist dieser nicht von AbstractParser sondern von ExcludeParser abgeleitet. Diese Vererbung ermöglicht es, die bestehenden Auswertungsmethoden der kritischen Paare wieder zu benutzen.

Betrachtet man in den Parseralgorithmen den Abschnitt, in dem abhängige[1] Regeln angewendet werden, so muß an dieser Stelle der aktuelle Ansatz auf den Stack gelegt werden. Außerdem muß auch in einer Liste verglichen werden, ob der aktuelle Ansatz schon einmal versucht wurde. Diese Liste wird durch die Klasse RuleInstance repräsentiert. Da in die Liste nur Ansätze hinzugefügt werden und gefragt wird, ob ein bestimmter Ansatz schon vohanden ist, braucht die Klasse RuleInstance nur die Methoden add(Match m) und isIn(Match m) zur Verfügung zu stellen.

Die Benutzung der Parserevents erfolgt analog zu den Standardevents aus anderen Paketen wie Swing oder AWT. Interessierte Klassen müssen dazu nur das Interface ParserEventListener implementieren. Die Klasse ParserErrorEvent liefert als zusätzliche Information den Graphen und einen Text, wenn der Fehler auftritt.

Die Erzeugung eines Objekts erfolgt nach dem Erbauermuster aus [GHJV95]. Die Erzeugung findet in der Klasse ParserFactory statt. Die Methoden haben alle gemein, daß sie durch mindestens ein Optionsobjekt konfiguriert werden. Diese Konfigurierung ermöglicht es, Objekte eines speziellen Typs zu erzeugen. Die detailierte Beschreibung befindet sich wieder in der API.

5.2.1 Layer

Dank der Objektorientierung können die Parser, die mit Layern arbeiten, durch einfache Unterklassenbildung realisiert werden. Die vorher vorgestellten Parser ExcludeParser, SimpleExcludeParser und SimpleParser erhalten eine entsprechende Unterklassen[2] mit vorangestelltem Layered. In diesen Unterklassen muß nur die Methode parse überschrieben werden. Die Verwendund der Layerfunktion verändert nur wenig die Arbeitsweise des Parsers, wie es in Kapitel 5.1.1 beschrieben ist. Außerdem kommt nur noch ein Attribut für die Layerfunktion hinzu.

Die Layerfunktion an sich ist in der Klasse LayerFunction enthalten. Nach den Zugriffsmethoden auf die Layer sind die Methoden checkLayer(), getStartLayer() und invertLayer(Hashtable layer) zu beachten. Die Methode checkLayer() überprüft, ob die eingegebenen Layer korrekt sind. Nach Definition 15 müssen die Layer größer null sein. Es wird aber nicht gefordert, daß der kleinste Layer gleich null ist. Daher liefert die Methode getStartLayer() den kleinsten Layer. Nach Definition wird jede Regel genau einem Layer zugeodnet. Um eine Regel aus einem bestimmten Layer auszuwählen, ist eine Invertierung der Layerfunktion nötig. Diese Invertierung wird durch die Methode invertLayer(Hashtable layer) vorgenommen.

5.2.2 Graphische Benutzeroberfläche

Das Design der Oberfläche gestaltet sich recht einfach. Das mag unter anderem daran liegen, daß nur eine Anzeige nötig ist, die den Prozeß des Parsers zeigt. Außerdem ist nur noch die Anzeige des Stopgraphen interessant. Weitere Informationen, wie sie etwa in der kritischen Paaranalyse benötigt wurden, sind bei dem Parser nicht nötig. Der Arbeitsgraph

[1] Im SimpleParser wird angenommen jede Regel ist abhängig.

[2] LayeredExcludeParser, LayeredSimpleExcludeParser und LayeredSimpleParser

und der Stopgraph sollen parallel angezeigt werden. Dafür kann man sich eine ähnliche Realisierung wie für den Überlappungsgraphdesktop (s. 4.12) vorstellen. Die Anzeige der Graphgrammatikbäume und der linken Regelseiten ist nicht nötig.

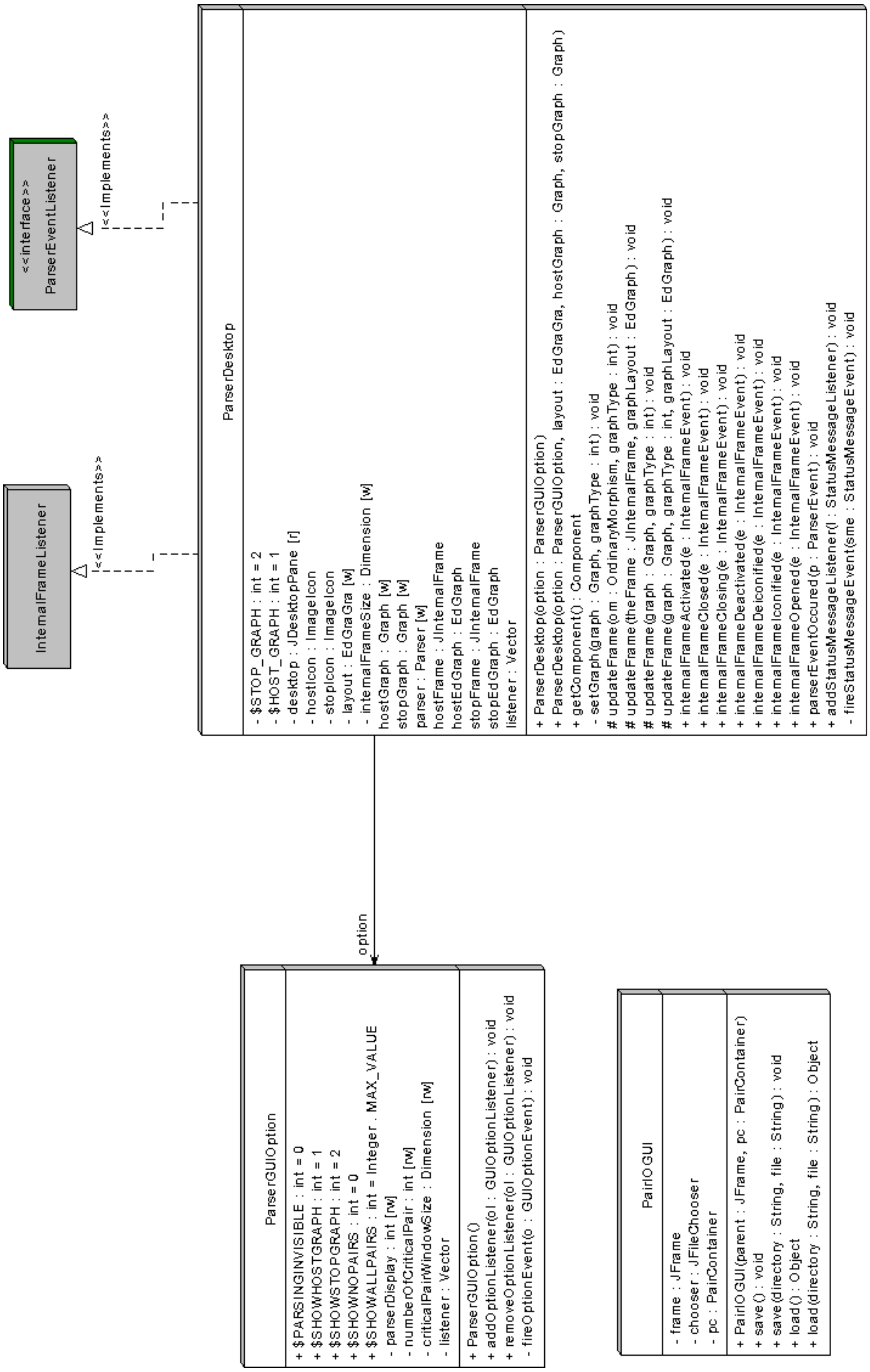

Abbildung 5.9: Klassendiagramm des Parserdesktops aus `agg.gui.parser`

Abbildung 5.9 zeigt die Klassen für die graphische Benuteroberfläche. Wie auch schon bei den kritischen Paaren ersetzt diese Oberfläche vollständig die klassische Oberfläche des AGG. Die Klasse ParserDesktop enthält dabei sowohl den Desktop als auch die internen Fenster für den Arbeits- und Stopgraphen. Über die Klasse ParserOption kann die Anzeige der Graphen konfiguriert werden. Zum Beispiel ist ein leerer Stopgraph nicht so interssant, um ihn anzeigen zu lassen. Diese Anzeige kann in den Optionen abgeschaltet werden. Ein weiteres Feature ist die Priorisierung des Stopgraphen. In der Regel enthält der Stopgraph relativ wenig Graphobjekte, im Extremfall sogar gar keine. Daher benötigt er in der Größe der Darstellung wenig Platz. Damit der Stopgraph nicht unter dem Fenster des Arbeitsgraphen begraben wird, liegt das Fenster des Stopgraphen immer über dem des Arbeitsgraphen.

5.3 Implementierung

Bei der Umsetzung des Design hat man immer wieder mit den Eigenschaften von AGG zu kämpfen. Dieses Problem tritt schon bei der Initialisierung des Parsers auf. Da AGG bei einer Regelanwendung mit in-place Transformation arbeitet, würde der Arbeitsgraph verändert werden. Es ist im allgemeinen aber nicht wünschenswert, den eigentlichen Arbeitsgraphen zu verändern oder sogar zu zerstören. Daher muß der Arbeitsgraph vor dem Parsieren kopiert werden.

Der nächste Schritt besteht darin eine Schleife zu konstruieren, die solange läuft, bis entweder der Arbeitsgraph isomorph zum Stopgraph oder der Stack leer ist und es keinen möglichen Ansatz mehr für eine Regel gibt.

Innerhalb dieser Schleife befindet sich ein kleiner und ein etwas größerer Block. Der kleine und erste Block versucht, nur eine konfliktfreie Regel anzuwenden.

Der zweite Block ist komplizierter und demzufolge auch größer. Dieser zweite Block darf allerdings nur betreten werden, wenn keine Regel angewendet wurde. Dadurch soll sichergestellt werden, daß nach einer Regelanwendung wieder zuerst der erste Block durchlaufen wird, damit so früh wie möglich konfliktfreie Regeln angewendet werden. Als erste Aktion im zweiten Block muß ein Ansatz einer beliebigen abhängigen Regel gefunden werden. Dieser Ansatz muß geprüft werden, ob er kritisch ist oder nicht (vgl. Abbildung 5.1 und 5.2). Sollte sich dieser Ansatz als unkritisch herausstellen, so kann er sofort angewendet werden. Andererseits wenn der Ansatz kritisch ist, muß geprüft werden, ob dieser Ansatz schon einmal versucht wurde, also dieser Ansatz schon in der Liste der kritischen Paare existiert. Diese Liste wird durch die Klasse RuleInstance repräsentiert. Sollte dieser Ansatz schon existieren, wird er verworfen und nach einem anderen gesucht. Sollte der Ansatz also noch nicht existieren, so wird er für spätere Zwecke gehalten. Trozdem soll noch nach einem anderen nicht kritischen Ansatz gesucht werden.

Sollte es keinen unkritischen Ansatz mehr geben, so muß die gesamte Backtracking-Maschinerie anlaufen. Das heißt, alle wichtigen Informationen müssen protokolliet werden. Zum einen gibt es den momentanen Arbeitsgraph, zum anderen die bisherigen kritischen Ansätze, was auf den Stack gelegt werden muß. Um danach weiter parsieren zu können, muß der Arbeitsgraph wieder kopiert werden. Die Liste mit den bisherigen kritischen Ansätzen muß nicht kopiert werden. Es reicht aus, eine neue leere Liste zu erzeugen. Die Frage, warum die nicht kopiert werden muß, läßt sich leicht erklären. Zu diesem Zeitpunkt ergibt sich das Szenario, daß eine Regelanwendung kurz bevorsteht. Das heißt aber, der Graph wird verändert und dieser Match kann nicht ein zweites Mal angewendet werden. Für die restlichen Ansätze in der Liste gilt aber, daß sie gerade durch die vorherige Regelanwendung zum Ziel führen. Also muß die Liste wieder leer sein.

Bis jetzt wurde immer davon ausgegangen, daß ein Ansatz einer Regel existiert. Sollte sich nach dieser Ansatzsuche herausstellen, daß kein Ansatz mehr gefunden werden kann, so muß "ein Schritt zurück" gemacht werden. Die lezten Daten auf dem Stack müssen wieder hergestellt werden. Da die aktuellen Daten in eine Sackgasse geführt haben, können sie verworfen werden. Es ist hier allerdings wichtig darauf zu achten, daß der Stack nicht leer ist, da der Arbeitsgraph dann nicht parsiert werden konnte. Es wird dann ein ParserErrorEvent erzeugt.

Wie man mit dieser Umsetzung einen Graphen parsieren kann, zeigt das Sequenzdiagramm in Abbildung 5.10 examplarisch an einem Parser mit kritischer Paaranalyse für die Excludebeziehung.

5.3.1 Zeitverhalten der verschiedenen Parser

Untersucht wurden hier nur der SimpleParser und der ExcludeParser. Daß der SimpleExcludeParser nicht mit in diesem Test auftritt, liegt an folgender Tatsache: Widererwartend lief der Simpleparser so schnell gegenüber dem ExcludeParser. Daher sollte ein Parser mit mehr Backtracking und weniger Optimierung zwischen den Ergebnissen vom Simpleparser und ExcludeParser sein.

