Konzeption und Implementierung eines Verfahrens zum Nachweis der Konsistenz in einer attributierten Graphgrammatik

Diplomarbeit
bei Prof. H. Ehrig

Berichter
Dr. Gabriele Taentzer

vorgelegt von Michael Matz
am Fachbereich Informatik der
Technischen Universität Berlin

Berlin, 12. August 2001

Inhaltsverzeichnis

Abstract

Das Ziel dieser Arbeit ist die Konzeption und Implementierung eines Verfahren, mit dem Konsistenz in einer attributierten Graphgrammatik nachgewiesen und zugesichert werden kann. Die Implementation erfolgt im Rahmen des `AGG` Systems, welches bereits Mittel zum Erstellen und Simulieren von Graphgrammatiken bereitstellt. Dazu werden bekannte Verfahren für den Single-Pushout-Ansatz um Attributierung erweitert, wobei sich herausstellt, dass sie fast ohne Änderung übernommen werden können. Im Hinblick auf Implementierbarkeit wird jedoch in einigen wesentlichen Teilen von jenen Verfahren abgewichen. Namentlich werden Nachbedingungen nicht vorgezogen, anstattdessen jedoch können bestimmte Verklebungen weggelassen werden. Diese theoretischen Ergebnisse werden in Java implementiert und in `AGG` integriert, und anschließend einem kurzen Test unterworfen. Der Software-Teil, also Sourcecode, Kompilat und Beispieleingabedateien befinden sich auf der CD-ROM, die Teil dieser Arbeit ist.

Überblick

Der schriftliche Teil der Arbeit gliedert sich in folgende Kapitel:

- In Kapitel 1 wird eine informelle Einführung in attributierte Graphen und Graphtransformation, sowie in Integritätsbedingungen und Konsistenz gegeben.
- Kapitel 2 stellt dies auf eine theoretische formale Grundlage, soweit sie für die Implementierung und das nächste Kapitel benötigt wird.
- Kapitel 3 baut darauf auf und stellt das implementierte Verfahren vom theoretischen Standpunkt vor. Dort wird auch der Beweis für dessen Korrektheit geführt.
- Kapitel 4 beschreibt die Konzeption des Softwareteils, samt den Designentscheidungen, und die Implementation anhand der UML-Diagramme der neuen Klassen, sowie anhand deren Beschreibung. Es wird darauf eingegangen, wie das GUI aussehen soll, und wie es implementiert ist. Es stellt also gleichzeitig die Dokumentation für Benutzer dar.
- Kapitel 5 beinhaltet die Beschreibung zweier Testläufe für spezielle Graphgrammatiken, sowie einige daraus resultierende Feststellung hinsichtlich Benutzbarkeit und Performance.
- Kapitel 6 schliesslich fasst die Ergebnisse zusammen, und gibt noch einmal eine Zusammenfassung der offenen Fragen, bzw. Vorschläge für untersuchenswerte Veränderungen.

Kapitel 1

Attributierte Graphen und Konsistenz

Dieses Kapitel stellt eine Einführung in attributierte Graphen, Graphersetzung, Graphgrammatiken und Konsistenz dar. Die hierin vorgestellten Konzepte werden im nächsten Kapitel formalisiert, weshalb hier weitgehend darauf verzichtet wird.

Das Ding auf dem gearbeitet wird ist ein attributierter Graph. Ein Graph an sich zeichnet sich durch den Besitz von Knoten und die diese Knoten verbindenden Kanten aus, die bei uns gerichtet sind. Zwischen zwei Knoten kann mehr als eine Kante existieren. Alle Graphelemente (Knoten und Kanten) haben einen Typ. In `AGG` werden attributierte Graphen verwendet, die pro Knoten und Kante eine Menge von Attributen haben können.

Ein Element einer solchen Attributmenge, also ein Attribut, besteht aus Typ, Name und Wert. Formal ist ein solchen Attribut Element einer Algebra, was hier jedoch noch nicht interessieren soll. Werte von Attributen sind generell Terme. In machen Kontexten sind bloss atomare Terme (Konstanten und Variable), in manchen sogar bloss Konstanten als Wert zugelassen [1]. Dabei muss dieser Term natürlich den korrekten Typ besitzen.

Der Typ eines Graphelementes besteht aus Typen und Namen der Elemente seiner Attributmenge und einem Typnamen. In `AGG` wird zusätzlich noch Färbung und Form (ausgefüllt, rund, eckig usw.) der Graphelemente zur Typbestimmung herangezogen.

Abbildung 1.1 zeigt ein Beispiel für einen attributierten Graphen. Die beiden rechteckigen Knoten haben den selben Typ, im rechten wird für das Attribut "zahl" die Variable x verwendet, um ovalen Knoten wird mit dieser Variable für Attribut "num" der Term $(x + 1) * 2$ gebildet.

[1] In `AGG` kann der Wert auch ganz weggelassen werden, das Attribut ist dann also 'leer'

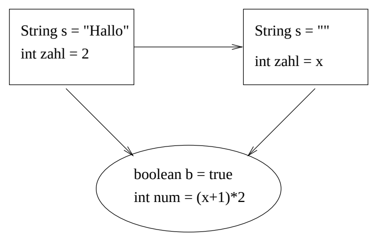

Abbildung 1.1: Attributierter Graph

Diese attributtierten Graphen sind also die Objekte auf denen wir arbeiten wollen, wir brauchen also noch ein Mittel, um sie verändern oder Aussagen über sie treffen zu können. Also um mind. zwei solcher Graphen in Beziehung stellen zu können. Dieses Mittel sind Abbildungen zwischen zwei Graphen, genannt Morphismen. Gegeben zwei attributierte Graphen (Original und Image) gibt ein Morphismus zwischen beiden an, ob und wie Graphelemente von Original auf Graphelemente von Image abgebildet werden.

Ein solcher Morphismus muss ein paar Mindestanforderungen erfüllen und kann darüber hinaus noch weitere Eigenschaften haben. Zu erstgenanntem gehört die Tatsache, daß Source und Target einer abzubildenden Kante auf Source und Target deren Abbilds abgebildet werden. Weiterhin können ausschließlich Graphelemente desselben Typs aufeinander abgebildet werden.

Morphismen sind normale Abbildungen, können also alle besonderen Eigenschaften von ihnen haben, z.B. könenn sie partiell, total, surjektiv oder injektiv sein.

Etwas Beachtenswertes ist die Abbildung von attributierten Graphelementen aufeinander. Hierbei wird eine Termidentifikation vorgenommen, man ändert also unter Umständen die Algebra der Gesamtheit von Source, Target und Morphismus in eine Quotiententermalgebra. Dies ist nur in bestimmten Kontexten zugelassen. Ein Sonderfall ist z.B. wenn eine Konstante auf eine andere abgebildet wird, und die Algebra keine Termalgebra ist. In diesem Fall müssen beide Konstanten denselben Wert haben.

Ein Beispiel für einen Morphismus zeigt Abb. 1.2. Der gestrichelte Pfeil gibt an, welche Knoten auf welche abbgebildet werden. Ausserdem haben aufeinander abgebildete Graphelemente dieselbe Zahl ("1:" im Beispiel) inbeschrieben. Der Morphismus im Bild ist total und injektiv.

Das Mittel um Graphen verändern zu können sind Regeln. Eine Regel besteht aus zwei Graphen, der linken und rechten Seite, und einem Morphismus zwischen beiden. Regeln kann man auf einen Graph (den so genannten

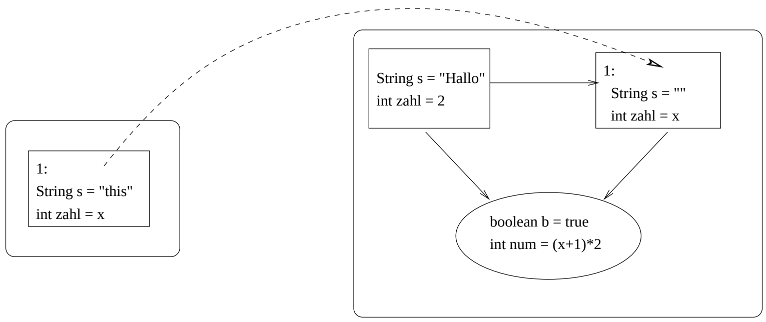

Abbildung 1.2: attributierter Morphismus

Hostgraph) anwenden, indem man die linke Regelseite im Hostgraphen sucht (möglicherweise gibt es mehrere Vorkommen, dann entscheidet man sich für eins), und dann genau diese Graphelemente durch die rechte Regelseite ersetzt. Dieses Suchen der linken Seite wird durch Suchen eines Morphismus von linker Regelseite in Hostgraphen simuliert, dann weiss man schon mal, welche Graphelemente von linker Seite mit welchen des Hostgraphen korrespondieren sollen. Dieser Morphismus wird Match genannt, und muss ein paar Anforderungen erfüllen. So muß er z.B. total sein, damit auch die **komplette** linke Seite im Host gefunden wird. Hat man also einen Match kann man die Regel anwenden, indem man nun für Elemente der linken Seite bestimmt ob und wie sie durch Elemente der rechten ersetzt werden (durch den Regelmorphismus). Elemente der rechten Seite die kein Urbild im Regelmorphismus haben, kommen einfach zum Hostgraph dazu, Elemente der linken Seite, die nicht durch die Regel abgebildet wurden werden gelöscht.

Eine Erweiterung zu normalen Regeln sind Anwendungsbedingungen. Sie stellen eine Bedingung dar, wann ein Match für eine Regel gültig ist. Falls die Bedingungen nicht erfüllt sind, darf die Regel auf diesem Hostgraphen mit diesem Ansatz nicht angewandt werden. Eine oft benutzte Anwendungsbedingung sind negative Anwedungsbedingungen (NAC von engl. negative application condition). Sie bestehen aus einem Graphen und einem Morphismus von linker Regelseite in diesen Graphen. Die Semantik einer NAC ist, dass man eine Regel an einem Match nur dann anwenden darf, wenn man an dieser Stelle nicht auch den Graphen der NAC im Hostgraph findet. (Das genaue Mittel sind wieder Morphismen die bestimmten Eigenschaften genügen müssen, siehe nächstes Kapitel)

Die Behandlung von Attributen ist wieder eine Betrachtung wert. Die Algebra der linken und rechten Seite ist die Termalgebra, in den Attributen sind nur Konstanten zugelassen, also Terme. Im Hostgraphen ist eine beliebige Algebra zugelassen, Attribute sind ebenfalls bloss Konstanten. Der Match identifiziert nun Terme der linken Seite mit Konstanten des Hostgraphen.

Anstatt sich jetzt allerdings mit Quotiententermalgebren und Algebrahomomorphismen herumzuplagen stelle man sich dies lieber so vor, dass diese Termidentifikation in eine Variablenbelegung, die allen Variablen der linken Seite eine Konstante in der Hostgraphalgebra zuweist, mündet. Damit sind dann auch die Terme der rechten Regelseite zu Konstanten (ebenfalls der Hostgraphalgebra) auswertbar, und genau diese sind es dann, die in den Graphelementen des veränderten Hostgraphen landen. Damit dies klappt, duerfen in der rechten Regelseite natürlich nur Variablen verwendet werden, die durch einen Match der linken Seite belegt werden können. Ausserdem ist das Berechnen einer Belegung schwierig (sogar unentscheidbar), wenn zugelassen wird, dass beliebige Terme in der linken Seite stehen. Man beschränkt sich also darauf, in der linken Seite als Terme nur atomare Terme zuzulassen, womit durch den Match die Variablenbelegung direkt angegeben wird.

Auch Gleichungen über Terme können Anwendungsbedingungen sein. Die Semantik ist dann die, dass eine Regel nur angewendet werden darf, wenn durch den Match die Variablen so belegt werden, dass alle Gleichungen (oder Ungleichungen) erfüllt werden.

Der schon betrachte Morphismus aus Abb. 1.2 stellt eine einfache Regel dar. Die Semantik derselben ist in etwa die, dass an jeden rechteckigen Knoten, der eine Zeichenkette mit Inhalt "this" und eine beliebige Zahl enthält, zwei neue Knoten vorangestellt und angehangen werden, der String gelöscht wird, und in den neuen ovalen Knoten eine Zahl eingetragen wird, die von derjenigen im Originalknoten abhängt.

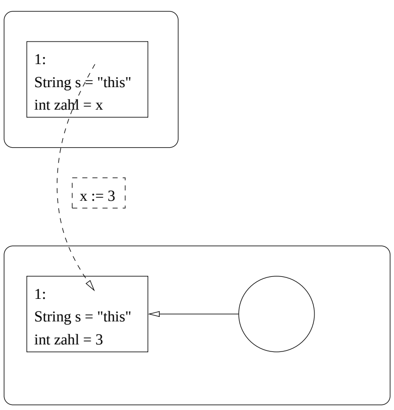

Abbildung 1.3: Match für Regel aus Abb. 1.2

Ein Match für diese Regel ist in Abb. 1.3 dargestellt. Oben die linke Regelseite, unten der Hostgraph. An den den Morphismus darstellenden Pfeil ist die Variablenzuweisung geschrieben, die daraus resultiert, dass Matche

A-gültig sein müssen. Dies ist also die Gleichungsmenge, die zur Bildung der Quotiententermalgebra herangezogen werden würde, sich dies als Variablenzuweisung vorzustellen ist allerdings deutlich durchsichtiger.

Würde man die obige Regel an dem Match aus Abb. 1.3 anwenden, so stellte sich die Situation wie in Abb. 1.4 dar. Oben istg wieder die Regel zu erkennen (Graphen L und R), unten der Hostgraph G und der Zielgraph H. Man beachte besondern, wie der Term $(x+1)*2$ in R durch 8 in H ersetzt wurde, da wegen $x = 3$ (als Terme) in derselben Algebra auch $(x+1)*2 = 8$ gelten muss. Es sind im Bild nur die Abbildungen der Knoten dargestellt. Die Kanten werden auch abgebildet, aber dies würde das Bild nur unnötig verkomplizieren. Man sieht deutlich, wie die Objekte aus G, die von der Regel nicht berührt werden in H immer noch vorhenden sind, und wie der Kontext des rechteckigen Knotens aus G entsprechend der Regel verändert wurde.

