Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation

Michael Rudolf

Diplomarbeit

am Fachbereich Informatik der Technischen Universität Berlin
betreut von Prof. Dr. Hartmut Ehrig und Dr. Gabriele Taentzer

Dezember 1997

Zusammenfassung

Diese Arbeit beschreibt die objektorientierte Entwicklung des algebraischen Graphtransformationssystems AGG. Es handelt sich dabei um ein vollständig neuentwickeltes Nachfolgesystem des AGG von [Bey91], das den Graphbegriff der ALR-Graphen übernimmt, zusätzlich aber vereinfachte Schnittstellen für das Arbeiten mit herkömmlichen Graphen zur Verfügung stellt. Wesentliche Teile der Arbeit befassen sich mit der Integration von vorhandenen Bibliotheken für die Attributierung [Mel97] und Transformation [Wol97] von Graphen. Einen weiteren Schwerpunkt bildet das Problem der automatischen Ansatzsuche, das als Constraint Satisfaction Problem (CSP) formuliert und implementiert wird. Zu diesem Zweck wird ein allgemeines Framework zur Lösung von CSPs vorgestellt.

Das erstellte System soll als Grundlage für die praktische Erprobung theoretischer Ergebnisse aus dem Bereich der algebraischen Graphtransformation dienen.

An dieser Stelle möchte ich mich bei all jenen bedanken, ohne die diese Arbeit nicht so __________________ geworden wäre. Besonders erwähnen möchte ich:

- ♢ *Gabi Taentzer* für Kommentare und Diskussion.
- ♢ *Reiko Heckel* für den Pointer auf die CSPs und für seinen Ausdruck von [Kum92].
- ♢ *Boris Melamed* für magische Scripte.
- ♢ *Manuel Koch* für das große Lob.
- ♢ *Rosi Bardohl* für die Revanche.
- ♢ *meine Familie* für Verständnis und Rückendeckung, insbesondere
- ♢ *meine Mutter* für 182,5 $\times$ Vizevertretung und
- ♢ *meine Tante* für 181,5 $\times$ O-Saft frisch gepreßt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Definition 1.1 (*Graph*)
„Was ist ein *Graph?*" — „Gib einem Experten einen Bleistift und Papier, und bitte ihn, dir eine kurze Übersicht über sein Fachgebiet zu geben. Was du nach fünf Minuten in Händen hältst, *das* ist ein *Graph!*" △

Zugegeben, es gibt andere Definitionen für den Begriff des Graphen[1]. Diese Definition macht aber besonders deutlich, daß Graphen bei der Veranschaulichung komplexer Zusammenhänge auch im täglichen Leben eine Rolle spielen. Meistens allerdings, ohne daß die Graphen als solche überhaupt wahrgenommen würden. „Was ist ein Graph?" – Die Chancen stehen nicht schlecht, daß derselbe Experte, der uns eingangs – ohne es zu wollen – schon eine schöne Definition geliefert hat, auf diese Frage hin zum Lexikon greift. Wenn wir aber mit Graphen umgehen können, ohne zu wissen, was ein Graph eigentlich ist, dann scheinen diese Graphen ein sehr intuitives Beschreibungsmittel zu sein. Eins zu null für die Graphen.

Unser Experte hat nicht gezögert, uns die komplexen Zusammenhänge seines Spezialgebiets intuitiv durch Graphen zu erläutern. Aber warum unterhält er sich dann mit seinem Computer über nicht minder komplexe Probleme in Fortran? Nicht unbedingt, weil ihm das leichter fällt. Sondern, weil seine „Alltagsgraphen" nach Definition 1.1 doch ein wenig *zu* intuitiv sind, als daß eine Maschine sie interpretieren könnte. Außerdem hat der Experte gemerkt, daß er mit Graphen zwar Zustände, Situationen und Zusammenhänge auf dem Papier gut ordnen kann, sich mit der Beschreibung von Zustandsänderungen oder gar Berechnungen jedoch schwer tut. Es mangelt also an einer formalen Syntax und an Beschreibungsmethoden für Aspekte der Dynamik. Böse Schwächen in der Abwehr der Alltagsgraphen – Ausgleich, eins zu eins.

Doch die Graphen haben die Nachwuchsarbeit nicht vernachlässigt, die Auswechselbank ist gut besetzt: Mit formalen Graphbegriffen und Algorithmen auf Graphen beschäftigt sich die Wissenschaft schon seit mehreren hundert Jahren. Leonhard Euler benutzte anno 1736 Graphen bei der Lösung des „Königsberger Brückenproblems" und gilt damit als Pionier. An verschiedenen syntaktischen Formalisierungen des Graphbegriffs jedenfalls besteht schon lan-

[1] zum Beispiel die auf Seite 7 (Def. 2.1)

ge kein Mangel mehr. Erst vor ca. 25 Jahren begann allerdings die systematische Erforschung dynamischer Aspekte in Form von *Graphtransformation* [PR69, EPS73]. Die zugrundeliegende Idee ist dabei die regelbasierte Manipulation statischer Graphen, die sich am Vorbild der Veränderung von Symbolketten durch die Produktionsregeln einer Chomsky-Grammatik orientiert. Aufgrund dieser Analogie wird eine Menge von Graphregeln zusammen mit einem Arbeitsgraphen auch *Graphgrammatik* genannt. Im *algebraischen Ansatz zur Graphtransformation*, auf dem diese Arbeit beruht, werden die zugrundeliegenden Konzepte im Rahmen von Algebra und Kategorientheorie formalisiert [Ehr79, Löw93]. Ein großer Vorteil dieses Ansatzes ist seine gute theoretische Fundierung mit einer formalen Semantik, die auf der kategorientheoretischen Konstruktion des *Pushouts* beruht.

Die Motivation für die formale Beschreibung von Graphen und Graphtransformationen liegt dabei immer in dem Anliegen, die intuitive und anschauliche Aussagekraft von Graphen mit der maschinellen *Interpretierbarkeit* zu vereinen. Außerdem schafft eine Formalisierung die Grundlage für die Möglichkeit, bestimmte Eigenschaften der graphisch modellierten Systeme formal zu beweisen. So können algebraische Graphgrammatiken etwa auf Unabhängigkeits- und Nebenläufigkeitseigenschaften geprüft [Ehr79] oder in bezug auf Termination untersucht werden [Plu95], und es gibt Methoden zur Konsistenzanalyse und -sicherung [HW95]. Nach der Auswechslung der Alltagsgraphen gegen die algebraische Graphtransformation fällt folgerichtig der Treffer zur erneuten Führung: zwei zu eins für die Graphen.

Wenn wir uns noch einmal das Blatt Papier anschauen, auf dem uns der Experte sein Fachgebiet erklärt hat, dann sehen wir neben Strichen, Kästen und Punkten auch eine Menge von Namen und Zahlen, die meistens einem bestimmten graphischen Objekt zugeordnet sind; oft ist zum Beispiel ein Wort mit einem Kasten umrahmt. Wenn wir uns aber eine einfache formale Definition eines Graphen ansehen, dann ist dort nur von „Knoten" und „Kanten" die Rede. Erst das Konzept der *Attributierung* erweitert diesen Graphbegriff um die Möglichkeit, den graphischen Objekten andere Daten zuzuordnen, etwa Texte oder Zahlen, was im intuitiven Umgang mit Graphen so selbstverständlich ist. Die Transformation attributierter Graphen ermöglicht dann auch das Operieren auf den Attributen, so daß z.B. Berechnungen auf Zahlen durchgeführt werden können, was durch einfache Graphtransformation nicht sinnvoll modelliert werden kann. Selbstverständlich ist das Attributierungskonzept im Rahmen der algebraischen Graphtransformation formal definiert [LKW93].

Das Interesse des Experten ist geweckt. Er findet einen Artikel zur Einführung in die algebraische Graphtransformation, aber schon nach drei Seiten gibt er auf und fragt sich: „Was hat das noch mit dem intuitiven Umgang mit Graphen zu tun?" Doch er hat in dem Artikel auch ein Beispiel gefunden und das Prinzip der Transformation durch Regeln verstanden. Er hat auch gleich ein paar Regeln aufs Papier gemalt, die ein Problem betreffen, das ihn gerade beschäftigt. Dann hat er aber keine Programme auf seinem Rechner gefunden, die ihm die Eingabe und vor allem die Ausführung seines Regelsystems erlaubt hätten, und hat schließlich bald die Lust verloren. Wahrscheinlich hat er nicht sehr intensiv gesucht, denn sonst wäre ihm mit PROGRES [Zün95, Sch97] zumindest ein System aufgefallen, das bereits eine umfangreiche Entwicklungsumgebung für das Programmieren mit Graphtransformation bieten kann. PROGRES basiert jedoch nicht auf dem algebraischen Transformationsansatz und legt entsprechend weniger Wert auf die theoretische Fundierung. Unser Experte jedenfalls sitzt inzwischen wieder vor seinem Computer. Nachdem er sein Problem nun in Fortran formuliert hat, verbringt er viel Zeit mit dem Debugger, doch er sieht das positiv: Immerhin hat *der* eine graphische Benutzeroberfläche und ist intuitiv bedienbar! Mangelnde Werkzeugunterstützung für die algebraische Graphtransformation – ein schwerer Torwartfehler führt zum erneuten Ausgleich. Zwei zu zwei.

1.2 Ziele

Halbzeit. Gelegenheit, die Mannschaft neu einzustellen. Die taktische Anweisung für die Gegenoffensive lautet: *Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation*, so will es der Titel dieser Arbeit. Ihr Ziel soll es sein, einen Beitrag zu leisten, daß unser Experte in nicht allzu ferner Zukunft vielleicht nicht nur mit einem menschlichen Fragesteller, sondern auch mit seinem Computer auf der Basis von Graphen kommuniziert. Doch das Nahziel ist etwas bescheidener gesteckt; aus einer durch die Theorie verstärkten Abwehr heraus wollen wir vorsichtig in die Richtung der praktischen Anwendungen vorstoßen. Das zu entwickelnde System soll deshalb zunächst zur Veranschaulichung und Präsentation der theoretischen Ergebnisse aus dem Bereich der algebraischen Graphtransformation dienen. Darüber hinaus soll es die Grundlage bilden für die prototypische Erprobung praktischer Anwendungsmöglichkeiten.

Bereits 1991 wurde mit der Entwicklung des Algebraischen Graphgrammatik-Systems AGG begonnen [Bey91, LB93], das aber hauptsächlich die Funktionalität eines Grapheditors bot. Die Erfahrungen bei der Erweiterung des in der Programmiersprache Eiffel implementierten Systems durch [Win95] führten zu der Entscheidung, mit einer Neuimplementierung eines modularen Systemkerns in C++ zu beginnen [Rud96]. Die Implementierung einer Komponente zur Attributbehandlung erfolgte dann bereits in Java [Mel97].

Die vorliegende Arbeit beschäftigt sich nun mit der Portierung der Datenstrukturen aus [Rud96] nach Java, der Integration der besagten Attributkomponente und mit der Ergänzung um Funktionalitäten für Transformation und automatische Ansatzsuche. Dabei hat jeder Systemteil seine Besonderheiten:

Graphmodell: Die bereits aus dem ersten AGG-System bekannten ALR-Graphen erlauben durch ihr Abstraktionskonzept eine Meta-Modellierung, so daß eine Graphgrammatik im System als Graph dargestellt und entsprechend durch Graphregeln manipuliert werden kann. Außerdem enthalten die ALR-Graphen eine explizite Darstellung der Diagrammebene von kategoriellen Konstruktionen.

Attributierung: Neben dem Umgang mit Variablen und deren Belegung durch Ansätze ermöglicht die Attributkomponente aus [Mel97] die Attributierung von Graphobjekten mit herkömmlichen Programmen, deren Ausführung von der Graphtransformation gesteuert wird. Daß die Wahl der Programmiersprache dabei auf Java fiel und nicht auf Fortran, wird niemand ernsthaft als Nachteil ansehen.

Transformation: Die Transformation unter Verwendung der Colimit-Bibliothek von [Wol97] garantiert durch deren impliziten Korrektheitsnachweis die Übereinstimmung mit der Theorie und ermöglicht darüber hinaus die Berechnung von allgemeinen Colimiten im AGG-System.

Ansatzsuche: Die Formulierung der Ansatzsuche als Constraint Satisfaction Problem erlaubt eine Entkopplung des eigentlichen Algorithmus vom konkret verwendeten Graphmodell. Für die Lösungsalgorithmen kann nun auf die reichhaltigen Forschungsergebnisse aus dem Gebiet der Constraint Satisfaction zurückgegriffen werden, z.B. in [Pro93, Dec94].

Eine separate Editorkomponente mit graphischer Oberfläche für die Transformation befindet sich parallel dazu in der Entwicklung, ist aber nicht Gegenstand dieser Arbeit.

1.3 Gliederung

Die Arbeit beginnt in **Kapitel 2** mit einer Einführung in die Transformation attributierter Graphen nach dem algebraischen Ansatz. Außerdem ist hier ein Beispiel für einen Transformationsschritt im AGG-System angegeben. **Kapitel 3** beschreibt anschließend die beim Systementwurf verwendeten Methoden und Notationen und die Programmierumgebung.

Der Überblick über die Strukturierung des AGG-Systems in **Kapitel 4** dient gleichzeitig als Orientierung für die anschließenden drei **Kapitel 5–7**, die sich mit dem Entwurf der Systemteile für Graphdatenstrukturen, Transformation und Ansatzsuche beschäftigen und damit den Kern der Arbeit bilden. Jedes dieser Kapitel gibt zunächst eine Einführung in die theoretischen und konzeptionellen Grundlagen, bevor die eigentliche Entwurfsbeschreibung jeweils mit der Anforderungsdefinition und einem Überblick über die Anwendungsschnittstelle des Teilsystems beginnt. Da jedes Teilsystem sich mit unterschiedlichen Aspekten der Attributierung beschäftigt, erfolgt die Beschreibung der Integration der Attributkomponente entsprechend über die Kapitel verteilt.

Natürlich fließen in den Systementwurf die Implementierungserfahrungen mit ein, so daß die Entwurfsbeschreibung gleichzeitig als Beschreibung des implementierten Systems verstanden werden kann; entsprechend geben die Anforderungsdefinitionen immer auch einen Überblick über die tatsächlich implementierte Funktionalität. Gleiches gilt für **Kapitel 8**, das die Anwendungsschnittstelle für die Transformation einfacher Graphen beschreibt. In diesem Zusammenhang wird hier noch einmal kurz auf alle Funktionalitäten eingegangen, welche in den vorhergehenden Kapiteln auf der Basis von ALR-Graphen entwickelt wurden.

Die Anmerkungen zur Implementierung in **Kapitel 9** beziehen sich vor allem auf die wenigen Abweichungen von der Beschreibung des Entwurfs, und **Kapitel 10** schließlich faßt die Ergebnisse zusammen und erläutert die Perspektiven. Der **Anhang** enthält eine umfassende Dokumentation aller wichtigen Schnittstellen der Implementierung in Java.

Anpfiff zur zweiten Halbzeit.

Kapitel 2

Einführung in die attributierte Graphtransformation

Dieses Kapitel gibt eine Einführung in die wichtigsten Begriffe der attributierten algebraischen Graphtransformation nach dem sogenannten *Single-Pushout-Ansatz (SPO)* [Löw93, Roz97]. Wir geben eine informelle Beschreibung der Konstruktion eines Graphtransformationsschritts und betrachten einige Beispiele. Einige Bildschirmabzüge von einer Beispieltransformation im AGG-System vermitteln schließlich einen Eindruck von der Umsetzung der Begriffe im implementierten System.

2.1 Graphen und Morphismen

Als Grundlage definieren wir zunächst den Begriff eines einfachen Graphen, dessen Knoten und Kanten über einer gegebenen Labelmenge typisiert sind.

Definition 2.1 (*Einfacher Graph*)
Ein *einfacher Graph* G ist ein Tupel $G = (G_V, G_E, L, s, t, \ell)$ mit

- einer endlichen Knotenmenge G_V (für „vertex“) und einer endlichen Kantenmenge G_E (für „edge“) mit $G_V \cap G_E = \emptyset$,
- zwei totalen Abbildungen $s, t : G_E \to G_V$ (für „source“ und „target“),
- einer Labelmenge L und
- einer totalen Abbildung $\ell : G_V \cup G_E \to L$.

Als Sammelbegriff für Knoten und Kanten sprechen wir auch von *Graphobjekten* oder, noch kürzer, einfach von den *Objekten* eines Graphen. △

Beziehungen zwischen Graphen werden durch Morphismen ausgedrückt:

Definition 2.2 (*Graphmorphismus*)
Ein *Graphmorphismus* (kurz: Morphismus) $m : L \to G$ zwischen zwei einfachen Graphen

$L = (L_V, L_E, L_L, s_L, t_L, \ell_L)$ und $G = (G_V, G_E, L_G, s_G, t_G, \ell_G)$ ist ein Paar von partiellen Abbildungen $m = (m_V : L_V \to G_V,\ m_E : L_E \to G_E)$, für das die folgenden Eigenschaften gelten:

1. $\forall v \in dom(m_V) :\ \ell_L(v) = \ell_G(m_V(v))$
2. $\forall e \in dom(m_E) :\ \ell_L(e) = \ell_G(m_E(e))$
3. $\forall e \in dom(m_E) :$
   - $s_L(e) \in dom(m_V)\ \wedge\ m_V(s_L(e)) = s_G(m_E(e))$
   - $t_L(e) \in dom(m_V)\ \wedge\ m_V(t_L(e)) = t_G(m_E(e))$

Hier steht $dom(f),\ f : A \to B$ für den Definitionsbereich einer partiellen Abbildung f, also für diejenige Teilmenge von A, für die die Abbildung definiert ist.

m heißt *total*, wenn $dom(m_V) = L_V$ und $dom(m_E) = L_E$. L wird als *Original-* oder *Quellgraph* von m bezeichnet, G entsprechend als *Bild-* oder *Zielgraph*. Für das *Bild eines Graphobjekts o unter dem Morphismus* schreiben wir unter Vernachlässigung der Indizes von Knoten- und Kantenabbildung einfach $m(o)$. △

Die Bedingungen 1 bis 3 heißen *Verträglichkeitsbedingungen*, man spricht auch von *Operationsverträglichkeit* eines Morphismus bezüglich der Operationen s, t und ℓ. Dabei wird oft noch weiter unterschieden zwischen *Typverträglichkeit* (Bed. 1 und 2) und *Strukturverträglichkeit* (Bed. 3). Aufgrund der Forderung von Bed. 3, daß ein Morphismus zusammen mit einer Kante immer auch deren Source- und Targetknoten abbilden muß, zeichnet der Definitionsbereich eines Morphismus m immer einen Teilgraphen $dom(m)$ von L aus; entsprechendes gilt für den Wertebereich $codom(m)$.

2.2 Attributierung

Für die Definition der Attributierung benötigen wir einige grundlegende Begriffe aus der Theorie der algebraischen Spezifikation. Da wir uns aber in die Theorie der Attributierung nicht näher vertiefen wollen, soll hier auch auf eine formale Definition der Begriffe „Signatur“ und „Algebra zur Signatur“ verzichtet werden; siehe dafür [EM85]. Zur Klärung der Terminologie sei gesagt, daß eine Signatur $Sig\ = (S, OP)$ den syntaktischen Rahmen für eine Klasse $Alg(Sig)$ von Algebren zur Signatur bildet. Sie besteht aus einer Menge S von Sortensymbolen und einer Menge OP von Operationssymbolen. Jede Algebra aus $Alg(Sig)$ „implementiert“ die Sortensymbole der Signatur durch Wertemengen, sog. *Trägermengen*, und die Operationssymbole durch entsprechende Operationen. Ein Sig-Homomorphismus zwischen zwei Sig-Algebren ist dann eine Familie von operationsverträglichen Abbildungen zwischen den Trägermengen der beiden Algebren.

Kommen wir nun zur erweiterten Definition eines Graphen, dessen Knoten und Kanten über einer gegebenen Algebra attributiert werden können:

Definition 2.3 (*Attributierter Graph*)
Sei A eine Algebra zu einer Signatur $AS = (S, OP)$ und $G = (G_V, G_E, L, s, t, \ell)$ ein einfacher Graph gemäß Def. 2.1. Ein Tupel $G_{AS} = (G_V, G_E, L, s, t, \ell, A, at, av)$ mit

- einer totalen Abbildung $at : L \to S$ (für „attribute type“) und
- einer totalen Abbildung $av : G_V \cup G_E \to AV$ (für „attribute value“), wobei $AV = \biguplus_{s \in S} A_s$ die disjunkte Vereinigung aller Trägermengen von A ist,

heißt *attributierter Graph*, wenn für $a\ell : AV \to S$ mit $\forall v \in A_s : a\ell(v) = s$ das folgende Diagramm kommutiert:

$$\begin{tikzcd}
L \arrow[r, "at"] & S \\
G_V \cup G_E \arrow[u, "\ell"] \arrow[r, "av"'] \arrow[ur, phantom, "(=)"] & AV \arrow[u, "a\ell"']
\end{tikzcd}$$

△

Diese Definition stellt eine Vereinfachung des Attributkonzepts aus [LKW93] dar, wie sie ähnlich auch in [Roz97] zu finden ist. Diese vereinfachte Sicht kommt der intuitiven Anschauung näher und entspricht auch eher den von der Attributkomponente [Mel97] implementierten Schnittstellen. Über die Attributierungsfunktion av wird jedem Graphobjekt genau ein Attributwert zugeordnet. Wenn wir mehrere Attribute an einem Objekt benötigen, was in der Praxis meistens der Fall ist, dann müssen wir diese bereits innerhalb der Algebra zu einem Tupel zusammenfassen. Dies entspricht genau der Sichtweise der Implementierung. Die Sorten der Attributalgebra entsprechen verschiedenen Typen von Attributen; ein Attributtyp könnte etwa ein Paar aus einem String und einer Zahl sein, ein anderer vielleicht ein boole'scher Wert. Alle Werte der Attributalgebra, gleich welchen Typs, werden zu einer Wertemenge AV zusammengefaßt. Die Abbildung $a\ell$ ordnet dann jedem Wert seinen Attributtyp zu, ganz analog zur Typisierungsoperation ℓ für die Graphobjekte. at ordnet schließlich jedem Graphobjekttyp genau einen Attributtyp zu, und die Forderung nach Kommutativität des Attributierungsdiagramms sorgt dafür, daß alle Graphobjekte mit demselben Graphobjekttyp auch nur mit Attributwerten zu immer demselben Attributtyp attributiert werden können.

Jetzt müssen wir noch erklären, was die Attributierung für die Beziehungen zwischen Graphen bedeutet:

Definition 2.4 (*Morphismus zwischen attributierten Graphen*) Seien $L_{AS} = (L_V, L_E, L_L, s_L, t_L, \ell_L, A_L, at_L, av_L)$ und $G_{AS} = (G_V, G_E, L_G, s_G, t_G, \ell_G, A_G, at_G, av_G)$ attributierte Graphen nach Def. 2.3. Ein Paar $m = (m_G, m_D)$ ist ein *Morphismus* $m : L_{AS} \to G_{AS}$ zwischen den Graphen L_{AS} und G_{AS}, wenn

1. $m_G : L \to G$ mit $L = (L_V, L_E, L_L, s_L, t_L, \ell_L)$ und $G = (G_V, G_E, L_G, s_G, t_G, \ell_G)$ Morphismus ist gemäß Def. 2.2,
2. $m_D : A_L \to A_G$ ein totaler AS-Homomorphismus ist und
3. $\forall x \in dom(m_G)$ das folgende Diagramm kommutiert:

$$\begin{tikzcd}
L_V \cup L_E \arrow[r, "m_G"] \arrow[d, "av_L"'] \arrow[dr, phantom, "(=)"] & G_V \cup G_E \arrow[d, "av_G"] \\
AV_L \arrow[r, "m_D"'] & AV_G
\end{tikzcd}$$

△

Die Forderung nach einem totalen Homomorphismus zwischen den Attributalgebren ist eine sehr strenge Voraussetzung; zusammen mit der geforderten Verträglichkeit des Morphismus mit der Attributierungsoperation bedeutet dies praktisch, daß vom Morphismus abgebildete Objekte ihre Attributwerte beibehalten müssen.

2.3 Regeln, Ansätze und Transformation

In diesem Abschnitt erklären wir die Grundbegriffe für die regelbasierte Transformation attributierter Graphen nach dem *Single-Pushout*-Verfahren [Löw93, LKW93]. Die Definitionen für die Transformation ohne Attribute können als Spezialfall angesehen werden, indem über einelementige Algebren zu einer Signatur mit nur einer Sorte attributiert wird. Zunächst definieren wir nun, was wir unter einer Graphregel verstehen wollen:

Definition 2.5 (*Graphregel*) Eine *Graphregel* ist ein Graphmorphismus $r : L \to R$. Die beiden Graphen L und R werden auch als *linke* und *rechte Seite* der Regel bezeichnet. △

Bemerkung 2.6 (*Rechnen auf Attributen*) Um die regelbasierte Änderung von Attributen auf intuitive Weise zu ermöglichen, gehen wir im folgenden davon aus, daß ein Regelmorphismus die Bedingung 3 aus Definition 2.4 *nicht* erfüllen muß. In der Theorie wird dieses Problem durch indirekte Attributierung über sog. *Attributträger* gelöst, vgl. Abschnitt 6.1.3. △

In der linken Seite einer Regel beschreiben wir eine Konstellation von Objekten, die wir verändern wollen; der Morphismus von der linken in die rechte Seite beschreibt die konkret vorzunehmenden Änderungen: Objekte, die über den Regelmorphismus abgebildet werden, werden erhalten, Objekte aus L, die nicht in $dom(r)$ liegen, werden gelöscht. Wenn es zwei Objekte x, y in $dom(r)$ gibt mit $x \neq y$, so daß $m(x) = m(y)$, dann sagt man, der Regelmorphismus *identifiziert* (oder *verklebt*) die beiden Objekte x und y; ein solcher Morphismus heißt *nicht-injektiv*. Zusätzlich kann es in der rechten Regelseite Objekte geben, die vom Regelmorphismus nicht erreicht werden; das bedeutet, daß diese Objekte von der Regel neu erzeugt werden sollen. Insgesamt beschreibt also ein Regelmorphismus, wie ein Graph vor bzw. nach der Anwendung der Regel aussehen soll.

Um eine Regel auf einen Arbeitsgraphen anwenden zu können, benötigen wir noch den Begriff des Ansatzes:

Definition 2.7 (*Ansatz einer Graphregel*) Ein *Ansatz* einer Graphregel $r : L \to R$ in einen Graphen G ist ein *totaler* Graphmorphismus $m : L \to G$. △

Ein Ansatz beschreibt also, wie die linke Seite einer Regel in einem Arbeitsgraphen wiedergefunden wird. Die Totalität eines Ansatzes stellt sicher, daß alle Voraussetzungen, die die linke Seite der Regel formuliert, im Arbeitsgraphen tatsächlich gegeben sind. Wichtig ist die Feststellung, daß der Arbeitsgraph wesentlich größer sein kann als die linke Regelseite; der Teil des Arbeitsgraphen, der nicht im Wertebereich des Ansatzmorphismus liegt, wird als

Kontext bezeichnet. Schon aus diesem Grund ist klar, daß es im allgemeinen nicht nur einen Ansatz für eine Regel in einen gegebenen Arbeitsgraphen gibt.

Jetzt fehlt uns nur noch das Ergebnis einer Regelanwendung auf einen Arbeitsgraphen an einem gegebenen Ansatz:

Definition 2.8 (*Transformationsschritt*) Das Ergebnis der Anwendung einer Graphregel $r : L \to R$ auf einen Graphen G an einem Ansatz $m : L \to G$ ist ein Graph H mit zwei Morphismen $r^* : G \to H$ und $m^* : R \to H$. Dieses Ergebnis wird eindeutig charakterisiert durch ein *Pushout*[1] in der Kategorie der attributierten Graphen mit partiellen Morphismen[2]:

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

Eine solche Regelanwendung wird als *Transformationsschritt* oder auch als *direkte Ableitung* bezeichnet. △

Anstatt näher auf den kategorientheoretischen Hintergrund einzugehen, geben wir eine kurze informelle Beschreibung der entsprechenden Konstruktion für den nicht attributierten Fall und betrachten anschließend ein Beispiel. [Löw93] beschreibt die Konstruktion formal und beweist die Pushout-Eigenschaften. Die Attribute können dann durch ein Pushout auf den Attributalgebren separat berechnet und anschließend eindeutig den Graphobjekten zugeordnet werden, vgl. [LKW93, CL95].

Konstruktion 2.9 (*Ergebnis eines Transformationsschritts*) Gegeben eine Regel $r : L \to R$ und ein Ansatz $m : L \to G$. Wir konstruieren den Ergebnisgraphen H der Regelanwendung in zwei Schritten:

1. **Verkleben und Hinzufügen.** Wir betrachten zunächst nur die Objekte aus $dom(r)$. Jedes dieser Objekte hat über den Ansatz m eine Entsprechung im Arbeitsgraphen G. Wir fügen nun die von der Regel zu erzeugenden Objekte zwischen den entsprechenden Objekten im Arbeitsgraphen ein. Außerdem verkleben wir diejenigen Objekte aus G, die der Regelmorphismus identifiziert. Wir erhalten als Zwischenergebnis einen Graphen Z, in dem G als Teilgraph enthalten ist.

2. **Löschen.** Wir löschen die Entsprechungen aller derjenigen Objekte aus L in Z, die nicht in $dom(r)$ liegen. Dabei kann es vorkommen, daß *hängende Kanten* zurückbleiben, das sind Kanten, deren Quelle oder Ziel gelöscht wurde. Solche Kanten werden ebenfalls gelöscht, denn sonst wäre das Ergebnis kein Graph mehr.

△

[1] Abschnitt 6.1.1 gibt Definitionen für die Begriffe *Kategorie* und *Pushout*.

[2] [LKW93] zeigen dies für die Attributierung über totalen Algebren, in [CL95] wird die Konstruktion auf partielle Algebren ausgeweitet

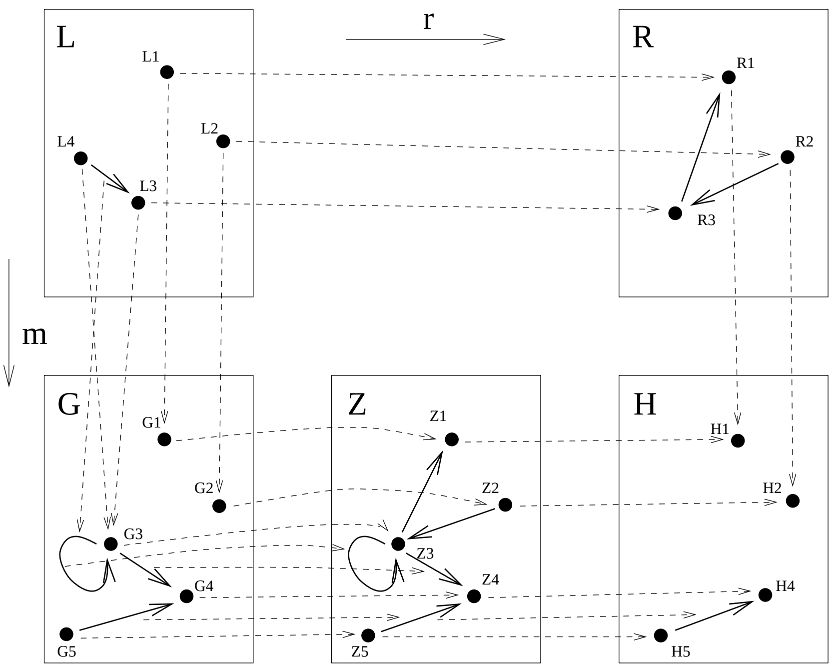


Abbildung 2.1: Ein Transformationsschritt an einem Ansatz mit Konflikten.

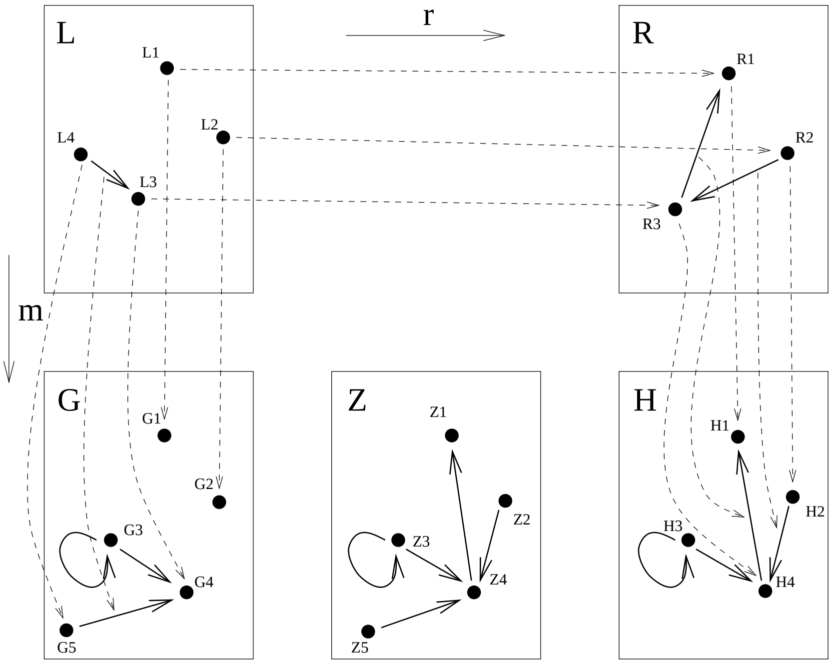


Abbildung 2.2: Ein Transformationsschritt an einem vorsichtigen Ansatz.

Beispiel 2.10 (*Transformationsschritt mit Konflikten*) In Abbildung 2.1 betrachten wir der Einfachheit halber ein Beispiel, in dem alle Graphobjekte denselben Typ haben; die Typinformation kann deshalb als redundant vernachlässigt werden. Die Regel $r : L \to R$ erhält die drei Knoten $L1, L2, L3$ und fügt zwischen ihnen zwei neue Kanten ein. Der Knoten $L4$ und die von $L4$ ausgehende Kante dagegen sollen gelöscht werden. Die Regel ist *injektiv*, d.h. es werde keine Objekte verklebt. Der Ansatz $m : L \to G$ dagegen ist *nicht-injektiv*, $L4$ und $L3$ werden gemeinsam auf $G3$ abgebildet. Die Kante zwischen den beiden Objekten in L wird entsprechend zu einer Schlaufe in G. Der Graph Z zeigt das Zwischenergebnis gemäß Konstruktion 2.9 nach dem ersten Schritt: Wir finden G als Teilgraph in Z wieder, und die gemäß der Regel einzufügenden Objekte wurden ergänzt.

Im zweiten Schritt der Konstruktion löschen wir diejenigen Objekte aus Z, deren Entsprechungen aus L nicht in $dom(r)$ liegen; das betrifft den Knoten $L4$ aus L und die von ihm ausgehende Kante. Diese Objekte finden wir als $Z3$ mit der entsprechenden Schlaufe in Z wieder. Wir löschen also die Schlaufe, und das anschließende Löschen von $Z3$ hinterläßt gleich drei hängende Kanten, die wir ebenfalls entfernen müssen. das Ergebnis ist der Graph H.