Die beiden Algorithmen SimpleParser und ExcludeParser hatten sich folgendem Szenario zu stellen: Die der Überprüfung zugrunde liegende Graphgrammatik bestand aus fünf Regeln, alle ohne negative Anwendungsbedingungen. Der Arbeitsgraph enthielt 13 Knoten mit insgesammt 17 Kanten bei aktivierter Dangling-Condition. Der Simpleparser benötigte gerade mal 1:20 Minuten. Der ExcludeParser schaffte diese Disziplin in nur 1:00 Minuten, mußte aber erst 14 Minuten lang die kritischen Paare berechnen. Eine vollkommen andere Situation trat bei ausgeschalteter Dangling-Condition ein. Hier war kein Parser zu bewegen, seinen Prozeß zu beenden. Die Vorgänge wurden mit einem OutOfMemoryError quitiert. Lediglich die Zeit, nach der der Fehler auftrat, variierte. So lieferte der Simpleparser schon nach fünf Minuten den Fehler, während sich der ExcludeParser sieben Minuten für die kritischen Paare Zeit ließ und dann noch mal acht Minuten bis zum Fehler brauchte.

Das unterschiedliche Verhalten der beiden Algorithmen ist in dem Design und der Implementierung von AGG zu suchen. Schon in früheren Versionen von AGG konnte ein Performanzverlust nach mehreren Regelanwendungen bzw. Morphismusbildungen beobachtet werden. Bedenkt man, wie stark AGG damit bei der kritischen Paaranalyse und beim Parsieren belastet wird, ist dieses Zeitverhalten der Parser weniger verwunderlich.

Layer

Durch die Layerfunktion ergibt sich eine gewisse Kontrollstruktur (Regeln niedriger Layer müssen vor Regeln höherer Layer angewendet werden). Sie verbessert nur verschwindend gering die Laufzeit des Parsers. Extreme Verbesserungen wurden allerdings in der Berechnung der kritischen Paare erzielt. Dieser Geschwindigkeitsschub katapultiert damit die Parser in den Bereich des Zumutbaren für den Benutzer.

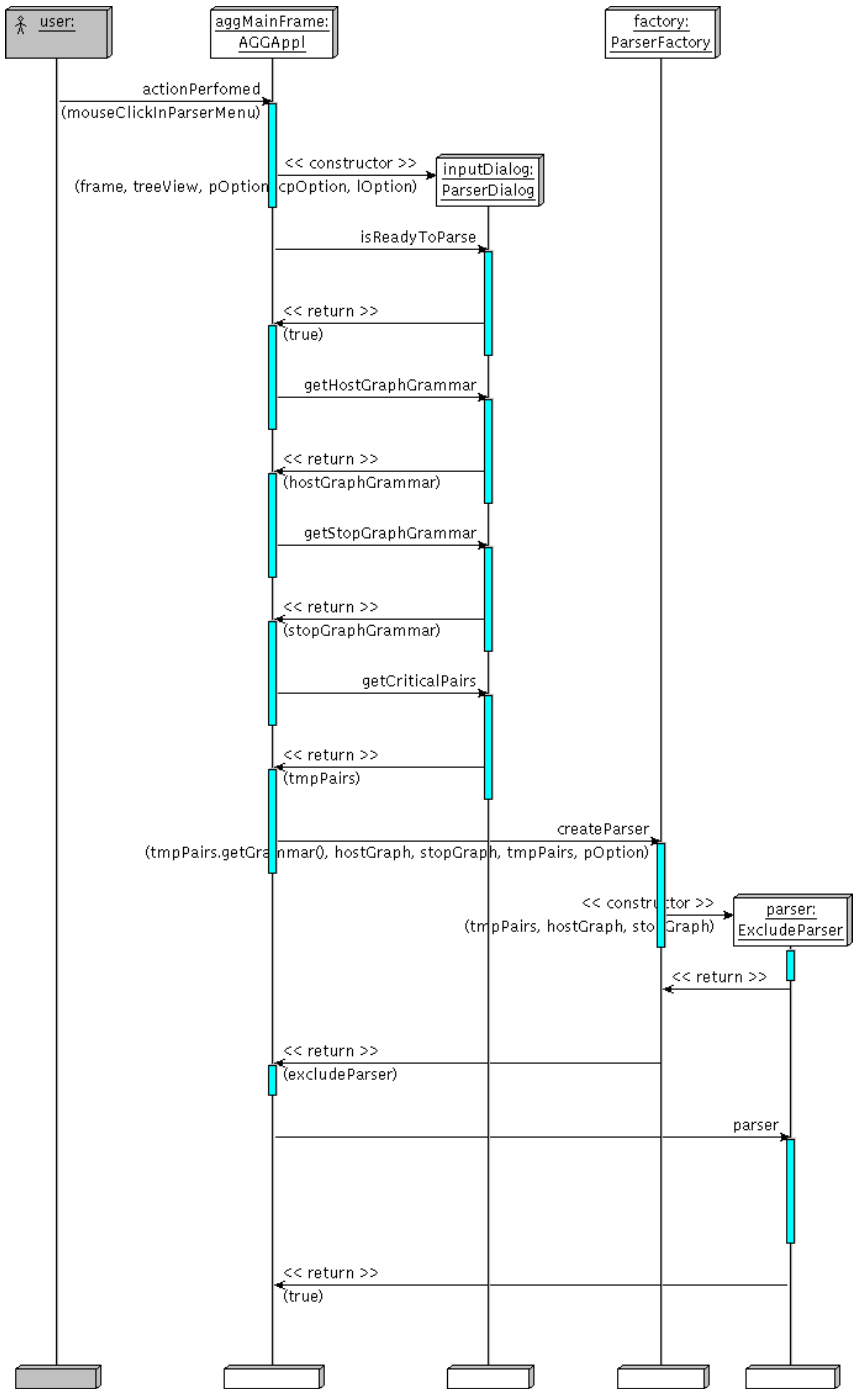

Abbildung 5.10: Sequenzdiagramm für den Aufruf eines Parsers

5.3.2 Graphische Benutzeroberfläche

Die Implementierung der graphischen Benutzeroberfläche erweist sich nicht als besonders problematisch. Die Erstellung des Desktops und der weiteren Oberfläche konnte sehr schnell realisiert werden, da die benutzten Komponenten wie zum Beispiel Dialogfenster schon als vorgefertigte Komponenten von Java in einer Bibliothek zur Verfügung gestellt werden.

Kapitel 6

Benutzerhandbuch

Nach der Entwicklung des Parser haben sich nun auch die Anforderungen an die graphische Benutzeroberfläche geändert. Die neu entstandenen Teil der Oberfläche wurden schon in den einzelnen Kapiteln kurz beschrieben. Als nächstes wird die Benutzung der neuen Oberfläche beschrieben.

Auf den ersten Blick hat sich nichts an der Oberfläche verändert. Abbildung 6.1 zeigt das AGG nach dem Starten. Es fallen nur zwei zusätzliche Menüpunkte (Parser und Preferences) auf.

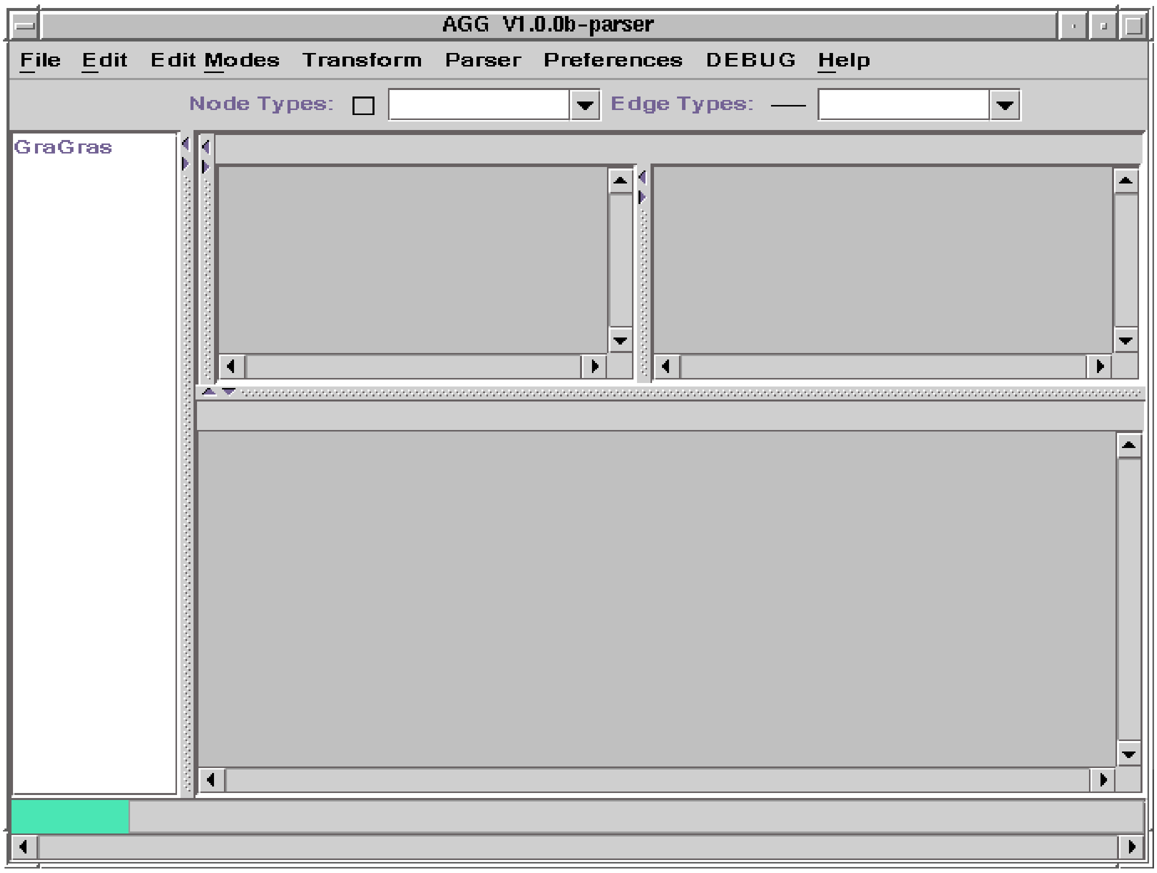

Abbildung 6.1: Neu gestartetes AGG

Obwohl diese Arbeit seinen Schwerpunkt auf dem Parser hat, beginnt das Parsieren mit den Preferences. Hier wird festgelegt, welche Strategien eingesetzt werden sollen.

6.1 Einstellungen (Preferences)

6.1.1 Parser (Parser)

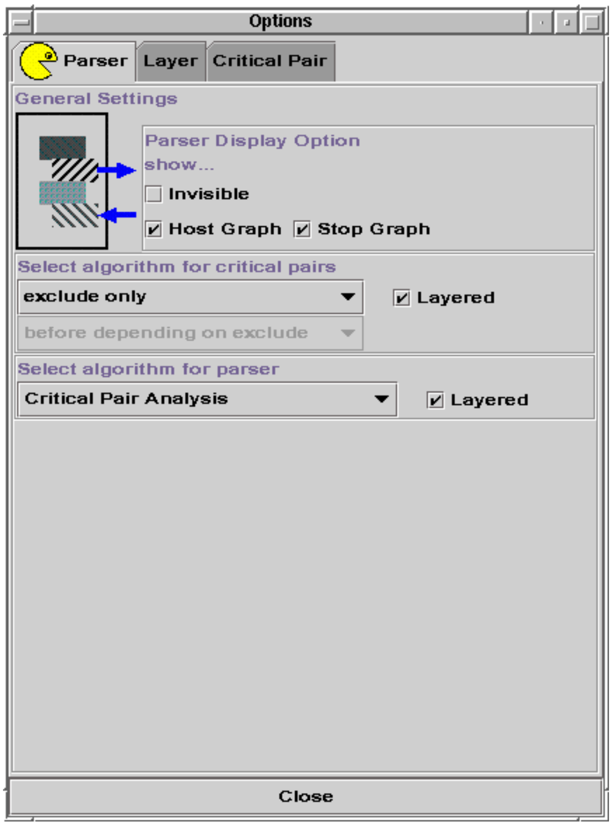

Abbildung 6.2: Einstellungen des Parsers

Abbildung 6.2 zeigt die Einstellungsmöglichkeiten für den Parser. Die Parseroptionen unterteilen sich in drei Punkte. Die *Parser Display Option* zeigt an, ob und inwieweit der Parsingprozeß angezeigt werden soll. Die beiden Checkboxen *Invisible* und *Host Graph* sind zu einer Gruppe zusammengefaßt. Das heißt, nur einer der beiden kann angewählt werden. Ist *Invisible* angewählt so wird zusätzlich das Feld *Stop Graph* deaktiviert; es sind also keine Eingaben hier möglich. Dies ist auch intuitiv verständlich, da die Veränderung der Anzeige des Stopgraphen beim unsichtbaren Parsieren keinen Sinn macht. Anders sieht diese Situiation bei *Host Graph* aus. Ist dieses Feld markiert, so wird mindestens der Arbeitsgraph beim Parsieren angezeigt. Hier kann es nun der Wunsch des Benutzers sein, den Stopgraphen mit anzeigen zu lassen, um eine Referenz zu haben, in wie weit sich der Arbeitsgraph schon dem Stopgraphen angenähert hat.