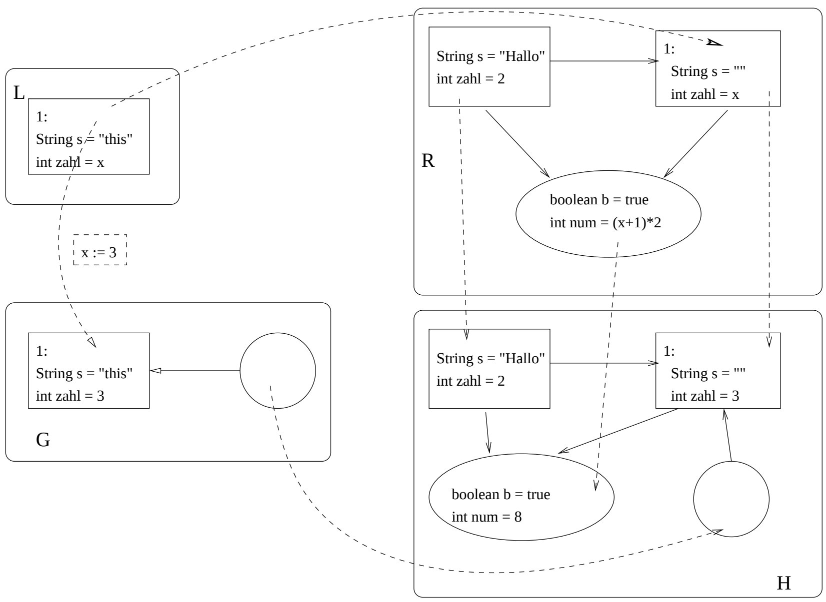

Abbildung 1.4: Anwendung der Regel aus Abb. 1.2 auf Match aus 1.3

Eine endliche Menge von Regeln zusammen mit einem Graphen, dem sogenannten Startgraph, nennt man Graphgrammatik. Gegeben einen Graph, den aktuellen Arbeitsgraph, kann man aus ihm durch Anwendung einer Regel aus dieser Menge einen neuen Arbeitsgraphen erzeugen, auf den man wieder Regeln anwenden kann. Eine solche Regelanwendung in eine direkte Ableitung. Eine beliebige Ableitung von einem Graphen besteht aus einer Folge von direkten Ableitung, deren Hostgraph jeweils der Zielgraph der vorhergehenden Ableitung ist, der Hostgraph der ersten Ableitung natürlich der Graph selbst. Alle auf diese Weise vom Startgraphen der Grammatik mit

Hilfe von Regeln der Grammatik ableitbaren Graphen sind die Elemente der Sprache dieser Grammatik. Eine Graphgrammatik stellt also ein konstruktives Mittel zur Beschreibung einer Graphmenge (nämlich deren Sprache) dar.

Ein anderes Mittel zur Beschreibung einer Graphmenge soll nun vorgestellt werden. Es sind dies Grapheinschränkungen. Eine Grapheinschänkung besteht wie eine Regel aus zwei Graphen, der Prämisse und der Konklusion, und einem Morphismus zwischen Prämisse und Konklusion. Dieser Morphismus soll total und injektiv sein, es ist also eine Graphinklusion. Ein Graph erfüllt nun eine solche Einschränkung, wenn an allen Stellen im Graphen, an denen die Prämisse vorkommt, auch die Konklusion zu finden ist. In dieser Arbeit beschränken wir uns darauf, das diese Vorkommen (dargestellt durch einen Morphismus) total und injektiv sein sollen.

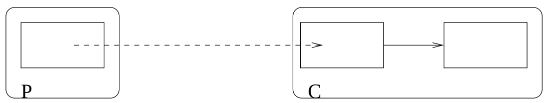

Abbildung 1.5: Einfache Graphinklusion

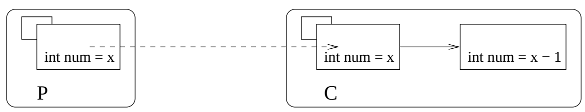

Abbildung 1.6: attributierte Graphinklusion

Zwei Graphinklusionen sind in Abbildungen 1.5 und 1.6 dargestellt, wobei erstere für jeden Knoten fordert, dass an ihm mind. ein weiterer Knoten hängt (von endlichen Graphen also nur erfüllbar, wenn alle Knoten in Zyklen liegen). Die zweite Inkusion beinhaltet auch Attributbedingungen und fordert für jeden speziell markierten Knoten (mit einer Kante zu sich selbst), dass er eine Verbindung zu einem Knoten hat, der den (arithemtischen) Vorgänger des eigenen Attributes enthält.

In Abb. 1.7 sind drei Graphen dargestellt. Dabei erfüllen $G1$ und $G2$ die Graphinklusion aus Abb. 1.6 ($G1$, da 2 eben $3 - 1$ ist, und nur der Knoten mit 3 markiert ist; $G2$ weil die Prämisse garnicht erst erfüllt wird), Graph $G3$ jedoch nicht (der Knoten 2 ist zwar markiert, aber an ihm hängt kein Knoten, der 1 enthält).

Gegeben eine endliche Menge von Grapheinschränkungen können wir eine Menge von korrekten Graphen auszeichnen, die alle Grapheinschränkungen erfüllen. Hat man nun eine Graphgrammatik und eine Menge von Gra-

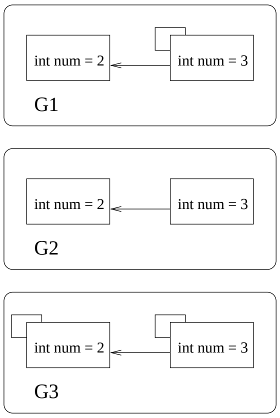

Abbildung 1.7: $G1$ und $G2$ erfüllen Abb. 1.6, $G3$ nicht

pheinschränkungen, kann man sich fragen, ob alle von der Grammatik erzeugten Graphen die Grapheinschränkungen erfüllen, i.e. die Sprache der Grammatik eine Untermenge korrekten Graphen ist. Ist dies der Fall nennt man die Grammatik konsistent bezüglich der Grapheinschränkungen. In Kapitel 3 wird ein Verfahren vorgestellt, mit dem man Regeln durch spezielle Anwendungsbedingungen so erweitern kann, dass die dann erzeugte Grammatik auf jeden Fall konsistent ist. Man beachte, dass die Sprache einer Grammatik selten gleich der Menge der korrekten Graphen gleich sein wird, sondern immer eine Untermenge, da schon durch Änderung des Startgraphen eine komplett andere Sprache erzeugt werden kann, die immer noch konsistent sein kann.

Kapitel 2

Formalisierung

Für den Rest dieses Kapitels sei eine Typmenge T fest vorgegeben, sowie eine Signatur Σ_{attr}, die die Signatur für die Attributwerte darstellt. Attribute sind also Elemente von Algebren zu dieser Signatur, die aber ansonsten nicht weiter spezifiziert ist. Wir unterscheiden keine Knoten- und Kantentypen.

2.1 Attributierte Graphen

Die Art, wie die Graphelemente zu Attributen kommen ist auf den ersten Blick gewöhnungsbedürftig: Pro Typ gibt es eine Menge von speziellen Knoten, den sogenannten Carriern, die die Attributwerte enthalten. Es gibt einen Carrierknoten pro Kombination von Attributwerten, also potentiell unendlich viele. Jedem normalen Graphelement (also den Nicht-Carriern) wird ein Carrier zugeordnet, der dann die Werte der Attribute dieses Graphelements widerspiegelt. Änderungen von Attributen eines Graphelementes werden nun dadurch realisiert, dass die Zuordnung des Carriers auf denjenigen umgehangen wird, der genau die gewünschten Attribute enthält. Das Ändern von Attributen ist also ein **graphische** Operation. Dies alles mündet in folgende Definition:

Definition 2.1. *Ein* **attributierter Graph** $G = (G_V, G_E, G_C, s^G, t^G, c^G, typ^G, G_A)$ *zu einem Typ T ist also eine Familie von Mengen, als da wären: G_V der Menge von Knoten, G_E der Menge von Kanten, G_C die Menge von Carriern, source- und target Abbildung: $s^G, t^G : G_E \to G_V$, die pro Kante deren Beginn- und Endknoten angibt, der Zuordnung von Carriern zu Graphelementen $c^G : (G_V \cup G_E) \to G_C$ und schliesslich der Typisierung $typ^G : G_C \to T$. G_A ist eine Σ_{attr}-Algebra. Wir definieren noch die Typisierung von Knoten und Kanten folgendermassen: $n^G := (typ^G \circ c^G)^{|G_V} : G_V \to T$ und $a^G := (typ^G \circ c^G)^{|G_E} : G_E \to T$.*

Man beachte das **alle** Graphen desselben Typs, welche Attribute derselben Algebra zu Σ_{attr} haben, dieselbe Menge G_C besitzen. Man könnte sie also auch aus dem Graphen herausnehmen und anstattdessen wie den Typ T extern lassen. Weiterhin wollen wir aus attributierten Graphen normale ohne Attribute machen und Subgraphen bilden können:

Definition 2.2. *Der* **nackte Graph** G^N *eines attributierten Graphen* G *ist die Familie* $(G_V, G_E, s^G, t^G, n^G, a^G)$.

Definition 2.3. *Ein Graph* F *ist* **Subgraph** *des attributierten Graphen* G, *geschrieben* $F \subseteq G$, *falls gilt:* $F_V \subseteq G_V$, $F_E \subseteq G_E$, $s^F = s^{G|F_V}$, $t_F = s^{G|F_V}$, $c^F = c^{G|(F_V \cup F_E)}$, $F_C = G_C$ *und* $F_A = G_A$. *D.h. auch die Attributierung darf sich für die erhaltenen Graphelemente nicht ändern.*

Jetzt wollen wir Morphismen auf attributierten Graphen definieren. Dazu müssen wir erstens die graphische Struktur abbilden, und zweitens die Algebren, die den Attributen zu Grunde liegen:

Definition 2.4. *Ein* **totaler attributierter Graphmorphismus** $f : F \to G$, *oder auch* $F \xrightarrow{f} G$ *ist ein Paar* (f^G, f^D) *wobei* f^G *ein totaler Graphmorphismus* $F^N \xrightarrow{f^G} G^N$ *ist, und* $f^D : F_A \to G_A$ *ein totaler* Σ_{attr}*-Homomorphismus. Ein* **totaler Graphmorphismus** f^G *ist ein Paar von Funktionen* $f^G_V : F_V \to G_V$ *und* $f^G_E : F_E \to G_E$, *so dass* $f^G_V \circ s^F = s^G \circ f^G_E$ *und* $f^G_V \circ t^F = t^G \circ f^G_E$, *also Source und Target von abzubildenden Kanten auf Source und Target des Abbilds abgebildet werden. Ausserdem soll typgleich abgebildet werden, also* $f^G_V \circ n^F = n^G$ *und* $f^G_E \circ a^F = a^G$.

Falls sowohl f^G_V *als auch* f^G_E *injektiv (surjektiv, bijektiv) so nennen wir auch* f^G *so, und infolgedessen auch* f. *D.h. ein attributierter Graphmorphismus erbt diese Eigenschaft ausschliesslich vom graphischen Teil, egal, wie der Algebrahomomorphismus aussieht. Für spätere einfache Schreibweise wollen wir ausserdem* f^G *überladen, und nicht nur* (f^G_V, f^G_E) *so bezeichnen, sondern auch eine Abbildung* $f^G : (F_V \cup F_E) \to (G_V \cup G_E)$ *definieren, und zwar:* $\forall e \in F_V . f^G(e) := f^G_V(e)$ *und* $\forall e \in F_E . f^G(e) := f^G_E(e)$, *die so also generell Graphelemente in Graphelemente abbildet ohne zwischen Knoten und Kanten unterscheiden zu müssen.*

Bemerkung zu 2.4: Man kann den Σ_{attr}-Homomorphismus f^D auch als Abbildung von $F_C \to G_C$ auffassen. Durch ihn werden die einzelnen Werte in den Elementen des Carriers in die Zielalgebra übersetzt, was einen Carrier desselben Typs im Zielgraphen ergibt. Man kann f^D also zu einem $f^{D*} : F^C \to G^C$ erweitern. Insbesondere gilt: $f^D = id_{F_A} \Longrightarrow f^{D*} = id_{F_C}$. Daraus ergibt sich eine weitere Eigenschaft, die Morphismen haben können oder nicht: **A-Gültigkeit**. Ein Morphismus $f : F \to G$ ist A-gültig, falls gilt: $f^{D*} \circ c^F = c^G \circ f^G$. D.h. ein A-gültiger Morphismus darf ein Graphelement nur auf ein Graphelement abbilden, dessen Attributwerte gleich denen sind, wie sie bei Abbildung der Attribute der Urbilder dieses Graphelementes entstehen. Insbes. darf im Falle, dass $f^D = id_{F_A}$ ein Graphelement nur auf eines mit den gleichen Attributwerten abgebildet werden, damit f A-gültig ist.

Definition 2.5. *Ein* **attributierter Graphmorphismus** $F \xrightarrow{f} G$ *ist ein totaler attributierter Graphmorphismus von einem Subgraph* $d_f \subseteq F$ *nach* G. *Wenn wir den Unterschied explizit herausstellen wollen, nennen wir einen solchen Morphismus* **partiell**. $d_f = dom(f)$ *heisst* **Domain** *von* f.

Man kann einen partiellen attributierten Graphmorphismus f auch so auffassen, dass der Graphteil $f^G = (f_V^G, f_E^G)$ aus zwei partiellen Funktionen besteht, die die Bedingungen in 2.4 erfüllen, allerdings in der Kategorie der Mengen und partiellen Funktionen.

Definition 2.6. *Die* **Komposition** $g \circ f : F \to H$ *von zwei (partiellen) attributierten Graphmorphismen* $F \xrightarrow{f} G$ *und* $G \xrightarrow{g} H$ *ist definiert als* $g \circ f = ((g_V^G \circ_s f_V^G, g_E^G \circ_s f_E^G), g^D \circ_a f^D)$, *wobei* $\circ_s$ *die Komposition in der Kategorie der Mengen und partiellen Funktionen und* $\circ_a$ *die Komposition in der Kategorie der* Σ_{attr}*-Algebren und* Σ_{attr}*-Homomoprphismen ist.*

Man kann einen attributierten Graphen auch gleich als Algebra auffassen. Die zugrundeliegende Signatur würde $\Sigma = \Sigma_{GS} + \Sigma_{DS} + \Omega_{attr}$ sein. Dabei sind alle Operationen aus Σ_{GS} unär, und sie stellt die Graphsignatur dar. I.e. es gibt Sorten für Kanten und Knotentypen und die Carrier, und die Operationen simulieren s^G, t^G und c^G. $\Sigma_{DS} = \Sigma_{attr}$ ist wiederum für die Attribute zuständig. $\Omega_{attr} = (\emptyset, OP_\Omega)$ mit $OP_\Omega = (O_{s,s'})_{s \in S_{GS}, s' \in S_{DS}}$ stellt die Verbindung von Carriern zu Attributwerten dar. D.h. für jeden Carriertyp und jedes Attribut dieses Carriertyps gibt es eine Operation in OP_Ω, die den Wert dieses Attributes extrahiert, und den Namen des Attributes trägt.

Wenn man Graphen als Algebren definiert, so sind Graphmorphismen ganz normale partielle Σ-Homomorphismen die auf Σ_{DS} allerdings total sind. Dies zusammen mit den offensichtlichen Identitäten ergibt dann eine Kategorie $\underline{\mathbf{Alg}}(\Sigma)$. Also:

Satz 2.7. $\mathbf{AG_T(\Sigma_{attr})}$ *bezeichnet die Kategorie der attributierten Graphen des Typs T (siehe 2.1) und partiellen attributierten Graphmorphismen (siehe 2.5), wobei die Identitäten die Identität auf Knoten und Kanten, und der identische Algebrahomomorphismus ist. Falls T und* Σ_{attr} *aus dem Kontext klar sind bezeichnen wir diese Kategorie mit* $\underline{\mathbf{AGR}}$. $\underline{\mathbf{AGR}}$*enthält alle Colimiten.*

Der Beweis für den letzten Teil von 2.7 kann in [2] gefunden werden.