Der Effekt, den die Regel beschreibt, ist in diesem Ergebnis kaum noch wiederzuerkennen: Die beiden Kanten, die eingefügt werden sollten, tauchen im Ergebnis nirgends auf, und der Knoten $G3$, der laut Regelvorschrift erhalten werden sollte, wurde sogar gelöscht. Der Grund dafür liegt darin, daß durch die Identifikation von $L3$ und $L4$ über den Ansatz ein Widerspruch bezüglich $G3$ entsteht: Als Bild von $L3$ schreibt die Regel für den Knoten $G3$ vor, daß er erhalten bleiben soll, gleichzeitig muß er aber als Bild von $L4$ auch gelöscht werden. Wie wir gerade gesehen haben, löst die Konstruktion derartige *Konflikte* einfach durch Löschen. Dieses Verhalten ist typisch für die SPO-Transformation. △

Um pathologische Fälle wie den aus Beispiel 2.10 zu umgehen, kann man zusätzliche Bedingungen an den Ansatzmorphismus stellen:

Definition 2.11 (*Ansatzeinschränkungen*) Gegeben ein Ansatz $m : L \to G$ für eine Regel $r : L \to R$. Dann heißt m

injektiv, wenn $\forall x, y \in^3 L : \; m(x) = m(y) \Rightarrow x = y$.

löschinjektiv, kurz: **l-injektiv**, wenn $m|_{L\setminus dom(r)}$ injektiv ist.

konfliktfrei, wenn $m(x) = m(y)$ impliziert, daß entweder $x, y \in dom(r)$ oder $x, y \notin dom(r)$ gilt. Darüber hinaus erfüllt m die sogenannte **identification condition** genau dann, wenn er sowohl konfliktfrei als auch l-injektiv ist.

löschvollständig, kurz: **l-vollständig**, wenn für jede Kante $e \in G$ mit $s_G(e) \in codom(m|_{L\setminus dom(r)})$ oder $t_G(e) \in codom(m|_{L\setminus dom(r)})$ auch $e \in codom(m|_{L\setminus dom(r)})$ gilt. Diese Bedingung bewirkt, daß keine hängenden Kanten entstehen können, und ist auch unter dem Namen **dangling condition** bekannt.

vorsichtig, wenn m sowohl die *dangling condition* als auch die *identification condition* erfüllt. Diese Bedingung heißt auch **gluing condition**.

[3] Wir definieren $x \in L \Leftrightarrow x \in L_V \vee x \in L_E$.

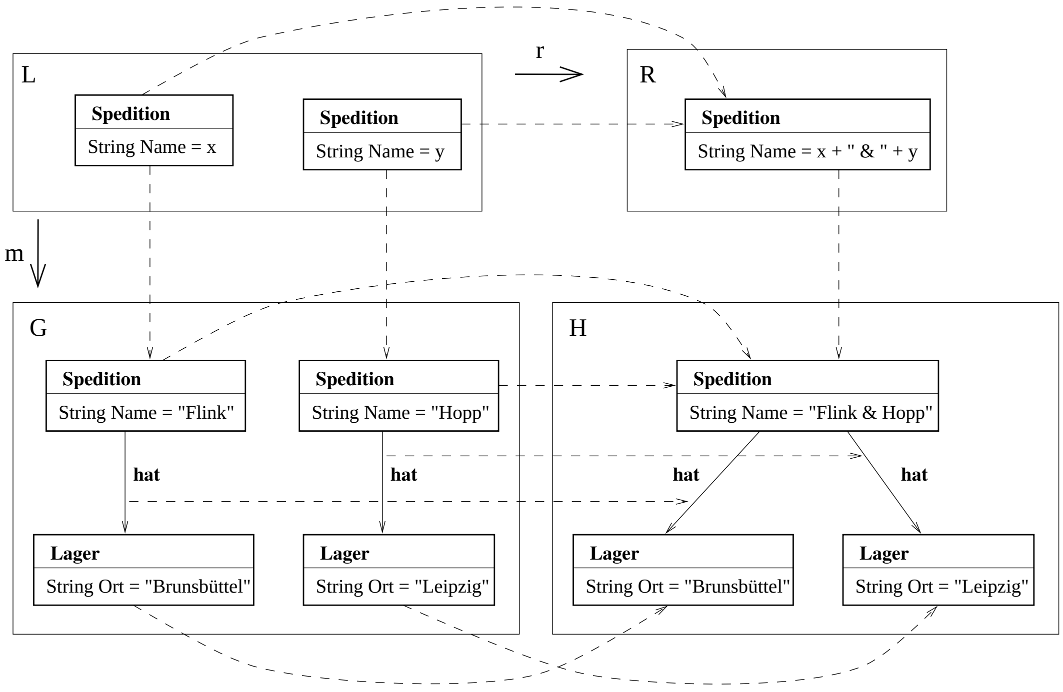


Abbildung 2.3: Transformation mit attributierten Graphen.

△

Beispiel 2.12 (*Vorsichtiger Transformationsschritt*) Der Ansatz aus Beispiel 2.10 in Abb. 2.1 erfüllte *keine* der oben definierten Ansatzbedingungen. Abbildung 2.2 zeigt einen Transformationsschritt mit derselben Regel, aber an einem Ansatz, der *alle* Bedingungen aus Def. 2.11 erfüllt. Die Morphismusbeziehungen zwischen den Graphen G, Z und H sind hier aus Gründen der Übersichtlichkeit nicht mehr durch gestrichelte Pfeile, sondern nur noch durch die Numerierung der Objekte und ihr Layout ausgedrückt. Im Unterschied zum vorigen Beispiel kommt der Effekt der Regel hier voll zur Geltung, was sich auch darin ausdrückt, daß der Morphismus von R in den Ergebnisgraphen H total ist. Außerdem finden keine impliziten Löschungen im Kontext von G mehr statt. △

Abschließend sei an einem einfachen Beispiel gezeigt, wie im Rahmen eines Graphtransformationsschrittes mit Attributen gerechnet werden kann.

Beispiel 2.13 (*Transformation mit attributierten Graphen*) In Abbildung 2.3 sind die Knoten der Graphen als Kästen gezeichnet, die fettgedruckten Namen entsprechen den Typen der Graphobjekte. Jeder Knoten ist mit einem Attribut vom Typ „String" versehen, dessen Wert nach dem Gleichheitszeichen angegeben ist. Die Regel soll ausdrücken, daß zwei Speditionsunternehmen fusionieren. Dazu wird eine nicht-injektive Regel verwendet, die die beiden Speditionsknoten verklebt. Die Namen der Unternehmen werden in den Regelgraphen durch Variablen angegeben, die später von einem Ansatz belegt werden. Der Name des neuen Speditionsunternehmens ergibt sich als Verknüpfung der Namen der fusionierten Unternehmen.

Der angegebene Arbeitsgraph G enthält zwei Speditionsunternehmen mit den Namen

„Flink“ und „Hopp“, die jeweils ein Lager in verschiedenen Städten besitzen. Nach der Regelanwendung haben die beiden Unternehmen fusioniert zu einer neuen Spedition „Flink & Hopp“, und der Besitz beider Ausgangsunternehmen ist in den Besitz der neuen Gesellschaft übergegangen.

Für weitere Einzelheiten über den Umgang mit Attributen in Regeln und Ansätzen siehe [Mel97] und Abschnitt 6.1.3. △

2.4 Beispieltransformation im AGG

Die Abbildungen 2.4 bis Abb. 2.6 zeigen eine Beispieltransformation mit der Entwicklerversion des Grapheditors vom AGG-System. Die graphische Darstellung von Knoten und Kanten entspricht weitestgehend der von Abb. 2.3 aus dem vorigen Abschnitt; allerdings werden die Attributtypen nicht angezeigt, um die Anzeige nicht zu überfrachten. Die Richtung der Kanten wird im Editor durch einen Farbwechsel angezeigt, was in den Bildschirmabzügen schlecht zu erkennen ist. Die Morphismen werden durch eine entsprechende Numerierung der Graphobjekte repräsentiert; auf diese Weise ist ein Ansatz zwischen der linken Regelseite und dem Arbeitsgraphen gegeben. Abb. 2.6 zeigt das Ergebnis des Transformationsschrittes an diesem Ansatz. Dieses Ergebnis wurde „in place“ berechnet, d.h., der Graph aus Abb. 2.5 wurde direkt manipuliert, anstatt wie in den Beispielen aus dem vorangegangenen Abschnitt einen neuen Ergebnisgraphen zu erzeugen und ihn durch einen Morphismus mit der Ausgangssituation in Beziehung zu setzen. Die Nummern im Ergebnisgraphen repräsentieren die „Reste“ des nun partiell gewordenen Ansatzes (vgl. Abschnitt 6.2.6 zur Konsistenzsicherung von Morphismen).

Gut zu erkennen ist noch einmal der Umgang mit Variablen in der Graphregel. Außerdem kann man sehen, daß Objekte beliebig viele Attribute haben können und daß in einer Regel nur solche Attribute angegeben werden müssen, die im jeweiligen Zusammenhang relevant sind. Für nicht angegebene Attributwerte wird implizit eine Variable angenommen, die nirgends referenziert wird.

Im Bild nicht dargestellt sind u.a. die folgenden Möglichkeiten:

- Nicht-injektive Regeln und Morphismen.
- Attributierung von Objekten der rechten Regelseite mit Aufrufen beliebiger Java-Methoden, die bei der Anwendung der Regel ausgeführt werden.
- Attributierung von Kanten.

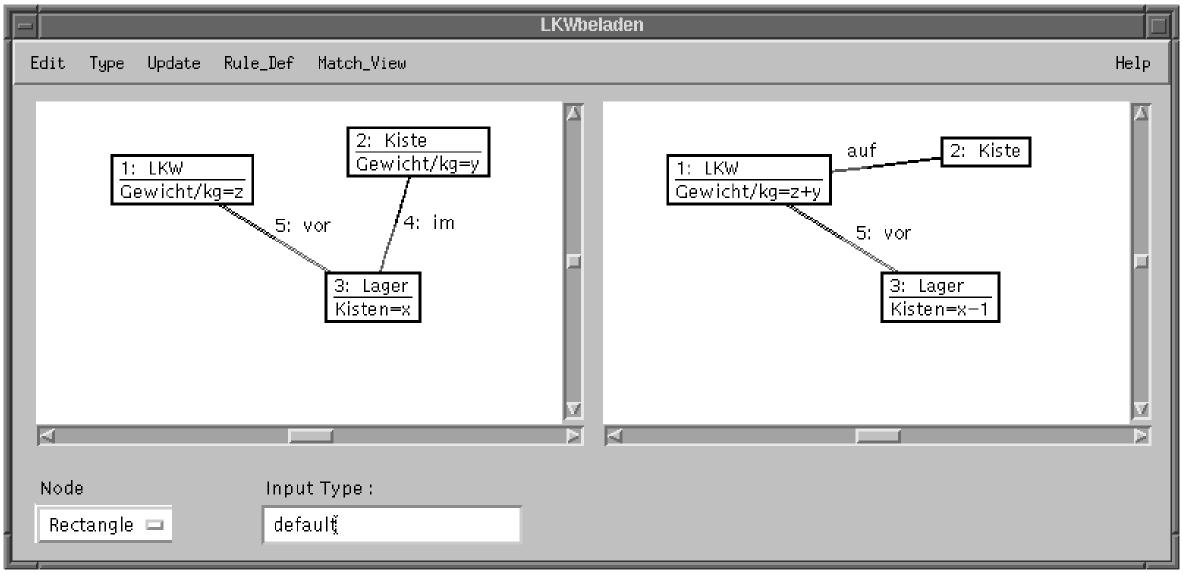


Abbildung 2.4: Eine Regel im Agg-System.

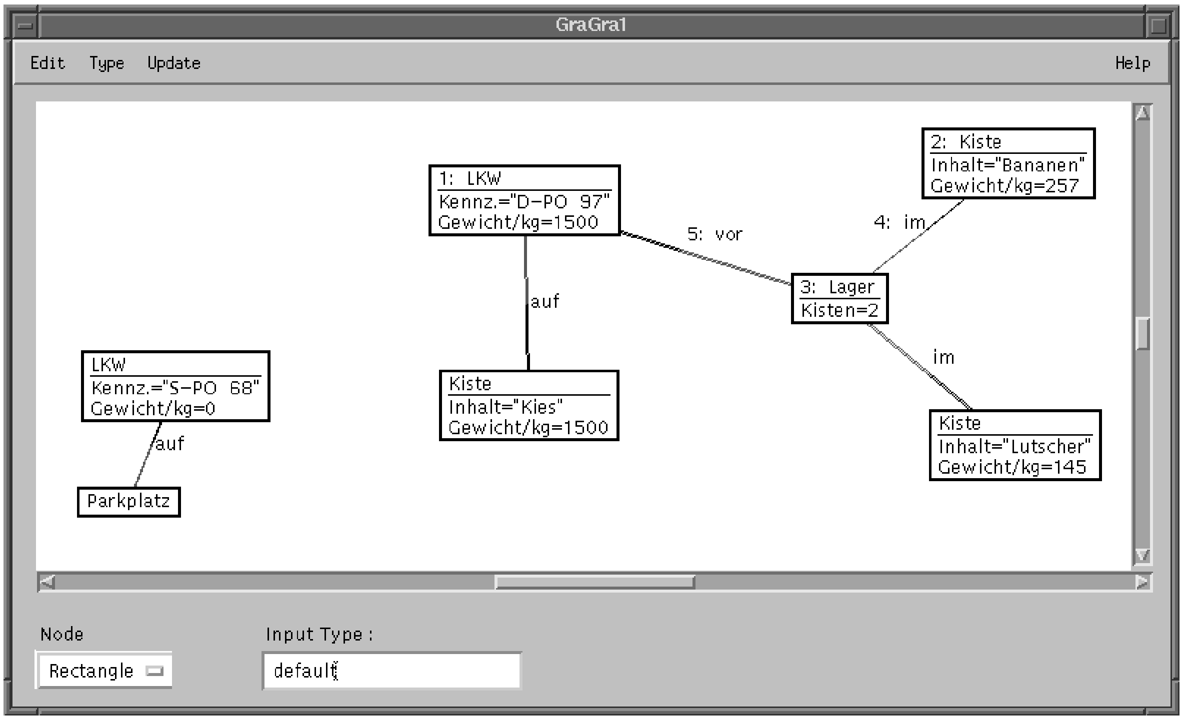


Abbildung 2.5: Der Arbeitsgraph vor der Regelanwendung.

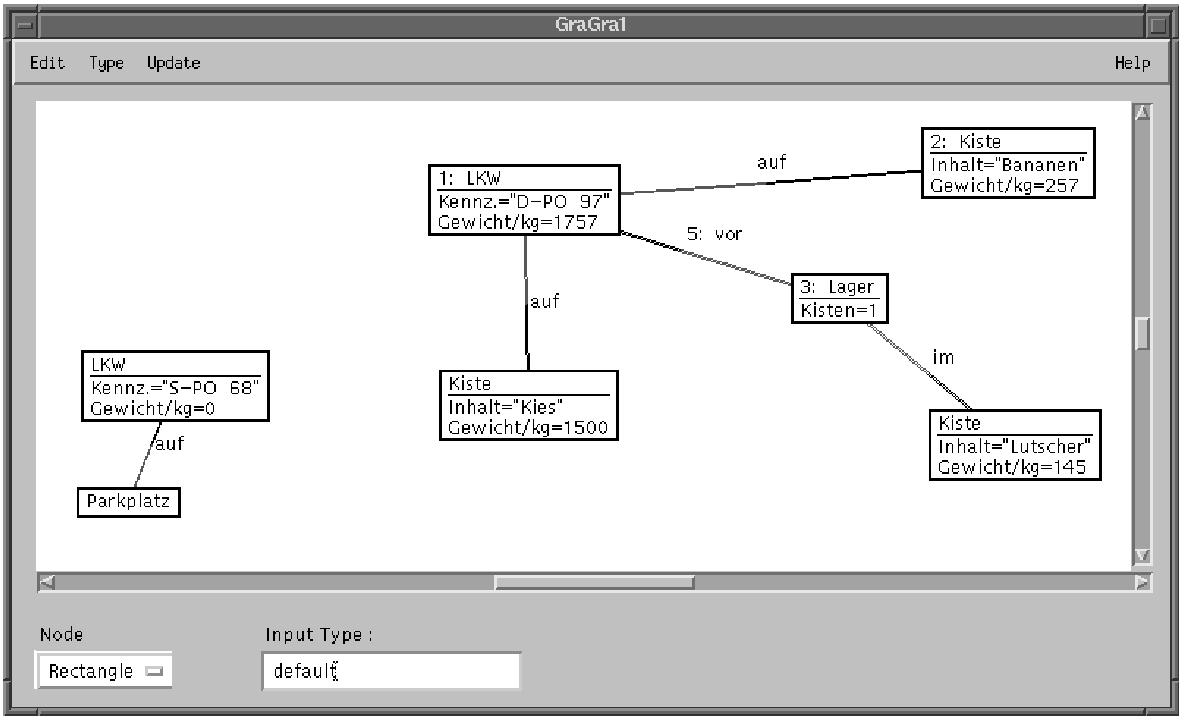


Abbildung 2.6: Der Ergebnisgraph nach der Regelanwendung.

Kapitel 3

Methoden und Werkzeuge

In diesem Kapitel beschreiben wir die für den objektorientierten Entwurf des AGG-Systems eingesetzten Methoden und Werkzeuge. Zunächst fassen wir kurz die in dieser Arbeit für die Entwurfsbeschreibung verwendete Notation der *Unified Modeling Language* zusammen. Anschließend geben wir einen Überblick über die bei der Entwicklung eingesetzten *Design Patterns* und kommentieren schließlich die Wahl der Programmiersprache *Java* und die damit gemachten Erfahrungen.

3.1 Unified Modeling Language (UML)

Für die Beschreibung des objektorientierten Entwurfs des AGG-Systems verwenden wir die Notation der *Unified Modeling Language* (UML). UML ist eine objektorientierte Modellierungssprache in der Tradition von OMT [RBP+91] und Booch [Boo91], und so fließen viele Aspekte dieser Vorgänger in UML mit ein. UML will eine Vereinheitlichung der für verschiedene Aspekte des Entwurfs verwendeten Notationen erreichen. Dazu wird zunächst eine Grundmenge an zentralen Modellierungselementen und deren Notation definiert, über der dann die Notationen für statische und dynamische Aspekte, für Klassen- und Instanzebene usw. quasi generisch aufgebaut werden. Dabei ist es ein zentrales Anliegen von UML, daß die entwickelten Notationen von Entwicklungsumgebungen für objektorientierten Software-Entwurf auf dem Rechner unterstützt werden können. In diesem Zusammenhang werden viele Optionen für die konkrete Darstellung offengelassen, während versucht wird, die darunterliegende Semantik möglichst weitgehend festzulegen.

Von den vielen verschiedenen Diagrammarten, die UML für die Modellierung verschiedener Aspekte eines Systems zur Verfügung stellt, werden wir nur zwei verwenden: Die sogenannten *Static Structure Diagrams*, die wir auch als *Klassendiagramme* bezeichnen, und zur Modellierung von dynamischen Aspekten die *Sequence Diagrams*, die auch als *Aufrufgraphen* bekannt sind. Wir geben nun einen kurzen Überblick über die Notation dieser Diagrammarten, für weitere Einzelheiten sei auf [Rat97] verwiesen.

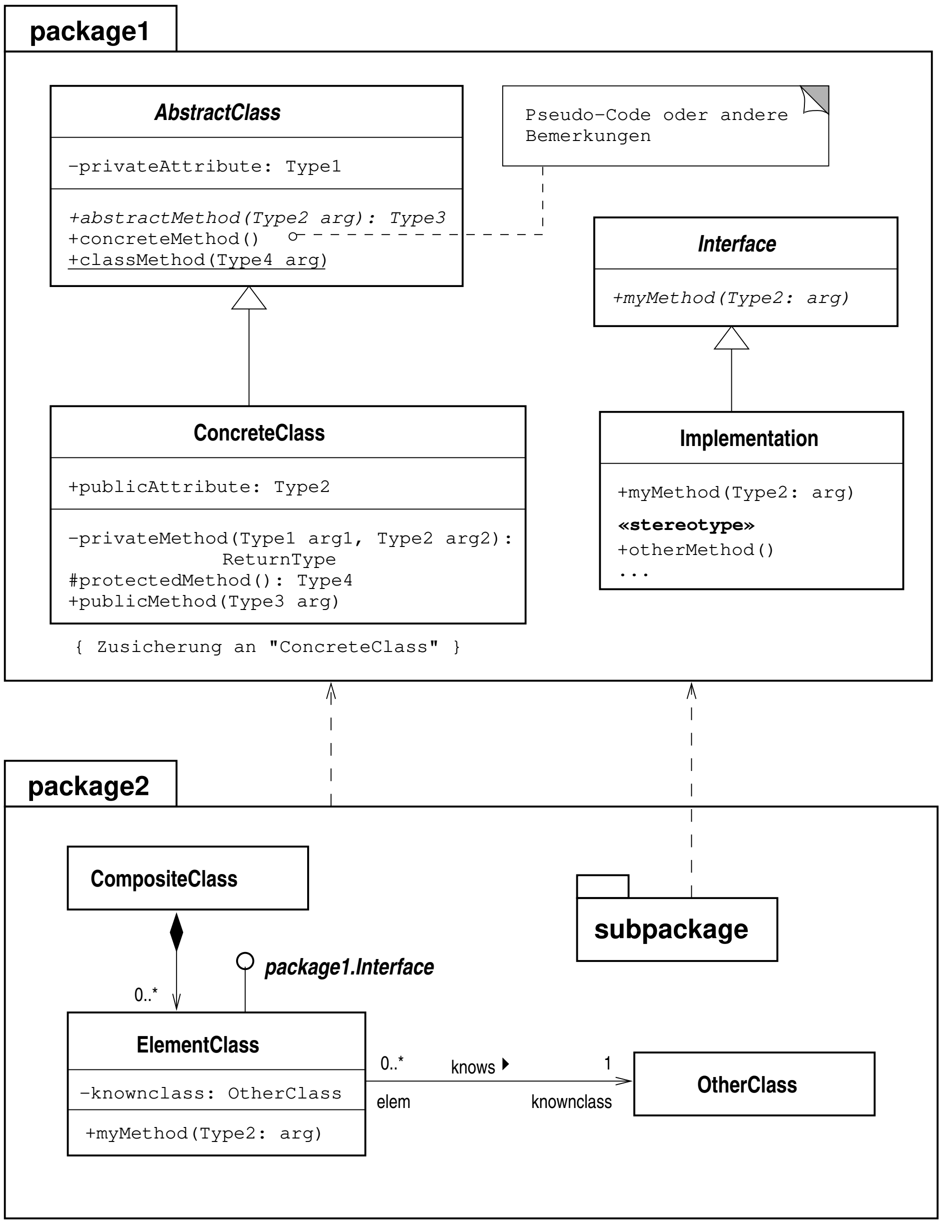


Abbildung 3.1: Notation von Klassendiagrammen.

Klassendiagramme

Im Mittelpunkt der Klassendiagramme (Abbildung 3.1) stehen die Begriffe *Klasse, Interface* und *Package*. **Klassen** werden wie schon in OMT durch Kästen dargestellt, in deren Kopf fettgedruckt der Name der Klasse angegeben ist. Die Namen von abstrakten Klassen und von Interfaces werden kursiv gesetzt. Außerdem kann eine Klasse Angaben über ihre Methoden und Variablen enthalten. In beiden Fällen wird zwischen öffentlichen, geschützten und privaten Einträgen unterschieden, jeweils gekennzeichnet durch ein vorangestelltes `+`, `#` oder `-`. Die Semantik des jeweiligen Schutzmodus wird gegeben durch die verwendete Programmiersprache, in diesem Fall also von Java. Die Typisierung der Variablen und der Rückgabewerte von Methoden wird nach einem Doppelpunkt angegeben, bei der Typisierung von Methodenparametern folgen wir stattdessen den Konventionen von Java. Abstrakte Methoden werden kursiv gestellt, Klassenvariablen und -Methoden werden unterstrichen. Grundsätzlich ist es erlaubt, beliebige Bestandteile einer Klasse zu „verstecken", es werden also immer nur diejenigen Variablen und Methoden einer Klasse angegeben, die im jeweiligen Zusammenhang relevant sind. Oft wird durch ein „..." explizit darauf hingewiesen, daß die Aufstellung einer Liste nicht vollständig ist.

Zusicherungen, die sich auf die Instanzen einer Klasse in einem bestimmten Zusammenhang beziehen, können in geschweiften Klammern geschrieben und der Klasse zugeordnet werden. Über sogenannte *stereotypes* kann die UML Notation in gewisser Weise dynamisch erweitert werden; eine in „«»" eingeschlossene Bezeichnung wird einem Notationselement zugeordnet und deutet seine spezielle Bedeutung an. Wir nutzen diese Notation beispielsweise, um die Konstruktoren in der Methodenliste einer Klasse auszuzeichnen.

Vererbungsbeziehungen zwischen Klassen werden durch einen Pfeil mit einer Dreiecksspitze notiert; die Spezialisierung erfolgt entgegen der Pfeilrichtung. Mit der gleichen Notation wird die Implementierung eines Interfaces dargestellt. Eine Notationsvariante erlaubt die Repräsentation eines Interfaces durch einen kleinen Kreis, der mit der implementierenden Klasse verbunden wird.

Außer der Vererbung, die eine Beziehung auf Klassenebene beschreibt, stehen Notationen zur Verfügung, um **Assoziationen** zwischen den Instanzen von Klassen anzugeben. Die allgemeine Form ist hier ein einfacher Pfeil zwischen zwei Klassen, an dem auf beiden Seiten sog. Rollenbezeichnungen und Kardinalitäten angegeben werden können. Betrachten wir das Beispiel in Abb. 3.1: Das Diagramm gibt an, daß jede Instanz der Klasse `package2.OtherClass` in Beziehung steht zu `0..*`, also beliebig vielen Instanzen der Klasse `ElementClass`. Diesen Instanzen wird die Rollenbezeichnung „elem" zugewiesen. Umgekehrt ist jede Instanz von `ElementClass` mit genau einer Instanz von `OtherClass` assoziiert. Die Richtung des Pfeiles, der die Assoziation symbolisiert, gibt dabei über den abstrakten Begriff der Assoziation hinaus an, daß `ElementClass` durch eine entsprechende Variable konkreten Zugriff auf `OtherClass` hat, was umgekehrt nicht der Fall ist. Mit einem Namen in der Mitte des Assoziationspfeiles kann zusätzlich angegeben werden, wie eine Assoziation zu lesen ist, in diesem Fall: „`ElementClass` knows `OtherClass`". Meistens wird dieser Name weggelassen, weil sich die Art der Beziehung schon aus den Rollenbezeichnungen ergibt. Oft geben wir mehrere Rollenbezeichner, durch Kommata getrennt, für eine einzige Assoziation an. Dies ist eine vereinfachende Notation für die entsprechende Anzahl von einzelnen Assoziationen. Die angegebene Kardinalität entspricht der jeder einzelnen Assoziation, gibt also *nicht* die Summe der Einzelkardinalitäten an.

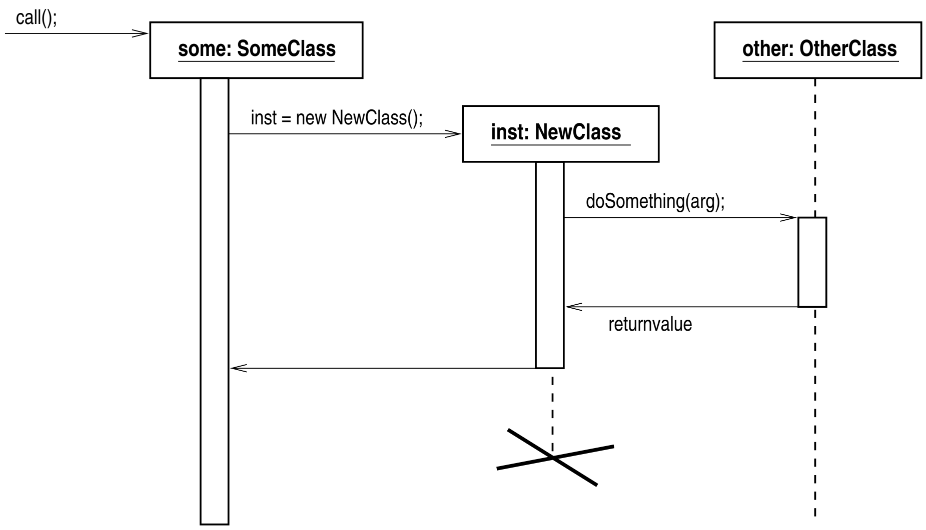


Abbildung 3.2: Notation von Sequenzdiagrammen.

Spezielle Formen der Assoziation sind **Aggregation und Komposition**, also *besteht-aus*-Beziehungen. Die strenge Form einer solchen Beziehung, die Komposition, wird durch eine schwarz ausgefüllte Raute an einem Assoziationspfeil notiert. In Abbildung 3.1 besteht etwa eine Instanz von `package2.CompositeClass` aus beliebig vielen Instanzen von `ElementClass`. Die Kardinalität auf Seiten eines Kompositums ist dabei implizit immer „1“. Die strenge Form bedeutet insbesondere eine Abhängigkeit in der Lebensdauer der Instanzen: Stirbt das Kompositum, so sterben auch seine Elemente. Dies gilt nicht für die schwache Form, die Aggregation; diese wird ebenso durch eine Raute notiert, die aber nicht gefüllt ist. Die Übergänge zwischen schwacher Komposition und einfacher Assoziation sind fließend und hängen oft nur von der Sichtweise ab.

Eine Menge von Klassen und Interfaces kann zu einem **Package** zusammengefaßt werden, und ein Package kann wiederum Subpackages enthalten. Packages definieren den Namensraum für die enthaltenen Klassen, so daß für die eindeutige Identifikation einer Klasse ggf. ein *Pfad* angegeben werden muß. Wir notieren die Pfade wie in Java, also z.B. `package2.OtherClass`. Eine *benutzt*-Beziehung zwischen Packages kann durch gestrichelte Pfeile angegeben werden. Ähnlich wie für die Klassen gilt auch für die Packages, daß jeweils nur diejenigen Ausschnitte aus einem vollständigen Package dargestellt werden, die in einem bestimmten Zusammenhang gerade interessieren.

Sequenzdiagramme

Sequenzdiagramme dienen zur Beschreibung der Interaktion zwischen Objekten, also von Instanzen der Klassen aus der statischen Beschreibung. Instanzen werden dargestellt wie Klassen, nur wird hier der Klassenname unterstrichen. Oft wird dem Objekt noch ein eigener Name gegeben, der dem Klassennamen vorangestellt wird. Entlang der vertikalen Achse eines Sequenzdiagramms verläuft die Zeit, gestrichelte Linien stellen die Lebensdauer der Objekte dar. Ein Methodenaufruf wird als Pfeil notiert, und die zeitliche Ausdehnung der Ausführung dieses Aufrufs symbolisiert dann ein weißer Balken. Innerhalb der Methodenausführung können weitere Aufrufe von Methoden auf anderen Objekten stattfinden. Terminiert eine Methode, so wird die Rückkehr des Kontrollflusses zum Aufrufer durch einen Pfeil in umgekehrter Aufrufrichtung dargestellt; dabei kann ein eventueller Rückgabewert benannt werden. Das Ende der Lebensdauer eines Objektes wird durch ein Kreuz markiert. Im Falle von Java geschieht das Löschen von Objekten asynchron durch eine Garbage-Collection.

3.2 Design Pattern

Das Prinzip von *Design Patterns* ist es, die Lösungsideen von bestimmten Problemen, die bei der objektorientierten Softwareentwicklung immer wieder auftreten, gewissermaßen als „Kochrezepte“ zu formulieren. Im allgemeinen betreffen diese Kochrezepte Fragen der Entkopplung und Abstraktion, durch die eine bessere Wartbarkeit und Wiederverwendbarkeit erreicht werden kann. [GHJV95] haben mit ihrem Buch über Design Patterns sicher zu recht eine große Pattern-Euphorie ausgelöst, denn diese Patterns geben viele Anregungen für gutes objektorientiertes Softwaredesign und fassen viele Erfahrungen zusammen, die auf diese Weise nicht mehr jeder Entwickler von neuem machen muß. Nicht zuletzt erlauben es die Begriffe, die für die einzelnen Patterns geprägt wurden, auf einer abstrakteren Ebene über Softwaredesign zu diskutieren. Dabei darf aber nicht vergessen werden, daß eine Ansammlung von Design-Patterns allein noch kein gutes Softwaredesign ausmacht. Patterns haben u.a. den Nachteil, daß sie durch die Einführung zusätzlicher Abstraktion auch die Anzahl der Klassen im System erhöhen; außerdem kann sich die Komplexität der Objektinteraktionen erheblich vergrößern, was bedeutet, daß sich die „Lesbarkeit“ eines Entwurfs nicht unbedingt verbessert. Schließlich kann das erhöhte Maß an Abstraktion auch die Effizienz beeinträchtigen, was sowohl Speicher- als auch Zeitaspekte betrifft. Es muß also in jedem Einzelfall geprüft werden, ob die Vorteile der Verwendung eines Patterns die Nachteile tatsächlich überwiegen.

Im folgenden geben wir einen kurzen Überblick über die beim Entwurf des AGG-Systems eingesetzten Design Patterns; für die ausführliche Beschreibung siehe [GHJV95].

Iterator

Ein Iterator ist ein Objekt, das man als Abstraktion von beliebigen konkreten Aggregationsobjekten, auch *Container* genannt, ansehen kann. Container in diesem Sinne sind z.B. Arrays, Listen, Vektoren, Mengen, Stacks usw. Ein Iterator erlaubt über eine schlanke Schnittstelle den sequentiellen Zugriff auf die Elemente eines solchen Containers. Durch die Verwendung von Iteratoren für den Zugriff auf Container kann der konkret verwendete Container z.B. aus Optimierungsgründen ausgetauscht werden, ohne den Zugriff zu beeinflussen. Dadurch ermöglichen Iteratoren auch die Implementierung von generischen Algorithmen über beliebigen Containern.

Alle modernen objektorientierten Programmiersprachen haben heute bereits ein Iteratorkonzept im Bibliotheksumfang. In Java heißt die entsprechende Iteratorschnittstelle `Enumeration`.

Factory Method

Factory Methoden sind spezielle Methoden, die die Erzeugung konkreter Instanzen kapseln. Der Rückgabewert einer Factory Methode ist als abstrakte Klasse oder als Interface typisiert, und abstrahiert damit von der konkreten Klasse, die innerhalb der Methoden mit Hilfe ihres Konstruktors erzeugt wird. Subklassen der Klasse, die die Factory Methode implementiert, können die Implementierung dieser Methode überschreiben und stattdessen eine andere konkrete Klasse instanziieren, die ebenfalls dem verlangten abstrakten Typ entspricht.

Das wichtigste Ziel dieses Patterns ist es, zu verhindern, daß die Instanziierung von Objekten einer konkreten Klasse über das ganze System verstreut wird. Nur so kann die konkrete Implementierungsklasse später ausgetauscht werden, ohne im gesamten System Änderungen vornehmen zu müssen.

Strategy

Als Strategy wird ein abstraktes Interface für eine Familie von Algorithmen bezeichnet. Betrachten wir ein Beispiel: Für die Implementierung eines Texteditors werden verschiedene Formatierungsalgorithmen gebraucht: linksbündig, rechtsbündig, zentriert usw. Als abstraktes Interface für diese Algorithmen reicht eine einzige Methode `format()` aus; wir nennen dieses Interface `FormatStrategy`. Dann implementieren wir jeden Algorithmus in der `format()`-Methode von entsprechenden Implementierungsklassen `FormatLeft`, `FormatRight` usw. Der Editor braucht nun zum Formatieren eines Textes nur das abstrakte Wissen um eine `FormatStrategy`. Nun kann von außen, z.B. von der graphischen Benutzeroberfläche aus, eine konkrete Strategy ausgewählt und dem Editor übergeben werden. Das Formatierungsverhalten wird also beeinflußt, ohne daß der Editor selbst davon Kenntnis hat. Dadurch wird es möglich, später bei Bedarf neue Formatierungsalgorithmen zu integrieren, ohne an der eigentlichen Implementierung des Editors auch nur eine Zeile zu ändern.

Wichtigstes Ziel dieses Patterns ist die dynamische Austauschbarkeit der Strategien zum einen, und zum anderen die bessere Wartbarkeit der über eine schlanke Schnittstelle entkoppelten Implementierung von komplexen Algorithmen.