Eine weitere Einstellung ist die Auswahl des Algorithmus der kritischen Paare. Generell kann hier unterschieden werden zwischen gelayerten und ungelayerten kritischen Paaren. Diese Auswahl wird durch die Checkbox *layered* ausgedrückt. Die eigentliche

Auswahl des Algorithmus befindet sich in der oberen Combobox. Dort kann zwischen *exclude only* und *exclude and before* ausgewählt werden. *exclude* bezieht sich dabei genau auf die Berechnung der kritischen Paare, wie sie im vorherigen Kapitel vorgestellt wurde. Die Auswahl *exclude and before* ist eher als Ausbaustufe anzusehen. Die *before*-Relation erwies sich in einem studentischen Projekt als sehr rechenintensiv mit nur mäßigem Gewinn für die Parserlaufzeit. Daher wurde in dieser Arbeit darauf verzichtet diesen Algorithmus zu implementieren, so daß die untere Combobox mit *before depending on exclude* und *before independing on exclude* ohne Wirkung ist.

Die letzte Einstellung betrifft den Parser selbst. Auch hier ist es möglich einen gelayerten Parser auszuwählen, in dem wieder die Checkbox *layered* angewählt wird. Da ein gelayerter Parser ohne gelayerte kritische Paare genauso wenig Sinn wie umgekehrt macht, sind die beiden Checkboxen synchronisiert. Das heißt, wird eine der beiden Checkboxen an- oder abgewählt, so wird die andere Checkbox automatisch in den selben Zustand gesetzt.

6.1.2 Layerfunktion (`Layer`)

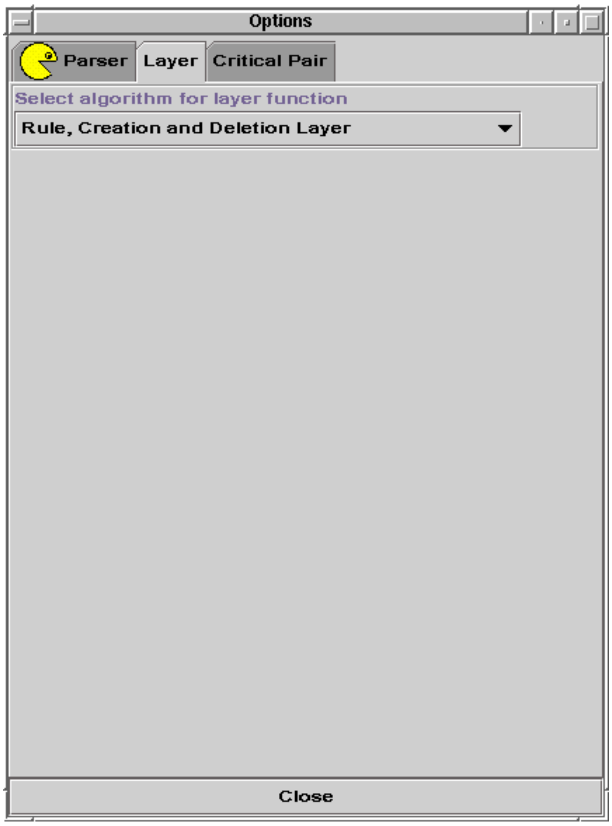

Abbildung 6.3: Einstellungen der Layerfunktion

Weitere Einstellungen finden sich unter dem Karteikartenreiter *Layer*. Abbildung 6.3 zeigt die Einstellungsmöglichkeiten für die Layerfunktion. Diese Auswahl des Algorithmus der Layerfunktion entspricht genau dem in Abschnitt 2.8. Ist bei den kritischen Paaren oder dem Parser die Layerfunktion nicht aktiviert, so hat auch diese Auswahl keine Funktion.

6.1.3 Kritische Paare (`Critical Pair`)

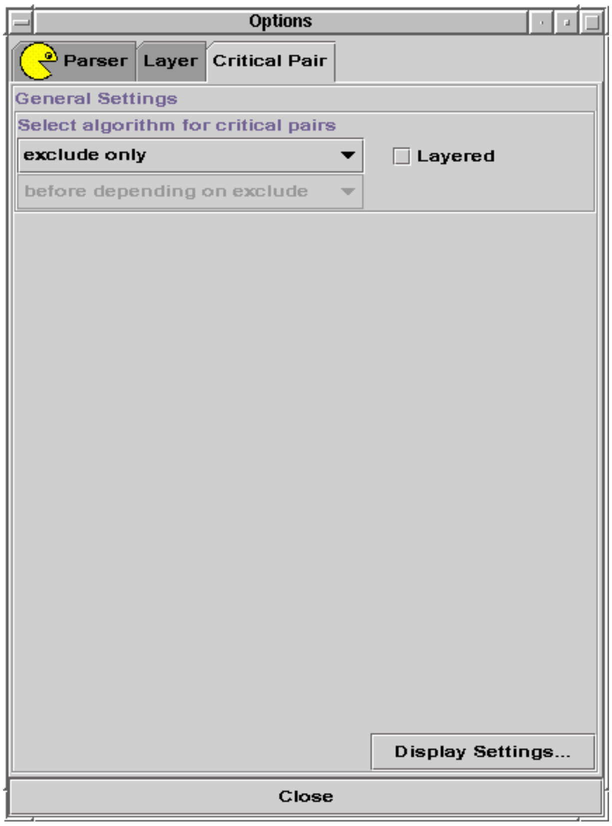

Abbildung 6.4: Einstellungen des Algorithmus der kritischen Paareanalyse

Abbildung 6.4 zeigt die Einstellungsmöglichkeiten für die kritische Paaranalyse. Die Einstellungen für die kritischen Paare enthalten zum einen Optionen für den Algorithmus und zum anderen Optionen für die graphische Wiedergabe auf dem Bildschirm. Die Einstellungen des Algorithmus ensprechen genau dem des Parsers. Die Auswahl des Algorithmus der kritischen Paare überträgt sich direkt auf die Einstellungen des Parsers.

Um die Ausgabe der kritischen Paare zu beeinflußen, muß zunächst die Anzeige am *Display Settings* Knopf umgestellt werden. Abbildung 6.5 zeigt die Einstellungsmöglichkeiten für diese Ausgabe. Hier stehen zwei Möglichkeiten zur Auswahl. Da es bei der kritischen Paarberechnung vorkommen kann, daß ein Paar 30 oder mehr Überlappungsgraphen enthält, kann der Benutzer eine maximale Anzahl von kritischen Paaren festlegen. Im Extremfall heißt das, der Benutzer kann den Schieberegler auf keine Anzahl (*None*) stellen. Ein weiteres Feature ist die initiale Größe des Fensters der Überlappungsgraphen. Diese Größe kann mit Hilfe des horizontalen und vertikalen Schiebereglers eingestellt werden. Um eine bessere Übersicht auf dem Bildschirm zu erhalten ist diese Darstellung um den Faktor zwei kleiner. Die eingestellte Größe ist in der späteren Oberfläche doppelt so groß. Mit dem Knopf *General Settings* gelangt man wieder zurück zur Einstellung des Algorithmus.

Sollten alle Einstellungen zur Zufriedenheit getan sein, so kann der Benutzer von jeder Stelle aus die Enstellungen durch den Knopf *Close* verlassen.

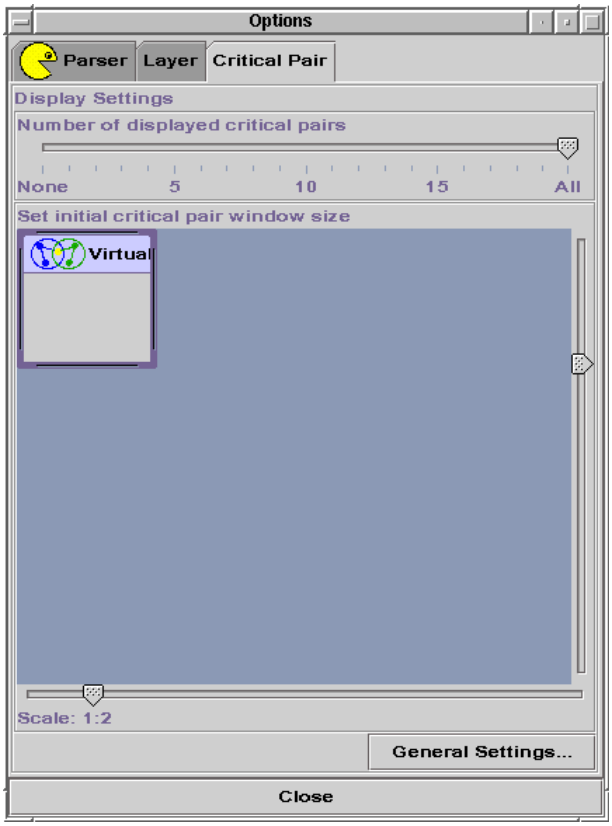

Abbildung 6.5: Einstellungen der Anzeige der kritischen Paare

6.2 Kritische Paaranalyse

Sind alle Einstellungen getroffen, kann der Benutzer wählen, ob er sich die kritischen Paare ansieht oder den Parser startet. Entscheidet sich der Benutzer für den ersten Punkt, so steht ihm diese Möglichkeit im Menu Parser unter Critical Pair zur Verfügung. Es erscheint ein weiteres Untermenü, in dem muß der Benutzer sich entscheiden, ob er die kritischen Paare generieren läßt (Generate) oder lädt (Load). Sollen die kritischen Paare neu generiert werden, so wird die Graphgrammatik benutzt, die gerade selektiert ist. Abbildung 6.6 zeigt das AGG mit entsprechender Grammatik. Wenn die kritischen Paare geladen werden sollen, so erscheint ein entsprechender Dateidialog. Zusätzlich wird die Benutzung der kritischen Paare durch ein Häckchen am Eintrag Critical Pair signalisiert. Es erscheint dann die Benutzeroberfläche der kritischen Paaranalyse, die auch in Abbildung 6.7 zu sehen ist.

Auf der linken Seite des Fensters erkennt der Benutzer den Baum für die Auswahl der Graphgrammatik wieder. Dieser Baum zeigt jedoch nur die selektierte Graphgrammatik mit ihren dazugehörigen Regeln. Der Arbeitsgraph wird nicht angezeigt. Da die Analyse der kritischen Paare auf zwei Regeln beruht, ist ein ein zweiter Baum vorhanden, der exakt die selbe Graphgrammatik anzeigt, wie der erste. Die in Definition 17 bezeichnete Regel r_1, findet sich in dem oberen Baum wieder, entsprechend befindet sich Regel r_2 im unteren Baum.

Zur Betrachtung von kritischen Paaren muß man nun die gewünschten Regeln durch Selektieren im Baum auswählen. Abbildung 6.8 zeigt zwei ausgewählte Regeln mit den

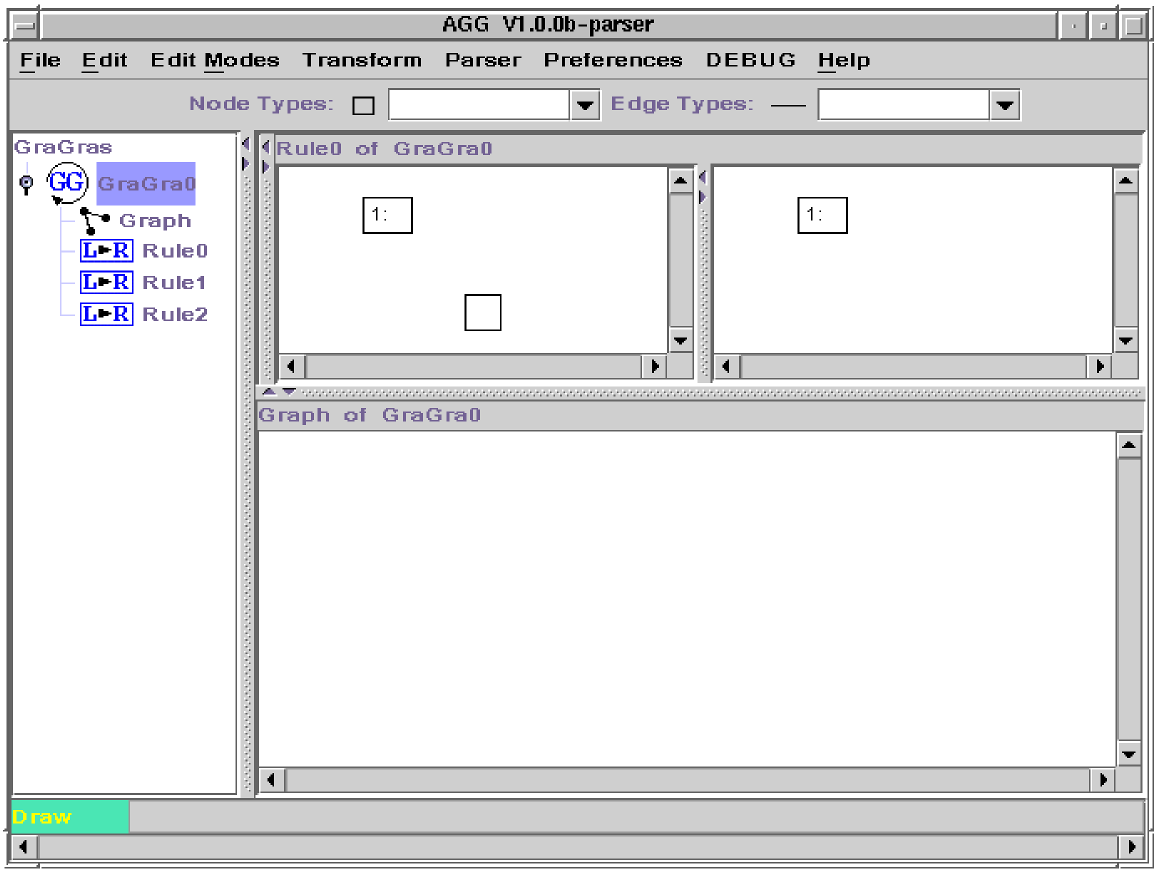

Abbildung 6.6: AGG mit einer Graphgrammatik

dazugehörigen Überlappungsgraphen.