2.2 graphische Regeln und Transformation

Im folgenden werden wir definieren, was graphische Regeln sind, wie sich damit Graphen ändern lassen, und wie damit eine Sprache aufgebaut wird. Zuerst also die Regel und Match:

Definition 2.8. *Eine* **graphische Regel** $L \xrightarrow{r} R$ *ist ein attributierter graphischer Morphismus in dem* $L_A = R_A =$ *Termalgebra zu* Σ_{attr}*. D.h. in Regeln (oder besser in linker und rechter Seite) dürfen Terme verwendet werden. Weiterhin sollen in L nur atomare Terme verwendet werden. Letzteres ist ausschliesslich der einfacheren Implementierbarkeit geschuldet. L heisst linke, R rechte Seite von r. Ein Morphismus* $L \xrightarrow{m} G$ *heisst* **Match** *von L in den Graphen G, falls er total und A-gültig ist.*

Es ist hier noch nicht von Bedeutung, wie man an einen Match m kommt, insbesondere, wie er A-gültig zu machen ist. Darauf wird kurz in Kapitel 4 eingegangen. Wann immer wir im folgenden einen Match brauchen setzen wir sein Vorhandensein einfach voraus.

Wenn man also eine Regel $L \xrightarrow{r} R$ und einen Match $L \xrightarrow{m} G$ gegeben hat weiss man (a) wo man die linke Seite von r im Hostgraphen G wiederfindet, und (b) wie man dieses Vorkommen verändern muss, um die Regel anzuwenden:

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

Abbildung 2.1: Direkte Ableitung G nach H

Definition 2.9. *Die* **direkte Ableitung** $G \overset{r,m}{\Longrightarrow} H$ *von Graph G mit Regel* $L \xrightarrow{r} R$ *am Match* $L \xrightarrow{m} G$ *nach Graph H wird durch das Pushout von r und m gebildet. Vergleiche Abbildung 2.1. Der Morphismus m∗ heisst Comatch, und ist ebenfalls A-gültig.*

Gegeben eine Menge R von Regeln, so heisst eine indizierte Menge $(G_i \overset{r_i,m_i}{\Longrightarrow} H_i)_{1 \leq i \leq n}$ *mit* $r_i \in R$ *von direkten Ableitungen eine* **Ableitungssequenz***, falls gilt:* $\forall i \in \mathbb{N}.(1 < i \leq n \rightarrow H_{i-1} = G_i)$*.* G_1 *heisst Startgraph dieser Ableitungssequenz,* H_n *Endgraph. Ein Graph H ist aus G mit Regeln aus R* **ableitbar** *(*$G \overset{R}{\Longrightarrow}^* H$*), falls es eine Ableitungssequenz*

$G = G_1 \overset{r_i, m_i}{\Longrightarrow}{}^* G_n = H$ gibt. Man beachte, dass $n = 1$ sein kann, die Ableitungssequenz dann leer ist, und demzufolge $G = H$ ist, auch wenn nicht mal Regeln vorhanden sind.

Definition 2.10. *Eine* **Graphgrammatik** *$Gra = (S, R)$ besteht aus dem Startgraphen S und einer Menge von Regeln R. Die von dieser Grammatik erzeugte Sprache ist die Menge der aus S ableitbaren Sprachen: $L_{Gra} = \{G | S \overset{R}{\Longrightarrow}{}^* G\}$.*

2.3 Grapheinschränkungen und Anwendungbedingungen

Nachdem wir jetzt also die Mittel haben, konstruktiv eine Menge von Graphen zu erzeugen (nämlich eine Graphgrammatik) wollen wir nun eine Menge von Graphen dadurch auszeichnen, dass sie bestimmte Eigenschaften erfüllen. Wir wollen dazu sogenannte Grapheinschränkungen benutzen, die sozusagen bei Vorhandensein von bestimmten Strukturen im Graphen (die Voraussetzung), auch bestimmte andere Strukturen an diesen Stellen fordern (die Folgerung). Also:

Definition 2.11. *Eine* **atomare Grapheinschränkung** *oder* **Graphinklusion** *ist ein Morphismus $P \xrightarrow{a} C \in Mor_{\underline{\mathbf{AGR}}}$, der total und injektiv ist. Der Graph P heisst Prämisse, C Konklusion der Grapheinschränkung. Ein Graph G* **erfüllt** *a ($G \models a$), falls gilt: $\forall P \xrightarrow{p} G$ total, injektiv $.\exists C \xrightarrow{q} G$ total, injektiv $.q \circ a = p$. Die p gehen also alle kompletten 'Vorkommen' von P in G durch, und für jedes solche muss auch C in G komplett vorkommen (durch q), und zwar an derselben 'Stelle' wie P (durch $q \circ a = p$). Siehe auch Abbildung 2.2.*

$$\begin{tikzcd}
P \arrow[r, "a"] \arrow[d, "p"'] \arrow[dr, phantom, "="] & C \arrow[dl, "q"] \\
G &
\end{tikzcd}$$

Abbildung 2.2: Graphinklusion

Bemerkung zu 2.11: Wie der Namenteil **atomar** schon andeutet sind die Graphinklusionen bloss die Grundlage für die Grapheinschränkungen, wie wir sie benutzen wollen. Bei uns besteht eine Grapheinschränkung aus einer endlichen Menge von Graphinklusionen, und einer booleschen Formel, die diese Menge als Variablen benutzt. Einen Graph gegeben, kann man diese Variablen belegen, je nachdem, ob der Graph die Graphinklusion erfüllt oder nicht, und damit dann auch die ganze Formel auswerten. Dieser Graph erfüllt dann eine solche Grapheinschränkung, wenn die Formel zu *true* ausgewertet wird. Da wir später die gleiche Formel für eine andere Menge von Variablen wiederverwenden wollen, führen wir einen Zwischenschritt ein, nämlich ein Mapping von Namen zu Graphinklusionen. In der Formel werden dann bloss Namen verwendet.

Definition 2.12. *Sei N eine endliche Menge von Namen so sind alle $n \in N$* **atomare Formeln**. *Definieren wir nun induktiv die Menge* $\mathbf{Form(N)}$*:*

- $x \in N \Longrightarrow x \in Form(N)$
- $\{'true',' false'\} \subseteq Form(N)$
- $x \in Form(N) \Longrightarrow \neg x \in Form(N)$
- $x, y \in Form(N) \Longrightarrow x \vee y, x \wedge y \in Form(N)$
- *keine anderen Dinge sind* $\in Form(N)$

Dies sind also die ganz normal definierten booleschen Formeln über Variablen.

Wie oben schon angedeutet ist nun eine Grapheinschränkung eine boolesche Formel, deren atomare Formeln besonders ausgewertet werden:

Definition 2.13. *Sei N wieder eine endliche Menge von Namen, $I = (i_n)_{n \in N}$ eine N-indizierte Menge von Graphinklusionen und $f \in Form(N)$ eine boolesche Formel, so ist das Paar (f, I) eine* **Grapheinschränkung**. *Ein Graph G* **erfüllt** *(f, I) ($G \models (f, I)$), falls $f(I, G) = true$. Dabei ist $f(I, G)$ folgendermassen induktiv definiert:*

- $f = n \in N$ *eine atomare Formel* $\Longrightarrow f(I, G) := (G \models i_n)$
- $f = true \Longrightarrow f(I, G) := true$
- $f = false \Longrightarrow f(I, G) := false$
- *Seien* $a, b \in Form(N)$*:*
  - $f = \neg a \Longrightarrow f(I, G) := \neg a(I, G)$
  - $f = a * b \Longrightarrow f(I, G) := a(I, G) * b(I, G)$ *für* $* \in \{\vee, \wedge\}$

Das im nächsten Kapitel vorgestellte Verfahren zur Erzwingung von Konsistenz einer Grammatik zu einer gegebene Menge von Grapheinschränkungen beruht darauf, dass es alle Regeln der Grammatik durch Bedingungen erweitert, die die Menge der Matche für eine Regel einschränken. Diese sollen im folgenden definiert werden:

Definition 2.14. *Eine* **atomare bedingte Grapheinschränkung über Graph** L *oder* **atomare Anwendungsbedingung** $acc_L = (x : L \to X, \bigvee_{i \in I}(X \xrightarrow{c_i} C_i))$ *besteht aus einem Morphismus* x *und einer Disjunktion von Graphinklusionen* c_i*. Im Gegensatz zur unbedingten atomaren Grapheinschränkung können nur Morphismen* $m : L \to G$ *ein* acc_L **erfüllen** *(*$m \models_L acc_L$*), und zwar (siehe Abbildung 2.3), falls* $\forall X \xrightarrow{p} G$ *total, injektiv mit* $p \circ x = m \to \exists i \in I, C_i \xrightarrow{q} G$ *total, injektiv und* $q \circ c_i = p$*. Bildlich gesprochen: wenn* X *an der Stelle wo* L *in* G *gefunden wurde, vorkommt (die Vorbedingung), dann muss auch mindestens ein* C_i *an dieser Stelle vorkommen. Man beachte, dass* I *durchaus* $\emptyset$ *sein darf. In diesem Falle darf* X *nicht in* G *vorkommen (zumindest nicht an der Stelle, die* m *angibt), da kein* i *gefunden werden könnte.*

$$
\begin{tikzcd}
L \arrow[r, "x"] \arrow[d, "m"'] & X \arrow[r, "c_i"] \arrow[dl, "p"] & C_i \arrow[dll, "q"] \\
G & &
\end{tikzcd}
$$

Abbildung 2.3: atomare Anwendungsbedingung

Wiederum benutzen wir diese atomaren Anwendungsbedingungen als atomare Formeln für die Variablen einer booleschen Formel:

Definition 2.15. *Analog zu Definition 2.13 sei* N *wieder eine endliche Menge von Namen,* $A = (a_n)_{n \in N}$ *eine* N*-indizierte Menge von atomaren Anwendungsbedingungen über Graph* L *und* $f \in Form(N)$ *eine boolesche Formel, so ist das Paar* (f, A) *eine* **Anwendungsbedingung über** L*. Ein Morphismus* $L \xrightarrow{m} G$ *erfüllt* (f, A) *(*$m \models_L (f, A)$*), wenn* $f(A, m) = true$ *ist. Dabei wird* $f(A, m)$ *wieder induktiv definiert, wobei ich hier bloss die Auswertung für atomare Formeln angeben will, der Rest ergibt sich exakt so wie in Definition 2.13. Falls* $f = n \in N$ *also eine atomare Formel ist, so ist* $f(A, m) := (m \models_L a_n)$*.*

Jetzt müssen wir bloss noch diese Anwendungsbedingungen mit den Regeln verbinden.

Definition 2.16. *Sei* $p = L \xrightarrow{r} R$ *eine Regel nach Definition 2.8,* $A_L = (f, A)$ *eine Anwendungsbedingung über* L *und* $A_R = (g, B)$ *eine Anwendungsbedingung über* R*, so heisst* $\hat{p} := (p, A(p))$ *mit* $A(p) := (A_L, A_R)$ **bedingte Regel**, A_L **Vorbedingung** *und* A_R **Nachbedingung**.

Die **direkte bedingte Ableitung** $G \overset{\hat{p},m}{\Longrightarrow} H$ *ist durch die normale direkte Ableitung* $G \overset{p,m}{\Longrightarrow} H$ *gegeben (siehe Defi. 2.9), darf allerdings bloss vollzogen werden, falls* $m \models_L A_L$ *und* $m^* \models_R A_R$*, wobei* $R \xrightarrow{m^*} H$ *der Comatch ist.*

Wenn man in einer Grammatik die Regeln durch bedingte Regeln ersetzt, erhält man wiederum eine Grammatik, die auch eine Sprache erzeugt, ganz analog der Definition 2.10.

Kapitel 3

Umwandlung von Grapheinschränkungen in Anwendungsbedingungen

In diesem Kapitel soll es darum gehen, aus Grapheinschränkungen Anwendungsbedingungen für Regeln zu machen. Die so veränderten bedingten Regeln werden, wenn sie auf einen die Grapheinschränkungen erfüllenden Graph angewendet werden, diese Erfülltheit erhalten. Es wird sich sehr stark auf [1] gestützt, allerdings mit den für die Attributierung und den Single-Pushout Ansatz nötigen Änderungen. Der Hauptsatz hierbei besagt das dies möglich ist:

Satz 3.1. *Sei $p = L \xrightarrow{r} R$ eine Regel, und (f, A) eine Grapheinschränkung, mit $f \in Form(N)$ und $A = (a_n)_{n \in N}$. Dann induziert (f, A) eine Anwendungsbedingung $A_R = (f', B) = T((f, A))$ über R, so dass für alle Morphismen $m^* : R \to H$ gilt: $m^* \models_R (f', B) \iff H \models (f, A)$.*

Wir geben zuerst eine Konstruktion an, die darauf beruht, dass es möglich ist, zu jeder atomaren Grapheinschränkung eine Konjunktion von atomaren Anwendungsbedingungen zu konstruieren, was unten gezeigt wird.

Konstruktion 3.2. *Sei also p und (f, A) wie in 3.1 gegeben. Durch $A = (a_n)_{n \in N}$ ist implizit eine Menge N von Namen gegeben. Definiere nun $B' := (b_n)_{n \in N}$, wobei $\forall n \in N. b_n = T(a_n)$ nach Konstruktion 3.4 Konjunktionen von atomaren Anwendungsbedingungen sind, d.h. $b_n = \bigwedge_{s \in S_n} b_n^s$ mit b_n^s atomare Anwendungsbedingungen. Nun definiere man $B := \biguplus b_n^s$ als disjunkte Vereinigung aller beteiligten atomaren Anwendungsbedingungen, was auch gleich zu einer neuen Namensmenge N' führt, und bennene die einzelnen Variablen in den b_n entsprechend um. Setzt man nun diese Konjunktionen an Stelle der Variablennamen in f ein, erhält man eine neue*

Formel f' (i.e. $f' = f[n/b_n]$). Dann ist $T((f,A)) := (f',B)$ die gesuchte Anwendungsbedingung.

Beweis. Wir erlauben uns hier eine Laxheit, indem wir auch die Schreibweise $g(X,m)$ verwenden, wenn g eine normale boolesche Formel ist, X allerdings auch nur eine N-indizierte Menge von Formeln, die dann aber über atomare Anwendungsbedingungen. Wir lassen also Formeln als Atome zu. Die Auswertungsregeln sollten klar sein.