Observer

Wenn man ein Softwaresystem zwecks Modularisierung in Komponenten teilt, tritt häufig das Problem auf, daß bestimmte Komponenten bezüglich der Zustandsänderungen von anderen Komponenten konsistent gehalten werden müssen. Oft manifestiert sich das Problem in einer n:1-Objektbeziehung, wo n Objekte vom Zustand eines anderen abhängig sind. Das Observer-Pattern bietet eine Möglichkeit, dieses Problem zu lösen, und dabei die Abhängigkeiten zwischen den beteiligten Objekten auf ein Minimum zu reduzieren. Das zentrale Objekt, von dessen Zustand die anderen abhängen, wird im Kontext des Patterns als *Subject* oder auch als *Observable* bezeichnet, die n abhängigen Objekte heißen *Observer*. Für beide Seiten stellt das Pattern abstrakte Schnittstellen zur Verfügung: Auf der Observable-Seite gibt es Methoden, die den Observern erlauben, sich an- und abzumelden; auf der Seite der Observer gibt es eine Methode `update()`, die vom Observable aufgerufen werden soll, wann immer sich dessen Zustand ändert. Bei einem konkreten Observable, das die Schnittstelle zum An-

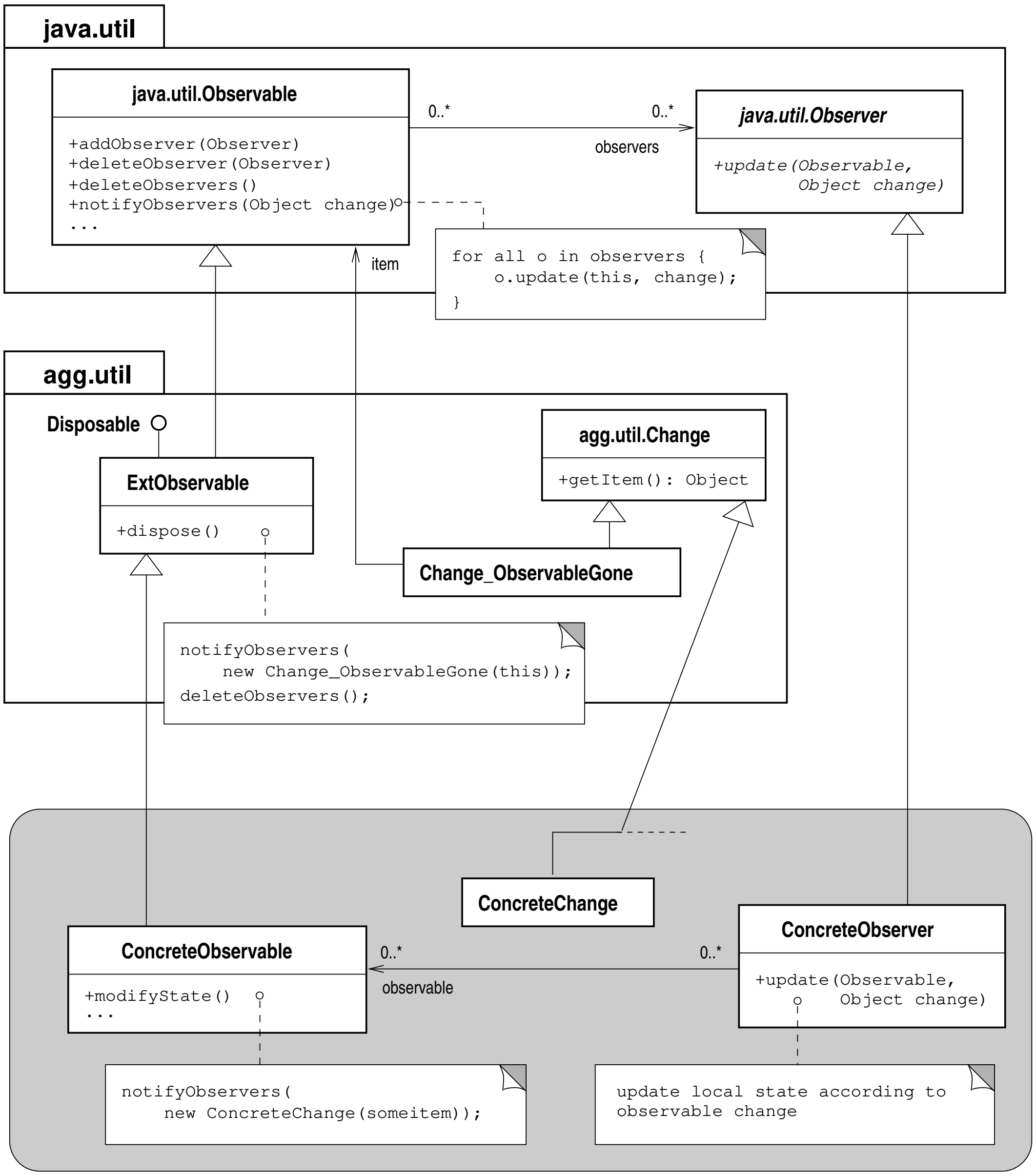


Abbildung 3.3: Erweiterung des Standard-Observer-Patterns von Java.

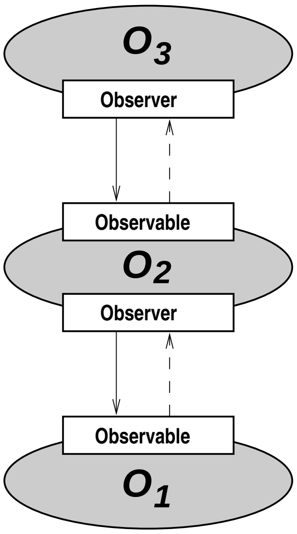


Abbildung 3.4: Schichtung von Observer-Pattern.

und Abmelden implementiert, kann sich nun jedes beliebige Objekt als Observer anmelden, die einzige Bedingung ist, daß es eine Implementierung der `update()`-Methode bereitstellt; von allen weiteren Eigenschaften wird abstrahiert. Das Observable selbst informiert dann bei jeder Änderung seines Zustands alle bei ihm angemeldeten Observer durch einen Aufruf von deren `update()`-Methode.

Durch das Observer-Pattern wird es möglich, gemeinsame Grundfunktionalitäten bzw. Datenbestände von mehreren Softwarekomponenten in ein Observable-Objekt auszulagern und durch beliebig viele Observer gleichzeitig zu nutzen. Dabei entsteht ein Effekt der Kooperation der Observer, ohne daß sie miteinander gekoppelt sein müssen, denn jeder Observer wird über den Update-Mechanismus über die Änderungen informiert, die von anderen Observern vorgenommen werden.

Eine wichtige Entscheidung, die bei der Anwendung des Observer-Patterns getroffen werden muß, ist, wieviel Information ein Observable über die konkrete Änderung seines Zustands an seine Observer weitergibt. Im einfachsten Fall wird nur die `update()`-Methode der Observer aufgerufen, ohne weitere Argumente zu übergeben. Man spricht in diesem Fall von einem *Pull*-Modell, denn die Observer müssen in der Folge selbst herausfinden, worin die Zustandsänderung des Observables bestand. Wenn der Zustand des Observables groß ist, kann das einen erheblichen Aufwand bedeuten. Das andere Extrem besteht im sog. *Push*-Modell, das besagt, daß ein Observable exakte Informationen über die konkrete Zustandsänderung als Update-Parameter übergibt, so daß von Seiten des Observers keine Ursachenforschung mehr betrieben werden muß. Dieses Vorgehen kann einen erhöhten Implementierungs- und Laufzeitaufwand auf Seiten des Observables bedeuten, der nicht zu rechtfertigen ist, wenn nicht davon ausgegangen werden kann, daß die Mehrzahl der Observer diese detaillierten Änderungsinformationen überhaupt benötigt. Zwischen den beiden Extremen des Push- und des Pull-Modells liegen beliebig viele Varianten, mehr oder weniger genaue Informationen über die Zustandsänderungen zu verschicken.

Das Observer-Pattern ist in der Standardbibliothek von Java bereits integriert; im Package `java.util` gibt es ein Interface `Observer` und eine Klasse `Observable`, die den An- und Abmeldemechanismus und die Obserververwaltung bereits vorimplementiert. Abbildung 3.3 zeigt die beiden Schnittstellen in UML Klassendiagrammnotation. In der Abbildung grau unterlegt ist eine hypothetische konkrete Implementierung von Observer und Observable.

Ein Problem bei der Implementierung des Observer-Patterns in Java stellen die zirkulären Referenzen dar, die dem Pattern inhärent sind: Auf der abstrakten Ebene kennt das Observable alle seine Observer, und auf der konkreten Ebene greift der Observer natürlich auf den Zustand des Observables zu. Objekte, die an zirkulären Referenzen beteiligt sind, können von der Garbage-Collection jedoch nur dann aus dem Speicher entfernt werden, wenn durch eine aufwendige Analyse des Speicherzustands festgestellt wurde, daß keines der an dem Zirkel beteiligten Objekte mehr von einem „lebendigen" Teil des Programms aus erreichbar ist. Aufgrund des hohen erforderlichen Aufwands wird eine solche Erreichbarkeitsanalyse vom Java-Laufzeitsystem nur sehr selten ausgelöst – im allgemeinen erst dann, wenn akuter Speicherplatzmangel besteht. Das bedeutet aber, daß zuvor auch ggf. vorhandener virtueller Speicher exzessiv genutzt wird, was üblicherweise deutliche Performance-Einbußen zur Folge hat. Objekte dagegen, für die eine einfache Zählung der Referenzen ergibt, daß sie nicht mehr referenziert werden, werden von der Garbage-Collection unabhängig vom Aufwand einer Erreichbarkeitsanalyse sofort freigegeben. Um diese effizientere Zugriffsmöglichkeit für den Garbage-Collector zu bieten, muß also zumindest eine der beiden Beziehungen im Observer-Pattern zunächst explizit gelöst werden. Dazu können wir einfach die Methode `deleteObservers()` auf dem Observable aufrufen, denn dadurch wird dieses veranlaßt, alle Referenzen auf seine Observer zu vergessen; damit wäre der Zirkel aufgelöst. Ein Problem entsteht aber z.B., sobald eine Schichtung von Observer-Beziehungen wie in Abbildung 3.4 vorliegt. Weil O_2 nichts davon erfährt, daß sein Observable O_1 ein `deleteObservers()` durchgeführt hat, wird O_2 weiterhin O_1 referenzieren, O_2 aber wird wiederum von seinen eigenen Observern referenziert. In dieser Situation kann die Garbage-Collection also aufgrund von einfacher Referenzzählung weder auf O_1 noch auf O_2 oder O_3 zugreifen, und so bleibt die gesamte Konfiguration als Speicherleiche erhalten, bis die nächste vollständige Erreichbarkeitsanalyse durchgeführt wird. Häufig wird aber ein Observer wie O_3 über die in Abbildung 3.4 dargestellten Beziehungen hinaus noch von anderen, möglicherweise „lebendigen" Programmteilen referenziert, so daß der Garbage-Collector selbst nach einer Erreichbarkeitsanalyse nicht auf die Objekte zugreifen kann.

Um dieses Problem zu lösen, spezialisieren wir die Klasse `Observable` zu einer Klasse `ExtObservable`, die eine Methode `dispose()` ergänzt. Diese Methode muß explizit aufgerufen werden, wenn das Observable aus dem Speicher gelöscht werden soll, und beseitigt durch einen Aufruf von `deleteObservers()` zunächst die zirkulären Referenzen auf der untersten Ebene. Gleichzeitig wird an alle Observer eine spezielle Änderungsinformation `Change_ObservableGone` verschickt, wodurch die Observer dazu aufgefordert werden, ihre Referenzen auf das Observable ebenfalls zu löschen. Dieses Verfahren wird dann im Falle einer Observerschichtung ggf. in den oberen Schichten fortgesetzt. Zu beachten ist dabei, daß jeder konkrete Observer die Behandlung der Änderungsinformation `Change_ObservableGone` explizit implementieren muß.

Das Problem der zirkulären Referenzen beschränkt sich nicht nur auf das Observer-Pattern, sondern kann in vielen Zusammenhängen auftreten. Jede Klasse, die an zirkulären Referenzen beteiligt ist, sollte deshalb durch die Implementierung des Interfaces `Disposable` anzeigen, daß sie explizit durch einen Aufruf ihrer `dispose()`-Methode aufgeräumt werden muß, um eine möglichst effiziente Speichernutzung zu gewährleisten.

3.3 Java

Für die Implementierung des neuen AGG-Systems wurde die von *Sun Microsystems* entwickelte objektorientierte Programmiersprache *Java* gewählt. Die Entscheidung für die Umstellung von der in [Rud96] verwendeten Sprache C++ war vor allem durch die Erwartung begründet, mit Java einen schnelleren Entwicklungsfortschritt zu erzielen. Diese Erwartungen haben sich im wesentlichen bestätigt. Entscheidende Ursachen hierfür sind:

- Das Bytecode-Compiler/Interpreter-Konzept, das kurze Compilezeiten ermöglicht.
- Der große Umfang an Standardbibliotheken, die u.a. eine GUI-Bibliothek (Abstract Window Toolkit, AWT) und eine Bibliothek für Verteilungskonzepte (Remote Method Invocation, RMI) beinhalten.
- Der gegenüber C++ verkleinerte Sprachumfang mit klareren Konzepten, der die Übersichtlichkeit erhöht.
- Keine Zeiger und keine Möglichkeit, zwischen *call-by-value* und *call-by-reference* zu wählen, was die Wahrscheinlichkeit von Programmierfehlern deutlich senkt.

Ein weiterer großer Vorteil der Programmiersprache Java ist ihre Plattformunabhängigkeit und die freie Verfügbarkeit der entsprechenden Laufzeit- und Entwicklungsumgebungen für alle gängigen Betriebssysteme. Schließlich war für die Entscheidung auch die weite Verbreitung der Sprache wichtig, die eine Unterstützung auch auf lange Sicht erwarten läßt.

Meistens wird als einer der größten Vorteile von Java gegenüber C++ die eingebaute Garbage-Collection erwähnt, die den Programmierer von der aufwendigen Verantwortung befreit, den Speicher für erzeugte Objekte auch selbst explizit wieder freigeben zu müssen. Vor dem Hintergrund der Probleme mit zirkulären Referenzen, die wir im Zusammenhang mit dem Observer-Pattern in Abschnitt 3.2 beschrieben haben, muß dieser Vorteil aber relativiert werden. Da derartige Referenzen explizit aufgelöst werden müssen, um eine effiziente Speichernutzung zu gewährleisten, liegt die Verantwortung für das Aufräumen des Speichers letztlich ähnlich wie schon in C++ wieder beim Programmierer.

Die wichtigsten Nachteile von Java, insbesondere gegenüber C++, sind:

- Die Ausführungsgeschwindigkeit eines interpretierten Java-Programms ist nicht zu vergleichen mit der eines compilierten C++-Programms. Auch der Speicherbedarf von Java-Programmen ist sehr viel größer.
- Keine Mehrfachvererbung.
- Keine parametrisierten Datentypen mit statischem Type-Checking à la C++-Templates. Dadurch verschlechtert sich die Lesbarkeit des Programmcodes, und das erforderliche dynamische Type-Checking hat negative Auswirkungen auf die Performance.

- Die Standardbibliotheken sind in einigen Belangen nicht leistungsfähig genug; insbesondere das AWT bietet zu wenig Unterstützung für komplexe Graphikoperationen. Alternative Bibliotheken sind kaum verfügbar.

Die Performance-Nachteile waren zum Zeitpunkt der Entscheidung für die Sprache Java bereits absehbar und haben sich in Anbetracht des Demonstrationscharakters des AGG-Systems bisher nicht als gravierendes Problem erwiesen. Für die Ausführung größerer Spezifikationen könnte sich das in Zukunft allerdings ändern. Dabei ist aber auch zu berücksichtigen, daß sich die Entwicklung der Bytecode-Compiler und Interpreter für Java noch in einem relativ frühen Stadium befindet, so daß hier weitere Verbesserungen zu erwarten sind. Außerdem sind bereits einige Übersetzer von Java nach C und sogenannte „Just-in-Time"-Compiler verfügbar, die den Java-Bytecode zur Laufzeit in echten nativen Maschinencode übersetzen. Es stehen also bereits heute eine Reihe von Optionen offen, und in Zukunft werden auf dem sich schnell entwickelnden Java-Markt sicher weitere hinzukommen.

Für die Entwicklung des AGG-Systems wurde das Java Development Kit (JDK) in der Version 1.1 verwendet. Zusätzlich kam die Java Generic Library (JGL) in der Version 2.0 zum Einsatz. Diese Bibliothek wurde von der Firma ObjectSpace entwickelt und ist frei verfügbar. Die JGL entspricht der Funktionalität der Standard Template Library (STL) unter C++, sie bietet also eine Sammlung von Containern, Iteratoren und generischen Algorithmen.

Kapitel 4

Systemstruktur

In den beiden folgenden Abschnitten geben wir einen Überblick über die Struktur des AGG-Systems. Dabei kommen wir über die allgemeine Philosophie der Architektur zu einer Kurzbeschreibung der einzelnen Packages.

4.1 Basis und Komponenten

Die einzelnen Klassen für den Entwurf des AGG-Systems werden ihren Aufgabenbereichen entsprechend zu *Packages* zusammengefaßt. Dabei wird der Package-Begriff, den UML für die Entwurfsphase bereitstellt, bei der Implementierung direkt durch das Package-Konzept der Programmiersprache Java umgesetzt. Anhand dieser Packages und ihrer Beziehungen gibt Abbildung 4.1 einen Überblick über die Systemstruktur. Die grau unterlegten Ellipsen drücken darüber hinaus eine weitere Gliederung der Packages in abstrakte *Systemkomponenten* aus. Diese weitere Strukturierungsebene ist rein ideeller Natur und hat weder in der Entwurfsmethodik noch in der Implementierung eine Entsprechung. Sie teilt das System in zwei Teile: eine *Basiskomponente*, auch kurz Basis genannt, und eine beliebige Anzahl von *Anwendungskomponenten* (kurz: Komponenten).

Aufgabe des Basissystems im AGG ist es, die Begriffe der Theorie der attributierten algebraischen Graphtransformation möglichst direkt und allgemein umzusetzen, ohne dabei auf ein konkretes Anwendungsgebiet hin zu spezialisieren oder zu optimieren. Die Konzeption und Implementierung dieses Basissystems bildet den Gegenstand der vorliegenden Arbeit. Der Begriff der Anwendungskomponente ist dagegen sehr weit gefaßt; jedes System, das auf die Funktionalitäten der Basiskomponente zugreift, gilt in diesem Zusammenhang als Anwendung. In bezug auf die Anwendungskomponenten hat die Basis also den Charakter einer *Bibliothek* für Graphdatenstrukturen und deren Transformation. Jede Anwendung kann das Graphmodell der Basis für ihre Zwecke interpretieren und um anwendungsspezifische Aspekte erweitern. Die Implementierung derartiger Erweiterungen erfolgt lokal in der jeweiligen Anwendungskomponente. Das aber bedeutet im allgemeinen, daß ein Bestand an lokalen Ergänzungsdaten entsteht, der in enger Beziehung zu den Basis-Graphdaten steht und bezüglich deren Veränderungen konsistent gehalten werden muß. Deshalb ist es typisch, wenn auch nicht zwingend, daß die Klassen einer Anwendungskomponente in einer Observer-Beziehung zur Basis stehen (vgl. Abschnitt 3.2, Observer-Pattern). Dadurch wird es außerdem

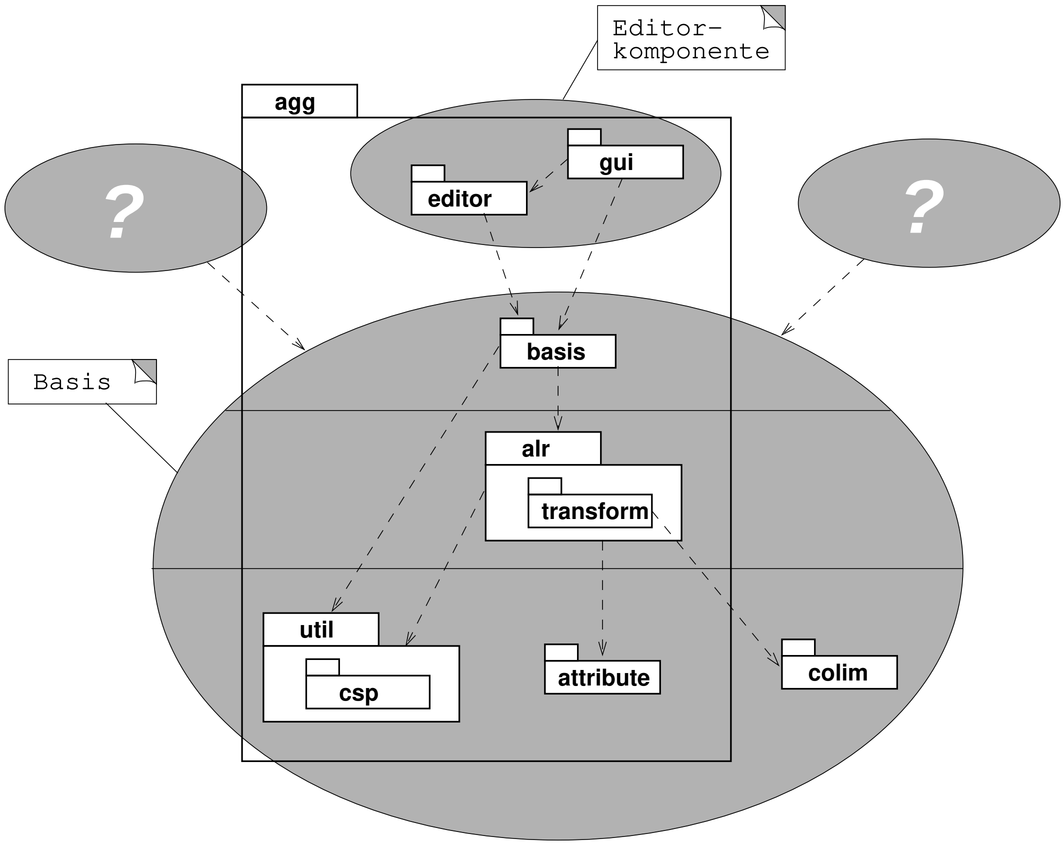


Abbildung 4.1: Struktur des AGG-Systems.

möglich, daß verschiedene Anwendungen gleichzeitig und kooperativ auf einer Basis agieren, auch wenn sie völlig unabhängig voneinander entwickelt wurden.

Am Beispiel der Editorkomponente aus Abb. 4.1 läßt sich die Beziehung zwischen Basis und Anwendungskomponenten gut nachvollziehen. Ein Grapheditor soll das Anzeigen und interaktive Editieren von Graphen ermöglichen. Dazu benötigt er Informationen über Layout und graphische Repräsentation, also z.B. über Bildschirmkoordinaten und Farbe der angezeigten Graphobjekte. Da die konkrete Bildschirmrepräsentation anwendungsabhängig ist, werden derartige Daten von der Basis nicht unterstützt. Die Editorkomponente baut deshalb lokale Strukturen auf, um jedes Graphobjekt aus der Basis um Repräsentationsaspekte zu ergänzen. Werden nun in der Basis Objekte erzeugt oder gelöscht, so muß auch der Editor seine zu dem entsprechenden Objekt gehörende Repräsentation löschen oder erzeugen. Solange der Editor davon ausgehen kann, daß er allein auf der Basis operiert, sind asynchrone Änderungen des Basiszustandes ausgeschlossen, d.h., der Editor kann allein aufgrund seines eigenen Verhaltens darauf schließen, wann er seinen Zustand mit dem der Basis neu synchronisieren muß. Sobald aber mehrere Komponenten auf einer Basis aktiv sind, seien es auch nur zwei Instanzen des Grapheditors, dann ist eine automatische Konsistenzsicherung nur noch mit Hilfe des Observer-Patterns möglich, wobei die Basis in der Rolle des Observables die Anwendungskomponenten als Observer aktiv über die Änderungen ihres Zustands informiert. So könnte z.B. eine als Observer realisierte Animationskomponente die Erzeugung eines Objekts, die vom Editor ausgelöst wurde, sofort durch eine Filmsequenz visualisieren, oder auch die Auswirkungen einer vom Editor aus angestoßenen Transformation.

Auf die einzelnen Packages und ihre Aufteilung in Schichten innerhalb des Basissystems gehen wir im folgenden Abschnitt ein.

4.2 Packages

Wir fassen nun kurz die Aufgaben der einzelnen Packages aus der Strukturübersicht in Abbildung 4.1 zusammen und verweisen dabei jeweils auf die Kapitel, in denen diese Packages ausführlich behandelt werden, bzw. auf die entsprechende Literatur.

Package `alr`. Hier wird die grundlegende Datenstruktur des AGG-Systems implementiert, der ALR-Graph. Er definiert eine Observable-Schnittstelle mit umfassenden Änderungsinformationen. Die zentralen Funktionalitäten von Ansatzsuche und Transformation werden über diesem Graphmodell implementiert. Die Beschreibung erfolgt in Kapitel 5.

Package `alr.transform`. Dieses Paket implementiert alles, was über die reinen ALR-Datenstrukturen hinaus für die Transformationsfunktionalität nötig ist, und beinhaltet die Schnittstellen für ALR-Morphismen, Ansatzsuche und Transformation. Die Morphismus-Repräsentation und die Transformation sind Thema von Kapitel 6, Kapitel 7 behandelt die Ansatzsuche.

Package `basis`. In diesem Paket finden sich Schnittstellen für Ansatzsuche und Transformation auf einfachen Graphen. Diese Funktionalitäten werden als Spezialfälle über den ALR-Implementierungen aus dem `alr`-Package realisiert. Diese Schnittstellen und das Konzept für ihre Implementierung werden in Kapitel 8 vorgestellt.

Package `util`. Im Package `util` werden einige Interfaces und abstrakte Klassen definiert, die für die Wiederverwendung in verschiedenen Zusammenhängen geeignet sind. Klassen aus diesem Paket werden in verschiedenen Kapiteln im jeweiligen Zusammenhang beschrieben.

Package `util.csp`. Aufgrund seiner prinzipiellen Wiederverwendbarkeit wurde die Implementierung des Frameworks zur Lösung von binären Constraint Satisfaction Problemen in ein Sub-Package von `util` ausgelagert. Den Entwurf des Frameworks und eines Lösungsalgorithmus behandeln die Abschnitte 7.3.3 und 7.3.4. Abschnitt 7.3.5 gibt ein Beispiel für die Anwendung des Frameworks.

Package `colim`. Dieses Package enthält die Implementierung der Colimit-Library zur Berechnung allgemeiner Colimiten auf ALPHA-Algebren [Wol97]. Wir verwenden diese Bibliothek für die Berechnung eines Graphtransformationsschrittes. Die Anbindung der Bibliothek beschreibt Abschnitt 6.2.7, Abschnitt 6.1.4 diskutiert die Beziehung zwischen ALPHA-Algebren und dem im AGG verwendeten Graphmodell der ALR-Graphen.

Package `attribute`. Die gesamte Attributfunktionalität von der Eingabe der Attribute über die Belegung von Variablen bis hin zur Berechnung des Transformationsergebnisses auf dem Attributteil wird ausgelagert in eine Attributkomponente [Mel97]. Das Thema der

Attributierung zieht sich durch alle Teile der Arbeit hindurch: Die Integration der Attribute in die Graphdatenstrukturen erfolgt in den Abschnitten 5.2.2 und 8.2, die Attributbehandlung in Morphismen aus theoretischer bzw. technischer Sicht beschreiben die Abschnitte 6.1.3 und 6.2.5, und Abschnitt 7.3.6 schließlich geht auf die Schwierigkeiten bei der Formulierung von Attribut-Constraints für die Ansatzsuche ein.

Packages `editor` und `gui`. Diese Packages bilden die Editorkomponente des AGG-Systems, die sich parallel zum Basissystem in der Entwicklung befindet. Im Package `gui` ist dabei auch eine Benutzeroberfläche für die Steuerung der Transformationsfunktionen vorgesehen. Auf die Editorkomponente kann in dieser Arbeit nicht näher eingegangen werden.

Package `agg`. Alle Packages, die im Zusammenhang mit AGG entwickelt wurden, sind in diesem Wurzelpaket zusammengefaßt. Die Ausnahme bildet hier die Colimit-Bibliothek, deren Entwicklung vom AGG-System unabhängig verlief.

Die Packages des Basissystems können drei verschiedenen Schichten zugeordnet werden, was in Abbildung 4.1 durch horizontale Schnitte angedeutet wird. Die unterste Schicht wird gebildet von den Packages `util`, `attribute` und `colim`. Wir können diese Schicht als *Bibliotheksschicht* auffassen, die durch eine vergleichsweise hohe Wiederverwendbarkeit in von Graphen und Graphtransformation unabhängigen Kontexten gekennzeichnet ist. Darüber liegt die *Implementierungsschicht* der ALR-Graphen, auf der die wesentlichen Funktionalitäten des Basissystems implementiert werden. Die oberste Schicht der Basiskomponente bildet die *Schnittstellenschicht* für einfache Graphen, die von der Implementierung durch ALR-Graphen abstrahiert. Die Wiederverwendbarkeit nimmt also in der Abbildung von unten nach oben mit wachsender Spezialisierung ab. In dieses Bild passen auch die Anwendungskomponenten als höchste Stufe der Spezialisierung, genau wie als unterste und allgemeinste Ebene die Bibliotheken JDK und JGL der Java-Entwicklungsumgebung, die hier nicht abgebildet sind.

Kapitel 5

Attributierte ALR-Graphen

Das AGG-System basiert auf einem hierarchischen Graphbegriff, den *ALR-Graphen*; die grundlegenden Funktionalitäten des Interpreters, Ansatzsuche und Transformation, werden über diesem Graphmodell realisiert. Die ALR-Graphen stellen eine Verallgemeinerung des in Kapitel 2 vorgestellten einfachen Graphbegriffs dar, indem sie mit der Abstraktionsabbildung neben der Source- und der Target-Abbildung eine weitere Operation auf den Knoten und Kanten eines Graphen einführen. In den folgenden Abschnitten geben wir eine Definition für dieses Graphmodell an und beschreiben dann die Umsetzung in objektorientierte Datentypen.

Der Konzeption und Implementierung der Basisdatenstrukturen in Form von ALR-Graphen galt bereits die Studienarbeit [Rud96]. Während sich die Entwurfs- und Implementierungsbedingungen durch die Umstellung der verwendeten Programmiersprache von C++ auf Java und den Abschied von der Integration des ursprünglichen AGG-Editors wesentlich verändert haben, können wir die dort entwickelten Konzepte und Schnittstellen aber zum großen Teil auf die veränderte Situation übertragen. Einige Konzepte, die in der Studienarbeit ausführlich behandelt wurden, werden wir hier deshalb etwas kürzer behandeln, wie z.B. den Einsatz des Observer-Patterns zur Unterstützung der modularen Erweiterbarkeit. Das Locking-Konzept auf ALR-Graphen, das in [Rud96] zur Sicherung gegen Schreiber/Leser-Konflikte unter nebenläufigen Bedingungen vorgesehen war, greifen wir aus Prioritätsgründen in dieser Arbeit nicht wieder auf; der derzeitige Systementwurf ist rein sequentiell. Gleichzeitig zu dieser Diplomarbeit entsteht aber in einem studentischen Projekt eine Version des AGG-Systems, in der versucht wird, die Ideen der verteilten Graphtransformation nach [Tae96, TK97] zu integrieren. Eine Evaluation, inwieweit die Locking-Funktionalität in diesem Zusammenhang von Bedeutung ist, steht noch aus.

5.1 Begriffe und Konzepte

5.1.1 ALR-Graphen

Die von Arlt, Löwe und Röder entwickelten *ALR-Graphen* [AR89, Löw93] führen eine Abstraktionsabbildung zwischen den Knoten und Kanten eines Graphen ein, mit der sich Graphen zu Knoten, Kantenbüschel zu Kanten abstrahieren lassen. Diese Abstraktion kann rekursiv angewandt werden, und durch bestimmte Bedingungen, die an die Abstraktionsabbildung gestellt werden, ergibt sich eine hierarchische Aufteilung des Graphen in Ebenen. Außerdem werden in einem ALR-Graphen die Knoten- und Kantenmengen zu einer gemeinsamen Menge von *Graphobjekten* zusammengefaßt. Knoten und Kanten bleiben über ihre Eigenschaften zwar weiterhin identifizierbar, aber es ergibt sich eine Verallgemeinerung insofern, als nun auch Kanten zwischen zwei Kanten oder einem Knoten und einer Kante zugelassen sind.

In Abschnitt 6.1.2 werden wir sehen, wie es diese Erweiterungen des Graphbegriffs ermöglichen, Morphismen zwischen ALR-Graphen direkt im Graphmodell zu repräsentieren, obwohl der Morphismus eigentlich ein Meta-Begriff ist, um die Beziehungen zwischen Graphen zu beschreiben. Ganz allgemein führt das dazu, daß sich beliebige Meta-Ebenen über ALR-Graphen wieder als ALR-Graph darstellen lassen. So kann z.B. eine ganze Graphgrammatik auf elegante Weise durch einen ALR-Graphen modelliert werden, wobei die Graphen der Grammatik wiederum Graphgrammatiken sein können. Auf die Modellierung einer Graphgrammatik kommen wir in Kapitel 8 zurück.

Die folgende Definition geht auf [Bey91] zurück:

Definition 5.1 (*ALR-Graph*) Ein Tupel $G = (O, \perp, h, a, s, t, L, \ell)$ mit

- einer endlichen Objektmenge O,
- zwei ausgezeichneten Objekten $\perp \in O$ und $h \in O$,
- drei totalen Abbildungen $a, s, t : O \to O$ (für „abstraction“, „source“ und „target“),
- einer Labelmenge L und
- einer totalen Abbildung $\ell : O \to L$

heißt *ALR-Graph*, wenn die folgenden Bedingungen erfüllt sind:

1. (Eigenschaften des Bottom-Elements)
   $a(\perp) = s(\perp) = t(\perp) = \perp$
2. (Objekt ist Knoten oder Kante)
   $\forall o \in O : \quad s(o) = \perp \iff t(o) = \perp$
3. (Knoten mit Knoten als Abstraktion)
   $\forall o \in O : \quad s(o) = \perp \implies s(a(o)) = \perp$
4. (Kante mit Knoten als Abstraktion)
   $\forall o \in O : \quad s(o) \neq \perp,\ s(a(o)) = \perp \implies a(s(o)) = a(t(o)) = a(o)$
5. (Kante mit Kante als Abstraktion)
   $\forall o \in O : \quad s(o) \neq \perp,\ s(a(o)) \neq \perp \implies a(s(o)) = s(a(o)) \wedge a(t(o)) = t(a(o))$
6. (Zyklenfreiheit der Abbildungen a, s, t)
   $\neg\exists n \in I\!N$, so daß ein Folge $(o_0, o_1, ..., o_n)$ existiert mit $o_i \in O \setminus \{\perp\}$, für die gilt: $(o_0, o_1), (o_1, o_2), \ldots, (o_{n-1}, o_n), (o_n, o_0) \in a \cup s \cup t$
7. (Eindeutigkeit des Top-Objekts)
   $\forall o \in O : \quad a(o) = \perp \iff o = h$

△

Die Labelmenge L zusammen mit der Abbildung ℓ erlaubt eine Typisierung der Objekte aus O. Knoten und Kanten aus der Objektmenge werden anhand ihrer Quell- und Zielobjekte unterschieden: Ein Objekt $o \in O$ ist genau dann ein Knoten, wenn $s(o) = \perp$; nach Bed. 2 ist dann auch schon $t(o) = \perp$. Anschaulich ausgedrückt bedeutet das, daß ein Knoten weder Quelle noch Ziel hat. Man kann sich deshalb s und t als partielle Operationen vorstellen, die für Knoten undefiniert sind. Das Bottom-Objekt $\perp$ hat lediglich die Funktion, diese eigentlich partiell definierten Abbildungen künstlich zu totalisieren. Dennoch bezeichnet man die Abbildungen oft als *undefiniert* an den Stellen, an denen ihr Wert $\perp$ ist.