Im oberen Teil der Benutzeroberfläche werden nun die linken Seiten der ausgewählten Regeln angezeigt. Dabei zeigt das linke Fenster den Teil der Regel r_1 und das rechte Fenster den Teil der Regel r_2 an. Im unteren Fenster erscheinen die Überlappungsgraphen in separaten Fenstern. Im Titel jedes Fensters wird zusätzlich noch die Nummer des Überlappungsgraphen angezeigt.

Die Visualisierung des Morphismus der linken Regelseite in den Überlappungsgraphen erfolgt auf die AGG typische Weise. Die Graphobjekte werden eindeutig nummeriert. Dabei fällt sofort auf, daß die Numerierung nicht unbedingt mit eins anfängt[1]. Es wird allerding nur der Morphismus für das selektierte Fenster angezeigt. Ein synchrones Betrachten mehrerer Morphismen ist nicht möglich. So schalten sich die Morphismusanzeige immer auf das selektierte Fenster, wobei das deselektierte Fenster die Morphismenanzeige verliert.

Sollten zwei Regeln konfliktfrei sein, so erscheint anstatt der Überlappungsgraphen ein Fenster mit der Meldung, diese beiden Regeln seien nicht kritisch (vgl. Abbildung 6.9).

Wenn die Arbeiten mit den kritischen Paaren abgeschlossen sind, kann der Benutzer die kritischen Paare speichern. Das erreicht man im Menü Parser unter dem Untermenü Critical Pair mit dem Eintrag Save. Wieder erscheint ein entsprechender Dateidialog.

Um wieder zur herkömmlichen AGG-Oberfläche zurückzugelangen, steht der Eintrag

[1] Siehe linke Regelseite von r_2 in Abbildung 6.8

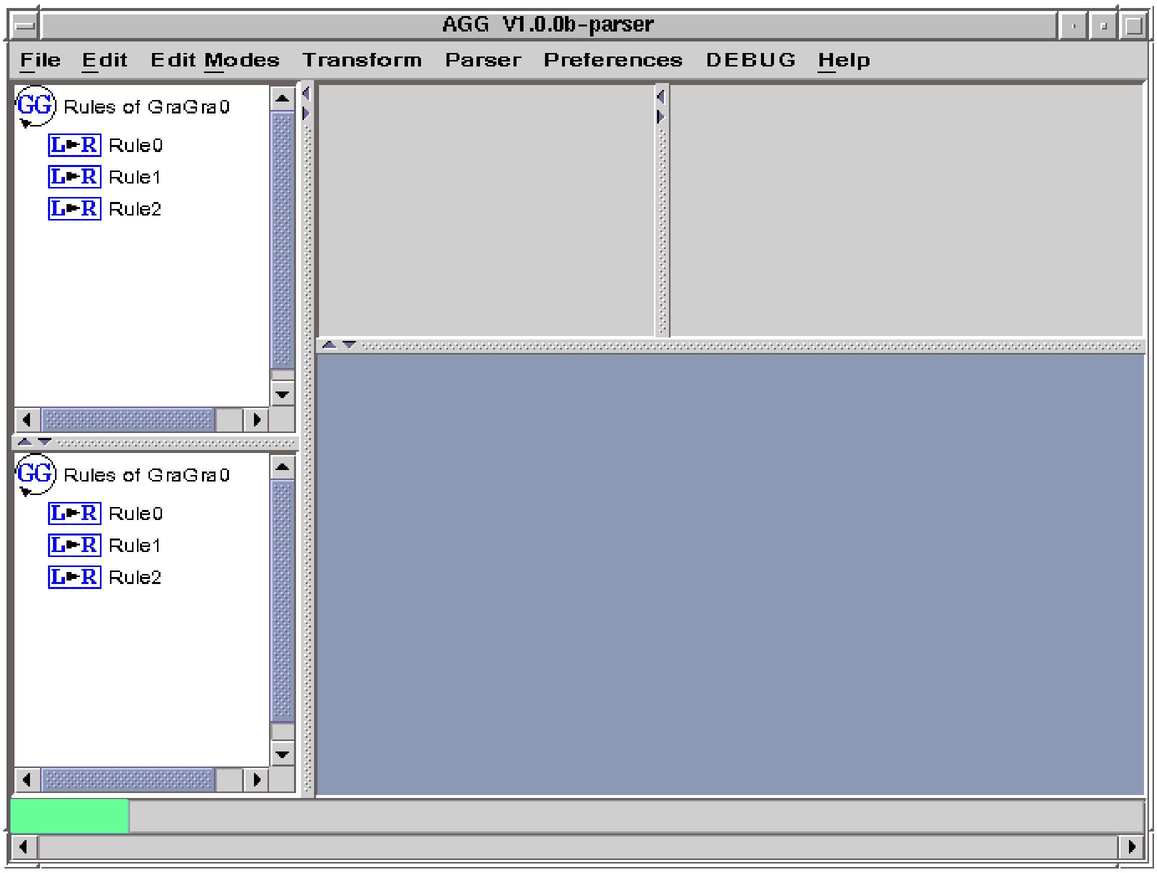

Abbildung 6.7: Der erste Blick auf die kritische Paaranalyse

back im Menü Parser im Untermenü Critical Pair zur Verfügung.

6.2.1 Layer

Wenn der Benutzer die Einstellung *Layer* angewählt hat, so wird vor dem Generieren der kritischen Paare ein Benutzerdialog geöffnet, der die Eingabe der Layer fordert. Der Dialog erscheint allerdings nicht, wenn die kritischen Paare aus einer Datei geladen werden. Dieser Benutzerdialog findet sich in Abbildung 6.10 wieder.

In den drei Abschnitten müssen jeweils die Layer für Regeln und Typen angegeben werden. Initial sind die Layer nicht korrekt. Dieses wird durch ein rotes Kreuz hinter dem Satz: "Are layers correct?". Durch den Knopf Check Layer werden die eingegebenen Layer geprüft. Sind die Layer korrekt, so wird das durch ein grünes Häkchen signalisiert. Der Close-Knopf schließt die Eingabe der Layer.

Sollten die Layer nicht korrekt sein, wenn der Close-Knopf gedrückt wird, dann springen die Einstellungen in die nicht gelayerte Einstellung zurück. Diese Tatsache wird durch ein kleines Nachrichtenfenster dem Benutzer mitgeteilt.

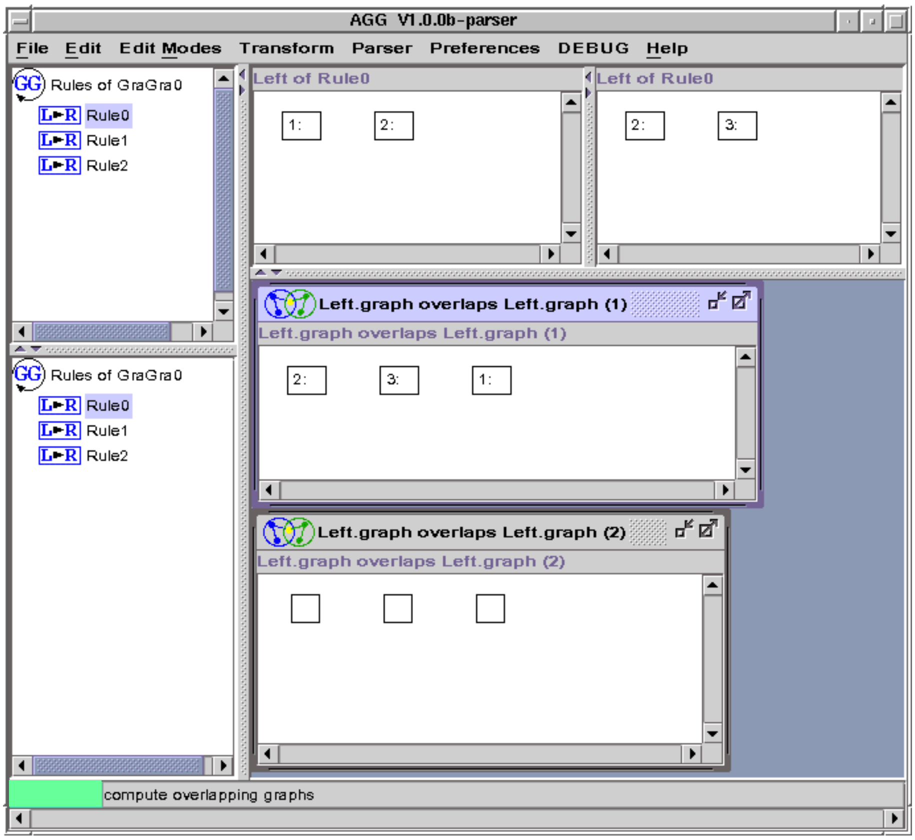

Abbildung 6.8: Ausgewählte Regeln in der kritischen Paaranalyse

Abbildung 6.9: Fenster für konfliktfreie Regeln

6.3 Parser

Soll ein Graph parsiert werden, so sind mindestens drei Daten notwendig: 1. der Arbeitsgraph, 2. der Stopgraph und 3. die parsierenden Regeln. Diese drei Daten werden beim Start des Parsers erfragt. Der Start erfolg durch den Eintrag Parser im Menü Parser.

Um die nötigen Startdaten zu setzen, erscheint ein Benutzerdialog, der in Abbildung 6.11 gezeigt wird.

Da die Daten nicht einzeln gehalten werden können, muß jedesmal eine komplette Graphgrammatik angegeben werden, aus der dann das entsprechende Datum extrahiert wird. D.h. um einen Arbeitsgraphen auszuwählen, muß die Graphgrammatik ausgewählt werden, die den Arbeitsgraphen enthält. Es ist natürlich möglich, daß die Graphgramma-

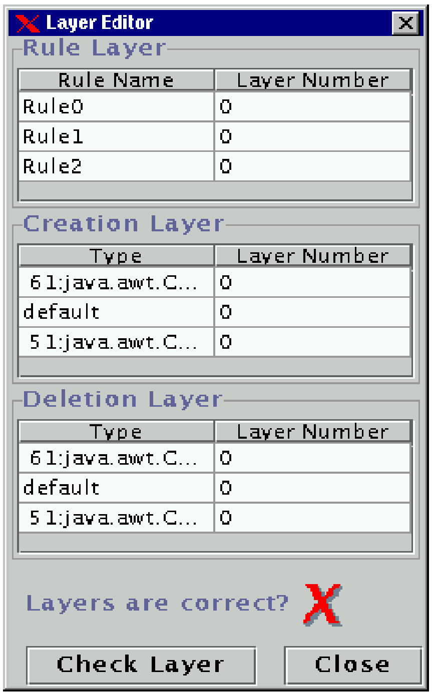

Abbildung 6.10: Editor der verschiedenen Layer

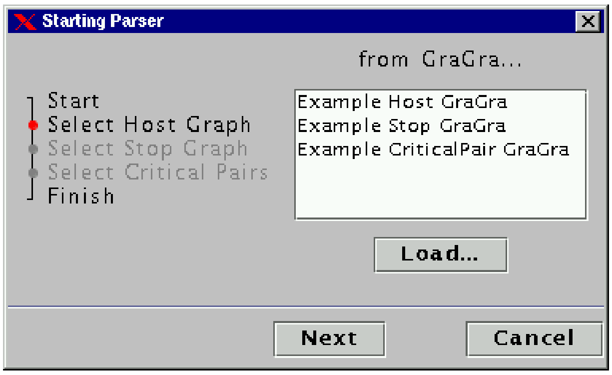

Abbildung 6.11: Startdialog des Parsers

tik sowohl den Arbeitsgraphen als auch die parsierenden Rageln enthält. In dem Fall wird die Graphgrammatik zweimal ausgewählt. Diese Auswahl kann zum einen über das Fenster der aktuell in Bearbeitung stehenden Grammatiken erfolgen. Eine andere Möglichkeit besteht aus dem Laden der Grammatik durch Drücken des Load-Knopfes.

Auf der linken Seite des Benutzerdialog wird der Stand der Eingabe durch einen roten Punkt makiert. Der Cancel-Knopf bricht jeder Zeit die Eingabe ab. Mit dem Knopf Next kann zum nächsten Punkt der Eingabe gesprungen werden. Abgearbeitete Punkte werden durch einen grünen Punkt markiert.

Der Auswahlpunkt *Select Critical Pairs* nimmt eine Sonderstellung ein. Die kritischen Paare stellen eine Einheit mit den parsierenden Regeln dar. Daher muß bei diesem Punkt die Graphgrammatik angegeben werden, die die parsierenden Regeln enthält. Sollte ein Parser in den Einstellungen augewählt worden sein, der keine kritischen Paare benötigt, so ist dieser Punkt die Stelle an der die parsierenden Regeln angegeben werden müssen.

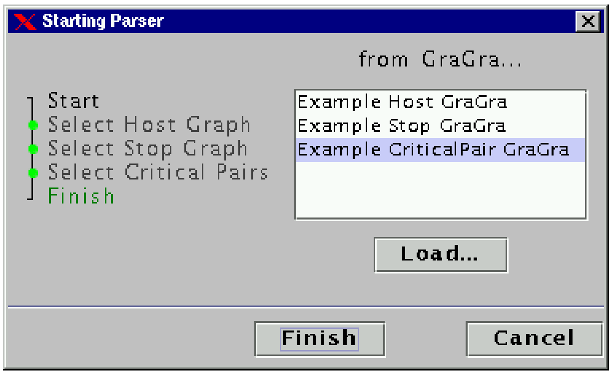

Abbildung 6.12: Startdialog des Parsers am Ende der Eingabe

Abbildung 6.12 zeigt den Eingabedialog am Ende der Eingabe. Alle benötigten Daten wurden eingegeben, das durch einen grünen Punkt angezeigt wird. Der Parser kann nun durch den Knopf Finish endgültig gestartet werden.