Nach Satz 3.3 gilt: $m^* \models_R T(a_n) \iff H \models a_n$. Für die Auswertung von f gilt dann: $\forall n \in N.g \in Form(N).g = n \to g(A,H) = (H \models a_n) = (m^* \models_R b_n) = g(B',m^*)$. Die Auswertung der atomaren Formeln in f ist also in beiden Interpretationen gleich, woraus auch die Gleichheit der Auswertung für die Gesamtformel folgt, also gilt: $f(A,H) = f(B',m^*)$. Da nun nach Konstruktion f' zu f äquivalent ist (es wurden bloss Umbenennungen vorgenommen), gilt auch $f(B',m^*) = f'(B,m^*)$, woraus wiederum folgt, dass $m^* \models_R (f',B) \iff H \models (f,A)$. □

Kommen wir also jetzt zu der Übersetzung von Graphinklusionen in Konjunktionen von atomaren Anwendungsbedingungen:

Satz 3.3. *Gegeben eine Graphinklusion $P \xrightarrow{c} C$ und ein Graph R , so existiert eine Konjunktion von atomaren Anwendungsbedingungen $T(c)$, so dass gilt: $\forall m^* : R \to H.m^* \models_R T(c) \iff H \models c$. Für $T(c) = \bigwedge_s s$ gilt: $m^* \models_R T(c) :\iff \forall s.m^* \models_R s$.*

$$\begin{tikzcd}
R \arrow[d, "s"'] & \\
S \arrow[d, "t"'] \arrow[dr, phantom, "(1)"] & P \arrow[l, "p"'] \arrow[d, "c"] \\
T \arrow[d, "\equiv"'] & C \arrow[l, "q"] \\
T_{/\equiv} &
\end{tikzcd}$$

Abbildung 3.1: Zur Konstruktion 3.4

Konstruktion 3.4. *Sei also $R \in Obj_{\underline{\mathbf{AGR}}}$ ein Graph, und $P \xrightarrow{c} C$ eine Graphinklusion, so definiere $T(c) := \bigwedge_S (R \xrightarrow{s} S, \bigvee_{i \in I} (S \xrightarrow{\equiv_i \circ t} T^i_{/\equiv}))$, wobei gilt:*

- *S ist Verklebung von R und P, i.e. $\exists s : R \to S$ und $\exists p : P \to S$, die zusammen surjektiv sind. Zusätzlich solle p injektiv sein und p und s A-gültig sein.*

- *$S \xrightarrow{t} T \xrightarrow{\equiv} T_{/\equiv}$ ist total und injektiv, (1) in Diagramm 3.1 ist Pushout von p und c, und $\equiv$ ist total und surjektiv, $T_{/\equiv}$ also eine Verklebung von T. Weiterhin sei $\equiv \circ q$ injektiv. (q ist auch total, da p total ist). In der Disjunktion stehen also alle Morphismen, die dies erfüllen, sie können sich allerdings (da t Teil des Pushouts ist) nur in $\equiv$ unterscheiden, weshalb da auch der Index i angefügt wurde.*

Bemerkung zur Konstruktion 3.4: Die Attribute verdienen wiederum eine genauere Betrachtung. Sowohl R als auch c enthalten Terme. Um Konfusion zu vermeiden sollte, bevor die Konstruktion angewendet wird, die Variablen aus c umbenannt werden (oder anderweitig sichergestellt werden, dass nicht Terme identifiziert werden, nur weil sie zufällig die gleichen Variablen enthalten). Weiterhin beachte man, dass p und s A-gültig sein sollen, um überhaupt eine Termidentifikation zu erreichen. Die Besonderheit hier ist, das R beliebige Terme enthält, P jedoch höchstens Variablen. Aus der geforderten A-Gültigkeit von s und p, und da S eine Verklebung ist, ergibt sich, das S_A, die Algebra in S, eine Quotiententermalgebra ist, die genau die Terme identifiziert, die in Carriern von Graphelementen in R und P stehen, die durch s und p auf gleiche Graphelemente in S abgebildet werden. Die Konstruktion des Pushouts stellt sicher, dass eine Termersetzung (oder besser Einsetzung) in den Graphelementen, die aus C nicht aber aus S kommen, stattfindet.

Beweis. Wir zeigen $H \not\models c \iff m^* \not\models_R T(c)$:

"$\Longrightarrow$": $H \not\models c$ bedeuted, dass zwar $\exists p' : P \to H$ injektiv, jedoch

$$\nexists q' : C \to H \text{ injektiv.} q' \circ c = p' \tag{3.1}$$

Wir versuchen nun im folgenden unter der Annahme dass $m^* \models_R T(c)$ ein solches q' zu konstruieren, womit die Annahme falsch wäre. Dazu bilden wir zuerst das Koprodukt $R + P$ mit den Injektionen in_R und in_P, und nehmen H als Vergleichsobjekt für $R + P$. Es existiert also ein eindeutiger (A-gültiger) Mittlermorphismus $f : R + P \to H$ mit

$$f \circ in_R = m^* \tag{3.2}$$

$$f \circ in_P = p' \tag{3.3}$$

f lässt sich nun in einen surjektiven $h : R + P \to S$ und einen injektiven $n : S \to H$ Teil faktorisieren, mit $n \circ h = f$. Man beachte wiederum, das diese Faktorisierung dazu führt, dass S_A eine Quotiententermalgebra zu $R_A + P_A$ ist, die die Terme in Carriern identifiziert, deren Graphelemente durch f auf dasselbe $e \in (H_V \cup H_E)$ abgebildet werden.

Definieren wir nun $s := h \circ in_R$ und $p := h \circ in_P$, also als h jeweils eingeschränkt auf R und P. Wegen h surjektiv und in_R, in_P zusammen surjektiv sind auch s, p zusammen surjektiv. Da f A-gültig ist, sind es auch s und p (Einschränkung im Definitionsbereich zerstört nicht die A-Gültigkeit). Damit ist S also eine Verklebung von R und P, womit der erste Teil der Anforderung aus Konstruktion 3.4 erfüllt ist, weshalb eine atomare Anwendungsbedingung aus $T(c)$ auf S basiert.

Nehmen wir also $m^* \models_R T(c)$, so muss insbesondere (da $T(c)$ eine Konjunktion ist), auch $m^* \models_R a = (s : R \to S, \bigvee(S \to T_{/\equiv_i}))$ gelten. n erfüllt die Vorbedingung von a, da $n \circ s = n \circ h \circ in_R = f \circ in_R = m^*$ gilt. Nach Definition 2.14 gibt es also ein $S \xrightarrow{\equiv_i \circ t} T_{/\equiv_i}$ und einen injektiven totalen Morphismus $T_{/\equiv_i} \xrightarrow{o} H$, so dass gilt

$$o \circ (\equiv_i \circ t) = n \tag{3.4}$$

Sei nun q der zu p parallele Morphismus im Pushout von p und c (er ist schon durch die Anwendungsbedingung gegeben) und $q' := o \circ \equiv_i \circ q$, so ist q' total injektiv, da q total, $(\equiv_i \circ q)$ injektiv und o injektiv ist. Für dieses q' prüfen wir nun die Bedingung 3.1:

$$q' \circ c = o \circ \equiv_i \circ q \circ c = o \circ \equiv_i \circ t \circ p = n \circ p = n \circ h \circ in_P = f \circ in_P = p'$$

Damit haben wir also ein $p' : C \to H$ gefunden, dass es nach 3.1 nicht geben dürfte, also muss die Annahme falsch sein, womit wir also $m^* \not\models_R T(c)$ erhalten.

"$\Longleftarrow$": Wir gehen also von $m^* \not\models_R T(c)$ aus, womit es mindestens eine atomare Anwendungsbedingung $a = (s : R \to S, \bigvee_{i \in I}(S \to T_{/\equiv_i}))$ in $T(c)$ gibt, mit $m^* \not\models_R a$. D.h. (nach Definition 2.14) dass zwar $\exists S \xrightarrow{n} H . n \circ s = m^*$, aber

$$\forall i \in I.\ \nexists T_{/\equiv_i} \xrightarrow{o} H . o \circ (\equiv_i \circ t) = n \tag{3.5}$$

Durch a sind auch die Morphismen p, c, t und q gegeben. Sei jetzt

$$p' := n \circ p : P \to H \tag{3.6}$$

so ist p' wegen n und p injektiv und total, also ein Ansatz für c.

Nehmen wir also $H \models c$ an, so muss $\exists C \xrightarrow{q'} H . q' \circ c = p'$ gelten wobei q' total und injektiv ist. Wegen $q' \circ c = p' = n \circ p$ kann man H als PO-Vergleichsobjekt für T heranziehen, womit also genau ein Morphismus $f : T \to H$ existiert, der

$$f \circ q = q' \tag{3.7}$$

$$f \circ t = n \tag{3.8}$$

erfüllt. Diesen faktorisieren wir nun in einen surjektiven ($\equiv: T \to T_{/\equiv}$) und einen injektiven ($o : T_{/\equiv} \to H$) Teil, womit dann $f = o\circ \equiv$ gilt. (Für eine Bemerkung über Attribute und resultierende Algebren bei einer solchen Faktorisierung siehe oben).

Untersuchen wir dieses spezielle $\equiv$ näher: Es gelten

$$o\circ \equiv \circ t = f \circ t = n \tag{3.9}$$

$$o\circ \equiv \circ q = f \circ q = q' \tag{3.10}$$

Ausserdem:

- $\equiv \circ t$ ist injektiv. Wäre es dies nicht, so gälte: $\exists x, y \in S. x \neq y. \equiv \circ t(x) =\equiv \circ t(y) \Longrightarrow o\circ \equiv \circ t(x) = o\circ \equiv \ t(y) \Longrightarrow n(x) = n(y)$, was falsch wäre, da n injektiv ist.
- $\equiv \circ q$ ist injektiv. Ansonsten ergäbe sich wiederum unter Benutzung von 3.10 ein Widerspruch zur Injektivität von q'.

Diese Punkte zusammen damit, dass $\equiv$ surjektiv ist führen dazu, dass $\exists i \in I. \equiv=\equiv_i$. Damit ist $\equiv \circ t$ also in der Disjunktion von a enthalten, womit 3.9 einen Widerspruch zu 3.5 darstellt. D.h. die Annahme $H \models c$ war falsch, es gilt also vielmehr $H \not\models c$. □

Die auf diese Art und Weise erhaltenen Nachbedingungen werden in anderen Papieren an dieser Stelle immer in Vorbedingungen umgewandelt, teils um die Äquivalenz von Nach- und Vorbedingungen zu zeigen, was andere Beweise vereinfacht, da man immer davon ausgehen kann, daß keine Nachbedingungen im Spiel sind. Wir werden diesen Weg in unserer Implementierung nicht beschreiten, obwohl dessen Implementierbarkeit untersucht werden sollte. Die Gründe sind mehrfach (siehe dazu auch Abbildung 3.2):

- Beim Vorziehen werden zwei Morphismen x und y (mit bestimmten Eigenschaften) dergestalt gesucht, dass (1) und (2) gültige Ableitungen $X \stackrel{r,x}{\Longrightarrow} S$ und $Y \stackrel{r_x,y}{\Longrightarrow} T_{/\equiv}$ sind, es wird also zu den Comatches s und $\equiv \circ t$ Matche gesucht. Davon kann es mehrere geben (nur endlich viele, aber immerhin), was die ohnehin schon reichliche Anzahl von Verklebungen und Morphismen auf der rechten Seite noch mehr vergrößert.
- Obwohl diese x und y sicher existieren, ist nicht ganz klar, wie man möglichst optimal danach suchen soll, damit die Konversion nicht noch länger dauert.

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "x"'] \arrow[dr, phantom, "(1)"] & R \arrow[d, "s"] \\
X \arrow[r, "r_x"] \arrow[d, "y"'] \arrow[dr, phantom, "(2)"] & S \arrow[d, "\equiv \circ t"] \\
Y \arrow[r, "r_y"] & T_{/\equiv}
\end{tikzcd}$$

Abbildung 3.2: Zum Vorziehen von Nachbedingungen

- Man kann die noch vorzustellende Vereinfachung nicht mehr verwenden.

Was hingegen keine Schwierigkeiten bereiten würde ist die Anwesenheit von Attributen: In P sind bloss Variablen zugelassen, die zudem durch c identisch vorhanden bleiben müssen. Falls nun in R komplexe Terme auftreten, auf die Variablen aus P gematcht werden, so werden diese Terme durch das Pushout erhalten bleiben, und höchstens als Subterm von anderen auftreten. Damit kann man die Regel invers anwenden, wenn man Termrewriting benutzt, man weiss welche Terme aus R auf welche Variablen aus L gematcht werden, und damit kann man alle diese Subterme aus S, T und $T_{/\equiv}$ durch diese Variablen ersetzen.

Ein Nachteil hat das Nicht-Vorziehen der Nachbedingung allerdings: man kann die Menge der Verklebungen (also die Menge der S) nicht mehr auf Relevanz prüfen, da dafür r_x benötigt wird. Wir stellen eine andere Möglichkeit vor, die Anzahl der zu untersuchenden Morphismen zu verkleinern, indem wir die Verklebungen von T wegfallen lassen. Es wäre interessant zu vergleichen, bei welcher Methode mehr Morphismen übrigbleiben, um sagen zu können, welche Anwendungsbedingung schneller überprüft werden könnte.

Betrachte Abbildung 3.3. Es stellt die Situation dar, dass ein Match nach H auch die Anwendungsbedingung erfüllte, und zwar wurde der injektive Morphismus f gefunden, so dass $f \circ \equiv t = n$. Wenn man nun $f' := f \circ \equiv$ setzt, so erfüllt f' die Gleichung $f' \circ t = n$, würde also ein Morphismus sein, der eine Anwendungsbedingung erfüllt, die aus der alten entsteht, indem man $\equiv$ einfach abschneidet.

Wenn wir also versuchen die Disjunktion $\bigvee S \xrightarrow{\equiv \circ t} T_{/\equiv}$ zu erfüllen, und deshalb alle $\equiv$ durchgehen, und nach einem injektiven f suchen, so wollen wir stattdessen ein $T \xrightarrow{f'} H$ mit folgenden Eigenschaften suchen: $f' \circ t = n$, $f' \circ t$ injektiv und $f' \circ q$ injektiv. Wenn wir ein solches finden, ist auch die Disjunktion erfüllt: Faktorisiere f' in $h \circ g$, mit g surjektiv, h injektiv, so ist $g \circ t$ und $g \circ q$ injektiv, da h total ist. Damit ist aber $g \circ t$ ein Element der Disjunktion (vergleiche Konstruktion 3.4), und h erfüllt dieses Element, und damit die Disjunktion. Wenn wir andererseits kein solches f' finden, so kann auch die Disjunktion nicht erfüllbar sein, wäre sie es, gäbe es nämlich ein $\equiv$ und ein f, welche sie erfüllen, welche sich wie unten zu einem f' komponieren lassen, welches obige Eigenschaften erfüllt, was wir nach Vorraussetzung nicht finden konnten.

Beide Verfahren sind also äquivalent, ohne die T-Verklebungen hat man allerdings weniger Morphismen zu untersuchen. Man beache noch, dass f' **nicht** injektiv sein soll, sondern ein beliebiger totaler Morphismus, der allerdings in gewissen Teilen injektiv ist, wie oben beschrieben. Ausserdem sei darauf verwiesen, dass man für diese Art von Anwendungsbedingungen den Morphismus q, der aus dem Pushout resultiert, merken muß.

$$\begin{tikzcd} T \arrow[rr, "f'"] \arrow[dr, "\equiv"'] & & H \\ & T_{/\equiv} \arrow[ur, "f"'] & \end{tikzcd}$$

Abbildung 3.3: Zur Elimination von $\equiv$

Kapitel 4

Implementierung

Dieses Kapitel stellt die konkrete Implementierung der oben beschriebenen Verfahren und die Designentscheidungen, die zu ihr geführt haben, vor. Dabei beachte man, daß sich die Software selbst, also Sourcen, Kompilat und Testdateien auf der zu dieser Arbeit gehörenden CD-ROM befinden (siehe auch Abschnitt 4.4).