Die Bedingungen 3 bis 5 fordern eine gewisse Verträglichkeit der Abstraktion mit den Source- und Target-Operationen. So darf z.B. eine Kante nur auf eine Kante mit gleicher Richtung abstrahiert werden. Aus den Bedingungen ergibt sich auch die implizite Ebenenzuordnung der Objekte, denn

$$\forall x \in O : \exists n \in I\!N \text{ mit } a^n(x) = \perp,$$

wobei $a^n(x)$ eine abkürzende Schreibweise für die n-malige Anwendung von a auf x ist. n bezeichnet also den Abstand eines Objektes vom Top-Objekt und identifiziert damit die Ebene, der es zugeordnet wird. Die Bedingungen erzwingen dabei, daß Source- und Target-Objekte einer Kante immer auf derselben Ebene liegen wie die Kante selbst.

Die Umkehrrelation der Abstraktionsfunktion kann man als *Verfeinerung* auffassen. In [Rud96] wurde für die Gesamtheit der direkten und indirekten Verfeinerungsobjekte eines Knotens der Begriff des *Verfeinerungsbaums* definiert. Wir verallgemeinern diesen Begriff auf Verfeinerungsbäume beliebiger Graphobjekte:

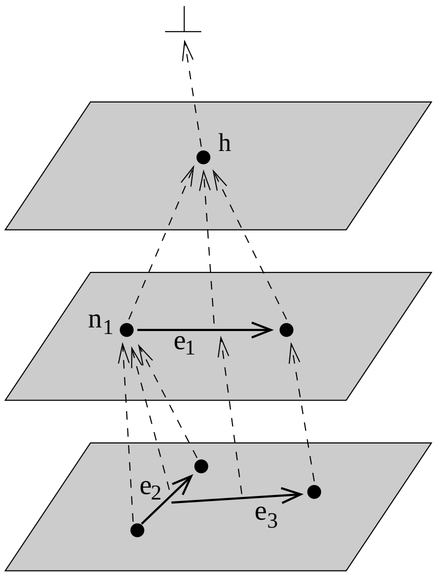
Definition 5.2 (*Verfeinerungsbaum*)
Sei $G = (O, \perp, h, a, s, t, L, \ell)$ ein ALR-Graph. Dann ist der *Verfeinerungsbaum* G_v eines Objekts $v \in O \setminus \{\perp\}$ definiert durch das Tupel $G_v = (O_v, \perp, v, a_v, s_v, t_v, L, \ell_v)$ mit

- $O_v = \{x \in O \mid \exists n \in I\!N : a^n(x) = v\} \cup \{\perp\}$
- $a_v : O_v \to O_v \quad \text{mit} \quad a_v(x) = \begin{cases} a(x), \textit{ wenn } & x \neq v \\ \perp, \textit{ wenn } & x = v \end{cases}$
- Zwei Fälle:
  1. (*v ist Knoten, d.h.* $s(v) = \perp$.) $\quad s_v, t_v : O_v \to O_v \quad \text{mit} \quad s_v = s|_{O_v}$ und $t_v = t|_{O_v}$
  2. (*v ist Kante, d.h.* $s(v) \neq \perp$.) Seien O_s und O_t die Objektmengen der Verfeinerungsbäume von $s(v)$ bzw. von $t(v)$. Dann ist

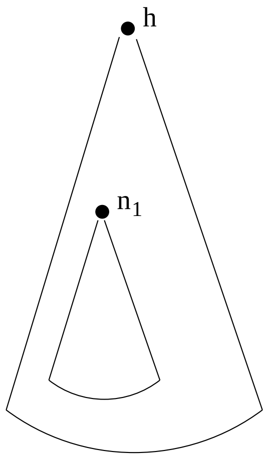
  $$\begin{array}{l} s_v : O_v \to O_s \quad \text{mit} \quad s_v = s|_{O_v} \text{ und} \\ t_v : O_v \to O_t \quad \text{mit} \quad t_v = t|_{O_v}. \end{array}$$

- $\ell_v : O_v \to L \quad \text{mit} \quad \ell_v = \ell|_{O_v}$.

h_v heißt *Top-Objekt* des Verfeinerungsbaums. △



(a) ein ALR-Graph



(b) schematische Darstellung des Verfeinerungsbaums von h

Abbildung 5.1: Ein ALR-Graph und die schematische Darstellung von Verfeinerungsbäumen.

Es ist leicht zu sehen, daß Verfeinerungsbäume von Knoten selbst ALR-Graphen sind:

Satz 5.3 (*Verfeinerungsbäume von Knoten sind ALR-Graphen*)
Sei $G = (O, \perp, h, a, s, t, L, \ell)$ ein ALR-Graph. Dann ist jeder Verfeinerungsbaum G_v eines Knotens $v \in O \setminus \{\perp\}$ selbst ein ALR-Graph. $\triangle$

Jeder Knoten in einem ALR-Graphen repräsentiert also selbst einen ALR-Graphen, und umgekehrt wird jeder ALR-Graph eindeutig durch sein Top-Objekt repräsentiert.

An den Verfeinerungsbäumen von Kanten fällt dagegen auf, daß die Source- und Targetabbildungen nicht abgeschlossen sind in der Objektmenge des Verfeinerungsbaums. Einen Verfeinerungsbaum einer Kante v können wir als Relation zwischen den Objekten der Verfeinerungsbäume von $s(v)$ und $t(v)$ ansehen, mit der Eigenschaft, daß nur Objekte in Relation gesetzt werden können, die auf derselben Ebene liegen.

Abbildung 5.1(a) zeigt ein Beispiel für einen ALR-Graphen, wobei die Abstraktionsabbildung durch gestrichelte Pfeile dargestellt ist. In Abb. 5.1(b) sehen wir die schematische Darstellung der Verfeinerungsbäume von h und n_1, aus der die Teilmengenbeziehung der Objektmengen der beiden Verfeinerungsbäume hervorgeht.

5.1.2 Attributierung

Analog zur Attributierung einfacher Graphen in Kapitel 2 erweitern wir auch die ALR-Graphen um die Möglichkeit, die Graphobjekte über einer gegebenen Attributalgebra zu attributieren:

Definition 5.4 (*Attributierter ALR-Graph*) Ein Tupel $G_{AS} = (G, A, at, av)$ mit

- einem ALR-Graphen $G = (O, \bot, h, a, s, t, L, \ell)$,
- einer Algebra A zu einer Signatur $AS = (S, OP)$,
- einer totalen Abbildung $at : L \to S$ (für „attribute type") und
- einer totalen Abbildung $av : O \to AV$ (für „attribute value"), wobei $AV = \biguplus_{s \in S} A_s$ die disjunkte Vereinigung aller Trägermengen von A ist,

heißt *attributierter ALR-Graph*, wenn für $a\ell : AV \to S$ mit $\forall v \in A_s : \; a\ell(v) = s$ das folgende Diagramm kommutiert:

$$\begin{tikzcd} L \arrow[r, "at"] \arrow[dr, phantom, "(=)"] & S \\ O \arrow[u, "\ell"] \arrow[r, "av"'] & AV \arrow[u, "a\ell"'] \end{tikzcd}$$

△

Entsprechend erweitern wir auch die Definition der Verfeinerungsbäume:

Definition 5.5 (*Attributierter Verfeinerungsbaum*)
Sei $G_{AS} = (G, A, at, av)$ mit $G = (O, \bot, h, a, s, t, L, \ell)$ ein attributierter ALR-Graph, $G_v = (O_v, \bot, h_v, a_v, s_v, t_v, L, \ell_v)$ Verfeinerungsbaum von $h_v \in O$. Dann ist $G_{ASv} = (G_v, A, at, av_v)$ mit

$$av_v : O_v \to AV, \; av_v = av|_{O_v}$$

ein *attributierter Verfeinerungsbaum* von h_v. △

Satz 5.3 gilt natürlich entsprechend für attributierte ALR-Strukturen. Im weiteren Verlauf der Arbeit sprechen wir auch im attributierten Fall einfach von ALR-Graphen.

5.2 Entwurf

Der Entwurf der grundlegenden Graphdatenstrukturen auf Basis von ALR-Graphen war bereits zentraler Gegenstand der Studienarbeit [Rud96]. Die folgenden Abschnitte bauen auf den Ergebnissen der Studienarbeit auf, sind aber dennoch so gestaltet, daß sie auch ohne Kenntnis dieser Arbeit verständlich sein sollen. Im Text werden wir an einigen Stellen für weitere Einzelheiten auf die entsprechenden Abschnitte in der Studienarbeit verweisen.

5.2.1 Anforderungsdefinition

Die Anforderungen an die Basisdatenstrukturen wurden bereits in [Rud96] ausführlich dokumentiert. Hier deshalb nur eine kurze Zusammenfassung mit einigen Anpassungen.

Funktionale Anforderungen

Typisierte, attributierte ALR-Graphen repräsentieren.
ALR-Graphen gemäß Def. 5.1 müssen erzeugt und verwaltet werden, die Attributierung von Knoten und Kanten soll mit Hilfe der Attributkomponente aus [Mel97] erfolgen.

Korrespondenz zur Theorie bewahren.
Um theoretische Ergebnisse aus dem Bereich der algebraischen Graphtransformation möglichst direkt praktisch umsetzen zu können, wollen wir uns, wo immer möglich, eng an den Definitionen der Theorie orientieren. Das bedeutet insbesondere, daß die Basisdatenstrukturen von theoriefremden Aspekten wie dem Layout eines Graphen oder der graphischen Repräsentation seiner Knoten und Kanten befreit werden sollten. Derartige Daten, die stark von einer konkreten Anwendungssicht abhängen, sollen deshalb auch lokal in der jeweiligen Anwendungskomponente gehalten werden. Die ergänzende lokale Datenhaltung in höheren Implementierungsschichten führt aber wiederum zu einem Bedarf an Mechanismen zur Konsistenzerhaltung solcher lokaler Daten gegenüber Zustandsänderungen auf den Basisdatenstrukturen, dem in Abschnitt 5.2.4 durch den Einsatz des Observer-Patterns entsprochen wird.

Verfeinerungsbäume unterstützen.
Die in Satz 5.3 formulierte Eigenschaft, daß Verfeinerungsbäume von Knoten in einem ALR-Graphen selbst wieder ALR-Graphen sind, ist entscheidend für die in Abschnitt 5.1.1 angesprochene Möglichkeit der Meta-Modellierung. Deshalb soll auch in der Implementierung auf den Verfeinerungsbaum eines Knotens zugegriffen werden können wie auf einen selbständigen ALR-Graphen.

Softwaretechnische Anforderungen

Flexibilität. Bereits bei den funktionalen Anforderungen haben wir festgestellt, daß die Forderung nach einer möglichst direkten Umsetzung der Theorie insbesondere bei den Graphstrukturen eine flexible Anpassungsmöglichkeit an die unzähligen Varianten der anwendungsbezogenen Repräsentation erfordert.

Modulare Erweiterbarkeit. Module, die Funktionalitäten auf den Basisstrukturen ergänzen, wollen wir ohne Änderungen an Basisdatentypen oder anderen Modulen hinzufügen können. Es gilt, die Schnittstellen zwischen den Systemteilen klein zu halten. Das Basissystem selbst sollte keinerlei Kenntnis von konkreten Anwendungsmodulen haben.

5.2.2 Anwendungsschnittstelle

Das Package `alr` stellt die Klassen für attributierte ALR-Graphen und Verfeinerungsbäume sowie für deren Knoten und Kanten zur Verfügung. Dabei benutzt es das Package `attribute`, das die gesamte Attributfunktionalität bereitstellt. Das Klassendiagramm in Abb. 5.2 gibt einen Überblick über die beiden Packages und ihre Beziehungen. `ALR_RefTree` bietet eine allgemeine Schnittstelle für Verfeinerungsbäume und repräsentiert deshalb nach Satz 5.3 sowohl

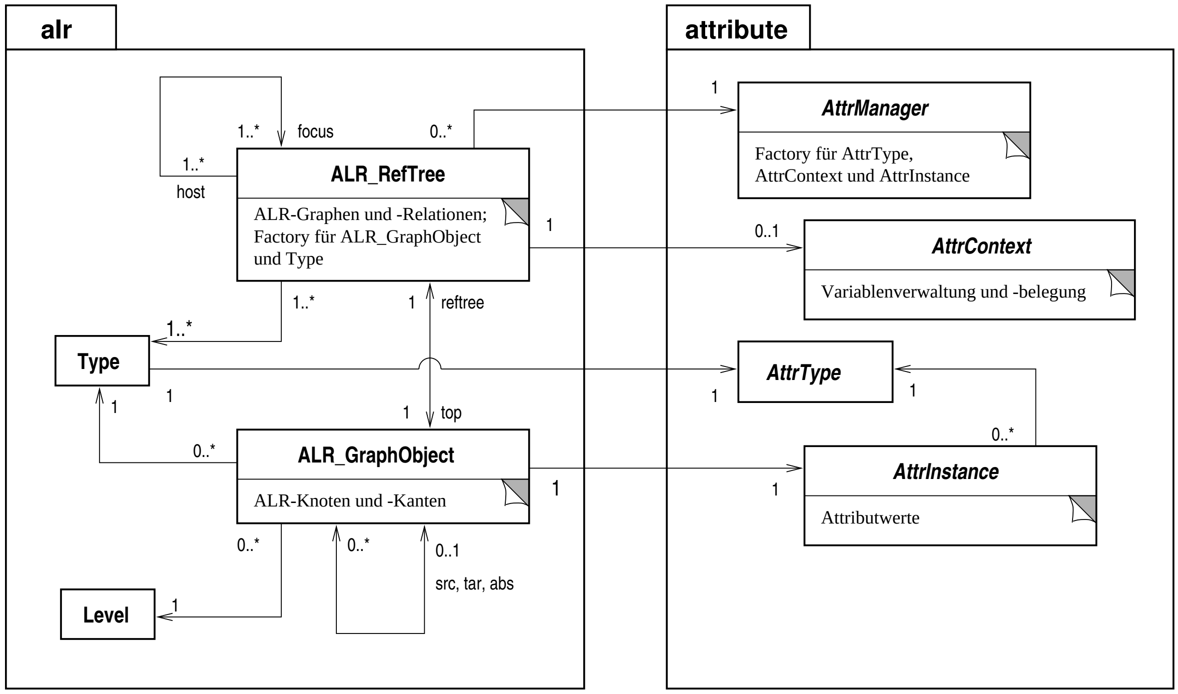


Abbildung 5.2: Übersichtsdiagramm zur Implementierung attributierter ALR-Strukturen.

ALR-Graphen als auch ALR-Relationen. Jedem Verfeinerungsbaum ist ein `AttrManager` zugeordnet, über den Attributtypen und -werte, aber auch Kontexte für Variablenbelegungen erzeugt werden können. Die Attributkontexte bekommen ihre Bedeutung aber erst im Zusammenhang mit Morphismen und Transformation und werden uns deshalb erst in Kapitel 6 beschäftigen. Jedem Verfeinerungsbaum ist außerdem genau ein `ALR_GraphObject` als Top-Objekt zugeordnet; die rekursiv gebildete Menge der Verfeinerungen dieses Objekts ergibt die Objektmenge des Verfeinerungsbaums. Die in der ALR-Graph-Definition implizit ausgedrückte Ebenenzuordnung eines Graphobjekts wird über die `Level`-Objekte explizit gemacht. Schließlich wird jedes Graphobjekt einem `Type` aus der Typmenge des Verfeinerungsbaums zugeordnet.

Wir beschreiben jetzt die Funktionalität der Klassen des `alr`-Packages anhand ihrer Interfaces.

`ALR_RefTree` (Abb. 5.3)

Die Prädikate `isGraph()` und `isRelation()` dienen zur Unterscheidung zwischen Verfeinerungsbäumen von Knoten, die ALR-Graphen repräsentieren (vgl. Def. 5.2), und Verfeinerungsbäumen von Kanten, die wir als Relationen zwischen Verfeinerungsbäumen auffassen. Mit `get`- und `setName()` kann ein Verfeinerungsbaum benannt werden.

`getTopObject()` liefert das Top-Objekt des Verfeinerungsbaums, auf dem die Methode aufgerufen wurde. Mit `getFocus()` können wir auf die Verfeinerung jedes Graphobjekts des aktuellen Verfeinerungsbaums über das `ALR_RefTree`-Interface zugreifen[1].

[1] Der Begriff „Fokus“ wurde in [Rud96] geprägt, als der Begriff des Verfeinerungsbaums noch nicht im Mittelpunkt stand, und kann in diesem Zusammenhang als synonym betrachtet werden.

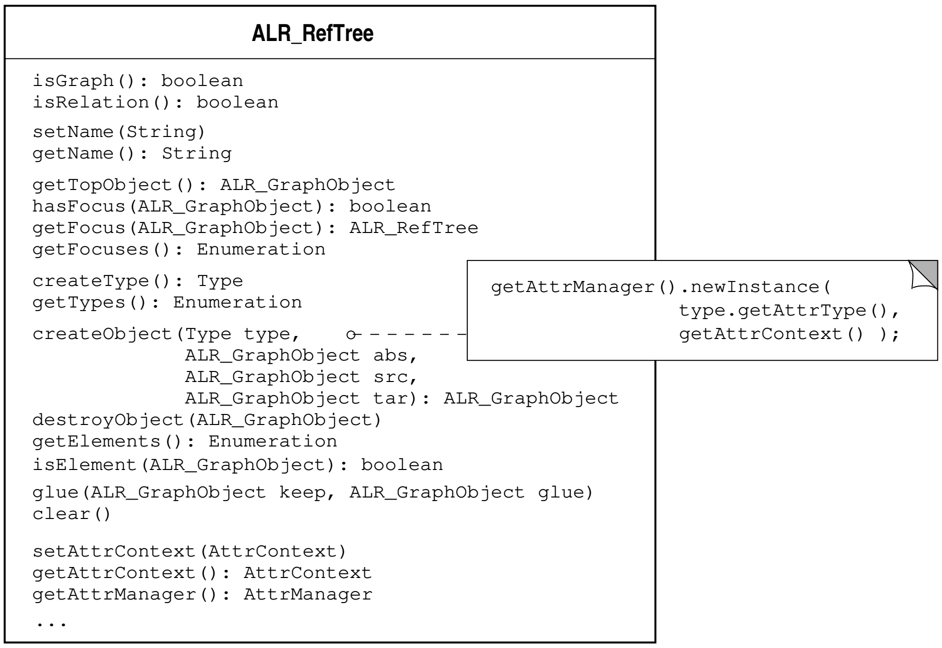


Abbildung 5.3: Das Interface der Klasse `ALR_RefTree`.

Aus Gründen der Speicherersparnis wird hier ein kleiner Trick angewandt: Im Gegensatz zur Theorie, wo zu jedem Graphobjekt `x` ein Verfeinerungsbaum *existiert*, wird hier eine neue Instanz von `ALR_RefTree` erst dann *erzeugt*, wenn zum erstenmal versucht wird, mit `getFocus(x)` darauf zuzugreifen. Bei allen folgenden Aufrufen wird dann lediglich die bereits existierende Instanz referenziert. Diese „lazy"-Instanziierung von Verfeinerungsbäumen ist für den Aufrufer von `getFocus()` transparent. Die Methoden `hasFocus()` und `getFocuses()` erlauben jedoch den Blick hinter die Kulissen: `hasFocus()` ermittelt, ob der Verfeinerungsbaum für ein bestimmtes Objekt bereits erzeugt wurde, `getFocuses()` liefert alle bereits erzeugten Verfeinerungsbäume von Objekten des aktuellen Verfeinerungsbaums. Diese Methoden sind in manchen Situationen aus Gründen der Speichereffizienz nützlich, um unnötige Instanziierungen von Verfeinerungsbäumen zu vermeiden. Für weitere Einzelheiten zum Fokus-Konzept siehe [Rud96], Def. 2.7 und Abschnitt 4.3.3[2].

`createType()` ist eine Factory-Method zum Erzeugen von `Type`-Objekten, die für die Typisierung von Graphobjekten verwendet werden. Automatisch wird für jeden neuen Typ auch eine Instanz von `AttrType` erzeugt, so daß eine 1:1-Beziehung zwischen den Instanzen dieser Klassen gewahrt bleibt. `getTypes()` liefert eine Aufzählung aller bereits vorhandenen Typen. Der Definition von Verfeinerungsbäumen folgend, haben alle Verfeinerungsbäume dieselbe Typmenge wie ihr „Host", also der ALR-Graph oder Verfeinerungsbaum, über dem sie definiert sind.

Mit der Factory-Method `createObject()` werden neue Knoten oder Kanten zur Objektmenge eines Verfeinerungsbaums hinzugefügt. Dabei muß für alle Abbildungen, die gemäß Definition auf Graphobjekten definiert sind, das Bild des neuen Objektes mit an-

[2]Das dortige Interface `BaseGraph` entspricht `ALR_RefTree`, und die beiden Implementierungen `SBaseConfig` und `SBaseFocus` wurden zur Implementierungsklasse `ALR_RefTreeImpl` zusammengefaßt.

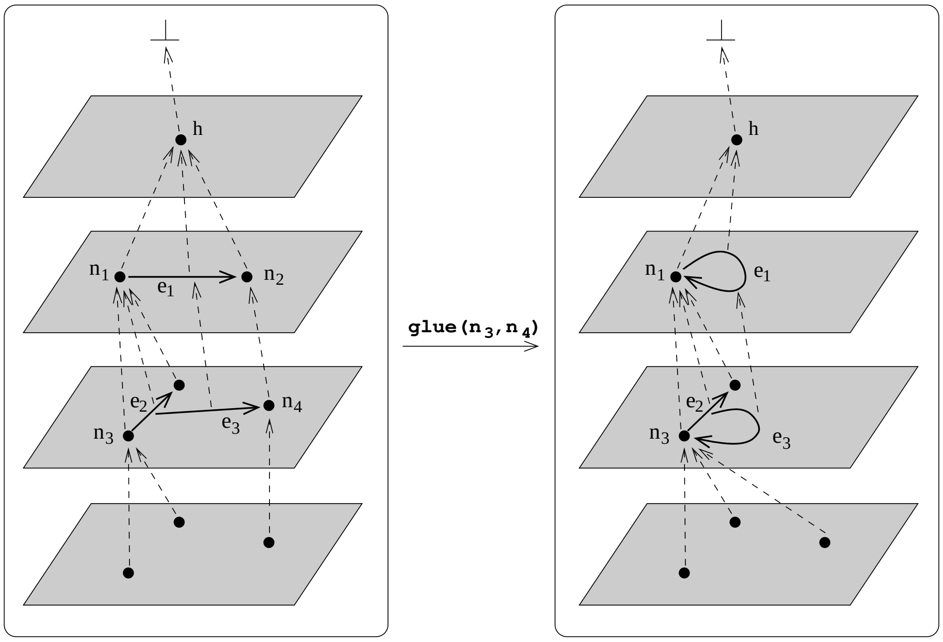


Abbildung 5.4: Beispiel für die Auswirkungen der `glue()`-Operation.

gegeben werden; ein nachträgliches Ändern dieser Abbildungsvorschriften ist aus Konsistenzgründen nicht möglich. Die angegebenen Bildwerte für die type-, abstraction-, source- und target-Abbildungen müssen eine Reihe von Vorbedingungen erfüllen, damit sie den Eigenschaften eines Verfeinerungsbaums genügen (vgl. Anhang (Schnittstellendoku)). Lediglich die Zuordnung des Attributwertes für das neue Objekt bildet eine Ausnahme: Es wird kein Wert übergeben, sondern automatisch ein leerer Standardwert vom Typ `AttrInstance` erzeugt. Abb. 5.3 zeigt den impliziten Aufruf der Methode `newInstance()` auf dem `AttrManager`. Das Interface `AttrInstance` bietet dann die Möglichkeit, einen Editor zur interaktiven Änderung des Attributwertes aufzurufen, oder aber den Wert im Rahmen einer Transformation aus den Werten bestimmter anderer Attribute zu berechnen (vgl. Kapitel 6).

Die Methode `destroyObject()` wird benutzt, um ein bestimmtes Objekt aus dem Verfeinerungsbaum zu löschen. Dabei werden alle Verfeinerungsobjekte und die ein- und ausgehenden Kanten rekursiv mitgelöscht, um die Konsistenz des Verfeinerungsbaums zu erhalten. Mit `getElements()` können alle Graphobjekte des aktuellen Verfeinerungsbaums aufgezählt werden, `isElement()` prüft die Zugehörigkeit eines Objekts. `clear()` löscht alle Graphobjekte bis auf das Top-Objekt des Verfeinerungsbaums.

Eine relativ komplexe Funktionalität hat die Methode `glue()`, die zwei Graphobjekte eines Verfeinerungsbaums „verklebt". Das bedeutet, die beiden Objekte werden zu einem einzigen verschmolzen, wobei der strukturelle Kontext beider Ausgangsobjekte auf das Ergebnisobjekt übergeht. Die Benennung der beiden formalen Parameter der Methode mit `keep` und `glue` deutet darauf hin, daß der Effekt der `glue()`-Operation nicht symmetrisch ist: Während alle source-, target- und abstraction-Operationen, die bisher auf das `glue`-Objekt zeigten, auf `keep` umgebogen werden und das `glue`-Objekt anschließend gelöscht wird, bleibt das `keep`-Objekt erhalten und repräsentiert nun die

Verklebung beider Objekte. Die Attributierung von `keep` bleibt erhalten, während die von `glue` verloren geht[3]. Wenn `keep` und `glue` unterschiedliche Abstraktionsobjekte haben, dann wird die `glue()`-Operation rekursiv auch auf diese Objekte angewandt, um eine wohldefinierte Abstraktionsabbildung für das Verklebungsergebnis zu erhalten. In Abb. 5.4 ist ein Beispiel für eine solche Kettenreaktion zu sehen. Natürlich gelten für die `glue()`-Operation eine Reihe von Vorbedingungen; so darf beispielsweise eine Kante nicht mit einem Knoten verklebt werden, und auch eine Verklebung über Ebenengrenzen hinweg ist nicht erlaubt. Außerdem müssen die zu verklebenden Objekte typgleich sein. Benötigt wird die Funktionalität der `glue()`-Operation, um bei der *In-place*-Graphtransformation (vgl. Kapitel 6) den Effekt von nicht-injektiven Regeln zu erzeugen. Die Semantik der Attributverklebung innerhalb von `glue()` ist für diesen Anwendungsfall irrelevant, da die Attributwerte des Verklebungsergebnisses allein durch die Regel bestimmt werden (zu Konfliktfällen und deren Lösung durch Vereinbarungen vgl. [Mel97], Abschnitt 3.1.3).

Instanzen von `ALR_RefTree` werden zur Modellierung von ALR-Graphen und -Morphismen benutzt. Zur Laufzeit übernehmen sie im Graphtransformationssystem so unterschiedliche Funktionen wie die einer Graphgrammatik, einer Regel bzw. einer Regelseite, eines Arbeitsgraphen oder eines Ansatzmorphismus (vgl. Abschnitt 8.1). Je nach Funktion werden aber auch verschiedene Attributierungsmöglichkeiten unterschieden: Variablen etwa sind nur in Regelgraphen zugelassen, nicht jedoch in Arbeitsgraphen, und Ansatzmorphismen haben gegenüber anderen Morphismen die besondere Eigenschaft, daß durch sie die Variablen der linken Regelseite mit Werten aus dem Arbeitsgraphen belegt werden. Die Attributkomponente muß deshalb wissen, in welchem funktionalen Kontext jeder Attributwert zu betrachten ist, und dazu dient der `AttrContext`. So gibt es spezielle Attributkontexte für linke und rechte Regelseiten sowie für Ansatzmorphismen. Diese Kontexte werden von einem `AttrManager` erzeugt, auf den mit der `getAttrManager()`-Methode eines Verfeinerungsbaums zugegriffen werden kann. Wird nun also zur Laufzeit beispielsweise eine Instanz von `ALR_RefTree` erzeugt, die die linke Seite einer Regel repräsentieren soll, dann wird über den `AttrManager` ein spezieller `AttrContext` für linke Regelseiten erzeugt und anschließend mit der Methode `setAttrContext()` als Attributkontext des Verfeinerungsbaums gesetzt. Der Code-Ausschnitt in Abb. 5.3 zeigt, wie in der Folge alle neu erzeugten Attributinstanzen in Beziehung zum aktuellen Attributkontext gesetzt werden.

`ALR_GraphObject` (Abb. 5.5)

Die Methoden `isNode()` und `isArc()` dienen zur Unterscheidung von Knoten und Kanten. Mit `getType()`, `getAttribute()` und `getLevel()` kann auf den Typ und den Attributwert eines Objekts bzw. auf die Ebene zugegriffen werden, auf der sich das Objekt befindet. `getSource()`, `getTarget()` und `getAbstraction()` liefern jeweils das Bild eines Objekts unter den drei Abbildungen s, t und a aus der Definition von ALR-Graphen. Zusätzlich repräsentieren wir mit den Methoden `getOutgoingArcs()`, `getIncomingArcs()` und `getRefinementObjects()` auch die Umkehrrelationen dieser drei Abbildungen. Der zusätzliche Parameter `top` bezeichnet das Top-Objekt des Verfeinerungsbaums, in dessen Kontext die ein- und ausgehenden Kanten betrachtet werden sollen; nur so können wir die Abgeschlossenheit der Umkehrrelationen in der Objektmen-

[3]Eine entsprechende `glue()`-Operation auf Attributen ist geplant.

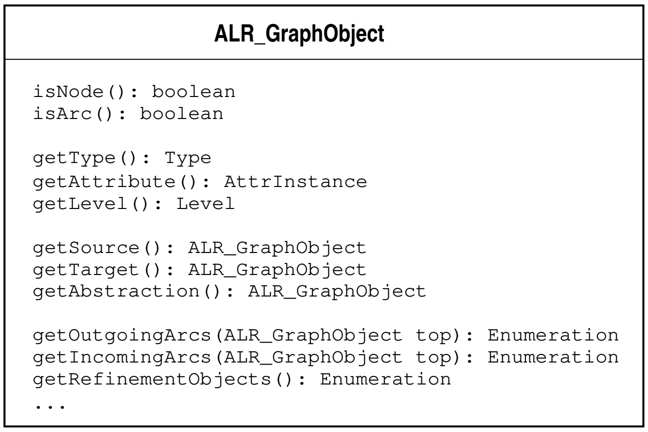


Abbildung 5.5: Das Interface der Klasse `ALR_GraphObject`.

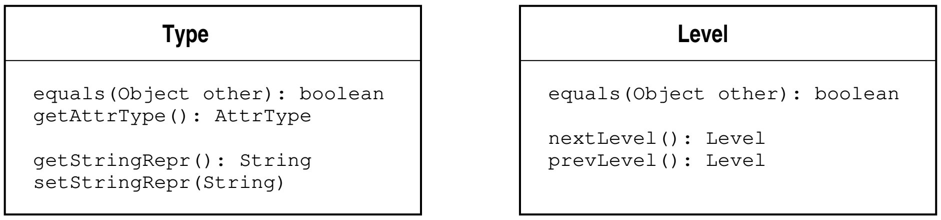


Abbildung 5.6: Die Interfaces der Klassen `Level` und `Type`.

ge des betrachteten Verfeinerungsbaums sicherstellen. Ein Beispiel anhand von Abb. 5.1 auf Seite 36: Wenn wir im Kontext des Verfeinerungsbaums von n_1 nach den ausgehenden Kanten von e_2 fragen, so ist e_3 keine gültige Antwort, weil dieses Objekt nicht zum gefragten Verfeinerungsbaum gehört. Wohl aber ist e_3 eine gültige Antwort im Kontext des Verfeinerungsbaums von h.

`Type` (Abb. 5.6)
: Seine wichtigste Methode erbt `Type` von `Object`: Mit `equals()` können wir vergleichen, ob zwei Typen gleich sind. Dies wird beispielsweise benötigt, um die Typverträglichkeit eines Morphismus zu prüfen. `getAttrType()` liefert den Typ der Attribute, die den Graphobjekten dieses Typs zugeordnet werden können. `get-` und `setStringRepr()` schließlich dienen dazu, einem Typen einen Namen zuzuweisen, der für die Repräsentation z.B. in einem Grapheditor verwendet werden kann. Dazu ist anzumerken, daß die Identität eines Typs nicht auf seinem Namen beruht, d.h., zwei Typen mit demselben Namen sind nicht notwendigerweise identisch.

`Level` (Abb. 5.6)
: Wie schon für `Type` gilt auch für `Level`, daß die entscheidende Funktionalität dieses Interfaces durch die Methode `equals()` realisiert wird. Hiermit kann festgestellt werden, ob sich zwei Graphobjekte auf derselben Ebene befinden oder nicht. Die Methoden `nextLevel()` und `prevLevel()` ermöglichen es, auf die direkten Nachbarn der aktuellen Ebene in der Ebenenhierarchie zuzugreifen.

5.2.3 Beziehungen zur Theorie

In der folgenden Tabelle stellen wir die Begriffe aus der Definition eines ALR-Verfeinerungsbaums nach Definition 5.5 (S. 37) ihren Entsprechungen in der Implementierung gegenüber. Sei $G_{AS} = (G, A, at, av)$ mit $G = (O, \perp, h, a, s, t, L, \ell)$ ein attributierter Verfeinerungsbaum. Dann gelten die folgenden Entsprechungen:

Theorie	Implementierung
G_{AS}	`ALR_RefTree g`
$x \in O$	`ALR_GraphObject x` mit `g.isElement(x)`
$m \in L$	`Type m` $\in$ `g.getTypes()`
O	`g.getElements()`
$\perp$	`null`
h	`g.getTopObject()`
$a(x)$	`x.getAbstraction()`
$s(x)$	`x.getSource()`
$t(x)$	`x.getTarget()`
L	`g.getTypes()`
$\ell(x)$	`x.getType()`
$AS = (S, OP)$	gegeben durch Java-Syntax
S	beliebige Tupel von Java-Typen, repräsentiert durch Instanzen von `AttrType`
OP	Java-Operationen und -Methoden (Syntax)
A	gegeben durch Java-Semantik
AV	beliebige Tupel von Java-Expressions oder -Werten, repräsentiert durch die Menge aller möglichen Instanzen von `AttrInstance`
$at(m)$	`m.getAttrType()`
$av(x)$	`x.getAttribute()`

Hierbei fällt auf, daß das $\perp$-Objekt aus der Definition in der Implementierung nicht als Instanz von `ALR_GraphObject` umgesetzt wird. Weil $\perp$ auch in der Theorie als Objekt keine Rolle spielt, sondern lediglich zur Totalisierung der Abbildungen verwendet wird, reicht es in der Implementierung aus, die Konstante `null` für eine undefinierte Referenz zu verwenden. Die Entsprechungen der Attributsignatur und -algebra sind abhängig vom konkreten Attribut-Handler; die Angaben in der Tabelle beziehen sich auf den in [Mel97] beschriebenen Java-Expression-Handler. Für die konkrete Attributalgebra kommen trotz festgelegtem Handler noch verschiedene Varianten in betracht, vgl. Abschnitt 6.2.3 (S. 66).