Wurden die kritischen Paare nicht geladen, sondern nur eine Grammatik angegeben, von der sie generiert werden sollen, so werden die kritischen Paare vor dem Beginn des Parsierens generiert. Dieser Vorgang kann einen Moment dauern.

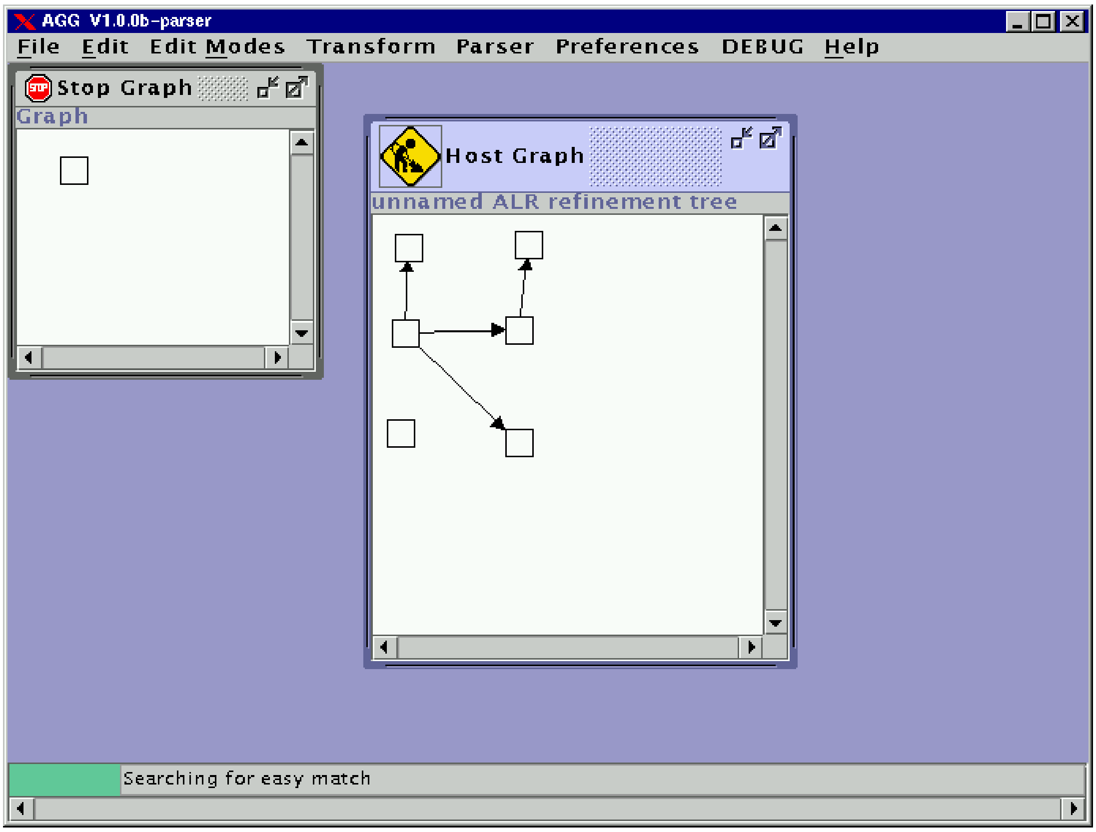

Abbildung 6.13: Die grafische Benutzeroberfläche beim Parsieren

Ist in den Einstellungen nicht das Hintergrundparsieren (*Invisible*) aktiviert, so erscheint die Oberfläche, die in Abbildung 6.13 zu sehen ist. Sollte die Anzeige des Stopgraphen nicht deaktiviert sein, so sind zwei Fenster zu sehen. Die Titel der Fenster bezeichnen ihren angezeigten Inhalt.

Der Benutzer kann nun den Prozeß des Parsierens mitverfolgen.

6.3.1 Layer

Wenn der Benutzer die Einstellung *Layer* angewählt hat, so wird nach dem Auswählen der Grammatik für die parsierenden Regeln ein Benutzerdialog geöffnet, der die Eingabe der Layer fordert. Der Dialog erscheint allerdings nicht, wenn die kritischen Paare aus einer Datei geladen werden. Dieser Benutzerdialog findet sich in Abbildung 6.10 wieder.

In den drei Abschnitten müssen jeweils die Layer für Regeln und Typen angegeben werden. Initial sind die Layer nicht korrekt. Dieses wird durch ein rotes Kreuz hinter dem Satz: "Are layers correct?". Durch den Knopf Check Layer werden die eingegebenen Layer geprüft. Sind die Layer korrekt, so wird das durch ein grünes Häkchen signalisiert. Der Close-Knopf schließt die Eingabe der Layer.

Sollten die Layer nicht korrekt sein, wenn der Close-Knopf gedrückt wird, dann springt der Parser in die nicht gelayerte Einstellung zurčk. Diese Tatsache wird durch ein kleines Nachrichtenfenster dem Benutzer mitgeteilt.

Kapitel 7

Mögliche Erweiterungen

Software ist nicht wie vor Jahren angenommen ein kurzlebiges Produkt. Viel mehr steht man vor dem Problem, daß Software neuen und vor allem gestiegenen Anforderungen angepaßt werden soll. Dieser Voraussicht soll dieses Kapitel Rechnung tragen.

Für die Pflege des Parsers bieten sich drei Erweiterungen. Diese Erweiterungen liegen ihrerseits in der Natur des Designs. Die drei Erweiterungen betreffen den Parseralgorithmus, die Haltung der kritischen Paare (Container) und den kritischen Paaralgorithmus selbst.

7.1 Parser

Durch die klar definierte Schnittstelle des Parsers kann ohne großen Aufwand ein neuer Algorithmus implementiert werden. Die Erweiterung soll exemplarisch an der Klasse LayeredSimpleExcludeParser erklärt werden. Zuvor soll nur nochmal kurz erläutert werden, was die Anforderungen waren.

Der Parseralgorithmus, der auf kritischer Paaranalyse basiert, sollte ein wenig vereinfacht werden. Das Problem war der Teil des Algorithmus, der einen kritischen Ansatz sucht. Sollte kein Ansatz mehr gefunden werden für eine konfliktfreie Regel, so mußte ein Ansatz für eine abhängige Regel gesucht werden. Wenn ein Ansatz gefunden wurde, der einen Konflikt auslöst, kann es nun sein, daß ein weiterer Ansatz existiert, der aber durch den gegebenen Arbeitsgraphen keinen Konflikt verursacht. Diese Suche nach einem weiteren Ansatz ist sehr zeitintensiv. So ist die Überlegung diese teure weitere Ansatzsuche abzuschalten.

Allerdings gibt es noch eine weitere Anforderung. Der Parser muß mit gelayerte Graphgrammatiken arbeiten können.

Diese beiden Erweiterungen müssen in den bestehenden Parser mit kritischer Paaranalyse einfliessen. Dieser Parser existiert bereits in der Klasse ExcludeParser. In dieser Klasse steckt also schon die komplette Verwaltung der kritischen Paare, Graphgrammitik und was der Parser sonst noch benötigt. Der erste Schritt zielt nun darauf ab, die teure Ansatzsuche abzuschalten. Diese Lösung findet sich in der Klasse SimpleExcludeParser wieder. Durch die Verfeinerung der Klasse ExcludeParser ist in der Klasse SimpleExcludeParser kein Verwaltungsarbeit zu leisten (s. auch Abbildung 5.6). Entsprechend verläuft auch der zweite Schritt der Erweiterung. Da die Ansatzsuche nun schon vereinfacht ist, muß für gelayerte Graphgrammatiken die Regelauswahl an die Layer gekoppelt werden. Das Ergebnis findet sich dann in der Klasse LayeredSimpleExcludeParser.

Generell kann also ein Parser entwickelt werden und muß dann nur das Interface Parser implementieren. Einige Dienste, die in einem Parser immer wieder auftreten, wie zum Beispiel die Regelanwendung, befinden sich in der Klasse AbstractParser. Wer diese Dienste nutzen will, der sollte also seinen Parser von AbstractParser ableiten. So ist es auch möglich über mehr spezielle Klassen Erweiterungen einzubauen.

Nach der Erweiterung und Erstellung der neuen Klassen muß jetzt auch noch der Benutzer die Möglichkeit bekommen, die Erweiterungen zu nutzen. Dazu sind zwei konzeptionelle Änderungen nötig. Die eine betrifft die Erzeugung des neuen Objekts, die andere die Auswahl durch den Benutzer.

Um ein neues Objekt zu erzeugen, sollte die Klasse ParserFactory um eine entsprechende Möglichkeit erweitert werden. Im Klartext bedeutet das, die Methode createParser(...) wird erweitert. Allerdings geschieht die Auswahl in dieser Methode durch die Optionen und sein Objekt. Die Klasse ParserOptionen muß also auch um eine entsprechende Möglichkeit erweitert werden. Der Erzeugung des neuen Objekts steht also nichts mehr im Weg.

Der Benutzer benötigt nur noch die Gelegenheit den neuen Parser aus seiner Benutzeroberfläche zu aktivieren. In dem bestehenden AGG ist dafür die Klasse ParserOptionGUI zuständig. Hier muß also in die Auswahlliste des Algorithmus ein benutzerfreundlicher Text eingetragen werden.

7.2 Kritischer Paarcontainer

Für die kritische Paaranalyse sind zwei Aspekte von Interesse. Zum einen benötigt man den Algorithmus zur Berechnung zum anderen einen Container, um die Berechnung zu halten.

An diese Container können je nach Anwendung unterschiedliche Anforderungen gestellt werden. So sind Optimierungen der Zugriffszeit oder des Speicherplatz denkbar. Andere Anwendungen verlangen vielleicht auch den Zugriff auf eine Datenbank.

Auf jeden Fall können unterschiedliche Container zur Verfügung gestellt werden, ohne den Algorithmus der Berechnung zu beachten. Ein Container muß nur generell das Interface PairContainer implementieren, so wie der Container ExcludePairContainer. Wie auch schon beim Parser mußte dieser Container um die Fähigkeit der gelayerten Graphgrammatiken erweitert werden. Dazu mußte lediglich der Aufruf des Algorithmus geändert werden. Dank der Objektorientierung muß sich auch hier die abgeleitete Klasse LayeredExcludePairContainer nicht um die Verwaltung zusätzlicher Daten kümmern.

Die Analogie zum Parser ist auch hier zu erkennen. Es muß auch bei einem neuen Container sichergestellt werden, daß der Benutzer die neue Erweiterung benutzen und entsprechende Objekte erzeugen kann. Die Erzeugung eines neuen Objekt sollte wieder der Klasse ParserFactory überlassen werden. Für die kritische Paare ist allerdings die Methode createEmptyCriticalPairs(...) verantwortlich, so daß hier die Erzeugung des neuen Containers statt findet. Auch diese Methode wird durch ein Optionsobjekt konfiguriert. Die Konsequenz ist also wieder eine Erweiterung der Klasse CriticalPairOption um einen neuen Container. Nun fehlt nur wieder die Möglichkeit des Benutzers diesen Container auch auszuwählen. Dadurch, daß die Benutzeroberfläche der Optionen für die kritische Paaranalyse und die Parsierung konsistent gehalten werden müssen, sind die Erweiterungen in zwei Klassen einzutragen. Die Benutzeroberfläche der Optionen der kritische Paaranalyse steht in der Klasse CriticalPairOptionGUI. Diese Klasse muß um einen benutzerfreundlichen Eintrag erweitert werden. Für die Konsistenshaltung ist auch noch ein Eintrag in der Benutzeroberfläche der Optionen des Parser nötig (ParserOptionGUI).

7.3 Kritischer Paaralgorithmus

Die Erweiterung des kritischen Paaralgorithmus erweist sich als ein klein wenig schwieriger. Diese Tatsache hat damit zu tun, daß für einen speziellen Algorithmus auch ein entsprechender Container gebraucht wird, der mit dem Algorithmus umgehen kann. Aber wenn man davon ausgeht, daß ein entsprechender Container existiert, so muß generell nur das Interface CriticalPair implementiert werden.

Entsprechend wurde auch die Excluderelation umgesetzt. Die Implementierung spiegelt die Klasse ExcludePair wider. Die Erweiterung um gelayerte Graphgrammatik enthält die Unterklasse LayeredExcludePair. Diese Klasse besteht quasi nur aus drei Zeilen. Die erste Zeile verwaltet das Layerobjekt. Die zweite Zeile checkt, ob bei einer Paarberechnung die beiden Regeln den selben Layer haben. Die dritte Zeile ruft dann die Paarberechnung aus der Oberklasse auf.

Da ein neuer Algorithmus an einen Container gebunden ist, erfolgt der Zugriff über den Container. Daher gelten für einen neuen Algorithmus die selben Bedingungen wie für einen Container. Die entsprechenden Bedingungen für eine Erweiterung wurden also schon in 7.2 erklärt.