Zuerst soll erläutert werden, wie die Anforderungen sind. Der Benutzer soll folgende Dinge (zusätzlich zu normalen `AGG` -Fähigkeiten) können:

- atomare Grapheinschränkungen (Graphinklusionen) eingeben
- Formeln eingeben, deren Variablen für Graphinklusionen stehen
- prüfen, ob der aktuelle Hostgraph die Integritätsbedingungen erfüllt
- alle eingegebenen Grapheinschränkungen in Anwendungsbedingungen für alle vorhandenen Regeln umwandeln
- solcherart erweiterte Regeln wie normale Regeln anwenden (so sie ihre Anwendungsbedingungen erfüllen)
- die Anwendungsbedingungen einer Regel, die aus einer kongreten Graphinklusion hervorgegangen sind, anschauen
- alle zusätzlich eingegebenen Information speichern und laden
- Graphinklusionen, Formeln oder Anwendungsbedingungen löschen

Sowohl durch die vorhandene Struktur in `AGG` , als auch durch obige Anforderungen ergeben sich zwei zu implementierende Komplexe: die Basisfunktionalität und die graphische Komponente zum Editieren und Anzeigen.

4.1 Basisfunktionalität

Um einen groben Überblick zu erhalten sei auf das Diagramm 4.2 verwiesen, welches die neuen Klassen für den nicht-graphischen Teil enthält, samt ihren Verbindungen untereinander und zum Rest des `AGG` -Systems (die Klassen oben rechts "hinter" dem gebogenen Strich gehören zum vorhandenen `AGG` -System).

`AGG` 's Funktionalität ist grob in drei Ebenen aufgeteilt, die alle mit Graphen und Morphismen zu tun haben:

- **ALR-refinement trees**: diese interne Komponenente implementiert die gesamte Funktionalität für Graphen, Graphtransformation und Attributverwaltung, und dem Ansteuern des Constraint Solvers zum Finden von Morphismen. Dies wird mit refinement trees implementiert, die nicht direkt als flache attributierte Graphen benutzt werden können. Hier steckt also sozusagen die Theorie.
- **xt_basis**: diese setzt auf obige auf, und wrappt sie fast vollständig. Dies ist die Komponente, die die eigentliche Basis von `AGG` darstellt, nur dass sie eben selbst mit Hilfe einer anderen implementiert ist. Sie stellt viele Hilfsfunktionen bereit, die die Arbeit mit Graphen erleichtert, und stellt insbes. auch das Mapping von refinement trees auf normale attributierte Graphen bereit.
- **Editor**: Für die meisten Klassen aus xt_basis, die Dinge implementieren, die theoretisch auch angezeigt werden können (z.B Knoten, Kanten, Graphen usw.) existiert hier noch einmal eine Wrapper-Klasse, die weitere Funktionalität für das GUI bereitstellt (sie zeigt selbst nichts an). Das GUI benutzt fast ausschließlich Klassen aus dieser Ebene.

Die Frage bei Erweiterungen, die sich also immer stellt, ist auf welcher Ebene man implementieren soll. Falls es nicht rein das GUI betrifft, sicherlich mindestens in **xt_basis** oder **ALR**, aber vorzugsweise nicht in beiden. Ich habe mich entschieden, auf **xt_basis** aufzusetzen.

Im folgenden eine Aufstellung der Basis-Klassen, die sich allesamt im neuen Paket `agg.cons.` befinden, samt einigen Hinweisen:

- `Evaluable:` Die wesentliche Aufgabe, die diese Komponente leistet, ist, über verschiedene Dinge eine Aussage zu treffen, ob sie gültig sind, oder nicht, abhängig von einem Parameter. Dies führt zum Interface `Evaluable`, welche nur eine einzige Methode enthält: `boolean eval(java.lang.Object o)`.

- `EvalSet`: Diese Klasse ist ein `Evaluable`, welches eine Menge von anderen `Evaluables` enthalten kann, und nur dann zu `true` auswertet, wenn alle Elemente dieser Menge dies auch tun. In anderen Worten bildet es eine logische *und*-Verknüpfung nach. Dieser Typ von `Evaluables` wird benutzt, um alle atomaren Anwendungsbedingungen zusammenzufassen, die aus einer Regel und einer Graphinklusion hervorgehen, da diese genau alle erfüllt sein müssen. Man hätte auch direkt eine Formel benutzen können (mit der Methode wie in Kapitel 3 beschrieben), aber auf diese Weise lassen sich bestehende Formeln für Grapheinschränkungen leichter auch für die Formeln in den Anwendungsbedingungen benutzen. Für den GUI-Teil existiert die Methode `Vector getSet()`, welche genau jene Menge zurückgibt.

- `AtomConstraint`: Diese Klasse repräsentiert eine Graphinklusion und implementiert ebenfalls `Evaluable`. Laut Definition ist sie also bloss ein spezieller Morphismus, wofür in xt_basis die Klasse `OrdinaryMorphism` steht, von der wir also erben. Die Methode `boolean isValid()` prüft den Morphismus, ob er injektiv und total, und ob er eingeschränkt auf die Domain die Identität ist, i.e. ob er tatsache eine Graphinklusion darstellt. Die `boolean eval(Object o)` Methode erwartet, daß das Argument eine `Graph` ist, und prüft dann, ob dieser Graph die Graphinklusion erfüllt. Das Verfahren entspricht exakt Definition 2.11.

- `AtomApplCond`: Atomare Anwendungsbedingungen werden hierdurch modelliert. Als solche bestehen sie also aus einem Morphismus für die Vorbedingung (erreichbar durch `getPreCond()`, und den Morphismen t und q aus Diagramm 3.1. Die Methode `eval(Object o)` erwartet einen Morphismus, der einen Comatch darsstellt, und prüft ihn auf Erfülltheit ebenfalls exakt nach Definition. Diese Klasse wird nur von der Klasse `Convert` (s.u.) erzeugt, da zu keinem anderen Zeitpunkt die nötigen drei Morphismen (insbes, q) vorhanden sind. Sie läßt sich also *nicht* zum Modellieren von normalen Nachbedingungen benutzen.

- `Formula`: Wie in den letzten beiden Kapiteln gesehen, wollen wir unsere Einschränkungen jeweils als Formeln über gewissen atomaren Einschränkungen aufbauen. Dazu dient diese Klasse. Sie stellt im Prinzip eine Blackbox dar, zu deren Konstruktion eine Menge (als `Vector`) von `Evaluables` und einen String benötigt wird, und deren Auswertung für einen Parameter dem Ergebnis entspricht, was man erhielte, wenn man alle Zahlen im String durch den Wahrheitswert der Auswertung des `Evaluables` an dieser Position im `Vector` ersetzt, und den resultierenden String als booleschen Ausdruck ansieht und berechnet. Die Methode `boolean isValid()` gibt an, ob die Formel gültig ist, wozu alle Subformeln gültig sein müssen, und sie selbst nicht undefiniert sein darf. Die Interpretation des Strings wird bei Konstruktion vollzogen und intern ein Baum von Formeln aufgebaut, an den man aber nicht herankommt. Falls der String syntaktisch inkorrekt ist, ist die erhaltene Formel undefiniert, und damit ungültig.

Die Methode `String getAsString(Vector v)` gibt einen String zurück, der genau der Formel entspricht, und füllt v mit den atomaren Formelbestandteilen. Da Zahlen im String nur relativ zu einem `Vector` interpretiert werden können, werden beide benötigt, um ein komplettes Bild der Formel zu erhalten. Man kann v schon vorher füllen, dann ist der zurückgegebene String relativ zu diesem v, allerdings werden Atomare, die noch nicht in v waren, trotzdem hinten angehangen.

`setFormula(Vector v, String s)` kann benutzt werden, um die komplette Formel, die ausgewertet wird, zu ersetzen, ohne ein neues Formelobjekt anlegen zu müssen.

- `Convert:` Diese Klasse implementiert das Verfahren zum Umwandeln von Graphinklusionen in eine Menge von atomaren Anwendungsbedingungen. Instanzen sind spezifisch für eine Regel und eine Graphinklusion zuständig. Die Menge der durch beide bestimmten Anwendungsbedingungen kann durch die Methode `Vector convert()` erhalten werden. Die Methode `pushout(m1,m2)` berechnet den Pushout von den Morphismen $m1$ und $m2$, wobei $m2$ als der Morphismus angesehen wird, der Variablen Werte zuweist. Zurückgegeben wird das Paar der resultierenden Morphismen, wobei der erste Morphismus parallel $m1$ ist. Diese Methode ist nur public, falls andere Teile in `AGG` mal Pushouts von Morphismen berechnen müssen, ohne sich immer erst eine Hilfregel konstruieren zu wollen, oder den Targetgraph dadurch zu zerstören. `pushout()` ist unabhängig der konkreten Instanz von `Convert`, könnte also `static` sein.

Um speicher- und ladbar zu sein implementieren die Klassen `AtomConstraint` und `Formel` das Interface `XMLObject`.

Um die durch obige Klassen implementierte Funktionalität in `AGG` bereitzustellen sind noch einige Änderungen in der xt_basis nötig:

- In der Klasse `BaseFactory` wurden Methoden hinzugefügt, um `AtomConstraint`s zu erzeugen, sowie um einen `OrdinaryMorphism` mit einem `MATCH_MAP` mapping zu versehen.
- Eine `Rule` enthält jetzt auch die Menge ihrer Anwendungsbedingungen (also `Formulas`), und die atomaren Anwendungsbedinungen, auf denen sie beruhen, sowie die Accessormethoden, um diese zu lesen, zurueckzusetzen, oder weiter Formeln addieren zu könenn.

- In der Klasse `Match` wird `isValid()` dergestalt geändert, dass bevor die Gültigkeit festgestellt wird, erstmal alle Formeln in der Regel mit dem Co-Match ausgewertet werden, der aus Anwendung dieses Matches resultieren würde. Ein Match ist dann bloss gültig, wenn alle Formeln *true* lieferten.

- `GraGra`: stellt Methoden bereit, um Graphinklusionen und Formeln über diesen zu erzeugen und zu löschen, und bindet dies in das bestehende Framework ein. Darüber hinaus gibt es die neuen Methoden `checkGraphConstraints()` zum Prüfen des Hostgraphen gegen alle Formeln, `checkAtomics()` zum Prüfen gegen alle Graphinklusionen, unabhängig von deren Verwendung in einer Formel, `convertAtomics()`, das unglücklich benannt ist, und für alle Regeln und alle Grapheinschränkungen (also nicht bloss atomare) die Anwendungsbedingungen berechnet, und sie in die Regeln einfügt.

Ein Hinweis zum Löschen von Graphinklusionen: eigentlich würden dadurch alle Formeln ungültig werden, die auf dieser Graphinklusion beruhen. Um sie nicht zu zerstören, wird einfach jedes Vorkommen dieser Inklusion durch die Konstante *true* ersetzt. Man kann die Formel immer noch löschen.

Ein weiterer Hinweis zur Implementierung von `Convert`: Im Verfahren ist es nötig alle Überlappungen der rechten Regelseite und der Prämisse zu bilden, wozu die Methode `OrdinaryMorphism.getOverlappings()` benutzt wird. In Tests stellte sich heraus, dass sie manchmal die gleiche Überlappung mehrfach zurückgibt, dafür dann aber eine andere vergißt. Dieser Fehler tritt nur sporadisch auf, und ich habe noch keine Möglichkeit gefunden, ihn vernünftig zu reproduzieren. Es liegt jedenfalls daran, dass `nextCompletion()` scheinbar nicht immer deterministisch arbeitet. Die Art und Weise, wie `getOverlappings()` dieses benutzt macht es erforderlich, dass bei gegebenem Morphismus zwei Sequenzen von `nextCompletion()` Aufrufen die Totalisierungen in exakt derselben Reihenfolge zurückgeben.

4.2 Pseudocode

Hier sollen die Algorithmen zum Erzeugen der Nachbedingungen und zum Überprüfen derselben angegeben werden. Zuerst das Erzeugen:

```
createConstraints:
  forall R in Rules:
    clearConstraints(R)
    S = empty
    forall A in Atomics:
```

```
      E = new EvalSet (convert (R, A))
      S.append E
    rof
    R.atoms += S
    forall F in Formulas:
      F' = transscribe (F, R.atoms)
      R.formulas += F'
    rof
  rof
```

Jetzt das Konvertieren eines Atomics für eine Regel in eine Menge von atomaren Anwendungsbedingungen:

```
convert (rule, atomic):
  result = empty
  O = allOverlappings (rule.right, atomic.left)
  forall (s, p) in O:
      // s: rule.right --> S, p: atomic.left --> S
    if (!isInjective (p)) continue
    (t, q) = pushout (atomic, p)
      // t: S --> T, q: atomic.right --> T
    result += new AtomApplCond (s, t, q)
  rof
  return result
```

Und schliesslich die Methode eine atomare Anwendungsbedingung gegen einen Comatch zu testen:

```
check (comatch, applcond):
    // comatch: R --> H
  (s, t, q) = applcond
  forall p in Completions(S --> H):
    if (p*s != comatch) continue
    f = p.completeDiagram (t)
    if (f*t != p) continue
    found = false
    if (! isTotal (f)):
      forall f' in Completions (f):
        if (isInjective (f'*t) && isInjective (f'*q)):
          found = true
          break
        fi
      rof
```

```
    else:
      if (isInjective (f*t) && isInjective (f*q)) found = true
    fi
    if (!found) return invalid
  rof
  return valid
```

Damit schliesst die Implementierung der Basisfunktionalität ab.

4.3 Graphische Komponente

Um die am Beginn des Kapitels vorgestellten Anforderungen implementieren zu können müssen einige Entscheidungen zur graphischen Repräsentation getroffen werden:

- Die Funktionalität zum Prüfen des Hostgraphen und zum Berechnen der Anwendungsbedingungen soll in einem Submenü des Hauptmenüs zugänglich sein.
- Alle Graphinklusionen und Formeln darüber sollen im TreeView in der GUI von `AGG` auftauchen, also in derselben Ebene wie aktuell schon Regeln. Sie sollen einen Namen tragen.
- Die aus einzelnen Graphinklusionen resultierenden Mengen von atomaren Anwendungsbedingungen sollen als Unterbäume der Regeln dargestellt werden.
- Formeln sollen über einen modalen Dialog editiert werden. Er zeigt alle aktuellen Graphinklusionen der Graphgrammatik samt einer laufenden Nummer an, und stellt ein Textfeld bereit in dem initial die alte Formel steht, die verändert werden kann. Die Zahlen in der eingegebenen Formel werden relativ zu der angezeigten Menge von Graphinklusionen interpretiert.
- Graphinklusionen sollen wegen ihrer Nähe zu Regeln genauso editiert werden, mit den offensichtlichen Einschränkungen, dass sie keine NACs besitzen, sowie, dass es Inklusionen sein müssen (d.h. die rechte Seite darf die linke bloss um Objekte erweitern, und nichts darf gelöscht werden). Gültigkeit wird allerdings erst beim Prüfen des Hostgraphen geprüft, da beim editieren temporär Nicht-Inklusionen entstehen können.
- Das Prüfen des Hostgraphen resultiert in einer Messagebox, welche das Ergebnis (Graphinklusionen ungültig, Hostgraph verletz, oder Hostgraph erfüllt Bedingungen).