Auch das Diagramm aus Definition 5.4, das besagt, daß Graphobjekte desselben Typs auch Attributwerte desselben Attributtyps tragen müssen, können wir in der Terminologie der Implementierung ausdrücken:

$$\begin{tikzcd}
\texttt{Type} \arrow[r, "\texttt{getAttrType()}"] & \texttt{AttrType} \\
\texttt{ALR_GraphObject} \arrow[u, "\texttt{getType()}"] \arrow[r, "\texttt{getAttribute()}"'] \arrow[ur, phantom, "(=)"] & \texttt{AttrInstance} \arrow[u, "\texttt{getType()}"']
\end{tikzcd}$$

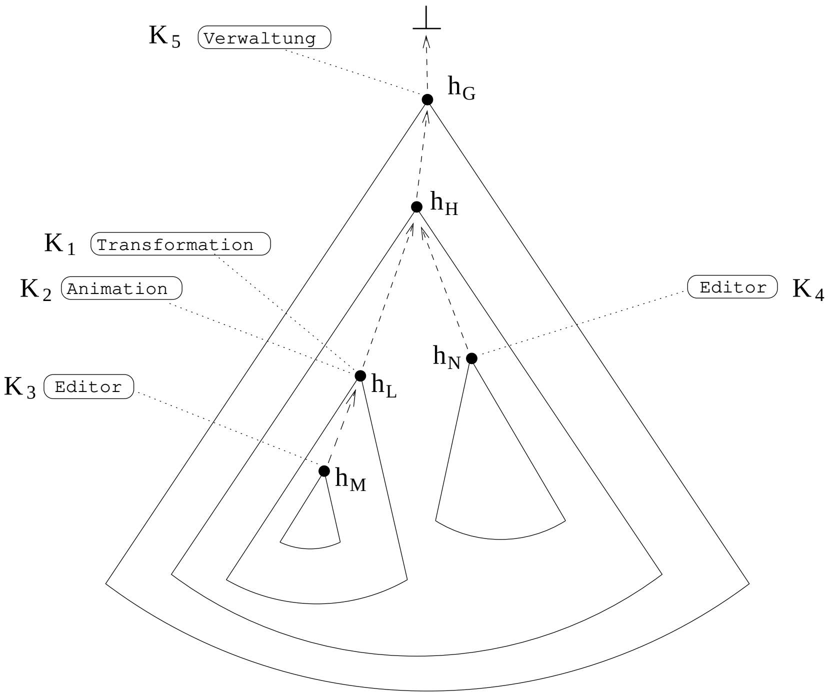


Abbildung 5.7: Automatische Konsistenzerhaltung von Erweiterungskomponenten.

Die Eckpunkte dieses Diagramms sind auch gut im Klassendiagramm aus Abb. 5.2 zu erkennen. Hier sind für die Assoziation zwischen `Type` und `AttrType` bzw. `ALR_GraphObject` und `AttrInstance` jeweils 1:1-Beziehungen angegeben, während in der Theorie auch nicht-injektive Abbildungen zugelassen sind. Diese Diskrepanz erklärt sich dadurch, daß in der Implementierung verschiedene Instanzen der Klassen `AttrType` bzw. `AttrInstance` kongruente Attributtypen bzw. -werte repräsentieren können, die in der Mengendarstellung der Theorie zusammenfallen.

5.2.4 Änderungsinformationen

`ALR_RefTree` ist eine Subklasse von `ExtObservable` (vgl. Abschnitt 3.2) und erlaubt als solche beliebig vielen Erweiterungskomponenten, sich als Observer anzumelden und fortan automatisch über alle Zustandsänderungen des Verfeinerungsbaums informiert zu werden. Durch die Anwendung des Observer-Patterns wird es beispielsweise möglich, einen Arbeitsgraphen zu transformieren und gleichzeitig die Auswirkungen der Transformation in einer Animation zu visualisieren, ohne daß Transformations- und Animationskomponente in irgendeiner Weise gekoppelt wären. Abbildung 5.7 zeigt ein Beispiel für eine mögliche Laufzeitsituation, in der verschiedene Anwendungskomponenten auf verschiedenen Verfeinerungsbäumen eines ALR-Graphen arbeiten. Komponenten wie Editoren (K_3, K_4) oder Animation (K_2), die mit Layout und Repräsentation von Graphen zu tun haben, werden sich typischerweise als Observer bei ihren jeweiligen Verfeinerungsbäumen anmelden; denn sie beobachten ihr Observationsobjekt gewöhnlich über einen längeren Zeitraum und müssen ihre lokalen repräsentationsbezogenen Daten konsistent halten mit eventuellen Veränderungen auf dem zugrundeliegenden Verfeinerungsbaum. Andere Komponenten dagegen, wie etwa die Transformationskomponente oder auch die Ansatzsuche, gehen für den kurzen Zeitraum ihrer Aktivität von einem exklusiven Schreibrecht auf dem Verfeinerungsbaum aus, auf dem sie gerade operieren, und haben deshalb kein Problem mit der Konsistenzerhaltung lokaler Daten.

Wichtig ist die Feststellung, daß zwischen geschachtelten Verfeinerungsbäumen wie etwa h_L und h_M Zustandsabhängigkeiten bestehen, die in der Teilmengenbeziehung ihrer Objektmengen begründet sind. Wird etwa mit dem Editor K_3 ein Objekt im Verfeinerungsbaum von h_M erzeugt, so gehört dieses Objekt auch zu dem ALR-Graphen mit dem Top-Objekt h_L. Operationen im Editor K_4 haben auf den Verfeinerungsbaum h_L dagegen grundsätzlich keinen Einfluß.

Als eine weitere mögliche Erweiterungskomponente, die sinnvollerweise als Observer zu implementieren wäre, zeigt Abb. 5.7 eine Verwaltungskomponente. Die Idee ist hier, daß ein Observer, der auf den obersten aller Verfeinerungsbäume angesetzt wird, über alle Zustandsänderungen auch der darunterliegenden Verfeinerungsbäume informiert wird. Eine solche Verwaltungskomponente könnte dann z.B. alle Zustandsänderungen protokollieren, um diese statistisch auszuwerten, oder aber um darüber eine „Undo"-Funktionalität zu implementieren. Durch die Möglichkeit der Meta-Modellierung könnte der ALR-Graph G mit dem Top-Objekt h_G auch ein ganze Graphgrammatik repräsentieren, und die Verwaltungskomponente könnte dann die Namen aller Regeln und Graphen aufsammeln, um sie über eine graphische Benutzeroberfläche zur Auswahl für nachfolgende Operationen bereitzustellen.

Das Prinzip des Observer-Patterns, das wir für die Kommunikation zwischen Basis und Erweiterungskomponenten einsetzen, haben wir bereits in Abschnitt 3.2 kennengelernt. In jeder konkreten Anwendungssituation dieses Patterns muß jedoch bezüglich des Umfangs der Änderungsinformationen, die das Observable an seine Observer verschickt, ein angemessener Kompromiß zwischen den Extremen des Push-Modells auf der einen und des Pull-Modells auf der anderen Seite gefunden werden. In unserem Fall ist klar, daß zumindest so viel Information über eine konkrete Zustandsänderung an die Observer weitergegeben werden muß, daß deren Synchronisation mit dem veränderten Zustand des Observables im Normalfall in konstanter Zeit, also unabhängig von der Größe des Verfeinerungsbaums erfolgen kann. Anderenfalls würde die Antwortzeit von modifizierenden Operationen auf den Basisstrukturen durch die anschließende Synchronisation der Observer stark beeinträchtigt. Ein Zuviel an Informationen, das von den Erweiterungskomponenten möglicherweise gar nicht genutzt wird, kann sich jedoch ebenfalls negativ auf die Performance auswirken. Abbildung 5.8 zeigt anhand eines Klassendiagramms die konkrete Anwendung des Observer-Patterns, wobei `ALR_RefTree` die Rolle des `Observables` übernimmt. Zu erkennen sind auch die sechs Subklassen von `Change`, die unsere anwendungsspezifischen Änderungsinformationen repräsentieren. Jede dieser Änderungsinformationen hat zwei Aspekte:

- Die *Art* der Zustandsänderung auf dem Observable wird beschrieben durch den Namen der konkreten Klasse. Beispiel: `ALR_Change_ObjectCreated` $\rightsquigarrow$ ein neues Graphobjekt wurde erzeugt.

- Das von der Änderung betroffene Objekt, das als *Item* bezeichnet wird. Auf das Item einer Änderungsinformation wird über die Methode `getItem()` zugegriffen, die in der allgemeinen Klasse `Change` definiert wird, weshalb der Rückgabewert auch ganz allgemein als `Object` typisiert ist. Jede Spezialisierung von `Change` muß spezifizieren, welchen konkreten Typ ihr Item hat. Beispiel: Eine Instanz von `ALR_Change_ObjectCreated` re-

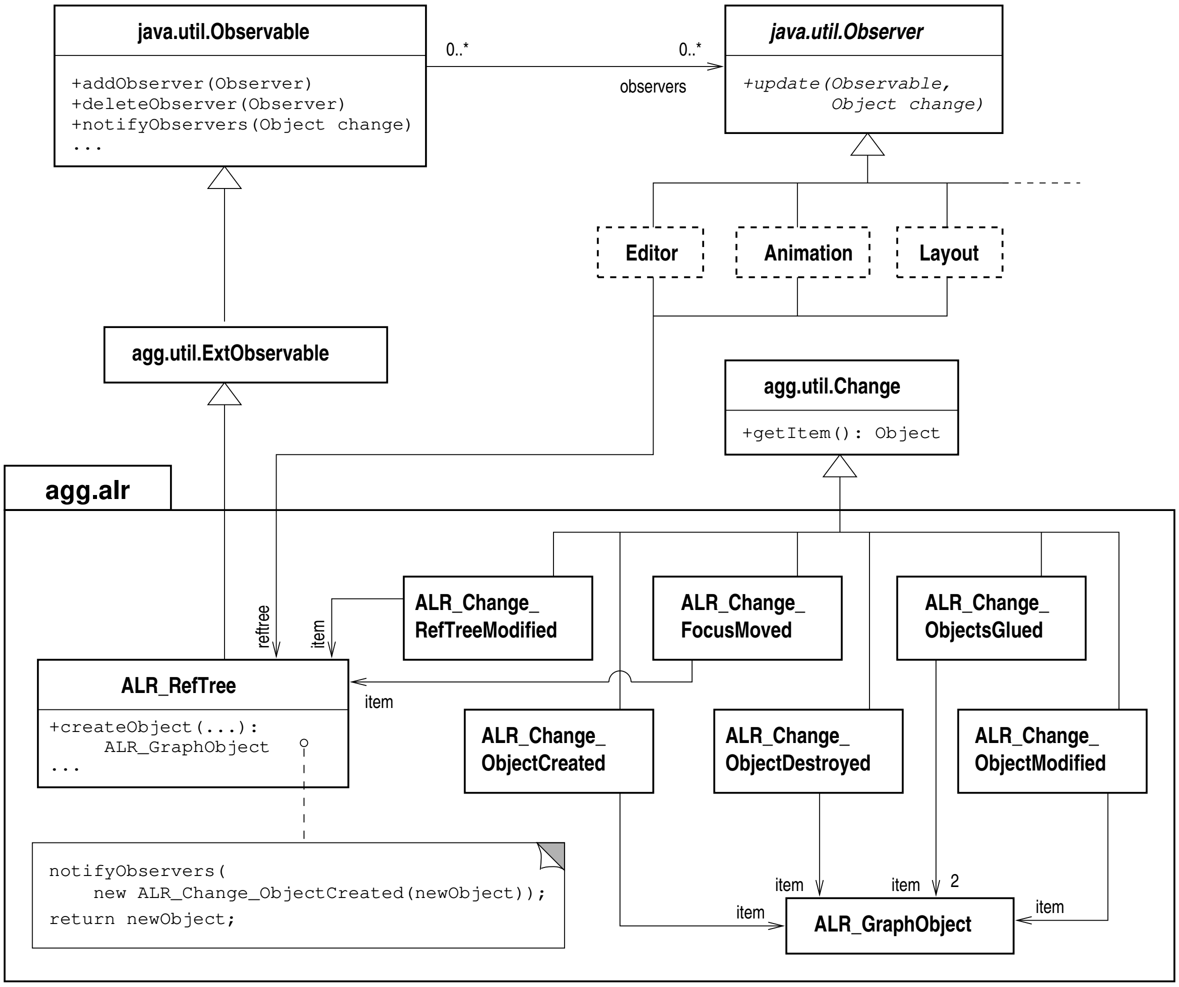


Abbildung 5.8: Anwendung des Observer-Patterns mit Änderungsinformationen.

Change	Item: Itemtyp	Bedeutung
ALR_Change_ ObjectCreated	x: ALR_GraphObject	x wurde erzeugt.
ALR_Change_ ObjectDestroyed	x: ALR_GraphObject	x wurde gelöscht.
ALR_Change_ ObjectModified	x: ALR_GraphObject	Die Attributierung von x wurde modifiziert.
ALR_Change_ ObjectsGlued	(x,y): Pair (von ALR_GraphObject)	x und y wurden durch einen Aufruf der Form glue(x,y) verklebt.
ALR_Change_ FocusMoved	h: ALR_RefTree	Das Top-Objekt u von h wurde durch einen Aufruf glue(u,t) mit dem Top-Objekt t von g verklebt.
ALR_Change_ RefTreeModified	g: ALR_RefTree	Der Name von g wurde geändert.
Change_ ObservableGone	g: ALR_RefTree	g wurde gelöscht (vgl. Abschnitt 3.2).
null	–	Undefinierte Zustandsänderung. Vom Observer wird eine vollständige Resynchronisation mit dem Zustand des Observables erwartet.

Tabelle 5.1: Semantik der Änderungsinformationen, die eine Instanz g von ALR_RefTree an ihre Observer schicken kann.

ferenziert als Item genau das Graphobjekt, das gerade erzeugt worden ist. Dieses Item hat den Typ ALR_GraphObject.

Der Code-Ausschnitt aus der Methode createObject() von ALR_RefTree zeigt, wie eine der Zustandsänderung entsprechende Änderungsinformation instanziiert, mit dem passenden Item parametrisiert und anschließend über den Aufruf von notifyObservers() an alle Observer verschickt wird. Im Klassendiagramm sind in den gestrichelten Kästen einige hypothetische Beispiele für Erweiterungskomponenten angegeben, die typischerweise als Observer zu implementieren wären.

Tabelle 5.1 beschreibt die Semantik der einzelnen Änderungsinformationen aus der Sicht eines Observers, der bei einer Instanz g von ALR_RefTree angemeldet ist. Der Observer erhält grundsätzlich nur solche Informationen, die den von ihm observierten Verfeinerungsbaum betreffen. Wenn also eine Änderungsinformation eintrifft, deren Item ein Graphobjekt ist, kann der Observer davon ausgehen, daß dieses Objekt ein Element seiner Objektmenge ist, oder – im Fall von ALR_Change_ObjectDestroyed – zumindest war. Die Änderungsinformationen beschreiben atomare Zustandsmodifikationen eines Verfeinerungsbaums. Ein Aufruf von clear() auf einer Instanz von ALR_RefTree generiert deshalb für jedes einzelne Graphobjekt eine ALR_Change_ObjectDestroyed-Nachricht. Auch destroyObject() verschickt für möglicherweise rekursiv gelöschte Objekte eigene Änderungsinformationen. Dieses Prinzip ist wichitg für die Implementierung der Reaktionen des Observers auf die einzelnen Änderungsinformationen in dessen update()-Methode. Weil die Informationen atomar und vollständig sind, kann eine aufwendige Analyse des Observable-Zustands auf Ursache und mögliche Folgewirkungen an dieser Stelle unterbleiben.

Beispiel 5.6 (*Aufruf von* `glue()` *und die ausgelösten Änderungsinformationen*)

Die kompliziertesten Reaktionen in bezug auf die Änderungsinformationen löst die Operation `glue()` aus. Deshalb betrachten wir anhand des Sequenzdiagramms in Abb. 5.9 einen Aufruf von `glue()` in einer Objektkonfiguration, die im wesentlichen der Abb. 5.7 entspricht, aber auf die drei Observer K_2, K_4 und K_5 eingeschränkt wird, von deren Anwendungsfunktionalität wir außerdem abstrahieren. Die Verfeinerungsbäume mit den Top-Objekten h_L, h_N und h_G bezeichnen wir entsprechend mit L, N und G. Wir beginnen die Aufrufsequenz mit dem Aufruf von `G.glue(`h_L, h_N`)`. Gemäß der asymmetrischen Funktionalität von `glue()` ist h_L das Objekt, auf das der strukturelle Kontext des anderen Objekts übertragen wird und das demnach die Verklebung beider Objekte repräsentiert, während h_N später gelöscht wird. Wir nennen deshalb das zu erhaltende Objekt `keep` und das nach der Verklebung zu löschende `glue`. Nachdem wir den Kontext des `glue`-Objekts auf `keep` übertragen haben, erzeugen wir eine Änderungsinformation `ALR_Change_ObjectsGlued` mit den beiden zu verklebenden Objekten als Parameter. Diese Änderungsinformation `change` soll nun an alle Observer verteilt werden, die von der Verklebung betroffen sind. Wir abstrahieren vom konkreten Verteilungsmechanismus, der über eine spezielle Infrastruktur zwischen den Graphobjekten implementiert wird (vgl. [Rud96], S. 41). Stattdessen zeigt Abb. 5.9 ein Pseudo-Objekt mit der Bezeichnung „change information propagation“, das mit der ensprechenden Pseudo-Methode `propagate()` aktiviert wird und die Verteilung der ihm übergebenen Änderungsinformationen an die betroffenen Observer organisiert. In diesem Fall wird durch Aufrufe der `notifyObservers()`-Methoden von L und G die Benachrichtigung aller Observer dieser Verfeinerungsbäume über die Verklebung der Graphobjekte h_L und h_N veranlaßt. Da in unserem Beispiel sowohl L als auch G nur jeweils einen Observer haben, erfolgt innerhalb von `notifyObservers()` auch nur ein einziger Aufruf der `update()`-Methode dieses Observers. Als typisches Beispiel für die durch ein `update()` ausgelöste Observer-Aktivität ist im Sequenzdiagramm „update display“ angegeben; eine Animationskomponente würde etwa die beiden betroffenen Graphobjekte aufeinander zu bewegen und verschmelzen. Auffällig ist, daß die Observer von N nicht benachrichtigt werden. Das liegt daran, daß das Zielobjekt der Verklebung `keep`$= h_L$ nicht zum Verfeinerungsbaum N gehört.

Nachdem die Verteilung der `ALR_Change_ObjectsGlued`-Information mit der Rückkehr der `propagate()`-Methode beendet ist, wird eine weitere Änderungsinformation erzeugt: `ALR_Change_FocusMoved` wird an alle Observer des Verfeinerungsbaums N verteilt, dessen Top-Objekt `glue`$= h_N$ im Verlauf der Verklebung gelöscht wird. Nach der Verklebung gehören alle Objekte, die sich vorher im Verfeinerungsbaum von h_N befanden, zum Verfeinerungsbaum L; L repräsentiert also genau so die Verklebung der Verfeinerungsbäume L und N, wie das Graphobjekt h_L die Verklebung von h_L und h_N repräsentiert. Die Änderungsinformation `ALR_Change_FocusMoved` veranlaßt deshalb die Observer von N – in unserem Beispiel ist das nur K_4 –, sich als Observer bei N ab- und dafür bei L anzumelden. Die Referenz auf das neue Observable L liefert `ALR_Change_FocusMoved` dabei als Item mit. Wichtig ist, daß K_4 bei dieser Gelegenheit alle seine Referenzen auf sein bisheriges Observable N löscht, damit dieses später von der Garbage-Collection erfaßt werden kann. Da in dem neuen Verfeinerungsbaum L, bei dem sich K_4 angemeldet hat, gegenüber seinem bisherigen Observable N viele neue Graphobjekte existieren, für die keine gesonderten Änderungsin-

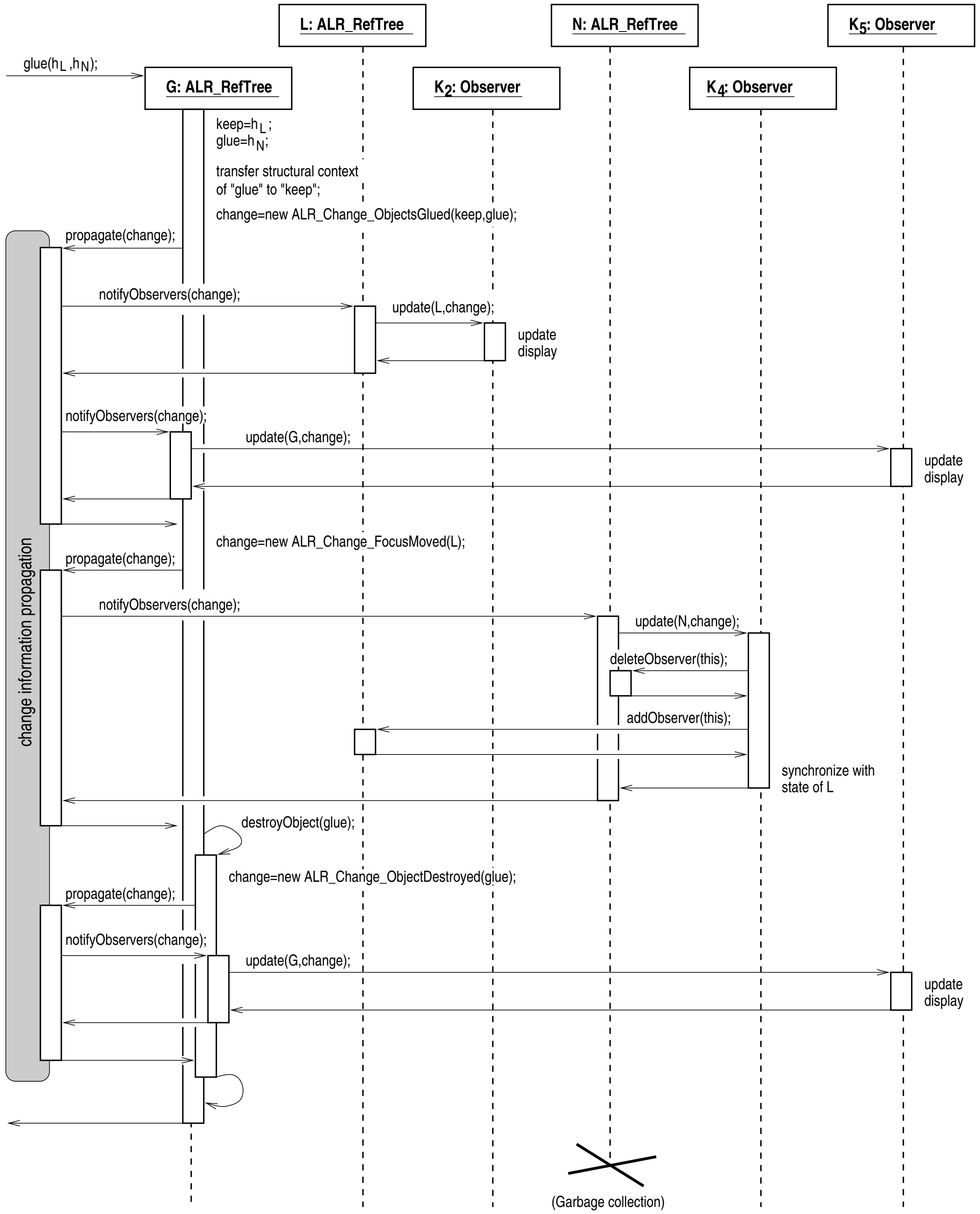


Abbildung 5.9: Beispiel für eine von `glue()` ausgelöste Update-Sequenz.

formationen verschickt werden, muß der Observer sich mit dem Zustand von L vollständig neu synchronisieren. Neben der undefinierten Änderungsinformation `null` (vgl. Tab. 5.1) ist `ALR_Change_FocusMoved` damit die einzige, die eine Resynchronisation mit linearem Aufwand erfordert. Außerdem ist `ALR_Change_FocusMoved` die einzige der im `alr`-Package definierten Änderungsinformationen, die nur *lokal* (zur Terminologie s. auch [Rud96]) verteilt wird, also nur an die Observer eines bestimmten Verfeinerungsbaums und nicht automatisch auch an alle umfassenden Verfeinerungsbäume. So erhält in unserem Beispiel auch K_5 keine entsprechende Information, denn diese hätte für ihn keinerlei Bedeutung, die über die bereits erhaltene `ALR_Change_ObjectsGlued`-Nachricht hinausginge.

Nachdem das Graphobjekt `glue`$= h_N$ bereits zu Beginn von seinem Kontext isoliert wurde und nun auch die Observer des zugehörigen Verfeinerungsbaums umgeleitet worden sind, wird h_N im letzten Schritt der `glue()`-Operation mit einem Aufruf von `destroyObject()` gelöscht. Dabei wird eine Änderungsinformation vom Typ `ALR_Change_ObjectDestroyed` erzeugt und mittels `propagate()` an die betroffenen Observer verschickt. Betroffen sind aber nur noch die Observer von G, da H keine Observer hat und auch die Observer von N sich bereits abgemeldet haben. Nach der Rückkehr von `propagate()` terminiert `destroyObject()` und damit auch `glue()`, und der Kontrollfluß kehrt zum Aufrufer zurück. Asynchron zum Kontrollfluß wird die Garbage-Collection später den von N und h_N belegten Speicher freigeben. $\triangle$

Kapitel 6

Transformation

Dieses Kapitel gilt dem Entwurf einer Transformationskomponente für das AGG-System. Wir wollen attributierte ALR-Graphen, wie wir sie in Kapitel 5 beschrieben haben, regelbasiert transformieren. Dazu führen wir zunächst den Begriff des *ALR-Morphismus* ein, der als Verfeinerungsbaum einer Kante in einem ALR-Graphen implementiert wird. Für die Berechnung des eigentlichen Transformationsschritts bedienen wir uns dann der *Colimit-Library* [Wol97]. Diese Bibliothek, die u.a. auch in einer Java-Implementierung vorliegt, berechnet allgemeine Colimiten auf sog. ALPHA-Algebren. Wir untersuchen deshalb, auf welche Weise sich ALR-Graphen als ALPHA-Algebren darstellen lassen, und warum das in ALPHA-Darstellung errechnete Ergebnis auch dem gewünschten Pushout auf ALR-Graphen entspricht. Die Definitionen der benötigten kategoriellen Grundbegriffe werden in einem gesonderten Abschnitt angegeben.

6.1 Begriffe und Konzepte

Das AGG-System transformiert ALR-Graphen nach dem *Single-Pushout*-Verfahren [Löw93, Roz97]: Ein direkter Transformationsschritt wird durch ein Pushout in der Kategorie der ALR-Graphen mit partiellen Morphismen beschrieben. In Abschnitt 6.1.1 geben wir die Definition der zentralen Begriffe *Kategorie* und *Pushout* wieder. Anschließend definieren wir den Begriff des *ALR-Morphismus* und betrachten eine Modellierung der Morphismen durch Verfeinerungsbäume von Kanten (vgl. Def. 5.2, S. 35). Abschnitt 6.1.4 schließlich dient der theoretischen Fundierung der Transformation von ALR-Graphen als ALPHA-Algebren in der Colimit-Bibliothek.

6.1.1 Kategorien und Pushouts

Der größte Teil der theoretischen Ergebnisse im Gebiet der algebraischen Graphtransformation geht auf die kategorientheoretische Beschreibung eines Graphtransformationsschrittes als Pushout in einer geeigneten Kategorie zurück. Da wir im Rahmen der theoretischen Fundierung der Transformationskomponente für das AGG-System die Begriffe „Kategorie“ und „Pushout“ noch häufiger gebrauchen werden, wollen wir sie an dieser Stelle kurz erläutern. Ausführliche Einführungen in die Kategorientheorie geben z.B. [Wal91] und [AM75].

Die Kategorientheorie ist ein abstraktes mathematisches Kalkül, das ähnlich der Mengentheorie oder der universellen Algebra das Beschreiben und Beweisen von Eigenschaften verschiedenster Strukturen in der Mathematik und Informatik erlaubt. Dabei begibt sich die Kategorientheorie auf ein nochmals erhöhtes Abstraktionsniveau, indem sie ihren jeweiligen Gegenstand nicht durch eine Auflösung in dessen Bestandteile zu beschreiben versucht, sondern ausschließlich über seine Beziehungen zu anderen Objekten des Gegenstandsbereichs. Damit haben wir schon die zentralen Begriffe der Kategorientheorie angesprochen: Die Gegenstände heißen *Objekte*, die Beziehungen werden durch *Morphismen* beschrieben, und eine *Kategorie* entspricht einem bestimmten Gegenstandsbereich.

Definition 6.1 (*Kategorie*) Eine Kategorie $\underline{\mathrm{KAT}}$ besteht aus

- einer Klasse von Objekten K,
- einer Menge M von Morphismen, wobei jeder Morphismus $f \in M$ eine ausgezeichnete Quelle A und ein ausgezeichnetes Ziel B aus K hat, geschrieben $f : A \to B$,
- einer Kompositionsoperation „$\circ$“, die aus zwei Morphismen $f : A \to B$ und $g : B \to C$ mit $A, B, C \in K$ einen Morphismus $(g \circ f : A \to C) \in M$ konstruiert und
- einem ausgezeichneten Morphismus $id_A : A \to A$ für jedes Objekt $A \in K$.

Darüber hinaus gelten die beiden folgenden Axiome:

1. *(Assoziativität der Komposition.)* Für alle $f : A \to B$, $g : B \to C$, $h : C \to D$ mit $f, g, h \in M$ gilt $h \circ (g \circ f) = (h \circ g) \circ f$.
2. *(Neutralität der Identität.)* Für alle $f \in M$ mit $f : A \to B$ gilt $id_B \circ f = f = f \circ id_A$.

Objekte einer Kategorie mit ihren Morphismusbeziehungen lassen sich gut durch Graphen veranschaulichen: Die Objekte werden als Knoten, die Morphismen als Kanten eines Graphen gezeichnet. Einen solchen Graphen nennen wir auch *Diagramm*. △

Eine einfache Kategorie, die gut geeignet ist, um eine Intuition für die Funktionsweise der kategorientheoretischen Begriffe zu entwickeln, ist die Kategorie $\underline{\mathrm{SET}}$ der Klasse aller Mengen mit den totalen Abbildungen zwischen Mengen als Morphismen. Aber auch die Klasse aller Mengen mit partiellen Abbildungen ergibt eine Kategorie $\underline{\mathrm{SET}^{\mathrm{P}}}$, was leicht nachzuweisen ist. Im Zusammenhang mit algebraischer Graphtransformation ist natürlich besonders interessant, daß die einfachen Graphen nach Def. 2.1 mit partiellen Graphmorphismen ebenfalls eine Kategorie bilden, ebenso wie ALR-Graphen und ALR-Morphismen [Roz97, Löw93, AR89].

Definition 6.2 (*Unterkategorie*) Eine Kategorie $\underline{\mathrm{KIT}}$ mit einer Objektklasse K_1 und einer Morphismenmenge M_1 ist Unterkategorie von einer Kategorie $\underline{\mathrm{KAT}}$ mit der Objektklasse K_2 und der Morphismenmenge M_2, wenn $K_1 \subseteq K_2$ und $M_1 \subseteq M_2$ mit $id_A^{\underline{\mathrm{KIT}}} = id_A^{\underline{\mathrm{KAT}}}$ für alle $A \in K_1$ und $f \circ_{\underline{\mathrm{KIT}}} g = f \circ_{\underline{\mathrm{KAT}}} g$ für alle $f, g \in M_1$. △

Die wichtigste kategorielle Konstruktion in bezug auf algebraische Graphtransformation ist das *Pushout*, das in den jeweiligen Graphkategorien einen direkten Graphtransformationsschritt beschreibt, also das Ergebnis der Anwendung einer Graphregel auf einen Arbeitsgraphen.

Definition 6.3 (*Pushout*) Das *Pushout* zweier Morphismen $f : A \to B$ und $g : A \to C$ in einer Kategorie <u>KAT</u> ist ein Objekt D, auch *Pushout-Objekt* genannt, zusammen mit zwei Pushoutmorphismen $f^* : C \to D$ und $g^* : B \to D$ so, daß folgendes gilt:

Kommutativität: $f^* \circ g = g^* \circ f$. In der Diagrammdarstellung wird die Kommutativität von Morphismen üblicherweise durch ein „(=)“ ausgedrückt.

universelle Eigenschaft: Für alle Paare von Morphismen $f' : C \to E$ und $g' : B \to E$ mit $f' \circ g = g' \circ f$ gibt es einen eindeutigen Morphismus $u : D \to E$, auch *universeller Morphismus* genannt, so daß $u \circ g^* = g'$ und $u \circ f^* = f'$.

Um darzustellen, daß es sich bei einem Diagramm A, B, C, D mit den Morphismen f, g, f^* und g^* um ein Pushout handelt, wird in der Diagrammdarstellung oft ein „(PO)“ in das sich ergebende Viereck geschrieben. △

Im allgemeinen Fall gibt es in einer Kategorie <u>KAT</u> nicht zu jedem beliebigen Paar von Morphismen $f : A \to B$ und $g : A \to C$ ein Pushout. Für eine Graphkategorie, in der wir die Transformation durch ein Pushout beschreiben wollen, ist diese Eigenschaft jedoch wünschenswert, damit jede Anwendung einer Graphregel auch ein definiertes Ergebnis hat. Wir sagen, eine Kategorie, die diese Eigenschaft erfüllt, *hat alle Pushouts*.

6.1.2 ALR-Morphismen

Um die Transformation von ALR-Graphen durch ein Pushout beschreiben zu können, benötigen wir zunächst eine geeignete Kategorie. Zu den ALR-Graphen aus Kapitel 5 als Objekte in der Kategorie fehlt uns noch die Definition eines entsprechenden Morphismusbegriffs. Wir benötigen partielle Morphismen zur Modellierung von löschenden Regeln, während Ansätze einer Regel in einen Arbeitsgraphen durch totale Morphismen dargestellt werden. Die nachfolgend definierten partiellen ALR-Morphismen bilden nach [AR89] zusammen mit den ALR-Graphen aus Def. 5.1 eine Kategorie. Totale ALR-Morphismen ergeben sich als Spezialfall.

Definition 6.4 (*ALR-Morphismus*) Gegeben zwei ALR-Graphen $G = (O_G, \perp_G, h_G, a_G, s_G, t_G, L_G, \ell_G)$ und $H = (O_H, \perp_H, h_H, a_H, s_H, t_H, L_H, \ell_H)$ nach Def. 5.1. Eine partielle Abbildung $m : O_G \to O_H$ heißt *ALR-Morphismus* zwischen den Graphen G und H, geschrieben $m : G \to H$, wenn sie die folgenden Eigenschaften hat:

1. $m(\perp_G) = \perp_H$
2. $m(h_G) = h_H$
3. $\forall x \in dom_{\mathcal{S}}(m)$:
   (a) $\ell_G(x) = \ell_H(m(x))$
   (b) $a_G(x) \in dom_{\mathcal{S}}(m) \;\wedge\; m(a_G(x)) = a_H(m(x))$
   (c) $s_G(x) \in dom_{\mathcal{S}}(m) \;\wedge\; m(s_G(x)) = s_H(m(x))$
   (d) $t_G(x) \in dom_{\mathcal{S}}(m) \;\wedge\; m(t_G(x)) = t_H(m(x))$

Dabei bezeichnet $dom_{\mathcal{S}}(m)$ die Teilmenge von O_G, auf der die Abbildung m definiert ist. $\triangle$

Aus dieser Definition folgt insbesondere die typische Eigenschaft für Graphmorphismen im allgemeinen, daß nämlich die Objektmenge $dom_{\mathcal{S}}(m)$ zusammen mit den auf diese Teilmenge eingeschränkten Abbildungen a_G, s_G, t_G, ℓ_G einen Teilgraphen von G darstellt; diesen Teilgraphen bezeichnen wir auch mit $dom_{\mathcal{G}}(m)$. Entsprechendes gilt auch für den Wertebereich von m.

Bereits im Zusammenhang mit der Definition von Verfeinerungsbäumen (Def. 5.2) haben wir festgestellt, daß wir den Verfeinerungsbaum einer Kante e als partielle *Relation* zwischen den Objektmengen der Verfeinerungsbäume von $s(e)$ und $t(e)$ auffassen können. Das heißt aber, daß sich insbesondere auch partielle *Abbildungen* als Verfeinerungsbäume der Kante e darstellen lassen. Seien nun $s(e)$ und $t(e)$ Knoten, dann sind ihre Verfeinerungsbäume nach Satz 5.3 ALR-Graphen, und es gibt für jeden ALR-Morphismus m zwischen diesen ALR-Graphen eine äquivalente Darstellung als Verfeinerungsbaum V_m von e. In der Implementierung werden die ALR-Morphismen auf diese Weise als Verfeinerungsbäume von Kanten repräsentiert.