Kapitel 8

Parsierung von visuellen Sprachen in GenGEd

In den vorherigen Kapiteln wurde das Design und die Implementierung eines Parsers für Graphgrammatiken vorgestellt. In diesem Kapitel wird das Design eines Parsers für visuelle Sprachen behandelt, der auf den vorgestellten Parser aufbaut, beziehungsweise ihn derart erweitert, so daß er für visuelle Sprachen eingesetzt werden kann. Dieser Parser soll von der GenGEd-Umgebung benutzt werden, so daß nicht mehr syntaxgesteuertes Editieren sondern auch Freihand-Editieren in einen durch GenGEd generierten grafischen Editor möglich ist. Zum allgemeinen Verständnis werden wir nachfolgend einen kurzen Überblick über GenGEd geben. Danach werden die Anforderungen an GenGEd betrachtet. Zum Schluß soll der Einsatz des Parsers modelliert werden. Weitere Informationen zu GenGEd befinden sich in [Bar99].

8.1 Einführung in GenGEd

GenGEd hat es sich zum Ziel gesetzt, Softwareentwicklungstechniken zu unterstützen, denen visuelle Sprachen zugrundeliegen. Diese visuellen Sprachen finden sich nicht nur in den Modellierungs- und Designabschnitten der Softwareentwicklung sondern auch in der Programmierung. In der Regel sind visuellen Sprachen statisch in entsprechende Entwicklungsumgebungen integriert. Das bedeutet, eine kleine Änderung der visuellen Sprache zieht eine aufwendige Anpassung der Entwicklungsumgebung nach sich. Die Kosten dieser aufwendigen Anpassung liegen sicherlich in einer zeitintensiven Neuentwicklung sowie der Einarbeitungszeit der Benutzer. Ein Generator für visuelle Entwicklungsumgebungen kann diese Kosten mindern. Solche visuellen Entwicklungsumgebungen beinhalten normalerweise mehrere Werkzeuge, die für verschiedene Zwecke eingesetzt werden können. Zu diesen Werkzeugen werden grafische Editoren gezählt, die das Editieren von visuellen Sätzen einer visuellen Sprache (auch Diagramm genannt) unterstützen.

Der Generierungsprozeß von grafischen Editoren erfolgt im allgemeinen aufgrund einer Spezifikation der visuellen Sprache. In der Literatur findet man bereits einige Ansätze zur textuellen Spezifikation von visuellen Sprachen [MM98], der Schwerpunkt von GenGEd ist jedoch die visuelle Spezifikation. Die Sprachspezifikation soll einen grafischen Editor so konfigurieren, daß Sätze dieser Sprache gezeichnet werden können.

Wie schon erwähnt gibt es nicht nur visuelle Programmiersprachen sondern auch visuelle Sprachen wie Diagrammsprachen, Ikon-Sprachen, um nur einige Arten von visuellen Sprachen zu nennen. Nicht zu letzt können auch Straßenschilder aus Kapitel 1 als visuelle Sprache aufgefaßt werden. GenGEd beschäftigt sich hauptsächlich mit der Syntax der visuellen Sprache nicht mit der Semantik. GenGEd unterscheidet zwischen konkreter und abstrakter Syntax. Die konkrete Syntax kann vereinfacht gesagt als Layout der Sprache angesehen werden. Die abstrakte Syntax hingegen liefert die Basis zur Definition der Semantik.

GenGEds Sprachspezifikationen teilen sich in ein Alphabet und einer Grammatik. Eine algebraische Graphstruktursignatur und ein Constraint-Satisfaction-Problem repräsentiert das Alphabet. Im wesentlichen werden in der Graphstruktursignatur die Sprachelemente und deren Verbindungen spezifiziert, wobei Sortensymbole die abstrakte Syntax der Sprachelemente repräsentiern und einstellige Operationssymbole die der Verbindungen. Die Grammatik wird durch eine algebraische Graphstrukturgrammatik dargestellt, was eine Verallgemeinerung der Begriffe aus Kapitel 2 darstellt.

Im momentanen Stand von GenGEd definieren die Regeln der Grammatik die Editieroperationen des generierten Editors. Dieses syntaxgesteuerte Arbeiten soll mit diesem Kapitel aufgebrochen werden, um ein Freihand-Editieren mit erhöhter Benutzerfreundlichkeit zu ermöglichen. Abbildung 8.1 zeigt die zur Zeit umgesetzte Architektur der GenGEd-Umgebung.

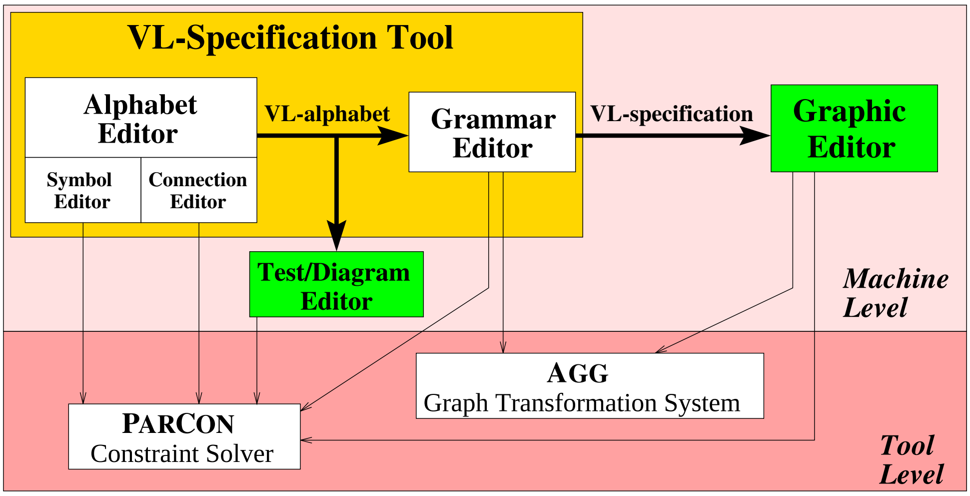

Abbildung 8.1: Die momentane Architektur der GenGEd-Umgebung

Wie in Abbildung 8.1 ersichtlich, besteht GenGEd aus mehreren Editoren und Werkzeugen. Zur Definition des Alphabets werden in dem Alphabeteditor zuerst die Sprachelemente gezeichnet. Die benutzerdefinierten Namen bilden die Typen der abstrakte Syntaxebene, und die Grafiken bilden die konkrete Syntaxebene. Die Sprachelemente sind die Basis zur Definition der Verbindungen. Auf der abstrakten Syntaxebene werden in dem Verbindungseditor für jeweils eine Verbindung zwei Namen ausgewählt und für die konkrete Syntaxebene eine Menge von grafischen Constraints zwischen den jeweiligen Grafiken definiert. Die Definition von grafischen Constraints resultiert in einem Constraint-Satisfaction-Problem, das von dem integrierten Constraintsolver Parcon [Gri96] gelöst wird.

Der Testeditor kann zur Überprüfung eines benutzerdefinierten Alphabets benutzt werden. Dieser Editor erlaubt das typ- und constraintbasierte Editieren von Diagrammen über einem Alphabet. Da diesem Editor noch keine vollständige Sprachspezifikation zugrundeliegt, können hier jedoch auch solche Diagramme gezeichnet werden, die nicht korrekt sind im Sinne der visuellen Sprache.

Basierend auf der Definition des Alphabets wird der Grammatikeditor konfiguriert. Zur visuellen und syntaxgesteuerten Definition der Sprachgrammatik werden automatisch sogenannte Alphabetregeln aus dem Alphabet generiert. Mit Hilfe der sprachspezifischen Alphabetregeln kann die Sprachgrammatik ebenfalls visuell definiert werden. Da die Definition der Sprachgrammatik auf algebraischer Graphtransformation basiert, wird für die zugrundeliegende Transformation das AGG-System [ERT99] eingesetzt. Die jeweiligen Constraint-Satisfaction-Probleme werden wie bei dem Alphabeteditor von Parcon gelöst.

Nachdem die Definition einer visuellen Sprache erfolgreich durchgeführt wurde, kann ein grafischer Editor mit dieser Sprachspezifikation konfiguriert werden, der syntaxgesteuert die Manipulation von Diagrammen unterstützt. Auch dieser Editor benutzt das Agg System für die zugrundeliegende Graphtransformation und Parcon zum Lösen der Constraint-Satisfaction-Probleme.

8.2 Anforderung an GenGEd für die Implementierung

Der in dieser Arbeit vorgestellte Parser verfügt über die Möglichkeit, gelayerte Regeln einer Graphgrammatik zu verwenden. Dieses Schichtenmodell stellt bestimmte Anforderungen an die Regeln einer Graphgrammatik wie zum Beispiel, daß jede Regel mindestens ein Graphobjekt löscht. Die genaue Beschreibung befindet sich in Kapitel 2.8 in Definition 15. Diese Layer müssen allerdings vom Designer einer visuellen Sprache eingegeben werden.

Weiterhin benötigt der Parser eine Graphgrammatik, die zum einen die parsierenden Regeln sowie den Stopgraphen enthält. Der nächste Punkt ist die Grammatik für visuelle Sprachen. Wie schon beschrieben wird in GenGEd ein algebraischer Ansatz verwendet. Es soll hier nicht die Art der Datenstrukturen im Fordergrund stehen sondern die Art der Regeln in der Grammatik. Zum syntaxgesteuerten Editieren sind zwei Arten von Regeln relevant: generierende und editierende Regeln, letztere beinhaltet auch einige Regeln zum Löschen von Sprachelementen und Verbindungen. Beiden Arten ist aber gemein, daß sie im wesentlichen ein neues Diagramm korrekt aufbauen. Ein wesentliches Merkmal und Neuerung für GenGEd ist die Benutzung von parsierenden Regeln. Sie zeichnen sich dadurch aus, daß sie aus einem vorhandenen Diagramm die Sprachelemente und Verbindungen löschen. Wie schon in Kapitel 4 beschrieben ist, können mit Hilfe dieser parsierenden Regeln automatisch kritische Paare berechnet werden. Eine *Parsierungsspezifikation* besteht somit aus folgenden Elementen:

- Die Parsierungsgrammatik, die parsierende Regeln und den Stopgraphen hält.
- Die kritischen Paare, die aus den Regeln der Parsierungsgrammatik generiert werden.
- Die Layerfunktion, wie sie in Definition 15 angegeben ist.

In GenGEd können visuelle Sätze einer visuellen Sprache in den sprachspezifischen grafischen Editor syntaxgesteuert manipiliert werden. Im Vergleich zu den bisher vorgestellten Konzepten entspricht ein visueller Satz somit einem Arbeitsgraphen, auf den die Regeln der entsprechenden Grammatik zur Transformation angewandt werden können. Ein visueller Satz baut sich in GenGEd aus zwei Komponenten auf: der konkreten und abstrakten Syntax. Die Struktur einer visuellen Sprache, auf der der Parser arbeitet, wird in GenGEd durch die abstrakte Syntax repräsentiert. Da der Parser auf attributierter Graphtransformation basiert, erwartet er einen attributierten Graphen zum Parsieren. Daher muß die Struktur, die abstrakte Syntax, die durch eine attributierte Graphstruktur repräsentiert ist, in einen attributierten Graphen umgewandelt werden. Der Umwandlungsprozeß ist ausführlich in Kapitel 6 in [Bar99] beschrieben und soll hier angewendet werden. Für das Parsieren kann die konkrete Syntax vernachläßigt werden.

Die Arbeitsweise von GenGEd ändert sich durch den Einsatz des Parsers radikal. Bisher war es nötig das Alphabet einer visuellen Sprache anzugeben, um darüber die entsprechende Grammatik zu definieren, die einen entsprechenden Editor generiert. Abbildung 8.2 zeigt die neue Architektur der GenGEd-Umgebung. Diese Architektur soll außerhalb dieser Arbeit realisiert werden.

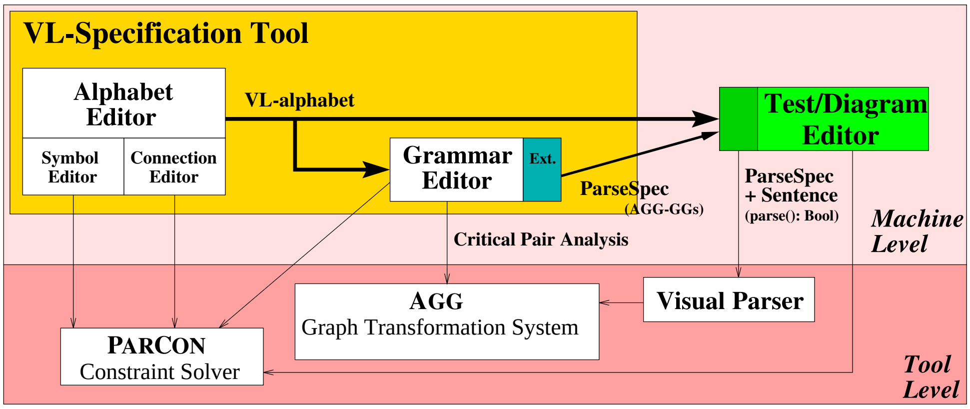

Abbildung 8.2: Die neue Architektur der GenGEd-Umgebung

Die neue Architektur zeichnet sich vor allem dadurch aus, daß kein neuer Editor für eine viselle Sprache generiert werden muß. Der Diagrammeditor kann mit einem definierten Alphabet so konfiguriert werden, daß ein Editor für eine visuelle Sprache zur Verfügung steht. Der Diagrammeditor kann damit parallel zur Entwicklung einer Grammatik für die visuelle Sprache benutzt werden. Für die Parsierung ist es nur notwendig eine Parsierungsspezifikation nachträglich zu laden. Der Diagrammeditor muß nur noch den eingebenen visuellen Satz in einen attributierten Graphen konvertieren. Um für das Editieren die Rechnerresourcen besser zu nutzen, ist es von Vorteil einen attributierten Graphen erst kurz vor dem Parsieren zu generieren. Eine kontinuierliche Erstellung eines Graphen während des Editierens, hat zum Nachteil, daß diese ganzen Daten während des Editierens erstellt werden müssen und somit den Prozessor belasten. Außerdem müssen diese zusätzlichen Daten während des Editierens mit gehalten werden, das einen Mehrverbrauch an Arbeitsspeicher bedeutet. Bei der Generierung der Daten kurz vor dem Parsieren wird dagegen nur kurz der Prozessor und der Arbeitsspeicher belastet.