- Ein alternatives GUI für das Browsen durch die Anwendungsbedingungen der Regeln soll bereitgestellt werden. Rationale hierfür ist, dass es potentiell relativ viele werden können (es kann schon pro Graphinklusion viel werden, und dadurch insgesamt sehr unübersichtlich).

  Es wird dafür eine andere Oberfläche entworfen, die bei Bedarf den Inhalt von `AGG` 's Hauptfenster ersetzt. Sie soll aus zwei Listen bestehen, die alle Regeln, bzw. Graphinklusionen der Graphgrammatik anzeigen, und damit eine Auswahl zulassen, einem Bereich, der ermöglicht, eine atomare Anwendungsbedingung (drei Morphismen) anzuzeigen, sowie ein paar Tasten zum Scrollen in der Liste der Überlappungen für die aktuelle Regel und Graphinklusion. Dieses GUI soll im Constraint-Submenü des Haupmenüs ein- und ausschaltbar sein.

Analog der vorhandenen Struktur in `AGG` wird für jede GUI-Komponente, die ein Element aus der xt_basis repräsentiert eine Wrapper-Klasse angelegt, die vom GUI benötigte Funktionalität (z.B. Namen) bereitstellt.

Bei uns sind dies nur `AtomConstraint` und `Formula`. `Convert` repräsentiert überhaupt nichts, `EvalSet` und `AtomApplCond` werden in den Anwendungsbedingungen für Regeln benutzt, brauchen aber für das Haupt-GUI keine Extrafunktionalität. Ergo ergeben sich zwei neue Klassen, die im Diagramm 4.3 dargestellt sind.

Wie oben beschrieben ist `EdAtomic` nur eine spezielle `EdRule`, ohne NACs und beruhend auf einem Morphismus anstatt einer `Rule`. Hier wäre es ganz offenbar schöner gewesen, die Funktionalität aus `EdRule` die nur für Morphismen zuständig ist, herauszufaktorisieren, und diese dann als `EdAtomic` benutzen, und `EdRule` selbst dann bloss um die nötige Funktionalität zu erweitern, anstatt diese Funktionalität quasi nachträglich abzuschalten.

`EdConstraint` wrappt eine `Formula`, und stellt bloss einen Namen für sie bereit. Um diese Klassen in das Framework zu bekommen wird `EdGraGra` so erweitert, daß es auch eine Menge von `EdAtomics` und von `EdConstraints` enthalten kann, die eben die Menge aller Graphinklusionen und Formeln in einer Graphgrammatik beschreiben.

Die Funktionalität des TreeViews im Haupt-GUI von `AGG` befindet sich in der Klasse `GraGraTreeView`, die entsprechend erweitert wird. I.e. zwei neue Popup-Menu Klassen für Graphinklusionen und für Formeln werden benötigt, deren Aktivierung im TreeView muß eingebaut werden, neue Kommandos für die auslösbaren Aktionen müssen her, und ähnliches. Der Regeleditor der schon für normale Regeln vorhanden ist, funktioniert, da `EdAtomic` eine `EdRule` ist, nach nur minimalen Anpassungen wie erwarten.

Der oben besprochene Dialog für Formeln ist in der Klasse `FormulaDialog` implementiert. Er wird im TreeView durch das Popupmenu von Formeln aktiviert.

Die Darstellung der atomaren Anwendungsbedingungen übernimmt die Klasse `TwoMorphs`, welche drei Graphen darstellen kann, samt den Markierungen für zwei Morphismen die eine Komposition darstellen müssen. Bei Bedarf ersetzt diese Darstellung den Regeleditor.

Das alternative GUI für Anwendungsbedingungen ist in den Klassen `AGGConstraints`, `ConstraintsGUI` und `RuleConstraint` implementiert. Dabei stellt erstere nur den Umschalter zwischen Haupt-GUI und diesem, und das Hauptmenü bereit. `ConstraintsGUI` implementiert ein `JPanel`, in dem sich links zwei Tree-Views zum Anzeigen und Auswählen von Regeln und Graphinklusionen befinden, und rechts das `RuleConstraint`-Panel. Wenn der Nutzer in die Treeviews klickt werden die aktuelle ausgewählte Regel und Graphinklusion diesem rechten Panel mitgeteilt.

`RuleConstraint` implementiert ebenfalls ein `JPanel`, und zwar zum Anzeigen einer Menge von atomaren Anwendungsbedingungen. Dazu befinden sich oben zwei Tasten zum Vor- und Zurückschalten des aktuell angezeigten Atomars, und darunter drei `SplitPanes` die die drei Morphismen des Atomars darstellen. Alle drei sind read-only. In Abbildung 4.1 ist dieses GUI zu sehen (hoffentlich).

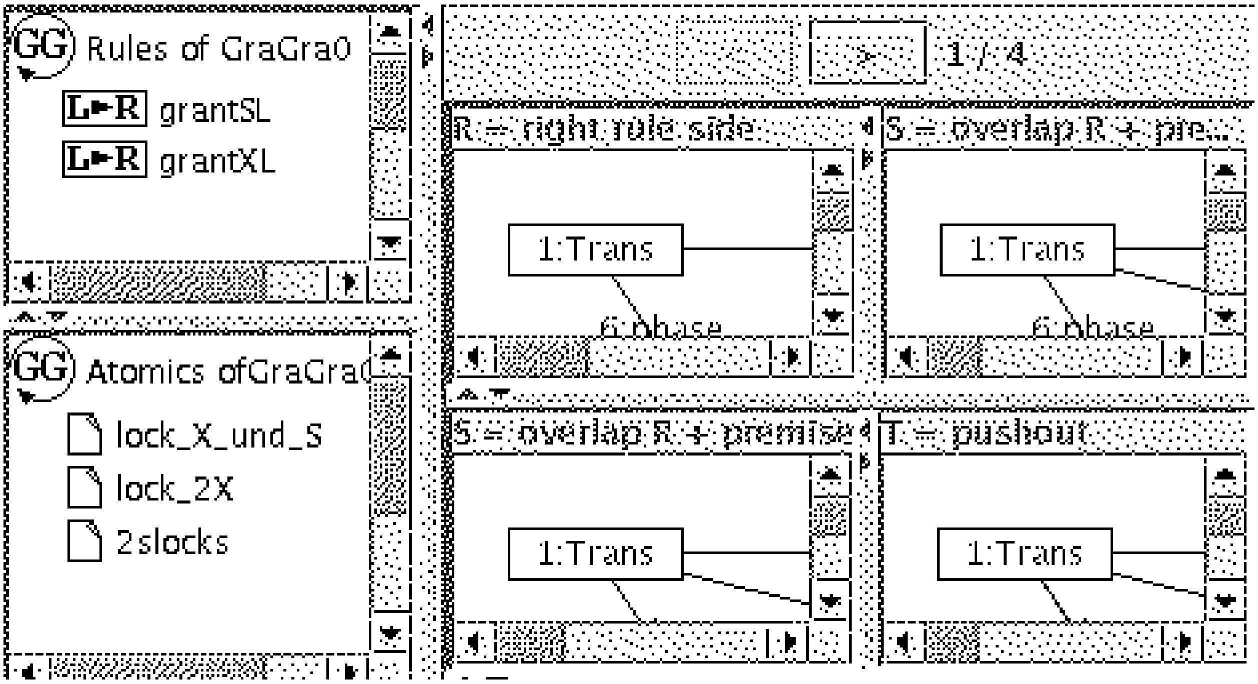

Abbildung 4.1: GUI zum Anzeigen der Anwendungsbedingungen

4.4 Die CD-ROM

Das Hauptverzeichnis der CD-ROM sieht folgendermaßen aus:

```
README
kompilat/
sourcen/
tests/
text/
```

Die Datei `README` gibt ebenfalls einen Überblick über den Inhalt der CD-ROM. Im Verzeichnis `tests/` befinden sich verschiedene Eingabedateien für `AGG` , darunter jenen, die im nächsten Kapitel referenziert werden. Im Verzeichnis `text/` befindet sich der Text dieser Arbeit in verschiedenen Formaten. Dazu gehören alle TEX-Sourcen, die eingebundenen Postscript-Dateien, deren Xfig-Sourcen, und das erzeugte Postscriptfile. Die Version auf CD-ROM ist allerdings reiner Zusatz. Woimmer sich dieses Dokument und die Version auf CD-ROM unterscheiden, ist dieses Dokument definitiv (im Falle, ich muß ein paar last minute Änderungen machen, aber die CD-ROM ist schon gebrannt ;-)) Im Verzeichnis `sourcen/` liegen die Java-Sourcen für

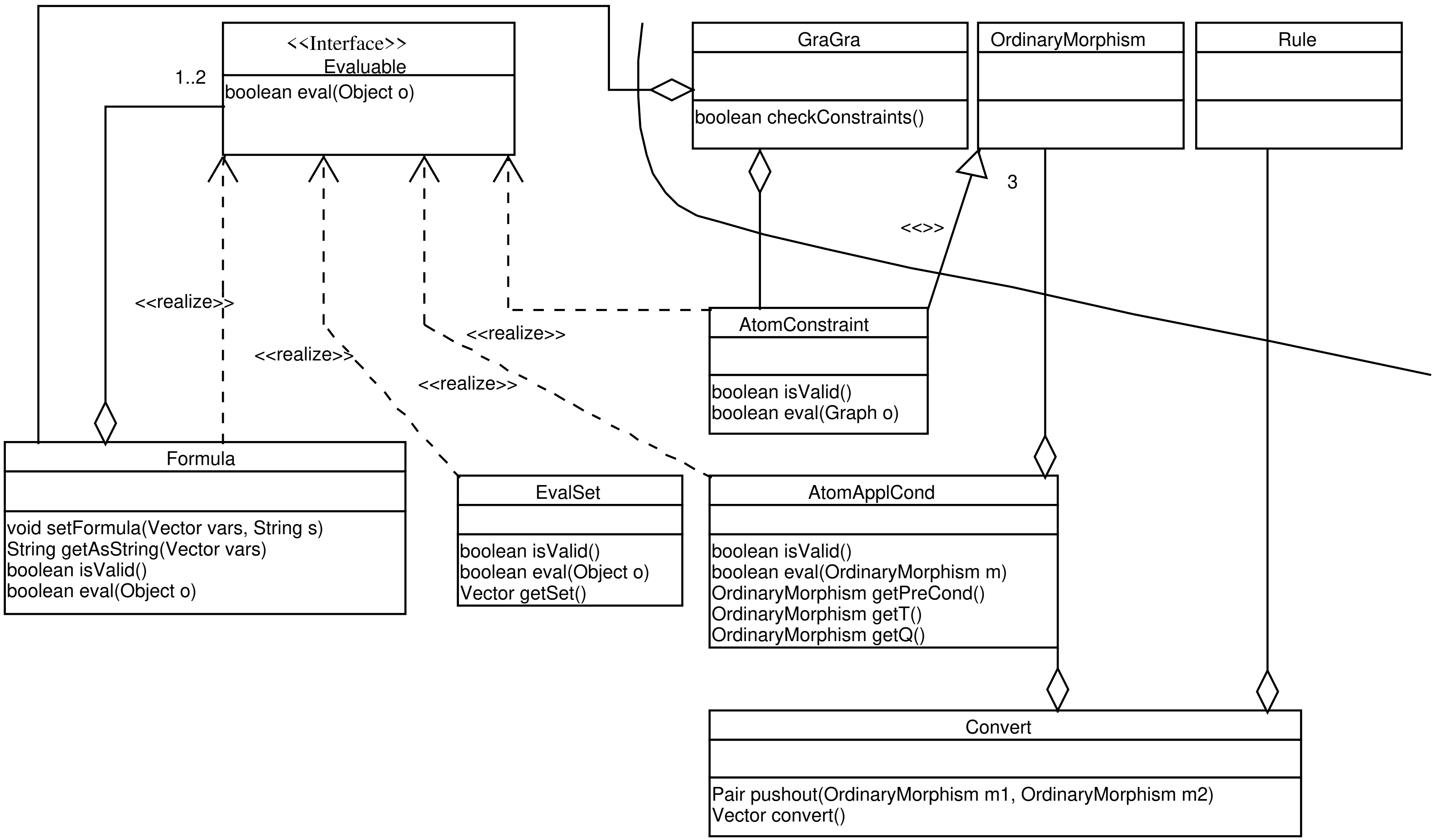

Abbildung 4.2: Klassendiagramm für die Basisfunktionalität

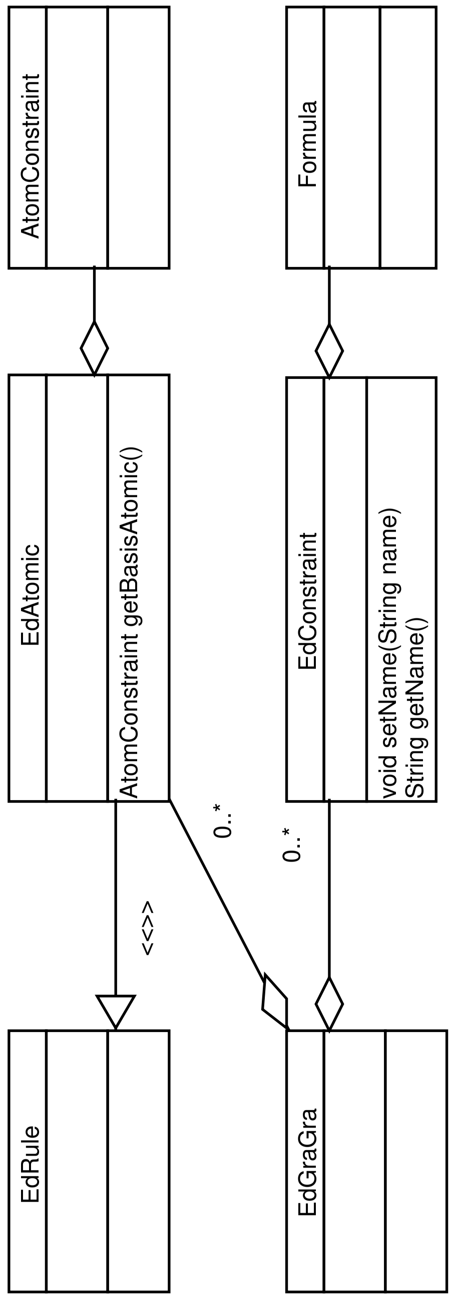

Abbildung 4.3: Klassendiagramm für die Editor Wrapper Klassen

das gesamte `AGG` System. Die Implementierung ist dort eingebunden. Diese Version von `AGG` basiert auf der Version aus dem CVS vom 12.7.2001, ein diff gegen diese Version erzeugt also alle Änderungen des Autors. Im Verzeichnis `kompilat/` schließlich liegen die `.class` Files der Dateien in `sourcen/`. Zur Einfachheit befindet sich dort auch eine Datei `agg.jar`. Damit kann `AGG` einfach mit

```
matz@hell > cd /cdrom/kompilat ; java -jar agg.jar
```

gestartet werden. Allerdings muss sich sowohl Xerces, als auch JGL im CLASSPATH befinden. Diese Version von `AGG` wurde mit Jikes übersetzt und sollte mit Java 1.2 und 1.3 laufen.