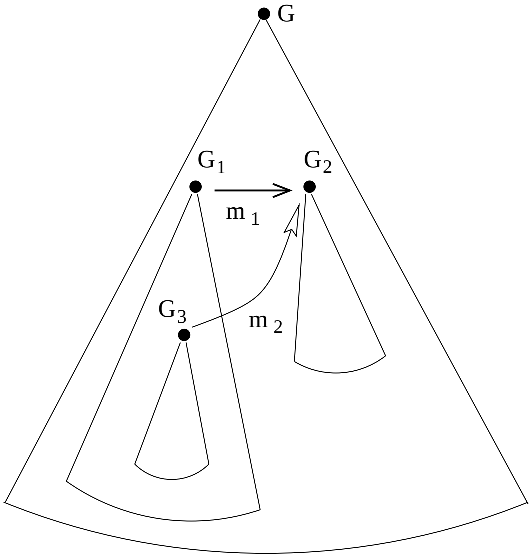
Konstruktion 6.5 (*Äquivalenter Verfeinerungsbaum zu einem ALR-Morphismus*)
Sei $G = (O, \perp, h, a, s, t, L, \ell)$ ein ALR-Graph, $e \in O$ eine Kante und $s(e), t(e)$ Knoten. Außerdem gelte $\neg\exists\ x \in O : a(x) = e$. Dann sind die Verfeinerungsbäume $S_e = (O_S, \perp_S, h_S = s(e), a_S, s_S, t_S, L_S, \ell_S)$ und $T_e = (O_T, \perp_T, h_T = t(e), a_T, s_T, t_T, L_T, \ell_T)$ ALR-Graphen. Für jeden ALR-Morphismus $m : S_e \longrightarrow T_e$ mit $dom_{\mathcal{S}}(m) = \{\perp_S, h_S, x_1, \ldots, x_n\}$ gibt es eine äquivalente Darstellung als Verfeinerungsbaum der Kante e. Wir konstruieren den äquivalenten Verfeinerungsbaum, indem wir den Graphen G wie folgt ergänzen:

- $O := O \uplus \{e_1, \ldots, e_n\}$, wobei das Symbol „$\uplus$“ für die disjunkte Vereingung von Mengen steht.
- $\forall i \in \{1, \ldots, n\}$:
  - $s(e_i) := x_i$
  - $t(e_i) := m(x_i)$
  - $a(e_i) := d \ \text{ mit } \ s(d) = a(x_i), \quad d \in \{e, e_1, \ldots, e_n\}$
  - $\ell(e_i) := k \ \text{ mit } \ k \in L$ beliebig.

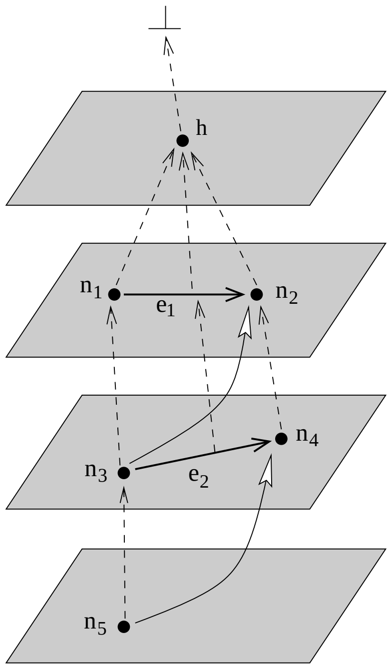
Daß das Tupel $G = (O, \perp, h, a, s, t, L, \ell)$ weiterhin ein ALR-Graph ist, folgt unmittelbar aus der Konstruktion. Der Verfeinerungsbaum $G_e = (O_E, \perp_E, h_E = e, a_E, s_E, t_E, L_E, \ell_E)$ ist dann eine äquivalente Darstellung des ALR-Morphismus m. $\triangle$

Die Äquivalenz der beiden Darstellungen ist offensichtlich, weil jedem Paar (x, y) von Objekten, die der Morphismus aufeinander abbildet, genau eine Kante d von G_e zugeordnet wird mit $s(d) = x$ und $t(e) = y$.

Bemerkung 6.6 (*ALR-Morphismen ohne Äquivalent*) Nicht ohne Grund sind wir bei der Konstruktion 6.5 von zwei ALR-Graphen ausgegangen, deren Top-Objekte bereits durch eine Kante in einem übergeordneten ALR-Graphen verbunden waren. Das garantierte uns, daß



(a) Schematische Darstellung von zwei Morphismen zwischen den Verfeinerungsbäumen eines ALR-Graphen.



(b) Ein konkreter Graph mit konkreten Morphismen zum Schema aus (a).

Abbildung 6.1: Der Morphismus m_2 ist nicht als Verfeinerungsbaum darstellbar.

sich die beiden Top-Objekte der Graphen, die wir durch einen Morphismus in Beziehung setzen wollten, auf derselben Ebene des übergeordneten Graphen befanden. Zusammen mit den Bedingungen 2 und 3b der Morphismus-Definition 6.4 war damit gewährleistet, daß auch alle übrigen Graphobjekte aus dem Domain des Morphismus auf derselben Ebene lagen wie ihre Bilder. Dies aber ist Voraussetzung, wenn wir zwischen Bild und Urbild eine Kante ziehen wollen, ohne die ALR-Eigenschaften des übergeordneten Graphen zu verletzen. Für Morphismen zwischen ALR-Graphen, deren Top-Objekte *nicht* auf derselben Ebene liegen, kann es dementsprechend *keine* äquivalente Verfeinerungsbaumdarstellung geben. Abbildung 6.1 veranschaulicht das Problem an einem einfachen Beispiel: Der Morphismus $m_1 : G_1 \longrightarrow G_2$ zwischen den ALR-Graphen mit den Top-Objekten n_1 und n_2 kann, wie in Abb. 6.1 angedeutet, durch den Verfeinerungsbaum der Kante e_1 repräsentiert werden, weil die beiden Top-Objekte auf derselben Ebene liegen. m_2 dagegen, mit den Abbildungsvorschriften $m_2(n_3) = n_2$ und $m_2(n_5) = n_4$ ein wohldefinierter ALR-Morphismus zwischen den Graphen G_3 und G_2, hat keine äquivalente Darstellung als Verfeinerungsbaum, weil wir für eine Kante zwischen n_3 und n_2 kein Abstraktionsobjekt finden können, so daß G ein ALR-Graph bleibt; verantwortlich dafür sind die Bedingungen 4 und 5 von Definition 5.1. $\triangle$

Mit den bisher definierten ALR-Morphismen können wir nur Beziehungen zwischen unattributierten ALR-Graphen beschreiben. Es steht also noch die Definition für den attributierten Fall aus:

Definition 6.7 (*ALR-Morphismus zwischen attributierten ALR-Graphen*) Seien $K_{AS} =$

(K, A_K, at_K, av_K) und $H_{AS} = (H, A_H, at_H, av_H)$ attributierte ALR-Graphen nach Def. 5.4 mit $K = (O_K, \perp_K, h_K, a_K, s_K, t_K, L_K, \ell_K)$ und $H = (O_H, \perp_H, h_H, a_H, s_H, t_H, L_H, \ell_H)$, A_K und A_H Algebren zur Signatur $AS = (S, OP)$. Ein Paar $m = (m_G, m_D)$ heißt *ALR-Morphismus* $m : K_{AS} \to H_{AS}$, wenn

- $m_G : K \to H$ ALR-Morphismus ist gemäß Def. 6.4,
- $m_D : A_K \to A_H$ ein totaler AS-Homomorphismus ist und
- $\forall x \in dom_S(m_G)$ das folgende Diagramm kommutiert:

$$\begin{tikzcd} O_K \arrow[r, "m_G"] \arrow[d, "av_K"'] \arrow[dr, phantom, "(=)"] & O_H \arrow[d, "av_H"] \\ AV_K \arrow[r, "m_D"'] & AV_H \end{tikzcd}$$

△

6.1.3 Attribute in Regeln und Ansätzen

Wir wollen uns in diesem Abschnitt ein wenig genauer mit den unterschiedlichen Rollen beschäftigen, welche die Attribute in Regel- bzw. Ansatzmorphismen spielen und dabei insbesondere untersuchen, in welcher Beziehung unsere Anschauung zu der theoretischen Fundierung dieser Konzepte steht. Diese Untersuchungen bilden die Grundlage für die Integration der Attributkomponente in den Entwurfsabschnitten 6.2.5 und 6.2.7.

In den Beispielen zur Einführung in die attributierte Graphtransformation in Kapitel 2 haben wir wie selbstverständlich Regeln benutzt, die auf Attributen rechnen. Grundvoraussetzung für das Rechnen mit Attributen ist aber die Möglichkeit, Regeln anzugeben, die den Attributwert eines Graphobjekts *verändern*. Abbildung 6.2 zeigt eine solche Regel, auf das Wesentliche reduziert, und macht dabei die Attributzuordnung av und die Daten- bzw. Graphteilmorphismen m_D und m_G explizit (vgl. Def. 6.7). Die beiden Graphen L und R sind über derselben Algebra Nat der natürlichen Zahlen attributiert. Aufgabe der Regel ist es, einen Knoten mit dem Attributwert „4“ im Arbeitsgraphen zu erhalten, seinen Attributwert jedoch auf „7“ zu ändern. Damit der Knoten bei einer Anwendung der Regel tatsächlich erhalten wird, ist er durch die Graphteilabbildung m_G mit dem Knoten auf der rechten Seite verbunden. Nun schreibt die Definition 6.7 aber vor, daß m_D ein totaler Homomorphismus auf der Datenalgebra sein muß, in unserem Fall also auf den natürlichen Zahlen. Wir müssen also insbesondere ein Bild für die Zahl „4“ unter m_D definieren. Eine Definition von $m_D(4) := 7$ wie in Abb. 6.2 ist aber leider nicht mit der Strukturverträglichkeit eines Homomorphismus vereinbar. Offensichtlich ist $m_D(4) := 4$ die einzig mögliche Definition, die die Homomorphiebedingungen erfüllt. Damit wäre aber wegen $av(m_G(p)) = 7 \neq m_D(av(p)) = 4$ die nach Def. 6.7 ebenfalls geforderte Attributierungsverträglichkeit von m verletzt.

Dieses Beispiel demonstriert eine allgemeine Eigenschaft der Theorie über attributierte Graphstrukturen nach [LKW93]: Der Attributwert eines Graphobjekts kann nicht geändert werden, ohne das Graphobjekt selbst zu löschen und neu zu erzeugen. Im SPO-Ansatz ist aber das Löschen eines Graphobjekts mit anschließendem Wiedererzeugen nicht äquivalent

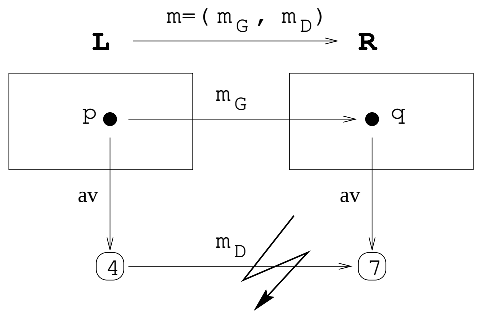


Abbildung 6.2: Die Theorie erlaubt keine Änderung der Attribute…

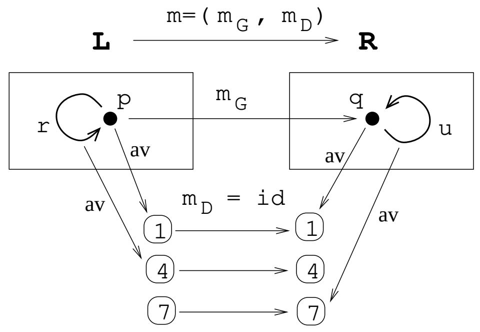


Abbildung 6.3: …sondern simuliert sie durch Löschen und Neuerzeugen von Graphobjekten mit veränderter Attributzuordnung.

zum direkten Erhalt, weil beim Löschen auch hängende Kanten implizit mitgelöscht werden. Um dennoch das Rechnen auf Attributwerten durch Regeln beschreiben zu können, bedient man sich eines Tricks: Anstatt ein Graphobjekt p direkt zu attributieren, wird eine Kante r mit $s(r) = t(r) = p$ aufgenommen, die ausschließlich als *Attributträger* fungiert. Diese Schlaufe kann bedenkenlos gelöscht und wieder eingefügt werden, wenn das Attribut geändert werden soll. Abbildung 6.3 veranschaulicht das Prinzip. Die Attributierung von p und q mit „1“ erfüllt dabei nur eine Dummy-Funktion, um die Attributierungsoperation av zu totalisieren. Daß in unserem Beispiel die Morphismusdefinition 6.7 als Datenteilmorphismus die Identität erzwingt, ist kein Zufall. In der Praxis gilt der allgemeine Grundsatz:

> Die beiden Graphen einer Regel werden grundsätzlich über derselben Algebra attributiert. Der Datenteil m_D des **Regelmorphismus** ist immer durch die Identität auf dieser Algebra gegeben. Um die Verwendung von Variablen in Graphregeln zu formalisieren, wird als Attributalgebra üblicherweise eine Termalgebra mit Variablen zur gegebenen Signatur eingesetzt.

Im Arbeitsgraphen dagegen werden keine Variablen zugelassen. Der Ansatzmorphismus belegt die Variablen der Regel, und die Transformation wertet die Terme der rechten Regelseite unter dieser Variablenbelegung aus. Im Gegensatz zu einem Regelmorphismus drückt ein Ansatz außerdem niemals die Veränderung von Attributwerten aus: Da der Graphteilmorphismus von einem Ansatz total sein muß, bleiben entsprechend der obigen Beobachtungen auch die Attributwerte gezwungenermaßen erhalten. Ansatz- und Regelmorphismen haben also bezüglich der Attributierung völlig unterschiedliche Funktionen. Entsprechend anders lautet auch der Grundsatz für den Datenteilmorphismus eines Ansatzes:

> Während die beiden Graphen einer Regel über einer Termalgebra mit Variablen attributiert sind, wird der Arbeitsgraph über einer beliebigen Wertealgebra zur selben Signatur attributiert. Ein **Ansatzmorphismus** induziert eine Belegung der Variablen der Termalgebra, und der Datenteilmorphismus m_D ist gegeben durch die eindeutige Fortsetzung dieser Variablenbelegung (vgl. [EM85]).

Bemerkung 6.8 (*Implementierung der Attributierungskonzepte*) Das in diesem Abschnitt vorgestellte Attributträgerkonzept ist durch technische Details der theoretischen Fundierung geprägt und verkompliziert den intuitiven Umgang mit Attributen. Deshalb wurde beim Entwurf der Attributkomponente in [Mel97] die Entscheidung getroffen, diese technischen Details zu kapseln und nach außen eine weitestgehend von der Intuition geprägte Schnittstelle zu präsentieren. Das bedeutet insbesondere, daß in der Graphrepräsentation keine Attributträger modelliert werden müssen. Die implementierte Semantik entspricht jedoch bis auf einen dokumentierten Konfliktfall dem zugrundeliegenden Attributträgerkonzept. $\triangle$

6.1.4 Transformation durch Pushout in der Kategorie ALPHA

In [Löw93] wird ein direkter Graphtransformationsschritt als *Pushout* beschrieben. Auch die Transformation von ALR-Graphen wird in [AR89] durch ein Pushout in der entsprechenden Kategorie erklärt[1]. Die Colimit-Bibliothek [Wol97] erlaubt die Berechnung allgemeiner Colimiten in der Kategorie der ALPHA-Algebren. Da der Pushout ein Spezialfall der allgemeinen Colimeskonstruktion ist, liegt es nahe, die Bibliothek als Transformationskomponente im AGG-System zu verwenden. Dazu geben wir in diesem Abschnitt eine Übersetzung der ALR-Graphen und -Morphismen in eine entsprechende ALPHA-Repräsentation an und begründen, daß das Pushout in der Kategorie der ALPHA-Algebren der entsprechenden Konstruktion auf ALR-Ebene entspricht.

ALPHA-Algebren werden in [CL95, Erd95] als „abstraktes Metamodell zur Darstellung beliebiger Graphen“ eingeführt. Sie dienen dort als Grundlage für eine Klassenbibliothek, die die Erstellung von Modellierungs- und Restrukturierungswerkzeugen unterstützen soll; einen Überblick über den Entwurf dieser Bibliothek gibt auch [EC96]. Der Begriff der ALPHA-Algebra wird dort wie folgt definiert[2]:

Definition 6.9 (ALPHA-*Algebra*) Eine ALPHA-*Algebra* ist eine Algebra zu der folgenden attributierten Graphsignatur (kurz: AGS, vgl. [LKW93]):

[1] [AR89] bezeichnen ALR-Graphen gemäß Def. 5.1 als *reguläre* ALR-Graphen. Es wird gezeigt, daß reguläre ALR-Graphen mit ihren Morphismen eine Kategorie bilden und daß das Ergebnis eines Pushouts in dieser Kategorie wieder ein regulärer ALR-Graph ist (Satz 3.7 in [AR89]).

[2] Der Einfachheit halber vernachlässigen wir hier, [EC96] folgend, die Operationen auf dem Datenteil der Signatur.

ags	ALPHA			
	graph part	**sorts**	*Item*	
		opns	$r_i : Item \to Item$	$(i \in \mathbb{N})$
	data part	**sorts**	*Data*	
	attr part	**opns**	$v_j : Item \to Data$	$(j \in \mathbb{N})$
end				

△

Im Graphteil der Signatur fällt sofort eine Ähnlichkeit zu den ALR-Graphen auf: Es gibt nur eine Sorte *Item*, die sowohl Knoten als auch Kanten repräsentiert. ALPHA-Algebren sind eine Verallgemeinerung der ALR-Graphen insofern, als sie nicht nur drei festgelegte Operationen *source, target* und *abstraction* auf ihrer Objektmenge erlauben, sondern eine beliebige Anzahl solcher einstelliger totaler Operationen r_i. Ebenso beliebig kann die Anzahl von Attributierungsoperationen v_j gewählt werden, wodurch jedem Element der *Item*-Menge einer ALPHA-Algebra beliebig viele Elemente einer Attributwertemenge zugeordnet werden können. Dabei ist aber zu berücksichtigen, daß es sich bei der oben angegebenen Signatur um ein Signaturschema handelt, daß erst durch konkrete Werte für i und j instanziiert werden muß. In einer Instanz dieses Schemas ist die Anzahl der Operationen zwar beliebig, aber fest.

Morphismen zwischen ALPHA-Algebren entsprechen einem Spezialfall der allgemeinen Definition für Morphismen zwischen AGS-Algebren aus [LKW93]:

Definition 6.10 (ALPHA-*Morphismus*) Gegeben zwei ALPHA-Algebren A_1 und A_2 mit den *Item*-Mengen $Item_1$ bzw. $Item_2$ und den Trägermengen $Data_1$ bzw. $Data_2$ zur Sorte *Data*. Ein ALPHA-*Morphismus* $m : A_1 \to A_2$ ist ein Paar (m_G, m_D) von Abbildungen, wobei

- $m_G : Item_1 \to Item_2$ eine mit den Graphteil-Operationen r_i verträgliche partielle Abbildung ist, während
- $m_D : Data_1 \to Data_2$ eine totale Abbildung darstellt.

Die Abbildungen auf dem Graph- und dem Datenteil müssen mit den Attributierungsoperationen in der jeweiligen Algebra verträglich sein, d.h. das folgende Diagramm muß kommutativ sein für alle Attributierungsoperationssymbole v_j aus der gegebenen Signatur:

$$\begin{tikzcd} Item_1 \arrow[r, "m_G"] \arrow[d, "v_{j_1}"'] \arrow[dr, phantom, "(=)"] & Item_2 \arrow[d, "v_{j_2}"] \\ Data_1 \arrow[r, "m_D"'] & Data_2 \end{tikzcd}$$

△

Da in der Implementierung des AGG-Systems die Attributkomponente von [Mel97] die Attributberechnungen übernehmen wird, betrachten wir im folgenden lediglich den Graphteil der ALPHA-Algebren und werden auch nur diesen von der Colimit-Bibliothek transformieren lassen. Dieses Vorgehen entspricht einer komponentenweisen Transformation von Graph- und Datenteil einer AGS-Algebra nach [LKW93]. Für die weiteren Überlegungen können wir deshalb die ursprüngliche AGS-Signatur ALPHA auf die folgende einfache Graphstruktursignatur[3] einschränken:

sig ALPHA-3
  sorts *Item*
  opns $r_1, r_2, r_3 : Item \to Item$
end

Wir wählen eine Signaturinstanz mit drei Operationssymbolen r_1 bis r_3, um den drei Operationen *abstraction, source* und *target* eines ALR-Graphen zu entsprechen. Morphismen zwischen ALPHA-3-Algebren reduzieren sich auf partielle operationsverträgliche Abbildungen zwischen deren *Item*-Mengen. ALPHA-3-Algebren und -Morphismen bilden zusammen eine Kategorie <u>ALPHA-3</u>, die alle Pushouts hat. Dies folgt aus den allgemeinen Eigenschaften von Graphstrukturen und deren Morphismen, vgl. [Löw93].

Offensichtlich ist jeder ALR-Graph $G = (O, \perp, h, a, s, t, L, \ell)$ eine Algebra zur Signatur ALPHA-3, wenn wir O als Trägermenge zur Sorte *Item* und a, s, t als Operationen für die Symbole r_1, r_2, r_3 auffassen. Dabei müssen wir allerdings in Kauf nehmen, daß uns der Zugriff auf die Typinformation des ALR-Graphen in der ALPHA-3-Algebra nicht mehr möglich ist. Außerdem ist jeder Morphismus zwischen zwei ALR-Graphen auch ein Morphismus zwischen den entsprechenden ALPHA-3-Algebren, denn die Operationsverträglichkeit wird auch von einem ALR-Morphismus verlangt (vgl. Def. 6.4). Umgekehrt ist aber natürlich weder jede ALPHA-3-Algebra ein ALR-Graph noch jeder ALPHA-3-Morphismus ein ALR-Morphismus, weil die ALR-Begriffe jeweils zusätzliche Konsistenzbedingungen erfordern, z.B. muß jeder ALR-Morphismus die Top-Objekte zweier ALR-Graphen aufeinander abbilden, und in einem ALR-Graphen muß das Abstraktionsziel eines Knotens immer ein Knoten sein.

Da die über einer festen Labelmenge L typisierten ALR-Graphen aus der Definition 5.1 mit den entsprechenden Morphismen (Def. 6.4) nach [AR89] eine Kategorie $\underline{\mathrm{ALR}^{\mathrm{L}}}$ bilden, die alle Pushouts hat[4], gilt dies auch für den untypisierten Fall, denn ein untypisierter Graph kann äquivalent als über einer einelementigen Labelmenge typisiert angesehen werden. Da wir schon gesehen haben, daß jeder ALR-Graph eine ALPHA-3-Algebra und jeder ALR-Morphismus auch ein ALPHA-3-Morphismus ist, ist die Kategorie <u>ALR</u> der untypisierten ALR-Graphen eine Unterkategorie von <u>ALPHA-3</u>.

Konstruktion 6.11 (*Pushout in* $\underline{\mathrm{ALR}^{L}}$) Das Pushout eines Diagramms

$$\begin{tikzcd} A \arrow[r, "m_B"] \arrow[d, "m_C"'] \arrow[dr, phantom, "(1)"] & B \\ C & \end{tikzcd}$$

in der Kategorie $\underline{\mathrm{ALR}^{\mathrm{L}}}$ können wir jetzt wie folgt konstruieren:

1. Berechne das Pushout $(D, c : C \to D, b : B \to D)$ des Diagramms (1) in <u>ALPHA-3</u> unter Vernachlässigung der Typinformation. Das Ergebnis ist auch Pushout in <u>ALR</u>, weil

[3] Graphstruktursignaturen sind Signaturen, die ausschließlich einstellige Operationssymbole enthalten (vgl. [Löw93]).

[4] Da es mit dem Graphen, der nur $\perp$ und ein Top-Objekt als Elemente hat, auch ein initiales Objekt in $\underline{\mathrm{ALR}^{\mathrm{L}}}$ gibt, ist die Kategorie darüber hinaus auch covollständig, vgl. [AHS90].

untypisierte ALR-Graphen Objekte in ALR sind, ALR Unterkategorie von ALPHA-3 ist und alle Pushouts hat.

$$\begin{tikzcd} A \arrow[r, "m_B"] \arrow[d, "m_C"'] \arrow[dr, phantom, "(PO)"] & B \arrow[d, "b"] \\ C \arrow[r, "c"'] & D \end{tikzcd}$$

2. Bestimme eindeutig die Typisierung von D so, daß D, c und d Objekte bzw. Morphismen in der Kategorie $\underline{\mathrm{ALR}^{\mathrm{L}}}$ sind:

$$\forall x \in D: \ \ell_D(x) = \begin{cases} \ell_C(y), & wenn\ \exists y \in C: \ c(y) = x \\ \ell_B(y), & wenn\ \exists y \in B: \ b(y) = x \end{cases}$$

ℓ_D ist total, weil c und b nach [Löw93] zusammen surjektiv sind. ℓ_D ist wohldefiniert, weil ein Objekt $x \in D$ aufgrund der universellen Eigenschaft des Pushouts nur dann Urbilder sowohl in C als auch in D haben kann, wenn es ein $z \in A$ gibt mit $b(m_B(z)) = c(m_C(z)) = x$. Dann haben aber wegen der Typverträglichkeit der Morphismen m_A und m_B beide Urbilder denselben Typ. ℓ_D ist außerdem eindeutig, weil jede andere Definition die Typverträglichkeit der Morphismen c oder b verletzen würde.

$\triangle$

Nun bleibt noch zu begründen, warum das Ergebnis dieser Konstruktion tatsächlich ein Pushout in $\underline{\mathrm{ALR}^{\mathrm{L}}}$ ist. Dazu müssen wir zeigen, daß das Ergebnisdiagramm kommutiert und daß die universelle Eigenschaft gilt (vgl. Def. 6.3). Die Kommutativität gilt offensichtlich, da das Diagramm als Pushout in ALPHA-3 konstruiert wurde. Für den Nachweis der universellen Eigenschaft betrachten wir die folgende Situation: Sei E typisierter ALR-Graph, $f: C \to E$ und $g: B \to E$ ALR-Morphismen mit $f \circ m_C = g \circ m_B$. Durch Vergessen der Typinformation erhalten wir eine entsprechende Situation in ALR, und weil das Diagramm (PO) nach Voraussetzung ein Pushout in ALR ist, existiert eindeutig ein Morphismus $d: D \to E$ mit $d \circ c = f$ und $d \circ b = g$. Diesen Morphismus können wir auf den typisierten Fall übertragen, wobei die Kommutativitäten erhalten bleiben. Die Frage ist jedoch, ob d überhaupt ein Morphismus in der Kategorie $\underline{\mathrm{ALR}^{\mathrm{L}}}$ ist, ob er also die zusätzliche Forderung nach Typverträglichkeit erfüllt. Nehmen wir einmal an, dies sei nicht der Fall. Dann gibt es ein $x \in D$ mit $\ell_D(x) \neq \ell_E(d(x))$. Da aber b und c zusammen surjektiv sind, gibt es ein Urbild y von x zumindest in einem der beiden Graphen C oder B. Sei also o.B.d.A. $y \in C$ mit $c(y) = x$. Nach Voraussetzung ist c typverträglich, also $\ell_C(y) = \ell_D(x)$. Wegen $f = d \circ c$ ist aber $f(y) = d(c(y)) = d(x)$, und weil f ebenfalls typverträglich ist, gilt weiter $\ell_C(y) = \ell_E(d(x))$ und damit schließlich $\ell_D(x) = \ell_E(d(x))$, was im direkten Widerspruch zur Annahme steht.

6.2 Entwurf

In diesem Abschnitt beschreiben wir die Anwendungsschnittstellen für ALR-Morphismen und Transformationsschritte. Außerdem gehen wir auf die Attributbehandlung und die Konsistenzsicherung der Morphismusimplementierung ein und beschäftigen uns mit der Anbindung der Colimit-Bibliothek. Auf der Basis dieser Bibliothek entwickeln wir eine Klasse für die Berechnung allgemeiner Colimiten auf ALR-Graphen und können dann die Schnittstelle für direkte Transformation als Spezialfall implementieren.

6.2.1 Anforderungsdefinition

Funktionale Anforderungen

Regel- und Ansatzmorphismen repräsentieren.
In Abschnitt 6.1.2 haben wir festgestellt, daß wir ALR-Morphismen durch Verfeinerungsbäume von Kanten modellieren können. Die entsprechende Klasse `ALR_RefTree` haben wir bereits in Kapitel 5 vorgestellt. Um aber einen komfortablen und sicheren Umgang mit dem so wichtigen Begriff des Morphismus zu ermöglichen, wollen wir diese Modellierung hinter einer abstrakten Schnittstelle verstecken. Diese Schnittstelle soll insbesondere sicherstellen, daß die darunterliegende Modellierung tatsächlich Morphismuseigenschaften hat.

Einfache Graphtransformationsschritte berechnen.
Die Grundfunktionalität eines Graphtransformationssystems besteht in der Anwendung einer Graphregel auf einen Arbeitsgraphen an einem vorgegebenen Ansatz. Die Transformationskomponente soll einen solchen einfachen Transformationsschritt nach dem *Single-Pushout-Verfahren* durchführen[5]. Auf diesen atomaren Transformationsschritten können dann Transaktionskonzepte und andere Kontrollmechanismen aufsetzen, was jedoch nicht mehr Thema dieser Arbeit ist.

Transformationsergebnis „in place" berechnen.
Bei der theoretischen Beschreibung eines Graphtransformationsschrittes durch ein Pushout wird das Ableitungsergebnis durch einen neuen Graphen repräsentiert, während der ursprüngliche Arbeitsgraph unversehrt bleibt. Der Ergebnisgraph wird dann über Morphismen mit dem Ausgangsgraphen und der rechten Regelseite in Beziehung gesetzt; so ergibt sich das typische Viereck des Pushout-Diagramms. Für die Implementierung hat es jedoch Vorteile, wenn sich die Effekte einer Regelanwendung unmittelbar auf der Graph-Instanz auswirken, auf die die Regel angesetzt wurde, auch wenn dabei der Zustand des Arbeitsgraphen *vor* der Regelanwendung verloren geht (*Inplace-Transformation*):

Raumeffizienz. Die Inplace-Transformation ist weniger speicherintensiv, denn bei der vollständigen Diagrammberechnung wird in jedem Schritt eine Kopie des Arbeitsgraphen erzeugt inklusive eines Morphismus, der die Beziehung zum Original herstellt.

Zeiteffizienz. Viele Anwendungskomponenten sind nur am aktuellen Zustand des Arbeitsgraphen und dessen Veränderungen interessiert; betrachten wir als Beispiel etwa eine Animationskomponente. Bei der *Diagramm-Transformation* bekäme die Komponente über den Observer-Mechanismus eine unverhältnismäßig große Flut an Änderungsinformationen zugeschickt, denn schließlich werden alle Graphobjekte, die den Ergebnisgraphen und seine Morphismen repräsentieren, in einem Transformationsschritt neu erzeugt. Die für die Animation relevanten Informationen müssen aus dieser Menge erst durch eine aufwendige Analyse gewonnen werden. Bei der Inplace-Transformation bekommt die Animationskomponente da-

[5] Eine Simulation des *Double-Pushout*-Verhaltens ist durch bestimmte Anforderungen (die sog. *gluing-condition*, vgl. Def. 2.11) an die Ansatzmorphismen leicht zu erreichen und ist als Erweiterung vorgesehen.

gegen genau die relevanten Änderungsinformationen zugeschickt, nämlich die, die gelöschte, hinzugefügte oder verklebte Objekte betreffen.

Es sind allerdings durchaus Anwendungskomponenten vorstellbar, die auf eine vollständige Berechnung des Pushout-Diagramms angewiesen sind, z.B. für die Analyse von Ableitungssequenzen. Deshalb ist für eine Ausbaustufe des Systems eine optionale Diagramm-Transformation vorgesehen.

Auch nicht-injektive Regel- und Ansatzmorphismen zulassen.
Im Interesse größtmöglicher Allgemeinheit bezüglich der Abbildung theoretischer Konzepte wollen wir uns nicht – wie viele bestehende Implementierungen von Graphtransformationssystemen – auf injektive Morphismen für Regeln und Ansätze einschränken.

Auf Attributen rechnen.
Während der Transformation müssen Attributwerte gemäß der Regelvorschrift gesetzt bzw. Attributausdrücke über Variablen berechnet werden, die vom aktuellen Ansatz belegt wurden (vgl. Beispiel 2.13 auf Seite 14).

Softwaretechnische Anforderungen

Korrektheit. Da es sich bei der Transformationskomponente um das Herzstück des Interpreters handelt, von dem die korrekte Ausführung der innerhalb des Systems spezifizierten Graphprogramme abhängt, kommt der Korrektheit dieses Systemteils bezüglich der formal definierten Transformationssemantik natürlich eine besonders große Bedeutung zu.

Effizienz. Der theoretische Aufwand eines Transformationsschrittes ist linear zur Größe der Regel. Solange auch die Implementierung im linearen Bereich bleibt, ist der Aufwand vernachlässigbar gegenüber dem potentiell exponentiellen Aufwand der Ansatzsuche. In Spezialfällen, etwa wenn die Ansätze vorgegeben sind, kann aber auch die Transformation selbst zum bestimmenden Aufwand werden.

Austauschbarkeit, Modularität. Natürlich sollte ein so zentraler Teil des Systems zu Wartungs- oder Optimierungszwecken notfalls austauschbar sein. Dies entspricht der Forderung nach einer wohldefinierten, schlanken Schnittstelle für die Transformation.

6.2.2 Anwendungsschnittstelle

Die Anwendungsschnittstelle zur Transformation von ALR-Graphen besteht aus den öffentlichen Interfaces der Klassen `ALR_Morphism`, `ALR_Match` und `ALR_Step` (Abb. 6.4). Die Klasse `ALR_Morphism` implementiert allgemeine ALR-Morphismen; mit den Methoden aus ihrem Interface werden wir uns noch im einzelnen befassen. `ALR_Match` ist eine Spezialisierung von `ALR_Morphism` zur Repräsentation von Ansatzmorphismen. Als einzige Schnittstellenerweiterung gegenüber `ALR_Morphism` steht die Methode `getRuleMorphism()` zur Verfügung, mit der der zu einem Ansatz gehörende Regelmorphismus abgefragt werden kann. `ALR_Step` schließlich bietet die Schnittstelle für die Ausführung des eigentlichen Transformationsschritts: die Klassenmethode `execute()` bekommt einen Ansatzmorphismus `match` übergeben, aus dem sowohl der Regelmorphismus (`match.getRuleMorphism()`) als auch der Arbeitsgraph

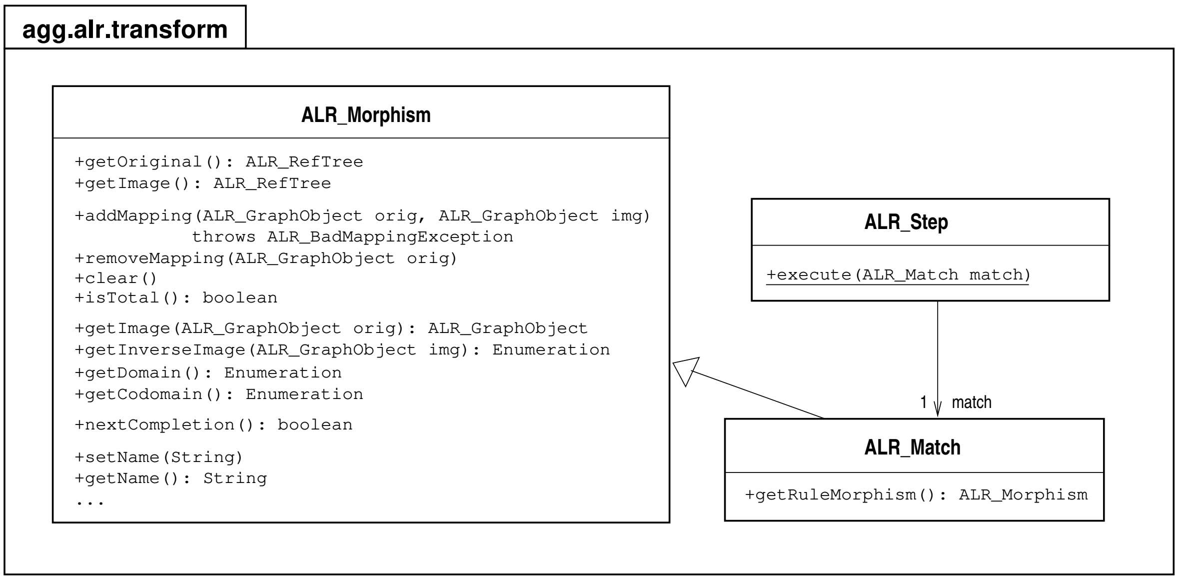


Abbildung 6.4: Die Interfaces für ALR-Morphismen und direkte Transformation.