Damit ist aber auch klar, daß der Diagrammeditor einen Benutzerdialog benötigt, der diese Generierung und danach den Parser startet. Dabei können zwei Situationen unterschieden werden: der erste und der wiederholte Aufruf des Parser. Der Aufruf des Parsers benötigt eine Graphgrammatik, einen Arbeits- und einen Stopgraphen. Für den Stopgraphen gibt es eine einfache Lösung: Er ist Teil der Parsierungsgrammatik. Die Grammatik kann oder besser sollte sogar schon vorher entwickelt und in einer Art Bibliothek zur Verfügung gestellt werden. Somit muß der Benutzer zum Parsieren nur nachträglich die Grammatik laden. Viel einfacher wird das Szenario bei wiederholtem Parsieren. Das einzige das Interaktion vom Benutzer forderte zwar die Grammatik, die jetzt aber schon bekannt ist. Das wiederholte Parsieren kann nun in einer zweiten Arbeitssitzung ausgeführt werden. Dem Benutzer war beim letzten Speichern nun schon die Parsierungsspezifikation bekannt.

Es ergibt sich nun ein Aspekt, die Datenhaltung, der noch diskutiert werden muß. Der Inhalt des Diagrammeditors kann auf zwei Arten abgespeichert werden. Zum einen ist es möglich den visuellen Satz mit der Parsierungsspezifikation in einer Datei zu halten. Ein anderer Weg wäre, zusätzlich zum visuellen Satz nur einen Verweis aus die Datei der Parserspezifikation zu halten. Die Vor- und Nachteile liegen offen auf der Hand. Im ersten Verfahren muß eine große Datei im zeiten Verfahren zwei kleine Dateien verwaltet werden. Wird nur eine Datei gehalten, so ist damit ein erhöhter Plattenplatzverbrauch verbunden. Besonders bei mehreren Benutzern wird klar, daß jeder eine identische Parsierungsspezifikation mit hält. Es existieren also bei n Benutzern auch n Spezifikationen. Allerdings sollte diese Art der Datenhaltung sehr zuverlässig sein, da zu einem bearbeiteten Diagramm auch die Spezifikation exisitiert. Zum anderen ist es auch weniger kompliziert nur eine Datei zu verwalten. Dem gegenüber steht die Verwaltung zweier Dateien, deren Verwaltung doch wohl etwas mehr Arbeitsaufand fordert. Auch die Verfügbarkeit getrennter Dateien birgt Risiken. Sollte der Server, auf dem die Parsierungsspezifikation abgelegt ist, ausfallen, so kann kein Benutzer sein Diagramm überprüfen. Allerding werden hier die Festplattenresourcen geschont. Bei n visuellen Sätzen einer Sprachspezifikation existieren nur n Dateien mit diesen Sätzen und nur eine Datei, die für alle Sätze die Parsierungsspezifikation hält. Es gibt aber noch einen weiteren Vorteil für die Benutzung zweier Dateien. Denkt man an das Beispiel der Verkehrsschilder, so wird deutlich, daß eine Sprachänderung zum Beispiel durch eine Gesetzesänderung nur die Aktualisierung einer Datei nach sich zieht. Wird nur eine Datei benutzt, so muß ein geeigneter Aktualisierungsmechanismus existieren, um die Parsierungsspezifikation auszutauschen. Auf jeden Fall müssen sich Parsierungsspezifikation und visueller Satz im Arbeitsspeicher befinden, dann kann das Parsieren direkt nach dem Generieren des Arbeitsgraphen gestartet werden.

Nachdem schon soviel über die Handhabung der Parsierungsspezifikation geschrieben wurde, wird es Zeit etwas näher auf die Erstellung der selbigen einzugehen. GenGEd stellt für diesen Zweck den Grammatikeditor zur Verfügung. Auf den ersten Blick scheint dieser Editor auch für die Erstellung von parsierenden Graphgrammatiken ausreichend zu sein. Nun kommt jedoch das Aber. Der Parser ist in der Lage Graphgrammatiken mit geschichteten Regeln zu verwenden. Diese Schichtung der Regeln fordert daher eine zusätzliche Interaktion des Benutzers. Zudem ist auch der Benutzerdialog von AGG abgebildet in Abbildung 6.10 für diese Zwecke nicht völlig geeignet. Die AGG-intern benutzten Typbezeichnungen sind dem Benutzer von GenGEd völlig fremd. Allerdings reicht diese Eingabe vom Konzept her völlig aus. Lediglich die Darstellung der GenGEd-Typen muß erreicht werden. Weiterhin kann der AGG-Parser mit kritischen Paaren umgehen. Diese Fähigkeit kann allerdings dem Benutzer von GenGEd weitestgehend verborgen bleiben.

So ist denkbar, daß nach der Erstellung der Regeln die kritischen Paare generiert werden und zusammen mit der Grammatik in der Parsierungsspezifikation abgespeichert werden.

8.3 Beispiel

Das Beispiel soll die Erstellung einer parsierenden Grammatik verdeutlichen. Das Beispiel ist ein Klassendiagramm nach UML, das auf Pakete und Klassen mit Assoziationen beschränkt wurde.

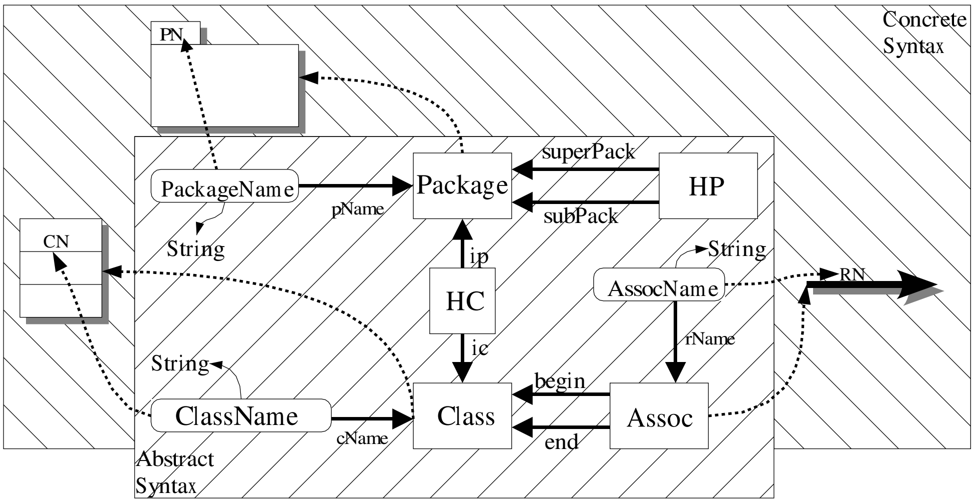

Abbildung 8.3: Darstellung eines Klassendiagramms in GenGEd

Die Abbildung 8.3 zeigt das Alphabet zu diesem Beispiel. Dieses Alphabet beschreibt ein einfaches Klassendiagramm, das aus Paketen und Klassen mit Assoziationen besteht. Pakete werden in der abstrakten Syntax durch den Knoten *Package* repräsentiert. Weiterhin können Pakete in einer Baumstruktur verschachtelt werden. Der Platzhalter *HP* wird für diese Schachtelung benutzt. Außerdem bietet dieser Platzhalter die Möglichkeit offen zu lassen, mit welchem Paket ein anderes Paket verbunden wird. Die Klassen, die durch die Knoten vom Typ *Class* dargestellt werden, sind über den Platzhalter *HC* einem Paket zugeordnet. Auch dieser Platzhalter *HC* sorgt dafür, daß Klassen flexibel einem Paket zugeordnet werden können. Das letzte Sprachelement ist die Assoziation, die durch den Knoten *Assoc* repräsentiert wird. Assoziationen können nur Klassen zugeordnet werden. Außerdem besitzt jedes Sprachelement noch einen Bezeichner, der durch ein Attribut (ClassName für Class, PackageName für Package, AssocName für Assoc) vom Typ String repräsentiert wird. Das Aussehen, die konkrete Syntax, wird der abstrakten Syntax durch einen gestrichelten Pfeil in Abbildung 8.3 zugeordnet.

Wie oben schon erwähnt, ist die konkrete Syntax nicht weiter relevant. Sie wird daher in den Regeln des Beispiels nicht weiter betrachtet. In diesem Beispiel ist es möglich, daß ein Paket des Klassendiagramms mehrere Unterpakete besitzen kann. Außerdem wird gefordert, daß jede Klasse in einem Paket einen eindeutigen Namen hat. In diesem Beispiel werden vier Regeln benötigt, um ein Klassendiagramm zu parsieren. Sie sollen kurz vorgestellt werden, bevor sie detailliert beschrieben werden.

Delete Package: löscht ein leeres Paket.

Delete Sub-Package: löscht ein leeres Subpaket.

Delete Class: löscht eine Klasse.

Delete Assoc: löscht eine Assoziation.

Diese Regeln entsprechen damit genau der Art, wie oben gefordert ist. Der Charakter der Regel zeichnet sich durch das Löschen eines Sprachelements aus.

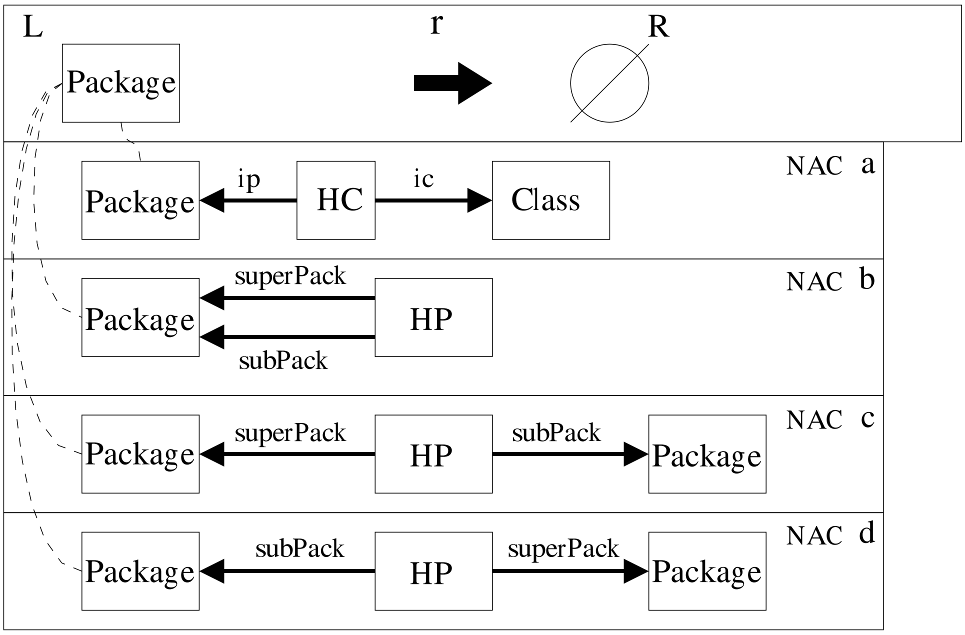

Abbildung 8.4: Regel: Delete Package

Die erste Regel Delete Package wird in Abbildung 8.4 dargestellt. Diese Regel soll Pakete aus einem Klassendiagramm löschen. Diese Pakete dürfen natürlich nur dann gelöscht werden, wenn sie leer sind. Daß ein Paket leer ist, bedeutet, daß keine Klasse bzw. Unterpaket mehr in diesem Paket existiert. Aber genauso wenig soll das zu löschende Paket Unterpaket eines anderen sein. Diese Aussagen werden durch die NACs a, c, d in Abbildung 8.4 repräsentiert. Die NAC b soll sicherstellen, daß ein Paket nicht Unterpaket zu sich selbst ist. Anhand dieser NAC wird auch eine Eigenschaft der benutzten Graphtransformation deutlich. Es werden für die NACs injektive Ansätze gesucht. Die beiden NACs c und d zeigen eine Eigenschaft von GenGEd, die im folgenden unterschieden wird. Wegen dem algebraischen Ansatz läßt GenGEd nur vollständige Diagramme zu. Um auszudrücken, daß ein Paket ein Unterpaket von irgendwem ist, würde es für die Graphtransformation ausreichen, einen Knoten *Paket* und einen Knoten *HP* durch eine Kante *subPack* zu verbinden. Diese Darstellung ist aufgrund des algebraischen Ansatzes in GenGEd kein erlaubtes Diagramm über dem Alphabet.