Kapitel 5

Benutzung der Konsistenzkomponente in `AGG`

Dieses Kapitel soll den Zugang des Nutzers zu der neuen Funktionalität vereinfachen. Es wird davon ausgegangen, daß er schon Erfahrung mit **`AGG`** hat.

Die typische Reihenfolge der Arbeitschritte ist folgende:

- Erzeugen einer neuen, leeren Graphgrammatik oder Laden einer bestehenden
- Anlagen der gewünschten Regeln
- Anlegen der atomaren Grapheinschränkungen
- Optionales einzelnes Prüfen der atomaren Grapheinschränkungen an einem Testhostgraphen
- Erzeugen der Constraints (also der booleschen Formeln). Evtl. Testen derselben am Hostgraphen.
- Erzeugen der Anwendungsbedingungen für die Regeln.
- Analysieren der erzeugten Anwendungsbedingungen oder Anwenden der Regeln.

5.1 Anlegen und Editieren von atomaren Grapheinschränkungen

Eine neue leere atomare Grapheinschränkung wird durch den entsprechenden Menüpunkt im Kontextmenü der Graphgrammatik angelegt. Editiert wird sie genau wie eine Regel (die ja auch bloss ein spezieller Morphismus ist), hat aber natürlich keine NACs. Da sie eine Graphinklusion sein muß, um gültig zu sein, empfiehlt es sich zuerst die linke Seite (die Vorraussetzung) zu editieren, und dann die rechte Seite samt den Mappings automatisch durch die `identic rule` Aktion (Taste `IR`) erzeugen zu lassen.

Im Kontextmenü für Atomics findet sich die Möglichkeit dieses Atomic gegen den aktuellen Hostgraphen zu testen. Erst zu diesem Zeitpunkt wird die Wohlgeformtheit des Atomics geprüft, während des Editierens kann es also temporär die Eigenschaften einer Graphinklusion verletzen. Das Ergebnis dieser Überprüfung (*Atomic invalid*, *Graph erfüllt nicht* oder *Graph erfüllt*) wird in der Statusleiste und in einer Messagebox dargestellt.

Für die genaue Definition, wann ein Graph eine Grapheinschränkung erfüllt sei auf Definition 2.11 verwiesen. In Kurzform: an allen Stellen an denen die Elemente der linken Seite der Grapheinschränkung im in Frage stehenden Graph vorkommen müssen auch die Elemente der rechten Seite vorhanden sein.

5.2 Anlegen und Editieren von Formeln (Constraints)

Ebenso wie Atomics werden Formeln durch die entsprechende Aktion im Kontextmenü der Graphgrammatik angelegt. Die Formel für das neu erzeugte Graphconstraint is *true*. Editiert wird sie in einem Dialogfenster, welches sich durch die `Edit` Aktion im Kontextmenü des Constraints aufrufen läßt. Siehe Abbildung 5.1.

Die boolesche Formel benutzt als Variablen ja die Grapheinschränkungen (siehe auch Definitionen 2.12 und 2.13), die also irgendwie zu bezeichnen sind. Die erste Idee wäre die Namen der Atomics selbst zu benutzen. Dies ist suboptimal: erstens müßte der Benutzer potentiell recht lange Namen evtl. mehrfach eingeben; zweitens müßte man die Bezeichner umständlich quoten (vielleicht durch Einschließen in "), falls Leerzeichen oder gar Formelzeichen in ihnen vorhanden sind.

Um die Eingabe von Formeln zu erleichtern wird eine auf den ersten Blick etwas sonderbare Funktionsweise gewählt. Jeder in der Graphgrammatik vorhandenen Grapheinschränkung wird eine Nummer zugeordnet (von 1 beginnend). Die kongrete Zuordnung ist im oberen Teil des Dialogs zu sehen. Die Formel benutzt nun diese Zahlen als Bezeichner der entsprechenden Atomics. Auf diese Weise kann man sie zügig eingeben, und sie wird nicht durch umständliche Quotingmechanismen unübersichtlich. Abgesehen von der Bezeichnung für Variablen entspricht die Syntax der Formeln denen

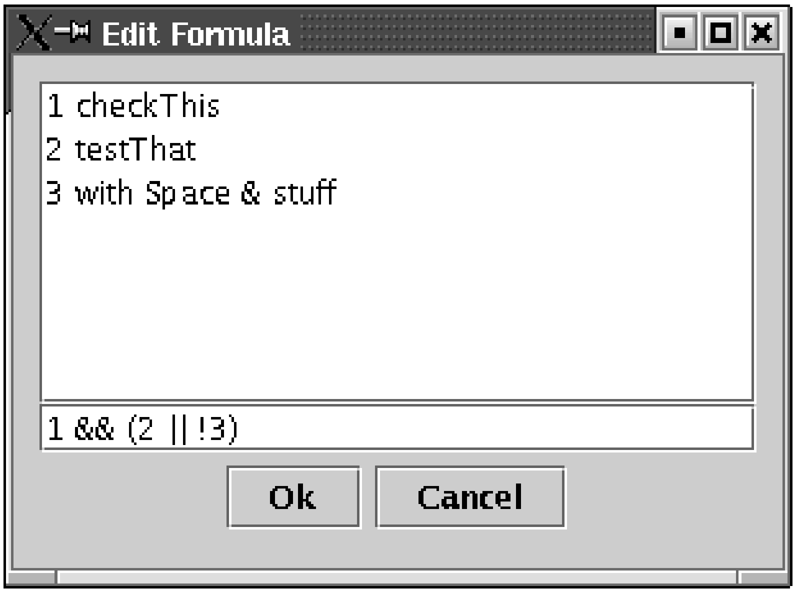

Abbildung 5.1: Formeleditor für Constraints

von `C` für boolesche Ausdrücke, komplett mit Klammern und den Operatoren !, &&, ||. Ausserdem haben die Konstanten $true$ und $false$ die offensichtliche Bedeutung.

Ein Beispiel: Sagen wir, wir haben zwei Atomics mit Namen `onlyThat` und "`ensure something`" und wollen beide in einer Verundung zusammenfassen. Die Formel dafür würde 1&&2 sein.

Die Formel selbst wird auch im Tooltip im Treeview angezeigt. Im Kontextmenü der Constraints befindet sich auch die Möglichkeit sie gegen den aktuellen Hostgraphen zu testen. Bei Auswahl wird zuerst die Gültigkeit der Formel und der beteiligten Atomics geprüft, dann wird pro Atomic der Graph auf Erfüllung desselben getestet, was je nach Ergebnis zu einem $true$ oder $false$ Wert für diese Variable in der Formel führt. Damit wird dann die Formel ausgewertet.

Wiederum ein Beispiel: Die Formel in Frage sei !1, und diese 1 entspricht Atomic `testThis`. Vorrausgesetzt, der aktuelle Hostgraph erfüllt `testThis` **nicht**, so erfüllt er aber die Formel.

5.3 Erzeugen und Anzeigen von Anwendungsbedingungen

Gesetzt den Fall, man hat einige Atomics und mindestens eine Formel erzeugt, so kann man daraus Anwendungsbedingungen für alle vorhandenen Regeln erstellen. Dies kann für ein ausgewähltes Atomic geschehen (aus seinem Kontextmenü heraus) oder für die Gesamtheit aller Formeln und Atomics (im Kontextmenü der Graphgrammatik).

Wie genauer in Konstruktion 3.4 beschrieben, wird pro Atomic und pro Regel eine Menge von atomaren Anwendungsbedingungen erstellt. Diese Menge erhält den gleichen Namen wie der Atomic aus dem sie resultierte. Auf diese Weise kann sie wiederum als atomare Formel interpretiert werden. Gegeben einen Comatch wird sie zu $true$ belegt, falls dieser **alle** Elemente der Menge erfüllt, wie in Definition 2.14 beschrieben. Ein Comatch insgesamt ist gültig, wenn er alle in der Graphgrammatik vorhandenen Formeln (i.e. die Constraints) erfüllt. Diese Formeln sind syntaktisch die gleichen, wie die oben beschriebenen, nur dass diesmal die Namen nicht für Grapheinschränkungen sondern Mengen von atomaren Anwendungsbedingungen stehen. (vergleiche Definition 2.15 und Konstruktion 3.2)

Falls nur ein ausgewähltes Atomic umgewandelt wurde, so wird also nur eine solche Menge erzeugt, und es wird keine Formel benutzt um Gültigkeit von Comatches zu prüfen. Anstattdessen erfüllt ein Comatch solch eine Nachbedingung, wenn er einfach die gesamte Menge erfüllt (d.h. es lässt sich so auffassen, dass die Regel nur eine Formel hat, welche selbst atomar ist, und die Menge bezeichnet).

In der GUI werden diese Mengen als Subbäume unter jeder Regel dargestellt (ihr Name erinnert an den Quellatomic), für jeden Atomic einer. In diesem Subbaum befinden sich die Elemente der Menge, also atomare Anwendungsbedingungen. Dargestellt werden sie durch Ihre Morphismen s und t (vrgl. Abb. 3.1). Man beachte, dass diese Graphpanels "read-only" sind. Im alternativen GUI wird ausserdem noch der Morphismus q dargestellt.

5.4 Löschen von Atomics oder Constraints

Im Kontextmenü kann man sowohl einzelne Atomics als auch Constraints (Formeln) löschen. Falls ein Atomic gelöscht wird, der noch in einer Formel referenziert wird, so wird diese Referenz durch die Konstante $true$ ersetzt.

Ausserdem beachte man, daß das Löschen oder Anlegen von Atomics oder Constraints dazu führt, dass alle erzeugten Nachbedingungen in allen Regeln ungültig und damit gelöscht werden. Falls der Benutzer Atomics ändert, so muss er selber dafür sorgen, dass die Nachbedingungen neu erzeugt werden, da sie in diesem Fall nicht gelöscht werden.

5.5 Anwenden der erweiterten Regeln

Falls Anwendungsbedingungen erzeugt wurden, können Regeln nur noch angewendet werden, falls der Comatch all Bedingungen erfüllt. Es werden also Nachbedingungen erzeugt. Intern wird also die Regel angewendet, und dann das Ergebnis überprüft. Falls es ungültig ist, wird es verworfen, und die gesamte Anwendung der Regel am gegebenen Match ist ungültig. Diese Beschränkung greift sowohl bei versuchter Anwendung einer kongreten Regel (an einem gegebenen oder selbst gesuchten Match), als auch bei automatischer Ausführung von Ableitungsschritten mit der gesamten Regelmenge.

Kapitel 6

Test des Konsistenzcheckers

Es werden nun einige Beispielsituationen vorgestellt, und was die Implementierung daraus macht. Dabei wird auch auf Probleme in der aktuellen Implementierung eingegangen. Ursprünglich war vorgesehen, die ACID-Eigenschaften für Transaktionen in Datenbanksystemen zu untersuchen. Die dazu notwendige Grammatik mit Regeln und Intergritätsbedingungen befindet sich auch auf der zu dieser Arbeit gehörenden CD-ROM (in `tests/acid2.ggx`), jedoch ergaben sich derartig viele Verklebungen (und selbst die erst, nachdem die Menge der Regeln und Einschränkungen empfindlich reduziert wurden, Datei `tests/acid_small_grant.ggx`), dass erstens `AGG` anfing in die Kniee zu gehen, und es zweitens unpraktisch erscheint alle resultierenden Anwendungsbedingungen von Hand zu überprüfen, und mit [1] zu vergleichen (ein Vergleich wäre nur beschränkt möglich, da wir keine Vorbedingungen erzeugen, sondern es bei Nachbedingungen belassen).

Da Umwandlung von Screenshots nach Postscript entweder riesige Dateien erzeugt, oder häßliche Ergebnisse (siehe die schon gebrachten Beispiele im letzten Kapitel, diese Dateien waren klein), werde ich hier keine Screenshots von `AGG` verwenden, sondern die entsprechenden Graphen mit Xfig nachzeichnen.

6.1 unattributiertes Beispiel

Betrachten wir also zunächst die Graphgrammatik aus Abbildung 6.1. Die zugrunde liegende `AGG` Datei ist `tests/ensure_after.ggc` :

Die Grammatik besteht nur aus einer Regel, die an einen Knoten einfach einen neuen Knoten anhängt. Die Grapheinschränkung fordert für jeden

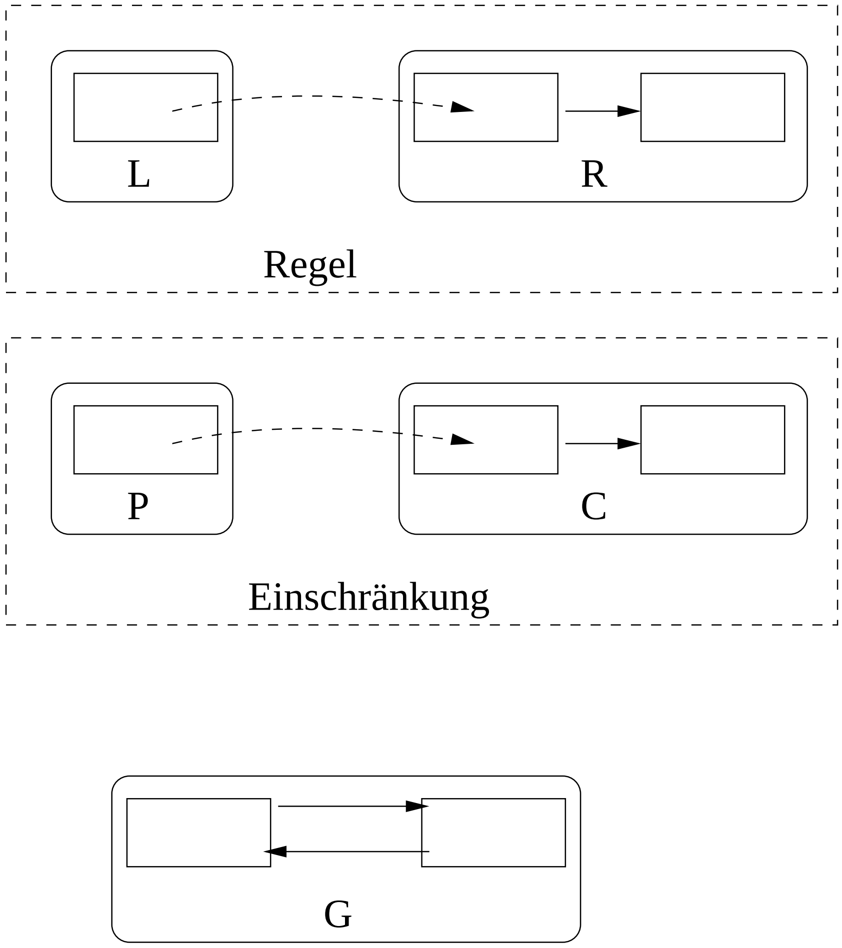

Abbildung 6.1: Einfache unattributierte GraGra mit Regel, Einschränkung und Hostgraph

Knoten eine ausgehende Kante samt daranhängendem Knoten. Der Startgraph ist der minimalste nicht leere Graph, der diese Bedingung erfüllt (alle Graphen, die sie erfüllen bestehen aus Zyklen, wie sich leicht überlegen läßt). Da die Bedingungen nämlich injektiv erfüllt sein sollen, erfüllt der Graph aus einem Knoten mit einer Loop (der minimalste nicht leere Graph, der einen Zyklus enthält) diese nicht.