(`match.getImage()`) für das entsprechende Pushout hervorgeht. Das Ergebnis der Transformation wird „in place“ berechnet, d.h., der Arbeitsgraph wird gemäß der Regelvorschrift modifiziert.

Kommen wir nun zur Beschreibung der wichtigsten Methoden des Interfaces von `ALR_Morphism`. Ein ALR-Morphismus beschreibt eine gerichtete Beziehung zwischen zwei ALR-Graphen. Die konkreten Quell- und Zielgraphen einer Instanz von `ALR_Morphism` werden bereits bei der Instanziierung des Morphismus in Gestalt von Konstruktorargumenten festgelegt. Mit Hilfe der Methoden `getOriginal()` bzw. `getImage()` kann dann zu jeder Zeit auf diese beiden Graphen zugegriffen werden. Die Rückgabewerte der beiden Methoden sind als `ALR_RefTree` typisiert, es gilt jedoch die Zusicherung, daß das Prädikat `isGraph()` auf diesen Verfeinerungsbäumen wahr ist. Das bedeutet nicht mehr und nicht weniger, als daß Quelle und Ziel eines Morphismus tatsächlich ALR-Graphen sind.

Nach Definition 6.4 ist ein ALR-Morphismus zwischen zwei ALR-Graphen eine Abbildung von der Objektmenge des Quell- in die Objektmenge des Zielgraphen, die gewisse Verträglichkeitseigenschaften hat. Mit der Methode `addMapping()` können wir elementweise die Abbildungsvorschrift definieren, indem wir einem Objekt `orig` aus dem Quellgraphen ein Bildobjekt `img` aus dem Zielgraphen zuordnen. War für `orig` bereits zuvor ein Bild definiert, so wird die vorherige Definition durch die neue ersetzt, um die Eindeutigkeit der Abbildung zu bewahren. `addMapping()` löst einen Fehlerzustand aus, wenn die angegebene Abbildungsvorschrift den Verträglichkeitsbedingungen des Morphismus nicht entspricht. Hierzu wird das Konzept der *Exceptions* in Java ausgenutzt: Es wird eine Instanz der Klasse `BadMappingException` erzeugt – in Java-Terminologie: *geworfen* –, die den aufgetretenen Fehler repräsentiert. Der Aufrufer von `addMapping()` kann diese Exception *auffangen* und eine entsprechende Fehlerbehandlung einleiten. Dabei kann über eine Methode `getMessage()` von der Exception eine Zeichenkette abgefragt werden, die die konkrete Fehlerursache beschreibt. An dieser Stelle kommen folgende Ursachen in Betracht:

1. Verletzung der Verträglichkeit mit den Operationen *source, target* oder *abstraction*; insbesondere der Versuch, Objekte aufeinander abzubilden, die auf unterschiedlichen Abstraktionsebenen liegen.

2. Verletzung der Typverträglichkeit bzw. der Versuch, einen Knoten auf eine Kante abzubilden oder umgekehrt.

3. Verletzung der Homomorphiebedingungen der Attributabbildung oder das Auftreten eines Konfliktes bezüglich der Attribute, vgl. [Mel97], Abschnitte 2.2.6 und 3.1.3.

Zum Entfernen einer einzelnen Abbildungsvorschrift dient die Methode `removeMapping()`: Nach einem Aufruf `m.removeMapping(orig)` ist das Bild des Objekts `orig` unter dem Morphismus `m` undefiniert, `orig` liegt also nicht mehr im Definitionsbereich des Morphismus. `clear()` entfernt alle vorhandenen Abbildungsvorschriften, der Definitionsbereich eines Morphismus ist nach einem Aufruf dieser Methode leer. Mit dem Prädikat `isTotal()` dagegen kann geprüft werden, ob ein Morphismus für jedes Objekt aus dem Quellgraphen definiert ist.

Das aktuelle Bild eines Objektes `orig` unter einem Morphismus `m` wird durch den Aufruf `m.getImage(orig)` abgefragt. Der Rückgabewert des Aufrufs ist `null`, falls `orig` nicht im Definitionsbereich von `m` liegt. Umgekehrt erhält man für ein Objekt `img` aus dem Zielgraphen von `m` über einen Aufruf der Form `m.getInverseImage(img)` eine Aufzählung seiner Urbilder unter dem Morphismus. Die Aufzählung ist leer, wenn `img` nicht im Wertebereich von `m` liegt. Die Methoden `getDomain()` und `getCodomain()` liefern Aufzählungen des Werte- bzw. Definitionsbereichs eines Morphismus.

`nextCompletion()` bildet die Schnittstelle zur Morphismusvervollständigung und wird in Kapitel 7 ausführlich besprochen. `getName()` und `setName()` schließlich bieten die Möglichkeit, einen Morphismus mit einem Namen zu versehen.

Bemerkung 6.12 (*Totalität von Ansatzmorphismen*) Nicht jede Instanz von `ALR_Match` ist ein Ansatz im Sinne der Theorie, denn die Theorie fordert von einem Ansatz insbesondere Totalität. Da ein Ansatz in der Implementierung wie jeder Morphismus über die Methoden `add`- und `removeMapping()` dynamisch auf- und abgebaut werden kann, ist die Totalität im allgemeinen nicht gegeben. Stattdessen muß gegebenenfalls mittels des Prädikats `isTotal()` auf diese Eigenschaft geprüft werden. △

6.2.3 Beziehungen zur Theorie

Wie schon bei den ALR-Graphen in Kapitel 5 wollen wir auch die Definition des ALR-Morphismus in tabellarischer Form ihrer Implementierung gegenüberstellen. Seien $G_{AS} = (G, A_G, at_G, av_G)$ und $H_{AS} = (H, A_H, at_H, av_H)$ attributierte ALR-Graphen, A_G und A_H Algebren zur Signatur $AS = (S, OP)$, und sei $m = (m_G, m_D) : G_{AS} \to H_{AS}$ ALR-Morphismus nach Def. 6.7 (S. 56). Dann gelten die folgenden Entsprechungen:

Theorie	Implementierung
m	`ALR_Morphism m`
G_{AS}	`m.getOriginal()`
H_{AS}	`m.getImage()`
AS	gegeben durch Java-Syntax
A_G, A_H	gegeben durch Java-Semantik
AV_G, AV_H	beliebige Tupel von Java-Expressions oder -Werten, repräsentiert durch die Menge aller möglichen Instanzen von `AttrInstance`
$dom(m_G)$	`m.getDomain()`
$codom(m_G)$	`m.getCodomain()`
$x \in dom(m_G)$	`ALR_GraphObject x` mit `m.getImage(x) != null`
$m_G(x)$	`m.getImage(x)`

Die Entsprechungen der Attributsignatur AS und der AS-Algebren A_G und A_H sind dabei abhängig vom ausgewählten konkreten Attribut-Handler; die Angaben in der Tabelle beziehen sich auf den in [Mel97] beschriebenen Java-Expression-Handler. Gemäß der in Abschnitt 6.1.3 (S. 57) formulierten Grundsätze hängt die Wahl der Attributalgebren A_G und A_H außerdem von der Rolle des konkreten Morphismus als Regel oder Ansatz ab: Im Falle eines Ansatzes entspräche A_H einer Algebra der Java-Werte, sonst bestehen beide Algebren aus unausgewerteten Java-Expressions, was dem Begriff der Termalgebra mit Variablen entspricht. Der Datenteilmorphismus m_D ist dann gegeben durch die Identität auf Java-Expressions für Regelmorphismen bzw. durch die Gleichheit der ausgewerteten Expressions mit den Werten im Arbeitsgraphen für Ansatzmorphismen.

Genaugenommen entspricht die Implementierung von Regelmorphismen, die Attribute ändern, nicht der theoretischen Definition 6.7 eines ALR-Morphismus, denn die Attributkomponente erlaubt es uns, vom Attributträgerprinzip (vgl. Abschnitt 6.1.3) zu abstrahieren. Praktisch bedeutet das, daß für Regelmorphismen in der Implementierung das Diagramm, das in der Morphismusdefinition die Attributierungsverträglichkeit beschreibt, *nicht* kommutieren muß. Damit sind in der Implementierung Attributierungsänderungen nach dem Muster von Abbildung 6.2 auf Seite 58 erlaubt, natürlich ohne daß die veränderten Attribute durch den Datenteilmorphismus m_D aufeinander abgebildet werden; m_D bleibt wie üblich durch die Identität gegeben. Für die Ansatzmorphismen ergibt sich keine derartige Diskrepanz zur Theorie, weil ein Ansatz grundsätzlich keine Attribute verändert.

6.2.4 Änderungsinformationen

Ganz analog zur Klasse `ALR_RefTree` bietet auch `ALR_Morphism` als Subklasse von `ExtObservable` anderen Klassen im Rahmen des Observer-Patterns die Möglichkeit, sich als Observer anzumelden, um von Änderungen auf dem Zustand des Morphismus informiert zu werden. Nachdem wir die allgemeine Funktionsweise des Observer-Patterns bereits in Abschnitt 3.2 kennengelernt haben und in Abschnitt 5.2.4 ausführlich auf die Repräsentation von Änderungsinformationen durch Subklassen von `agg.util.Change` eingegangen sind, können wir uns an dieser Stelle auf die Beschreibung der Semantik der von `ALR_Morphism` erzeugten konkreten Änderungsinformationen beschränken.

Das Klassendiagramm in Abbildung 6.5 zeigt die Implementierung von `ALR_Morphism`

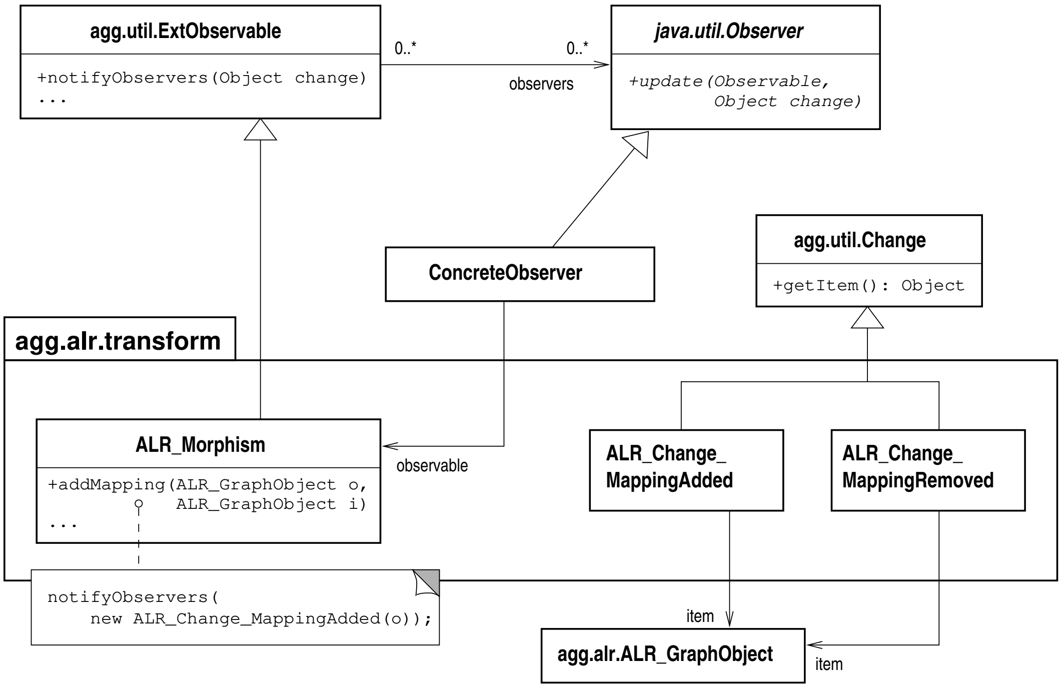


Abbildung 6.5: Änderungsinformationen des Observables `ALR_Morphism`.

als Subklasse von `ExtObservable` und die konkreten Änderungsinformationen mit ihren Item-Beziehungen. Die beiden Änderungsinformationen `ALR_Change_MappingAdded` bzw. `ALR_Change_MappingRemoved` betreffen die Aufnahme bzw. das Entfernen von Zuordnungsvorschriften für einzelne Objekte innerhalb des Morphismus. Der abgebildete Code-Ausschnitt zeigt, wie die Aufnahme einer solchen Zuordnungsvorschrift durch die Methode `addMapping()` die Verteilung der entsprechenden Änderungsinformation auslöst. Im Beispiel gehen wir vereinfachend davon aus, daß das übergebene Abbildungspaar die Morphismuseigenschaften erhält. Andernfalls würde, wie in der Schnittstellenbeschreibung in Abschnitt 6.2.2 dargestellt, eine Ausnahmebehandlung angestoßen und der Zustand des Morphismus bliebe unverändert; entsprechend würde auch keine Änderungsinformation verbreitet.

Tabelle 6.1 gibt eine vollständige Übersicht über die Änderungsinformationen, die ein Observer von einer Instanz `m` von `ALR_Morphism` erhalten kann. `Change_ObservableGone` wird von der Superklasse `ExtObservable` automatisch erzeugt, wenn das Observable gelöscht wird. Wie schon bei den Änderungsinformationen von `ALR_RefTree` in Abschnitt 5.2.4, so gilt auch hier die Philosophie, daß die spezifizierten Informationen atomar sind. Die komplexen Operationen `clear()` und `nextCompletion()` lösen entsprechend eine Menge von Änderungsinformationen aus, die in ihrer Gesamtheit die von der Operation bedingte Zustandsänderung vollständig beschreiben.

6.2.5 Attributbehandlung in `ALR_Morphism`

Je nachdem, ob eine Instanz von `ALR_Morphism` als Ansatz oder als Regel eingesetzt wird, müssen Variablen belegt, Attribute verglichen oder Konflikte vermieden werden. Da der Morphismus selbst keine Kenntnis von der konkreten Repräsentation der Attribute hat, muß die

Change	Item: Itemtyp	Bedeutung
`ALR_Change_MappingAdded`	`x: ALR_GraphObject`	Die Zuordnungsvorschrift `x` $\mapsto$ `m.getImage(x)` wurde aufgenommen.
`ALR_Change_MappingRemoved`	`x: ALR_GraphObject`	Die Zuordnungsvorschrift für `x` wurde entfernt.
`Change_ObservableGone`	`m: ALR_Morphism`	`m` wurde gelöscht (vgl. Abschnitt 3.2).
`null`	–	Undefinierte Zustandsänderung. Vom Observer wird eine vollständige Resynchronisation mit dem Zustand des Observables erwartet.

Tabelle 6.1: Semantik der Änderungsinformationen, die eine Instanz `m` von `ALR_Morphism` an ihre Observer schicken kann.

Attributkomponente diese Funktionalitäten übernehmen. Die diesbezügliche Kommunikation der Morphismusimplementierung mit der Attributkomponente wollen wir im folgenden anhand von Abbildung 6.6 beschreiben.

Zuerst einmal muß ein Morphismus der Attributkomponente mitteilen, ob er als Ansatz- oder Regelmorphismus fungiert, weil sich für die Attributbehandlung in diesen beiden Fällen erhebliche Unterschiede ergeben. Dies geschieht über sogenannte *Attributkontexte*, die mit der Factory-Methode `newContext()` der Klasse `AttrManager` erzeugt werden. Als Parameter wird bei der Erzeugung eines Attributkontexts eine der beiden in `AttrMapping` definierten Konstanten `PLAIN_MAP` oder `MATCH_MAP` als „Mapstyle" übergeben: `PLAIN_MAP` für einen Regel-, `MATCH_MAP` für einen Ansatzkontext. Mit der Methode `setAttrContext()` wird einem Morphismus sein Attributkontext zugewiesen. Diesen Kontext muß der Morphismus von nun an bei jeglicher Kommunikation mit der Attributkomponente sozusagen als Visitenkarte vorlegen, woraufhin diese ihr Verhalten jeweils auf ansatz- oder regelorientierte Attributbehandlung umschaltet. Der Morphismus selbst kann nach der einmaligen Festlegung des Attributkontextes die Behandlung von Attributen einheitlich gestalten und muß nicht mehr zwischen den beiden Fällen „Ansatz" oder „Regel" unterscheiden.

Wenn mit der Methode `addMapping()` eine neue Zuordnungsvorschrift zwischen zwei Graphobjekten `o` und `i` in einen Morphismus aufgenommen werden soll, wird automatisch die Methode `newMapping()` auf dem Attributmanager aufgerufen, um der Attributkomponente mitzuteilen, daß die entsprechenden Attributwerte der beiden Graphobjekte in Beziehung gesetzt werden sollen. Geschieht dies in einem Regelkontext, so ist diese Beziehung grundsätzlich zulässig, denn die Attributkomponente erlaubt uns abweichend von der Theorie die Änderung von Attributen auch auf durch den Regelmorphismus zu erhaltenden Graphobjekten (vgl. dazu die Abschnitte 6.1.3 und 6.2.3). Es gibt lediglich einige Konfliktfälle, die zu undefinierten Transformationsergebnissen führen würden, und die bei dieser Gelegenheit abgefangen werden. Die Erläuterung dieser Konflikte würde den Rahmen dieses Abschnitts sprengen, deshalb sei hier nur auf die entsprechenden Abschnitte 2.2.6 und 3.1.3 in [Mel97] verwiesen. Wenn ein solcher Konflikt auftritt, wirft die Methode `newMapping()` eine `AttrException`, die vom Morphismus aufgefangen und in eine `ALR_BadMappingException` umgewandelt wird, wie der Code-Ausschnitt in Abb. 6.6 andeutet.

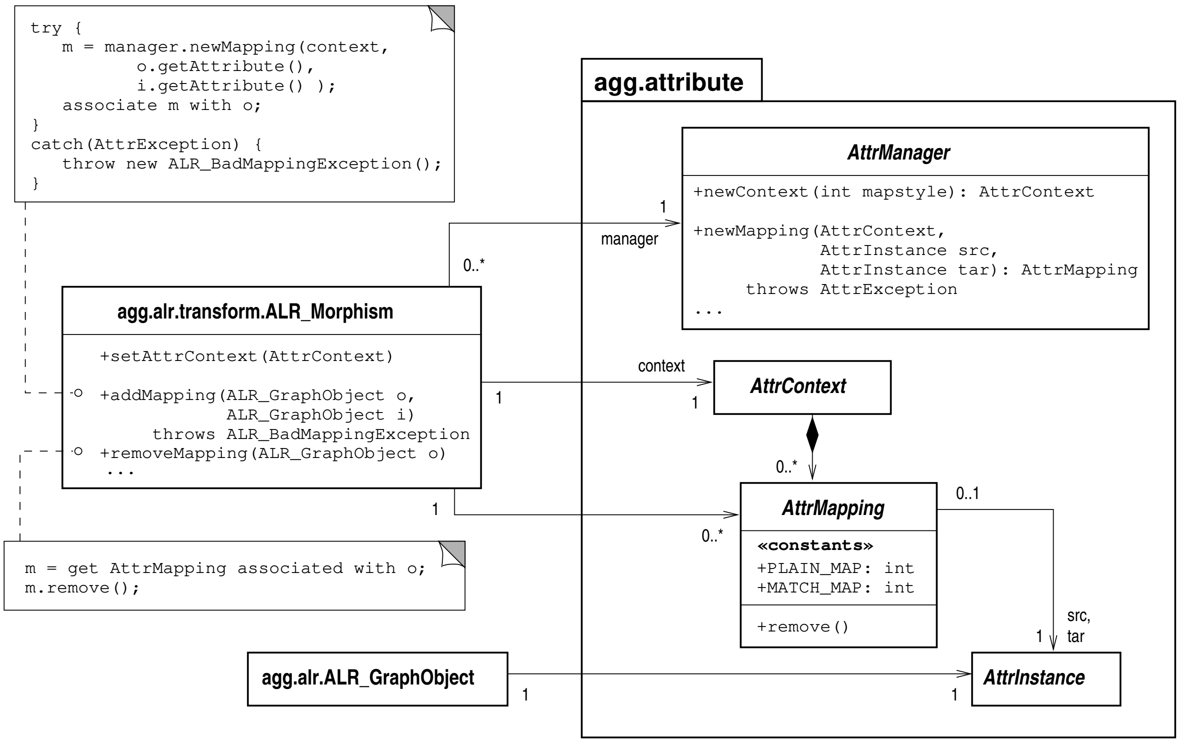


Abbildung 6.6: Verbindung zwischen `ALR_Morphism` und Attributkomponente.

Wird `newMapping()` mit einem Ansatzkontext aufgerufen, dann prüft die Attributkomponente, ob die beiden übergebenen Attribute im Sinne eines Ansatzes zueinander passen. Was das genau bedeutet, kann im Einzelfall vom konkreten Attributhandler abhängen. Im allgemeinen, und für den bisher verfügbaren Java-Expression-Handler im besonderen, heißt das: die beiden Attribute müssen entweder gleich sein, oder das erste Attribut ist eine Variable und wird durch den Wert des zweiten Attributs belegt. Sind diese Bedingungen nicht erfüllt, so wird wie gehabt eine `AttrException` ausgelöst.

`newMapping()` liefert als Rückgabewert eine Instanz der Klasse `AttrMapping`. Diese Klasse kapselt das Wissen der Attributkomponente um die Zuordnung zweier Attributinstanzen, deren zugehörige Graphobjekte durch einen Morphismus aufeinander abgebildet werden. Zu jeder Zuordnungsvorschrift $\mathtt{o} \mapsto \mathtt{i}$ zweier Graphobjekte in einem Morphismus existiert also genau eine Instanz von `AttrMapping`, die die entsprechenden Attributwerte einander zuordnet. Wird die Zuordnungsvorschrift der Graphobjekte durch einen Aufruf von `removeMapping()` entfernt, so muß auch das entsprechende Attributmapping aufgehoben werden. Diesem Zweck dient die Methode `remove()` der Klasse `AttrMapping`. Das bedeutet, daß der Morphismus sich merken muß, welches Attributmapping zu welcher Zuordnungsvorschrift auf dem Graphteil gehört. Dazu wird eine Hashtabelle angelegt, in der das Originalobjekt `o` einer jeden Zuordnungsvorschrift mit der entsprechenden Instanz von `AttrMapping` assoziiert wird.

Abschließend sei noch einmal darauf hingewiesen, daß durch die Methode `newMapping()` nicht etwa – analog zum Aufbau der Graphteilabbildung m_G durch `addMapping()` – der Datenteilmorphismus m_D eines ALR-Morphismus nach Def. 6.7 definiert wird. Der Datenteilmorphismus ist bereits implizit gegeben, sobald Attributhandler und Attributkontext festgelegt sind (vgl. Abschnitt 6.1.3). Die Attributmappings dienen vielmehr dazu, der Attributkomponente Informationen über den zugrundeliegenden Graphteilmorphismus zu verschaffen, denn aufgrund der 1 : 1-Beziehung zwischen Graphobjekten und ihren Attributinstanzen kann aus den Attributmappings die Struktur des gesamten Morphismus rekonstruiert werden. Damit stehen der Attributkomponente alle Informationen zur Verfügung, die sie benötigt, um zu entscheiden, ob sich die jeweilige Konstellation mit der zugrundeliegenden Attributträgersemantik verträgt oder ob ein Konfliktfall vorliegt.

6.2.6 Implementierung und Konsistenzsicherung von `ALR_Morphism`

Eine interessante Eigenschaft der ALR-Graphen ist die Möglichkeit, den Meta-Begriff des Morphismus auf elegante Weise selbst in einem ALR-Graphen zu repräsentieren. In Abschnitt 6.1.2 haben wir eine Konstruktion angegeben, die uns für einen ALR-Morphismus eine äquivalente Darstellung als Verfeinerungsbaum einer Kante in einem ALR-Graphen liefert. Auf dieser Konstruktion beruht die Implementierung der ALR-Morphismen im AGG-System. Die Verfeinerungsbaumdarstellung eines Morphismus bringt uns im wesentlichen zwei Vorteile:

- Wir bekommen die Möglichkeit der Meta-Modellierung im implementierten System: Wir können Regeln angeben, um Regeln oder ganze Graphgrammatiken zu transformieren. Dadurch ist es sogar vorstellbar, Funktionalitäten für das AGG-Systems im System selbst durch Graphtransformationsregeln zu implementieren.
- Dadurch, daß Morphismen und Graphen innerhalb derselben Datenstruktur repräsentiert werden, ergibt sich eine gewisse Vereinheitlichung im Umgang mit diesen Begriffen und damit die Chance, Code einzusparen. Ein Editor, der die Eingabe von ALR-Graphen unterstützt, kann beispielsweise immer auch für die Visualisierung und Manipulation von Morphismen benutzt werden, ohne daß dafür zusätzliche Operationen implementiert werden müßten.

Die Klasse `ALR_Morphism`, die wir in den vorausgegangenen Abschnitten beschrieben haben, stellt tatsächlich nur eine Art Adapter für die darunterliegende Verfeinerungsbaumdarstellung durch eine Instanz von `ALR_RefTree`. Dieser Adapter dient der Verständlichkeit und Bequemlichkeit, indem er das Interface bereitstellt, das ein Anwender von einem Morphismus erwartet. Außerdem sichert er die Konsistenz der Morphismusrepräsentation, denn nicht jeder Verfeinerungsbaum einer Kante erfüllt die von einem Morphismus verlangten Eigenschaften.

Die Sicherung der Konsistenz bezüglich der Morphismuseigenschaften ist dabei keine leichte Aufgabe, denn durch mögliche Modifikationen an den Original- und Bildgraphen eines Morphismus wird seine Konsistenz auf vielfältige Weise bedroht. Betrachten wir als Beispiel die beiden einfachen Ansatzmorphismen m_1 und m_2 in Abbildung 6.7. Von den beiden Regeln sind jeweils nur die linken Regelgraphen L_1 bzw. L_2 abgebildet. Beide Morphismen sind offensichtlich konsistent. Wenn wir nun beispielsweise das Attribut „Gewicht“ der linken Kiste im Graphen G auf „70“ ändern – das könnte etwa interaktiv in einem Grapheditor geschehen, aber auch durch einen Inplace-Transformationsschritt –, dann wird der Ansatz m_2 ungültig. Nicht jedoch m_1, denn $x = 70$ ist genauso wie $x = 50$ natürlich eine zulässige Variablenbelegung. Allerdings – wie erfährt die Attributkomponente davon, daß sich die Variablenbelegung geändert hat, wenn nicht irgendwer ihre Methode `newMapping()` aufruft? Aber nicht nur Attributänderungen sind ein Problem: Beinahe jede Modifikation im Definitions- oder Wertebereich eines

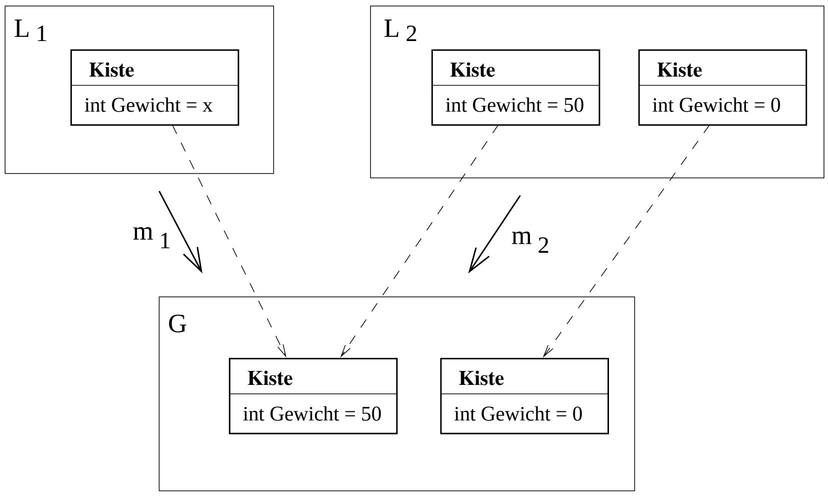


Abbildung 6.7: Zwei einfache Ansatzmorphismen.

Morphismus läßt diesen inkonsistent werden. Selbst Änderungen an den Original- und Bildgraphen, die außerhalb der Definitions- und Wertebereiche stattfinden, haben insofern Einfluß auf einen Morphismus, als sie den Suchraum für die Morphismusvervollständigung betreffen; hier muß ggf. der Berechnungszustand initialisiert werden, vgl. Abschnitt 7.3.2.

Eine einfache und pragmatische Lösung für dieses Konsistenzproblem wäre die Entscheidung, einen Morphismus für ungültig zu erklären, sobald seine Original- oder Bildgraphen manipuliert werden. In der Folge gäbe es zwei Möglichkeiten, mit einem solchen ungültigen Morphismus umzugehen: Entweder wir lassen es zu, daß Objekte der Klasse `ALR_Morphism` auch inkonsistente Morphismen repräsentieren, oder wir entscheiden uns dafür, derart inkonsistent gewordene Instanzen rigoros zu löschen. Werfen wir zunächst einen Blick auf die unangenehmen Konsequenzen der letzteren Alternative:

1. Nach einem Inplace-Transformationsschritt wäre der Ansatzmorphismus ungültig. Im Sinne der Effizienz wäre es jedoch wünschenswert, zumindest einen Teilansatz für die von der Regel erhaltenen Anteile zurückzubehalten, um bei einer späteren Anwendung derselben Regel zunächst eine Vervollständigung dieses Teilansatzes zu versuchen. In Kapitel 7 werden wir feststellen, daß gerade die Vorgabe von Teilansätzen eine Möglichkeit ist, um den exponentiellen Suchraum bei der Ansatzsuche entscheidend einzuschränken.

2. Bei der interaktiven Eingabe von Graphregeln ist es nicht akzeptabel, vom Benutzer zu verlangen, zunächst vollständig und fehlerfrei die beiden Regelgraphen einzugeben, und erst anschließend die Morphismusbeziehungen zu ergänzen. Es muß möglich sein, wechselweise den Morphismus und die Regelgraphen zu editieren, ohne bei jeder Modifikation der Graphen die bisher eingegebenen Morphismusbeziehungen zu verlieren.

Bliebe noch die Möglichkeit, die Repräsentation inkonsistenter Zustände durch die Klasse `ALR_Morphism` zuzulassen. Dann würden wir Methoden benötigen, um dynamisch die Konsistenz zu prüfen bzw. um inkonsistente Zustände zu reparieren. Das würde aber die Arbeit mit der Morphismusrepräsentation verkomplizieren, und jeder Algorithmus, der auf konsistente Morphismen angewiesen ist – z.B. Ansatzsuche und Transformation –, müßte selbst

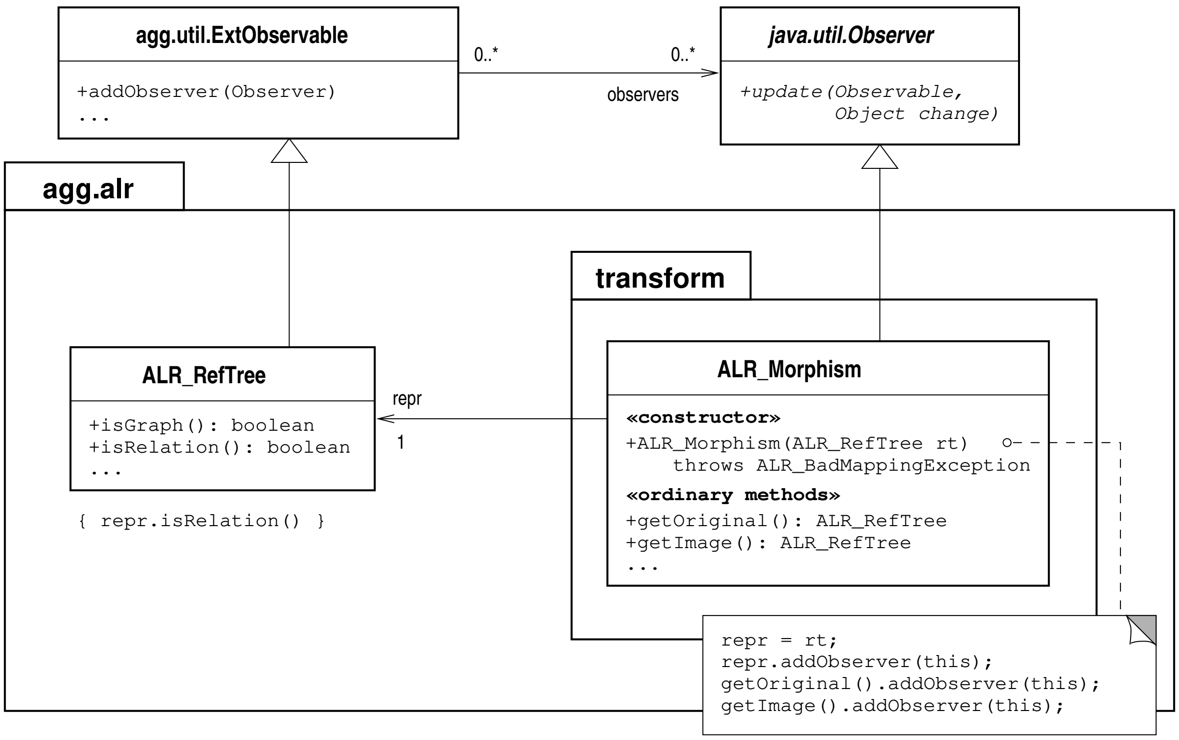


Abbildung 6.8: Implementierung von `ALR_Morphism` als Observer seiner `ALR_RefTree`-Repräsentation und seiner Original- und Bildgraphen.

die Verantwortung dafür übernehmen, daß inkonsistente Zustände entsprechend abgefangen werden. Eine derartige Abwälzung der Verantwortung auf eine beliebig große Anzahl von Anwendern bedeutet natürlich eine entsprechende Vervielfachung der Fehlerwahrscheinlichkeit. Wünschenswert ist daher vielmehr eine zentrale Konsistenzsicherung innerhalb der Morphismusimplementierung. Die Philosophie unserer Implementierung von `ALR_Morphism` lautet deshalb:

> Jede Instanz von `ALR_Morphism` repräsentiert zu jeder Zeit einen konsistenten ALR-Morphismus gemäß Def. 6.7[6]. Verletzungen der Morphismuseigenschaften durch Modifikationen an Original- und Bildgraphen des Morphismus werden automatisch repariert, indem die für die Verletzung verantwortlichen Zuordnungsvorschriften entfernt werden.