Die zweite Regel in Abbildung 8.5 soll alle Unterpakete eines Diagramms löschen. Dabei sollen die selben im wesenlichen alle Bedingungen wie in der ersten Regel gelten. Daher sind auch die NACs a - c identisch. Lediglich Die NAC d enthält eine kleine Änderung. Da

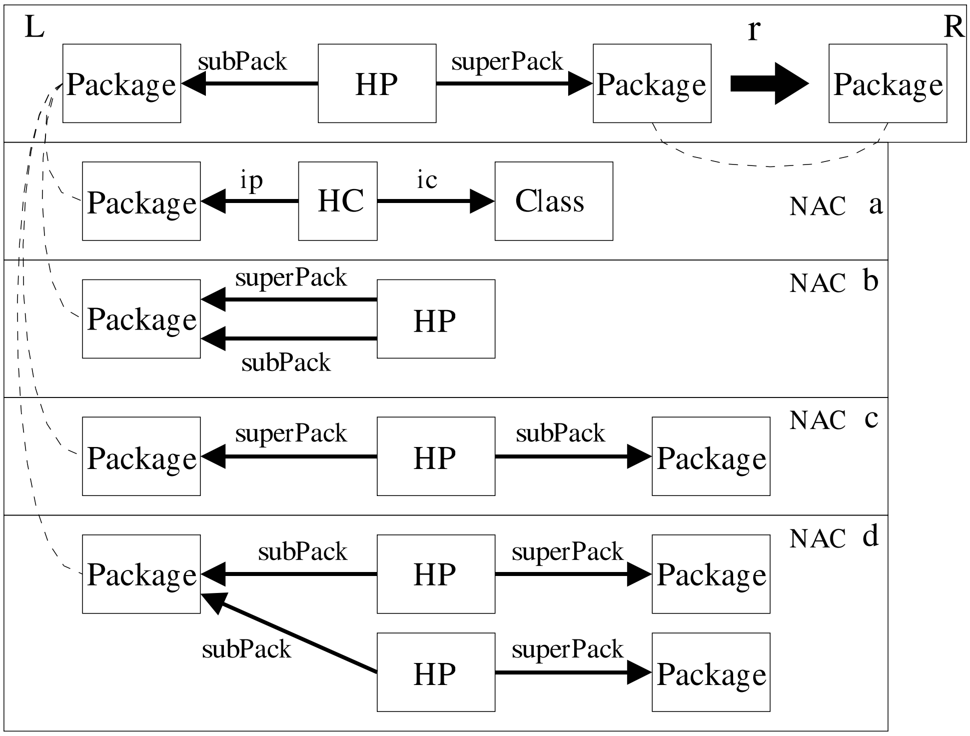

Abbildung 8.5: Regel: Delete Sub-Package

in dieser Regel Unterpakete gelöscht werden sollen, ist es falsch zu fordern, das zu löschende Paket darf kein Unterpaket sein, wie es in der ersten Regel mit NAC d ausgedrückt wird. Ein Paket aber kann nur direktes Unterpaket eines Oberpakets sein. Somit sind Baumstrukturen möglich aber keine gerichteten azyklischen Graphen. Mit diesen ersten beiden Regeln sollen beliebig tief verschachtelte Pakete gelöscht also erkannt werden.

Die dritte Regel in Abbildung 8.6 soll Klassen löschen. Ensprechend dem Alphabet muß eine Klasse immer genau einem Paket zugeordnet sein. Dieser Sachverhalt wird durch den Knoten *HC* mit den Kanten *ip* und *ic* beschrieben. Nun gibt es dazu die weitere Forderung, daß eine Klasse nur genau einem Paket zugeordet sein darf. Diese Forderung rechtfertigt die Existenz von NAC a. Klassen sollen in ihren Paket durch einen eindeutigen Namen identifiziert werden. Was bedeutet das? Nun, es dürfen nicht zwei Klassen in einem Paket existieren, die den gleichen Namen tragen. Findet man also zwei Knoten *Klasse* deren Attributbelegungen im Klassenname gleich sind, so ist die Eindeutigkeit nicht mehr gegeben. Genau das repräsentiert NAC b. Die NACs c und d sind im wesentlichen unspektakulär, da sie nur sicherstellen sollen, daß die zu löschende Klasse nicht mit anderen Klassen über Assoziationen verbunden ist.

Die vierte Regel in Abbildung 8.7 löscht Assoziationen und scheint einfach aufgebaut zu sein. Auf den ersten Blick betrachtet löscht sie nur Relationen zwischen zwei Klassen. Dabei werden nicht einmal irgendwelche Anforderungen an die Klassen oder Relationen gestellt. Betrachtet man aber nochmal die ersten drei Regeln so erkennt man, daß keine dieser Regeln angewendet werden kann, wenn eine Relation existiert. Fazit für die vierte Regel ist also: Sie ist die erste Regel, die beim Parsieren eines Klassendiagramms angewendet wird.

Bei genauer Betrachtung fällt auf, daß in diesem Beispiel zwei Platzhalter, die Knoten

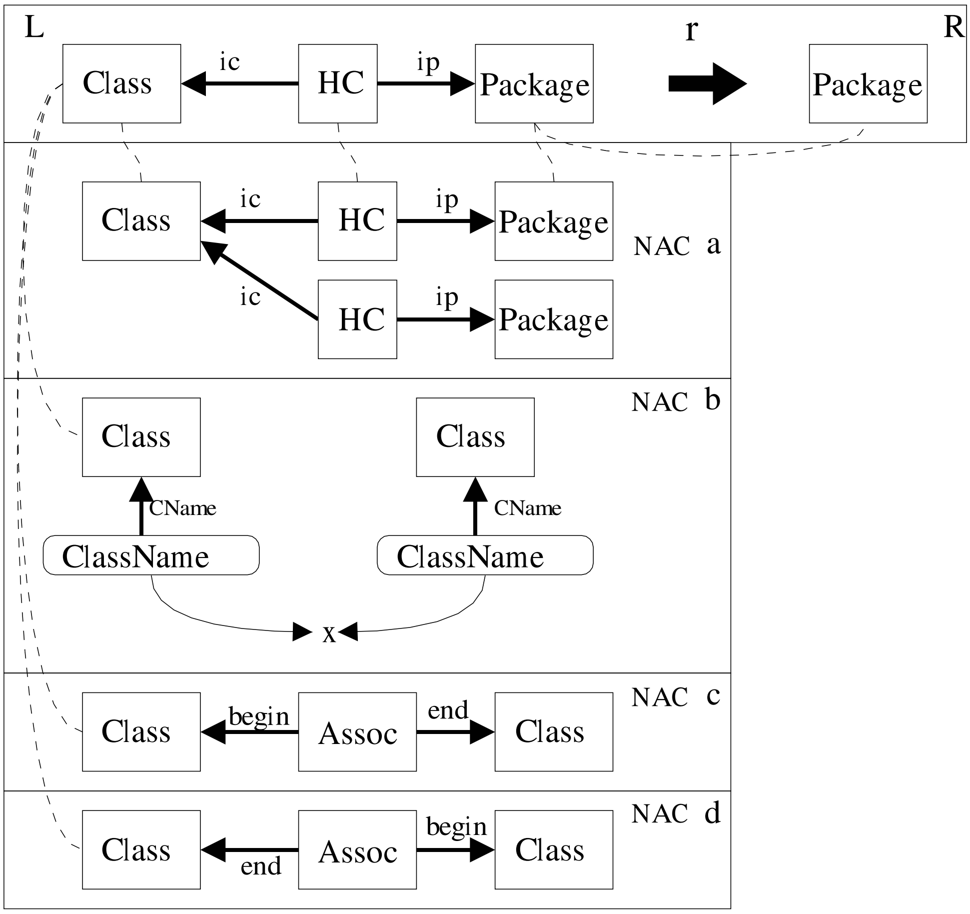

Abbildung 8.6: Regel: Delete Class

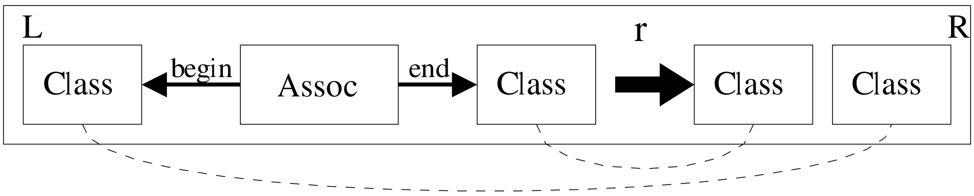

Abbildung 8.7: Regel: Delete Assoc

HP und *HC*, verwendet werden. Sie müssen eingefügt werden, da sonst mit GenGEd einige Forderungen der spezifizierten Sprache nicht umgesetzt werden können. Ersetzt man zum Beispiel den Knoten *HC* und die Kanten *ip* und *ic* durch eine Kante *isIn* vom Knoten *Class* zum Knoten *Package*, so läßt sich die Forderung, daß eine Klasse nur genau einem Paket zugeordnet wird nicht mehr als visuelle Sprache modellieren. Aber daher darauf zu schliessen, daß es sinnvoll ist, wenn nicht gar notwendig, daß keine direkten Beziehungen zwischen zwei Sprachelementen herzustellen, also nur ein Operationssymbol zu verwenden, ist falsch. Erweitert man dieses Beispiel um Stereotypen, die als eigene Knoten mit entsprechendem Attribut dargestellt werden, so können zum Beispiel beliebig viele Stereotypen an einer Klasse durch nur jeweils ein Operationssymbol *extends* hängen.

Kapitel 9

Zusammenfassung

In dieser Arbeit wurde vorgestellt, wie ein Parser basierend auf Graphtransformation arbeitet. Dabei wurde auch die Optimierung mittels kritischer Paaranalyse betrachtet, die sogar als eigenständige Komponente aufgefaßt werden kann. Die kritische Paaranalyse bietet dafür eine eigene graphische Benutzeroberfläche an. Ein weiterer Punkt der Optimierung war die Einführung des Schichtenmodells, die genannten Layer. Das Problem das beim Parsieren auftretende Wortproblem zu lösen, besteht vor allem darin, daß der Algorithmus nicht terminiert. Das Schichtenmodell wie es in Kapitel 2.8 definiert ist, schafft Abhilfe und stellt sicher, daß der Parser terminiert. Somit wurde durch diese Arbeit ein mächtiges Werkzeug zur Grapherkennung realisiert.

Die Graphtransformation wird in [Bar99] als Werkzeug für die Spezifikation visueller Sprachen benutzt. Durch die vorliegende Arbeit werden die Möglichkeiten für GenGEd stark erweitert, denn der Einsatz von Freihandeditoren ermöglicht eine erhöhte Transparenz der benutzten visuellen Sprache. Das Erlernen der sprachspezifischen Editierkommandos kann völlig entfallen. Ohne die Fähigkeiten eines solchen Editors zu schmälern, kann sich ein sprachspezifischer Editor wie ein gewöhnlicher Graphikeditor mit speziellen Formen gemäß eines Alphabets präsentieren. Ein Entwickler, der auf verschiedenen visuellen Sprachen angewiesen ist, kann somit effizienter arbeiten. Der Editor muß nur das entsprechende Alphabet laden.

Dieser Vorgang zeigt auch eine weitere Mächtigkeit des Parsers. Obwohl komplexe visuelle Sprachen zugrundeliegen, wird der Benutzer nicht mit deren spezifischen Eigenheiten während des Editierens konfrontiert. An dem Beispiel des Entwurfs eines Verkehrsschildes kann diese Eigenschaft verdeutlicht werden. Die Bestimmungen und Richtlinien aus der StVO werden völlig in die Spezifikation der visuellen Sprache übertragen. Man denke zum Beispiel an die Vorschrift ein Zusatzschild muß sich dicht unter einem Verkehrszeichen befinden. Der Designer eines neuen Verkehrsschildes sollte dieses Gesetz zwar kennen, sollte er es aber vergessen, so kann der Entwurf durch den Parser als fehlerhaft erkannt werden.

Im Laufe der Zeit ändern sich Gesetze. Solchen Änderungen ist leicht mit der Anpassung der visuellen Sprache beizukommen. Eine Änderung der Anforderung an den Parser ist schon eine größere Herausforderung. Vorausschauend wurden Design Pattern eingesetzt. Sie versprechen eine höhere Flexibilität und Wiederverwendbarkeit. Gerade diese Flexibilität und Wiederverwendbarkeit sind es, die die spätere Wartung von Software erleichtern. So wurden im Kapitel 7 schon einige sinnvolle Erweiterungen aufgezeigt.Diese Erweiterungen betreffen im wesentlichen den hier entwickelten Parser, für dessen Algorithmen am ehesten Fortschritte zu erwarten sind, sobald entsprechende Ergebnisse der Forschung vorliegen.

Literaturverzeichnis

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[Bar99] R. Bardohl, GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation.* Dissertation, Technische Universität Berlin, Fachbereich Informatik, 1999. Veröffentlicht durch Dr. Kovac Verlag, Hamburg, 2000.

[BST00] P. Bottoni, A. Schürr, and G. Taentzer, Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Langversion als Technischer Bericht SI-2000-06, University of Rom.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In *Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 1: Foundations*, pages 247–312. World Scientific, 1997.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer, The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. 1995.

[Gri96] P. Griebel. *ParCon - Paralleles Lösen von grafischen Constraints.* PhD thesis, Paderborn University, February 1996.

[LKW93] M. Löwe, M. Korff, und A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley & Sons Ltd, 1993.

[Mel98] B. Melamed. *Design and Realisation of an Attribute Manager for Conditional and Distributed Graph Transformation.* Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1998.

[MM98] K. Marriott and B. Meyer, editors. *Visual Language Theory.* Springer, New York, 1998.

[OMG01] Object Management Group, `http://www.omg.com`, 2001

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Vol 1: Foundations.* World Scientific, 1997.

[Rud97] M. Rudolf. *Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation.* Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1997. `http://tfs.cs.tu-berlin.de/publikationen/public1997.html`

[Sof01] Softera, `http://www.softera.com`, Softmodeler V2.5, 2001

[Sun00] Sun Microsystems, `http://java.sun.com/j2ee/j2sdkee/` Java Development Kit, 2000

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den