Wie sich ebenfalls erkennen läßt, darf die Regel nie angewendet werden. Der neu erzeugte Knoten würde nicht den geforderten Nachfolger besitzen. Für diese Regel und diese Einschränkung werden drei Überlappungen erzeugt, die in Abb. 6.2 dargestellt sind:

Dabei führt Überlappung S1 zu einem Dreieck, das fordert, dass an dem

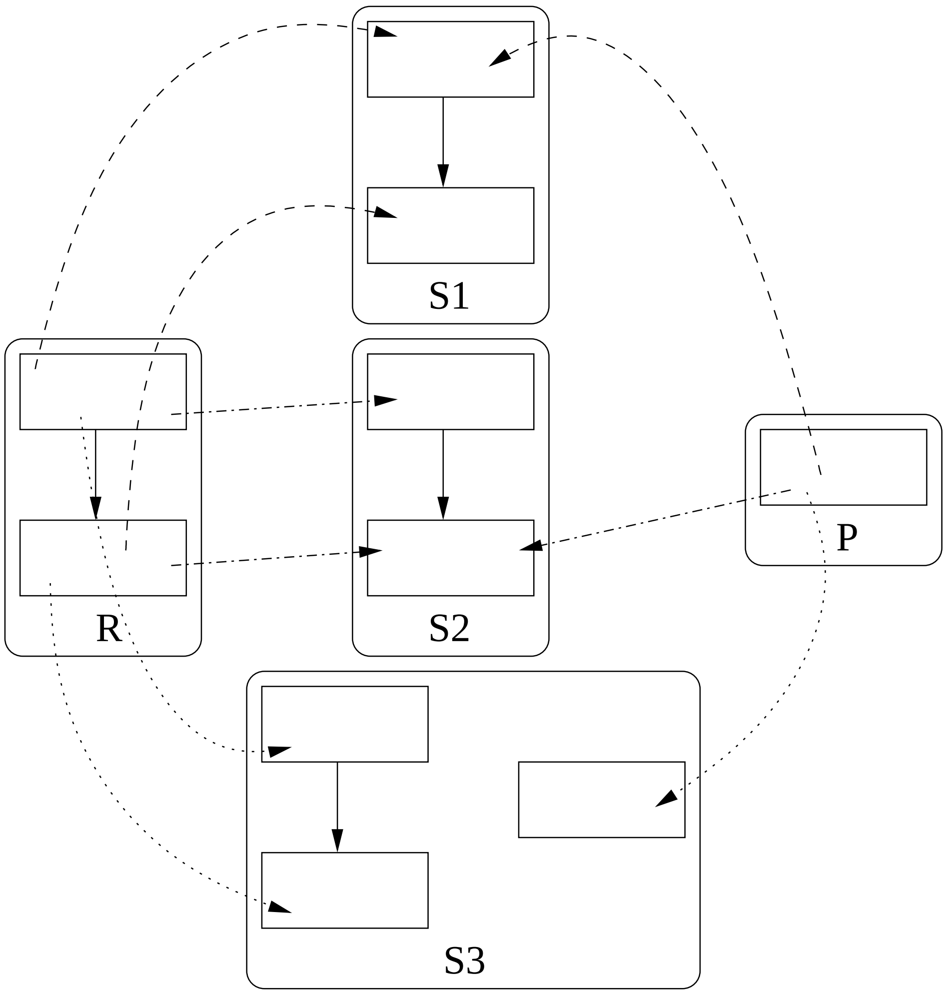

Abbildung 6.2: Überlappungen für die GraGra aus Abb. 6.1

Knoten, an den der neue angehangen wird, schon ein Knoten hängen mußte. Dies ist trivialerweise erfüllt, wenn der Sourcegraph die Einschränkungen erfüllte. Selbst wenn nicht, wird ja jetzt ein Knoten angehangen, womit die Bedingung an **diesem** Knoten dann erfüllt wäre. S1 ist also nutzlos.

Verklebung S3 ist ebenfalls nutzlos (es ist die disjunkte Vereinigung), da sie die Applikation der Regel nur gestattet, falls alle anderen Teile des Hostgraphen, die nicht an der Regelanwendung beteiligt waren, nun die Grapheinschränkung erfüllen. Ebenfalls trivial erfüllt, wenn der Graph die Einschränkungen vor Regelanwendung erfüllt.

Interessanter ist Verklebung S2. Sie führt im Pushout zu einer Kette von drei Knoten, die genau fordert, dass auch an dem neu erzeugten Knoten ein anderer Knoten hängen muss. In diesem einfachen Fall ist leicht einzusehen, dass die Regel dies unabhängig vom Kontext nicht leisten kann (es werden ja keine anderen Objekte erzeugt, zumindest eine neue Kante für den neuen Knoten müsste es aber sein). Dies führt dazu, daß Verklebung S2 jeden Co-Match verwirft, mithin also jeden Match, und damit die Regel an sich.

In `AGG` kann man sich davon überzeugen: Man lade obige Datei, erzeuge die Anwendungsbedingungen. Dann versuche man einen Match zu finden ("Next Completion"). `AGG` wird melden, daß eine solche nicht existiert. Bevor man dies tut sollte man sich überzeugen, dass tatsache diese drei Verklebungen erzeugt wurden. Wie oben schon angedeutet verrechnet sich `getOverlappings()` manchmal und entweder S1 oder S2 wird doppelt erzeugt, die andere dafür aber weggelassen.

Man beachte in obigem Beispiel, dass weitere Verklebungen von P und R nicht erzeugt werden. Auch das Zusammenziehen auf nur einen Knoten ist eine solche Verklebung, sowie das Zusammenziehen nur von R. Beide solche Verklebungen würden nur relevant werden, wenn nicht-injektive Co-matche zugelassen werden, trotzdem sollten sie erzeugt werden.

6.2 Beispiel mit Attributen

Jetzt kommen wir zu einer Graphgrammatik mit Attributen, die hier auch für die Einschränkung relevant sind. Die zugehörige Datei ist `tests/ensure_pred.ggx`, und sie ist in Abbildung 6.3 dargestellt:

Wie man erkennt spielen hier Kanten nur die Rolle als Markierer, die bestimmte Knoten auszeichnen, aber keine unterschiedlichen Knoten untereinander verbinden. Die Regel löscht einfach irgendeinen Knoten, solange dann noch einer übrig bleibt. Die Einschränkung verlangt, dass für jeden markierten Knoten, die eine Zahl als Attribut hat, ein beliebiger anderer Knoten im Graphen existiert, der den Vorgänger dieser Zahl enthält. Die Markierungen sind nötig, um irgendwo in dieser Kette aufhören zu können, da natürliche Zahlen nicht zyklisch sind, also erfüllende Graphen isomorph zu $\mathbb{N}$ sein müssten, wir aber nur endliche Graphen betrachten.

Der Effekt dieser Einschränkung ist, den Regelanwendungen eine gewisse Reihenfolge aufzuzwingen. Es darf kein Knoten gelöscht werden, solange noch ein Knoten existiert, der den Nachfolger der Zahl darin enthält. Ergo dürfen nur jeweils die Enden der Ketten gelöscht werden.

Die zwei Verklebungen die in dieser Grammatik resultieren sind trivial: Einmal werden beide Knoten der rechten Regelseite und Prämisse auf denselben Knoten abgebildet, einmal ist es die disjunkte Vereinigung. Die erste Verklebung führt dazu, dass die Regel nicht von einem Knoten den

eigenen Vorgänger entfernt. Die disjunkte Vereinigung führt dazu, das die Regel nicht den Vorgänger eines **anderen** Knoten löscht. Wegen der injektiven Erfülltheit ist der eine Fall nicht durch den anderen abgegolten (in egal welcher Richtung).

Auch dies kann man in `AGG` gut verfolgen. Man lade die obige Datei, erzeuge die Anwendungsbedingungen, und starte die automatische Transformation. `AGG` wird die beiden Ketten im Hostgraphen von hinten abbauen, bis nur ein Knoten übrig bleibt. Ohne die Anwendungsbedingungen würde `AGG` beliebig im Hostgraphen löschen. **Achtung:** Für dieses Beispiel muss die `dangling`-Bedingung für Matche in den Einstellungen ausgeschaltet werden. Andernfalls wird der zu löschende Knoten niemals auf einen der markierten gematcht, und im Falle dass die Nachbedingungen vorhanden sind können nur solche gelöscht werden, d.h. es würde überhaupt kein gültiger Match gefunden werden.

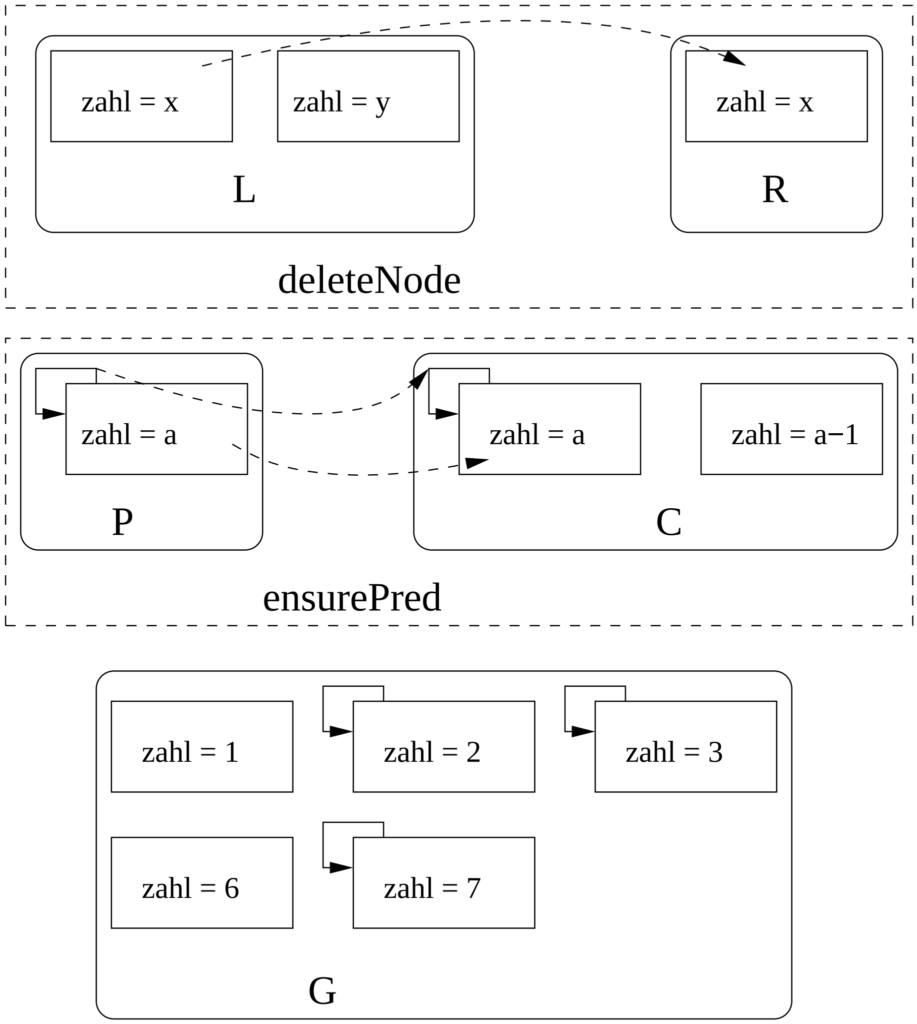

Abbildung 6.3: Einfache attributierte GraGra mit Regel, Einschränkung und Hostgraph

Kapitel 7

Fazit - Ausblick

Der Fehler in `getOverlappings()` sollte analysiert, und wenn nötig, beseitigt werden. Weiterhin ist die Anzeige der drei Morphismen für eine atomare Anwendungsbedingung nur suboptimal wie man schnell mitbekommt, wenn man eine große Anzahl von solchen zu untersuchen hat. Dies liegt zum Teil am unoptimalen automatischen Layout von `AGG` , zum Teil allerdings auch daran, das von den sechs dargestellten Graphen jeweils zwei identisch sind (S und T), was der Übersichtlichkeit schadet.

Wie das oben erwähnte Beispiel der ACID-Eigenschaften zeigt, und auch schon am kleinen ersten Beispiel sichtbar wird, werden äußerst viele redundante Verklebungen erzeugt. Es sollte sich also ein Mechanismus überlegt werden, der ähnlich dem in Kapitel 3 erläuterten Verfahren die unrelevanten Verklebungen herausfiltert, ohne aber das Vorziehen der Anwendungsbedingung zu erfordern. Alternativ sollte man das Verfahren zum Vorziehen implementieren und vergleichen, welche die geringere Anzahl von Morphismen erzeugt (und vielleicht sogar ebenfalls erlaubt, die Verklebungen von T wegzulassen).

Mit grösseren Grammatiken (und gross ist relativ, schon das relativ kleine Beispiel in `tests/acid_small_grant.ggx` zählt dazu) kann es durchaus eine beträchtliche Zeit dauern, bis alle Anwendungsbedingungen erzeugt sind. Dies liegt hauptsächlich an der Erzeugung aller Überlappungen, die eine Pushout-Konstruktion auf allen Subgraphen verwendet um Verklebungen zu erhalten. Das Pushout wird mit einer Regelanwendung simuliert, was relativ grossen Overhead erzeugt. Überhaupt ist der ganze Mechanismus in `AGG` , nämlich dass die ganzen Ref-Trees Observer ihrer Inhalte sind nicht unbedingt optimal, um mal schnell einen Morphismus anzulegen auf ihm zu arbeiten (oder auf einem seiner Graphen), und wieder wegzuwerfen. Die Observer wären hierbei nicht nötig, machen alles aber sehr zeitaufwendig.

Im Ergebnis läßt sich die Implementation für sehr kleine Grammatiken für den Konsistenznachweis verwenden, wobei dieser Nachweis allerdings wegen der hohen Zahl der redundanten Verklebungen für den Benutzer nicht unbedingt leicht ersichtlich ist. Er besteht eher darin, daß `AGG` zusichert, keine Graphen zu erzeugen, die inkosistent sind. Wie dies erreicht wird ist zwar in den Anwendungsbedingungen kodiert, allerdings nicht leicht zu erkennen.

Literaturverzeichnis

[1] Manuel Koch. Modellierung und nachweis der konsistenz von verteilten transaktionsmodellen für datenbanksysteme mit algebraischen graph-grammatiken. Technischer Bericht, TU Berlin, 1996.

[2] Manuel Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems.* Dissertation, TU Berlin, 2000.