Die Wahrung der Konsistenz bewirkt also im Zweifel eine zunehmende Partialität des Morphismus. Um diese Philosophie umzusetzen, bedienen wir uns einmal mehr des Observer-Patterns. Dabei nutzen wir die Observable-Schnittstelle und die entsprechenden Änderungsinformationen der Klasse `ALR_RefTree` aus: Abbildung 6.8 zeigt durch einen Code-Ausschnitt aus dem Konstruktor von `ALR_Morphism`, wie sich jede Instanz dieser Klasse zunächst als Observer bei ihrer jeweiligen Verfeinerungsbaumrepräsentation anmeldet. Anschließend läßt sich der Morphismus aber zusätzlich als Observer seines Original- und seines Bildgraphen registrie-

[6]Mit einer in der Praxis unwesentlichen Abweichung bezüglich der Attributierungsverträglichkeit, vgl. Abschnitt 6.2.3.

ren. Auf diese Weise wird der Morphismus von jeder Änderung auf diesen beiden Graphen informiert und kann gegebenenfalls mit den entsprechenden Maßnahmen zur Sicherung der Morphismuseigenschaften reagieren.

6.2.7 Anbindung der Colimit-Bibliothek

Bei der Entwicklung der Colimit-Bibliothek [Wol97, Wol96] wurde großer Wert auf die effiziente Implementierung der Datenstrukturen und Algorithmen gelegt. In seiner Dissertation [Wol97] weist D. Wolz außerdem die Korrektheit des verwendeten Algorithmus nach. Mit der Verwendung der Colimit-Bibliothek zur Berechnung eines Graphtransformationsschritts können wir deshalb den Forderungen nach Effizienz und Korrektheit aus der Anforderungsdefinition in Abschnitt 6.2.1 entsprechen. Da die Bibliothek darüber hinaus keinerlei Einschränkungen bezüglich des verwendeten Begriffs von partiellen Morphismen macht, erfüllt sie auch unsere Forderung nach der Möglichkeit, Transformationen mit nicht-injektiven Regeln und Ansätzen durchzuführen.

Wir wollen nicht verschweigen, daß durch die nötige Umwandlung der im AGG-System verwendeten Datenstrukturen für Graphen und Morphismen in die von der Colimit-Bibliothek erwartete Repräsentation Effizienzeinbußen entstehen. Der Aufwand für die Umwandlung ist aber ebenso wie die Colimesberechnung selbst linear, so daß der Aufwand in der Summe linear bleibt. Gemäß dieser Aufwandsabschätzung kann davon ausgegangen werden, daß eine direkte Implementierung eines einfachen Pushouts auf den Datenstrukturen des AGG-Systems etwa um den Faktor 2 bis 3 schneller sein könnte als die entsprechende Realisierung unter Verwendung der Colimit-Bibliothek. Die Verwendung der Bibliothek bietet uns dabei aber immer noch den Vorteil der nachgewiesenen Korrektheit und darüber hinaus die Flexibilität, bei Bedarf auch allgemeine Colimes-Konstruktionen zu berechnen. In der Praxis spielt eine um einen Linearfaktor verbesserte Laufzeit der Pushoutberechnung aber ohnehin kaum eine Rolle, denn der Gesamtaufwand eines Transformationsschrittes wird dominiert durch das NP-vollständige Problem der Ansatzsuche.

Die Colimit-Bibliothek basiert auf der Kategorie der ALPHA-Algebren, die wir in Abschnitt 6.1.4 kurz vorgestellt haben. Die ALPHA-Algebren weisen dasselbe theoretische Attributierungskonzept auf, wie es auch der Attributierung von einfachen Graphen und ALR-Graphen zugrundeliegt, und die Bibliothek unterstützt auch diesen Attributierungsbegriff und berücksichtigt die Attribute bei der Colimes-Berechnung. Für die Realisierung unserer Vorstellung vom Umgang mit Attributen bei einem Graphtransformationsschritt ist die von der Bibliothek angebotene Attributfunktionalität aber aus den folgenden Gründen nicht ausreichend:

1. Die Colimit-Bibliothek behandelt die Attributierung streng nach den Vorgaben der Theorie. Dies ist grundsätzlich zunächst als Vorteil zu werten, bedeutet aber in diesem Fall, daß die regelbasierte Änderung von Attributen an zu erhaltenden Objekten nur mit Hilfe einer expliziten Modellierung von Attributträgern realisiert werden könnte (vgl. Abschnitt 6.1.3).
2. Die Bibliothek bietet keine Unterstützung für das Rechnen mit Variablen und für das Auswerten von Termen.

Deshalb lassen wir in unserer Implementierung nur den Graphteil einer Transformation durch

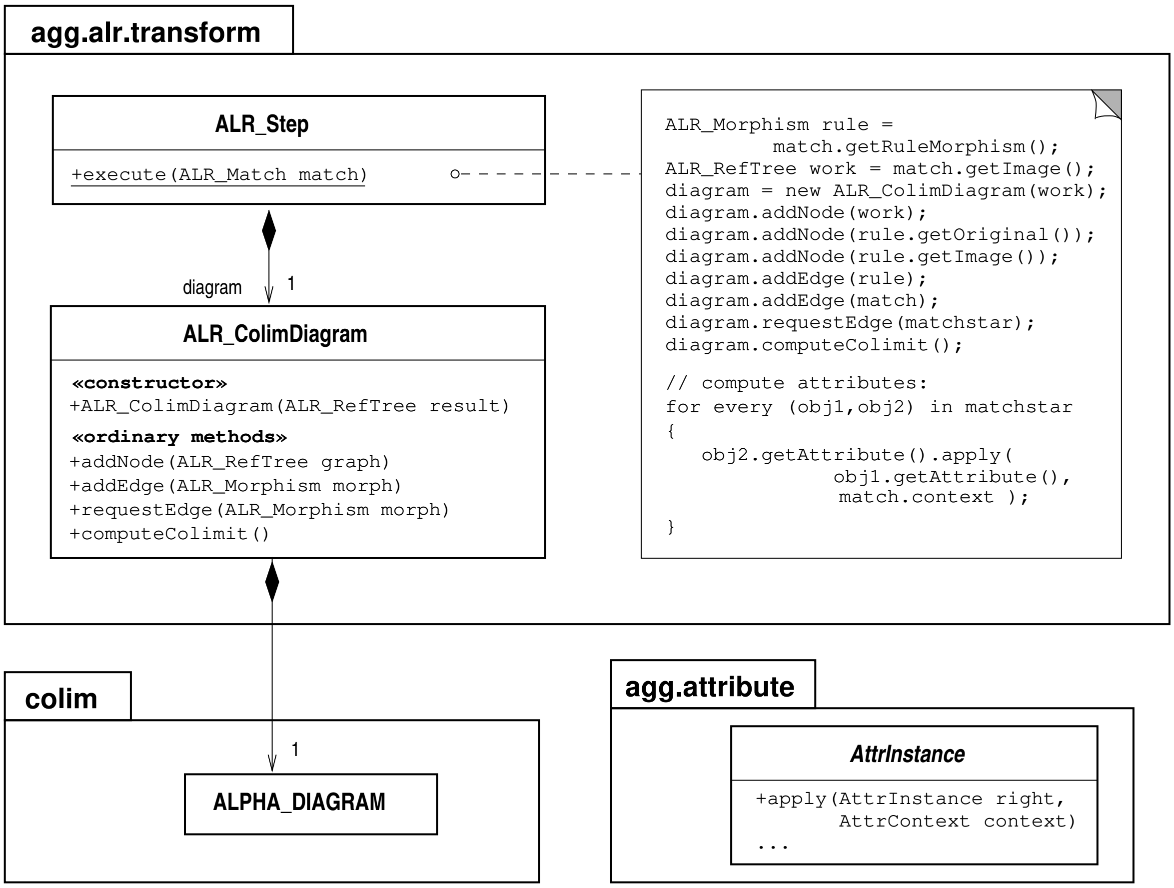


Abbildung 6.9: Implementierung eines Graphtransformationsschrittes.

die Colimit-Library berechnen, während die Berechnung der Attribute separat von der Attributkomponente [Mel97] durchgeführt wird. Die Attributkomponente bietet ein komfortables Konzept für das Rechnen mit Variablen in Graphregeln, wobei die Variablen durch Ansatzmorphismen belegt werden. Als Nachteil der separaten Attributbehandlung kann angesehen werden, daß wir bezüglich der Attribute nicht mehr vom Korrektheitsnachweis der Colimit-Bibliothek profitieren können, und daß die verwendete Attributkomponente die Attribute nur für einfache Pushouts berechnen kann.

Das Klassendiagramm in Abbildung 6.9 zeigt die Klassen und Packages, die an der Realisierung eines Graphtransformationsschrittes beteiligt sind. Auf der untersten Implementierungsebene befindet sich das Package `colim`, das die Colimit-Bibliothek beherbergt. Wir benutzen daraus die Klasse `ALPHA_DIAGRAM`, die die Berechnung von Colimiten auf beliebigen Diagrammen in der Kategorie Alpha realisiert. `ALR_ColimDiagram` führt die Berechnung von Colimiten beliebiger Diagramme in der Kategorie der typisierten, nicht attributierten ALR-Graphen auf den Alpha-Fall zurück. Die Schnittstelle der Klasse `ALR_Step` schließlich ist uns schon aus der Beschreibung der Anwendungsschnittstelle in Abschnitt 6.2.2 bekannt; die Berechnung des Pushouts, das einen Graphtransformationsschritt beschreibt, wird hier als Spezialfall über `ALR_ColimDiagram` implementiert. Anschließend berechnet `ALR_Step` durch entsprechende Aufrufe von Methoden der Attributkomponente die Attribute für das Trans-

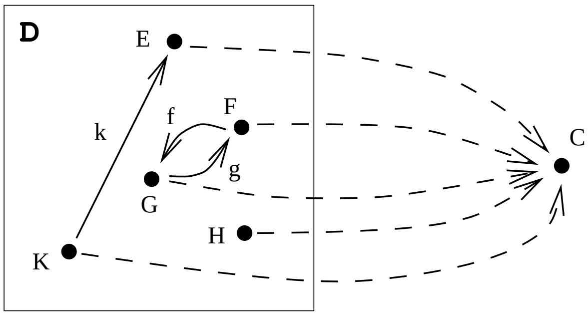


Abbildung 6.10: Beispiel für ein allgemeines Colimes-Diagramm.

formationsergebnis.

Um die Schnittstelle der Klasse `ALR_ColimDiagram` zu verstehen, müssen wir zunächst klären, was wir unter einem Colimes-Diagramm verstehen wollen. Ein *Diagramm* in einer Kategorie haben wir in Abschnitt 6.1.1 anschaulich als einen Graphen beschrieben, dessen Knoten Objekte und dessen Kanten Morphismen in der Kategorie repräsentieren. Ein *Colimes-Diagramm* ist dann ein Diagramm, das die Ausgangssituation und ggf. auch das Ergebnis einer Colimes-Bildung darstellt. Abbildung 6.10 zeigt eine solche Situation: Der Colimes eines beliebigen Diagramms D in einer Kategorie ist ein Knoten C im Diagramm – also ein Objekt in der Kategorie – zusammen mit je einer Kante von jedem Knoten des Ausgangsdiagramms zum Knoten C. Diese Kanten, im Bild gestrichelt gezeichnet, sind wiederum Morphismen in der Kategorie und werden *Colimes-Morphismen* genannt, während das Objekt C als *Colimes-Objekt* bezeichnet wird. Dazu sei angemerkt, daß die Colimes-Morphismen nach der kategorientheoretischen Definition des Colimes bestimmte Eigenschaften haben, und daß der Colimes nicht in jeder Kategorie zu jedem Diagramm existieren muß. Daß in der Kategorie $\underline{\mathrm{ALR}^{\mathrm{L}}}$ der typisierten, nicht attributierten ALR-Graphen alle Colimiten existieren, darauf haben wir in der Fußnote 4 auf Seite 61 bereits hingewiesen. Da uns an dieser Stelle aber letztlich nur das Pushout als Spezialfall des Colimes interessiert, verweisen wir für weitere Einzelheiten bezüglich des Colimes-Begriffs auf die einschlägige Literatur, z.B. [AM75, AHS90].

Kommen wir also zurück zur Schnittstellenbeschreibung der Klasse `ALR_ColimDiagram`. Zunächst einmal bietet die Klasse Methoden, um das Diagramm auszuzeichnen, dessen Colimes berechnet werden soll. Dies sind die Methoden `addNode()` und `addEdge()`, mit denen Knoten bzw. Kanten in das Ausgangsdiagramm aufgenommen werden. Für das Ausgangsdiagramm D der Colimesberechnung in Abb. 6.10 wäre zum Beispiel für jeden Knoten $X \in \{E, F, G, H, K\}$ einmal die Methode `addNode(`X`)` und für jede Kante $x \in \{f, g, k\}$ anschließend `addEdge(`x`)` aufzurufen. `ALR_ColimDiagram` berechnet Colimiten auf ALR-Graphen, folgerichtig sind die Kanten im Diagramm Instanzen von `ALR_Morphism`, während die Knoten als `ALR_RefTree` typisiert sind mit der zusätzlichen Bedingung, daß das Prädikat `isGraph()` wahr ist.

Nachdem das Diagramm auf diese Weise aufgebaut wurde, steht der Berechnung des Colimes durch den Aufruf `computeColimit()` fast nichts mehr im Wege. Die Frage ist nur: Wohin mit dem Colimes-Objekt? Und wollen wir wirklich jeden Colimes-Morphismus berechnen lassen?

Die erste Frage wird durch das obligatorische Konstruktorargument `result` beantwortet:

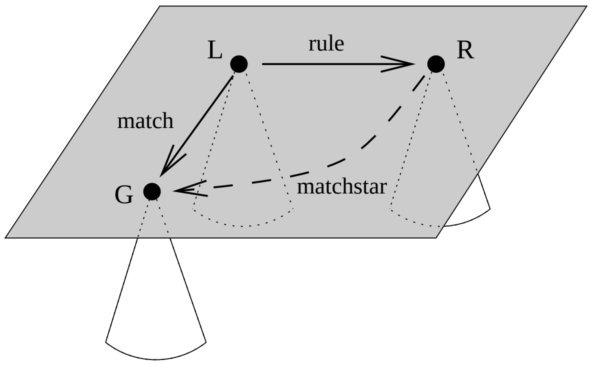


Abbildung 6.11: Beispiel für ein Inplace-Pushout-Diagramm auf ALR-Graphen.

Der übergebene ALR-Graph repräsentiert am Ende der Berechnung das Colimes-Objekt. Für das Beispiel aus Abb. 6.10 etwa würden wir einen neuen ALR-Graphen C als Platzhalter für das Colimes-Objekt erzeugen, und mit diesem leeren Graphen als Konstruktorargument eine Instanz von `ALR_ColimDiagram` erzeugen. Erst jetzt käme der Schritt, den wir bereits vorweggenommen haben, nämlich der Aufbau des Diagramms über die Methoden `addNode()` und `addEdge()`. Es ist aber auch erlaubt, einen Graphen aus dem Ausgangsdiagramm gleichzeitig als Repräsentant für das Colimes-Objekt zu bestimmen. Sagen wir beispielsweise, wir erzeugen eine Instanz von `ALR_ColimDiagram` mit dem Graphen G aus Abb. 6.10 als Konstruktorargument, und bauen anschließend das Diagramm auf wie zuvor, führen also insbesondere auch `addNode(`G`)` aus. Der Effekt eines anschließenden Aufrufs von `computeColimit()` ist eine *in-place*-Berechnung des Colimes-Objekts in G. Die Realisierung dieses Inplace-Effekts ist einfach: intern wird zunächst wie gewöhnlich ein neues Objekt C erzeugt und der Colimes-Morphismus $m : G \to C$ bestimmt. Anschließend werden alle Knoten und Kanten aus G gelöscht, die nicht im Definitionsbereich von m liegen, und alle diejenigen von C nach G kopiert, die nicht zum Wertebereich des Morphismus gehören. Zuletzt werden C und m verworfen.

Bleibt noch die Frage zu beantworten, welche Colimes-Morphismen tatsächlich berechnet werden sollen. Die Methode `requestEdge()` bietet hier die Möglichkeit, die gewünschten Colimes-Morphismen explizit anzufordern. Betrachten wir erneut das Beispiel aus Abb. 6.10 unter der Annahme, daß C als Platzhalter für das Colimes-Objekt gewählt wurde. Wenn wir nun erreichen wollen, daß z.B. der Colimes-Morphismus $c_K : K \to C$ berechnet wird, so müssen wir im Anschluß an den üblichen Diagrammaufbau einen leeren Morphismus c_K erzeugen und durch den Aufruf `requestEdge(`c_K`)` an die Diagramminstanz übergeben.

Die Klasse `ALR_Step` implementiert die Berechnung eines Graphtransformationsschrittes durch ein Pushout in der Kategorie der attributierten ALR-Graphen. Das Pushout ist aber nur ein Spezialfall des allgemeinen Colimes-Begriffs insofern, als beim Pushout das Ausgangsdiagramm eine festgelegte Struktur haben muß: Drei Knoten G, L, R und zwei Kanten $match$ und $rule$, die wie in Abbildung 6.11 gerichtet sind. Wir wollen nun anhand des Code-Ausschnitts aus Abb. 6.9 nachvollziehen, wie die Methode `execute()` von `ALR_Step` ein solches Pushout mit Hilfe der Funktionalität von `ALR_ColimDiagram` konstruiert. Als einziges Argument bekommt `execute()` den Ansatzmorphismus `match` übergeben, von dem zunächst der zugehörige Regelmorphismus `rule` und der Arbeitsgraph `work` erfragt wird. Dann wird eine Instanz von `ALR_ColimDiagram` erzeugt, und weil wir den Arbeitsgraphen „in place“ transformieren wollen, übergeben wir `work` als Konstruktorargument. In den folgenden Schritten wird das eigentliche Diagramm aufgebaut, und mit dem Aufruf `requestEdge(matchstar)` wird die Berechnung des Colimes-Morphismus von der rechten Regelseite in den Ergebnisgraphen angefordert, den wir für die separate Attributberechnung benötigen; Abb. 6.11 veranschaulicht die Konstellation. Anschließend führt `computeColimit()` die tatsächliche Berechnung von Colimes-Objekt und -Morphismus aus. Die Konstruktion des Pushouts in der Kategorie der nicht attributierten ALR-Graphen ist an dieser Stelle abgeschlossen; es folgt die separate Berechnung der Attribute des Ergebnisgraphen durch die Attributkomponente. Da wir den graphischen Anteil „in place" berechnet haben, haben alle Graphobjekte des Arbeitsgraphen, die außerhalb des Wertebereichs von `match` liegen, ihre Attribute unverändert behalten und müssen nicht weiter bearbeitet werden. Diejenigen Graphobjekte, die zwar vom Ansatz erfaßt, aber von der Regel gelöscht wurden, brauchen bezüglich der Attributierung ebenfalls nicht mehr berücksichtigt zu werden. Interessant sind also nur diejenigen Knoten und Kanten des Arbeitsgraphen, die von der Regel neu erzeugt oder aber erhalten und dabei potentiell modifiziert wurden. Aus den kategoriellen Eigenschaften eines Pushouts ergibt sich aber, daß genau diese Graphobjekte im Wertebereich des `matchstar`-Morphismus liegen, und daß die Attributwerte ihrer Urbilder unter dem Morphismus gerade die durch die Regel spezifizierten Ergebniswerte beschreiben. Im Prinzip müssen also nur die Attributwerte der Graphobjekte aus dem Wertebereich des `matchstar`-Morphismus mit den Attributwerten ihrer Urbilder überschrieben werden. Da die Werte von Attributen in der rechten Regelseite aber Variablen enthalten können, müssen sie zuvor noch unter der durch den Ansatzmorphismus gegebenen Variablenbelegung ausgewertet werden. Genau dieses Vorgehen wird in dem Programmausschnitt in Abb. 6.9 in einer Schleife über alle Abbildungspaare von je zwei Graphobjekten `(obj1,obj2)` aus `matchstar` realisiert: In jedem Schleifendurchlauf bewirkt ein Aufruf von `apply`, daß das Attribut des Graphobjekts `obj2` aus dem Arbeitsgraphen durch das Attribut des entsprechenden Graphobjekts `obj1` der rechten Regelseite ersetzt wird. Dabei wird das Attribut von `obj1` unter der Variablenbelegung ausgewertet, die durch den Attributkontext des Ansatzes gegeben ist.

Bemerkung 6.13 (*Explizite Diagrammebene in ALR-Graphen*) In Abbildung 6.11 ist eine weitere interessante Eigenschaft von ALR-Graphen zu erkennen: Da ALR-Graphen Verfeinerungsbäume von Knoten sind und ALR-Morphismen auch in der Implementierung durch Verfeinerungsbäume von Kanten dargestellt werden, ergibt sich bei jeder Colimeskonstruktion automatisch eine explizite Darstellung des Diagrammgraphen. Ein Editor, der das ebenenweise Anzeigen von ALR-Graphen unterstützt, eignet sich dadurch sehr gut zur Veranschaulichung kategorieller Konstruktionen. △

Kapitel 7

Ansatzsuche

In diesem Kapitel beschäftigen wir uns mit dem Problem der Ansatzsuche auf attributierten ALR-Graphen. Im Anschluß an eine Analyse naiver Lösungsverfahren formulieren wir die Ansatzsuche als Constraint Satisfaction Problem (CSP). Um diese Formulierung in der Implementierung umsetzen zu können, entwerfen wir ein Framework für die Lösung von allgemeinen CSPs. Als Lösungsalgorithmus beschreiben und implementieren wir ein Backjumping-Verfahren, das eine optimierte Form des einfachen Backtracking darstellt.

Bei der Realisierung der Ansatzsuche wird dabei nicht der Versuch unternommen, die bestmögliche Optimierung des eigentlichen Algorithmus zu erzielen; stattdessen liegt der Schwerpunkt auf der Entwicklung der Konzepte für die Formulierung der Ansatzsuche als CSP, sowie auf einer soliden softwaretechnischen Realisierung, die gewährleistet, daß dieser wichtige Teil des Systems leicht gewartet, optimiert oder ausgetauscht werden kann. Damit soll die Grundlage für weitere Arbeiten geschaffen werden, die sich dann auf die Sichtung der Literatur zum Thema Lösungsalgorithmen für CSPs und auf die Auswahl, die Optimierung und den Vergleich geeigneter Verfahren konzentrieren können.

7.1 Begriffe und Konzepte

Das Problem der Ansatzsuche besteht darin, ein homomorphes Bild der linken Regelseite im Arbeitsgraphen wiederzufinden. Dieses Problem ist in der Graphen- bzw. Komplexitätstheorie bekannt unter dem Namen *Subgraph Homomorphism Problem*, und man weiß, daß dieses Problem NP-vollständig ist [Meh84]. Die Transformation selbst, also die Umformung des Arbeitsgraphen gemäß der Regelvorschrift an einem gegebenen Ansatz, kann dagegen im allgemeinen in linearer Zeit durchgeführt werden. Das bedeutet, daß der Zeitaufwand eines *Graphtransformationsschrittes* durch die Ansatzsuche bestimmt wird.

Für die Praxis ist der exponentielle Aufwand, der sich im allgemeinen Fall aus der NP-Vollständigkeit für die Ansatzsuche ergibt, natürlich indiskutabel. Man versucht deshalb, zumindest den durchschnittlichen Aufwand auf polynomielles Niveau zu senken, beispielsweise indem man spezielle Graphklassen und darauf abgestimmte Heuristiken einsetzt.

Aus diesen Vorbetrachtungen wird deutlich, wie groß die Bedeutung einer effektiven Strategie zur Ansatzsuche für die praktische Einsetzbarkeit eines Graphtransformationssystems ist. Weiterhin ergibt sich, daß effektive Algorithmen zur Ansatzsuche üblicherweise konsequent die speziellen Eigenschaften des jeweils verwendeten konkreten Graphmodells ausnutzen. Hierin ist auch die Tatsache begründet, daß für neu entwickelte Implementierungen von Graphtransformation regelmäßig neue Arbeiten bezüglich der Ansatzsuche unternommen werden (z.B. [Bie97, Joh92]), anstatt existierende, in der Praxis bewährte effektive Algorithmen (aktuell z.B. [Zün96, Dör95]) zu übernehmen.

Wir werden im folgenden Abschnitt 7.1.1 zunächst erklären, was wir unter dem Begriff „Ansatzsuche" genau verstehen wollen. In den weiteren Unterabschnitten betrachten wir dann verschiedene Algorithmen für die Lösung von komplexen kombinatorischen Problemen, für die die Ansatzsuche ein Beispiel ist. Dabei arbeiten wir uns schrittweise von naiven Herangehensweisen über die Analyse ihrer Schwächen zu einem effektiveren Algorithmus vor. Außerdem geben wir eine kurze Einführung in den Bereich der *Constraint Satisfaction Probleme*, der einen geeigneten Rahmen an Begriffen und Methoden für die Lösung unseres Problems zu geben verspricht.

7.1.1 Ansatzsuche ist Morphismusvervollständigung

Im einfachsten Fall kann man definieren, daß jeder totale Morphismus zwischen der linken Seite einer Regel und dem Arbeitsgraphen ein Ansatz ist. In diesem Fall sind Regelmorphimus und rechte Regelseite für die Ansatzsuche irrelevant. Dann kann man sagen, die Aufgabe der Ansatzsuche bestehe darin, einen ggf. leeren partiellen Morphismus zu vervollständigen. Diese Vorstellung liegt allen folgenden Betrachtungen zugrunde.

Ein solcher Algorithmus zur Morphismusvervollständigung ist darüber hinaus nicht nur für die „eigentliche" Suche nach einem Ansatz zwischen Regel und Arbeitsgraph einsetzbar, sondern auch für die Überprüfung von negativen Anwendungsbedingungen oder für das Erweitern von Interface-Morphismen auf lokale Graphen im Rahmen von verteilter Graphtransformation [Tae96].

Häufig wird der Ansatzbegriff aber weiter eingeschränkt[1], so daß z.B. nur solche Morphismen als gültige Ansätze betrachtet werden, die

- injektiv sind,
- die sog. „gluing-condition" bezüglich einer gegebenen Regel erfüllen, wodurch in Rahmen der Single-Pushout-Theorie der Effekt einer Double-Pushout-Transformation simuliert werden kann, oder
- vorgegebene negative Anwendungsbedingungen erfüllen [HHT96].

Wir wollen es an dieser Stelle mit der pragmatischen Feststellung bewenden lassen, daß alle diese Ansatzbegriffe Spezialfälle sind, die sich in Kombination mit nachgeschalteten Tests ebenfalls durch den allgemeinen Algorithmus zur Morphismusvervollständigung erschlagen lassen. In der Praxis werden sich aber häufig wesentliche Effizienzvorteile erreichen lassen, wenn die Einschränkungen für den jeweiligen Ansatzbegriff direkt in den Algorithmus integriert werden. Dies führt zu der Anforderung an den Systementwurf, daß ggf. auch mehrere

[1] vgl. Def. 2.11: *Ansatzeinschränkungen*

Versionen des Algorithmus koexistieren sollten; vor allem aber wird dadurch die Forderung nach geeigneter Kapselung und Modularisierung der Ansatzsuche unterstrichen.

7.1.2 Generate & Test

Dem einfachsten Algorithmus zur Ansatzsuche liegt eine kombinatorische Sicht auf das Problem zugrunde: Gegeben seien zwei einfache Graphen L und G, die Anzahl ihrer Graphobjekte betrage $|L| = k$ bzw. $|G| = n$; der Einfachheit halber wollen wir hier nicht zwischen Knoten und Kanten unterscheiden. Dann gibt es maximal n^k Möglichkeiten (*Variationen*), die Objekte von L auf die Objekte von G abzubilden. Ist man lediglich an injektiven Abbildungen interessiert, reduziert sich die Anzahl der Variationen auf $\frac{n!}{(n-k)!}$. Alle diese Abbildungen werden nun eine nach der anderen konstruiert und auf ihre Homomorphieeigenschaft hin überprüft. Die Anzahl der Tests, die nötig sind, um einen Homomorphismus zu identifizieren, ist dabei abhängig vom konkreten Graphmodell. Für einfache, typisierte Graphen beispielsweise brauchen wir für jedes abgebildete Objekt einen Test auf Typverträglichkeit und für jede Kante zusätzlich zwei Tests auf Verträglichkeit mit den Operationen *source* bzw. *target*.

Dieser Algorithmus ist allerdings so einfach wie ineffizient. Das wird besonders deutlich in dem Fall, wenn keine der n^k Abbildungsvarianten ein Homomorphismus ist, wenn also kein Ansatz existiert. Um nämlich zu diesem Ergebnis zu kommen, muß der Generate-&-Test-Algorithmus grundsätzlich alle n^k Abbildungsvarianten überprüfen. Daß dieser Aufwand unnötig ist, läßt sich anhand des Diagramms in Abb. 7.1 leicht nachvollziehen. Das Diagramm stellt den Suchraum aller möglichen Abbildungen $m : L \to G$ zwischen zwei Graphen L und G dar, wobei die Objekte der Graphen mit l_1 bis l_k bzw. g_1 bis g_n bezeichnet werden. Jeder Weg im Diagramm von *Start* nach *Stop* entlang der Pfeilrichtung entspricht einer der n^k möglichen Abbildungen, wenn wir jedes g_{i_h} auf dem Weg als Bild dem entsprechenden Objekt l_h zuordnen: $m : l_h \mapsto g_{i_h}$.

Gehen wir nun davon aus, daß in der *Auswahlordnung*[2] $l_1 \dots l_k$ aus Abb. 7.1 der Ordnungsindex jeder Kante größer ist als die Indizes ihrer Source- und Target-Knoten. Dann können wir die Homomorphieeigenschaft einer gegebenen Abbildung überprüfen, indem wir entlang der Auswahlordnung Objekt für Objekt auf die Einhaltung der Bedingungen aus der entsprechenden Morphismusdefinition testen. Wird bei der Überprüfung der Abbildungsvorschrift für ein l_h festgestellt, daß eine der Homomorphiebedingungen verletzt ist, dann ist die aktuelle Abbildung kein Homomorphismus, und es kann mit der Untersuchung der nächsten Abbildung fortgefahren werden. Mehr noch: Alle weiteren Abbildungen, die sich in den Abbildungsvorschriften für l_1 bis l_h nicht von der gerade getesteten unterscheiden, können ebenfalls keine Homomorphismen sein und müssen nicht mehr untersucht werden; der Suchraum verkleinert sich um n^{k-h} Abbildungen! Der Generate-&-Test-Algorithmus aber berücksichtigt diese Optimierung nicht und durchsucht stur den gesamten Abbildungsraum. Dies ist die auffälligste Schwäche dieses Algorithmus, der wir im nächsten Abschnitt mit einem einfachen Backtracking-Verfahren begegnen werden.

[2] Die Namen *Suchordnung* und *Auswahlordnung* gehen auf [Arl90] zurück.

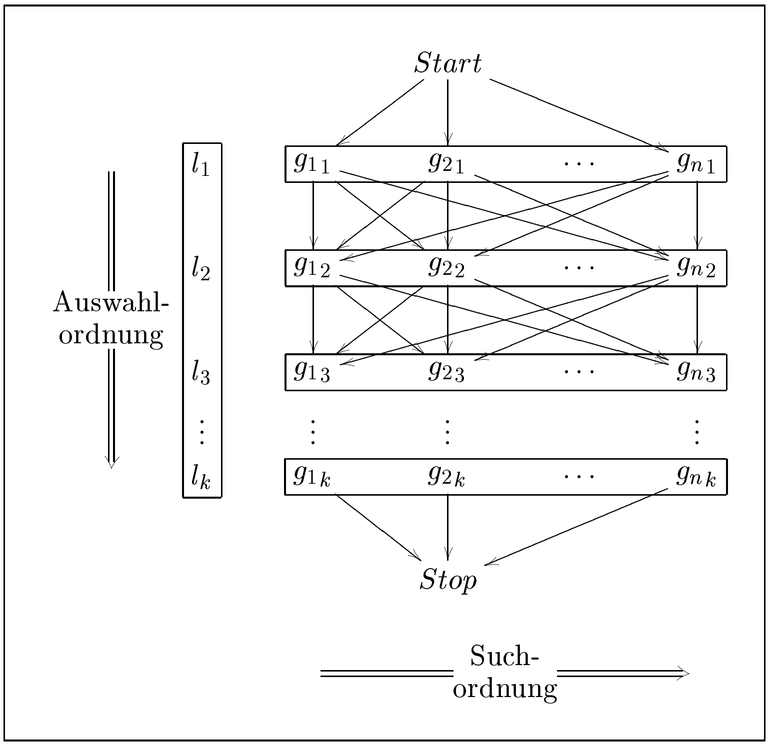


Abbildung 7.1: Der Suchraum möglicher Abbildungen.

7.1.3 Einfaches Backtracking

Im Gegensatz zum Generate-&-Test-Algorithmus wird beim einfachen Backtracking nicht erst eine totale Abbildung konstruiert, um sie anschließend auf ihre Homomorphieeigenschaften zu prüfen. Stattdessen wird eine anfangs leere Abbildung Schritt für Schritt um Abbildungsvorschriften für $l_1, l_2, \ldots, l_k$ ergänzt und nach jedem Schritt sofort auf Verletzung von Homomorphiebedingungen getestet. Dabei müssen wir wiederum voraussetzen, daß die *Auswahlordnung* Kanten hinter ihren Source- und Target-Knoten einordnet. Anderenfalls könnte die Situation eintreten, daß zu dem Zeitpunkt, wo das Verfahren der Kante ein Bild zuordnet, für deren Source- und Target-Objekte noch keine Bilder definiert sind. Der anschließende Homomorphietest, der die Verträglichkeit des Morphismus mit den Source- und Target-Operationen verlangt, würde grundsätzlich scheitern.

Wenn der Algorithmus nun eine Verletzung der Homomorphiebedingungen feststellt, wird ein *Backtracking*-Schritt durchgeführt: Die zuletzt ergänzte Abbildungsvorschrift $m : l_h \mapsto g_{ih}$ wird zurückgenommen und durch die gemäß der *Suchordnung* nächste Zuordnungsmöglichkeit $m : l_h \mapsto g_{i+1h}$ ersetzt. Wenn für l_h keine weiteren Zuordnungsmöglicheiten mehr bestehen, wird auch die Abbildungsvorschrift für l_{h-1} zurückgenommen und durch den nächsten Kandidaten ersetzt usw. Dieses Verfahren entspricht einer *Depth-First*-Suche in dem in Abb. 7.1 dargestellten Suchraum, der einen zusammengefalteten Baum repräsentiert.

Mit dem Backtracking-Verfahren kann der tatsächlich zu durchsuchende Teil des Suchraums gegenüber dem Generate-&-Test-Algorithmus im allgemeinen erheblich reduziert werden, denn durch jeden Backtracking-Schritt, der eine Zuordnungsvorschrift für ein l_h zurücknimmt, verringert sich die Anzahl der zu prüfenden Abbildungsvarianten um n^{k-h}. Mit

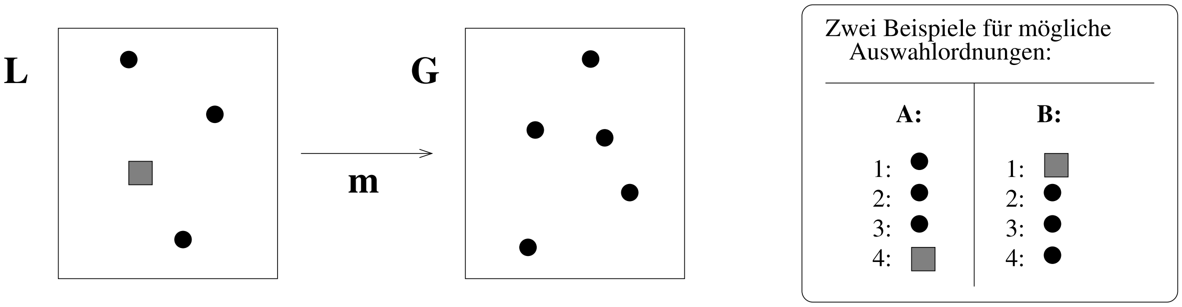


Abbildung 7.2: Beispiel zur Auswirkung der Auswahlordnung.

anderen Worten: mit jedem Backtracking-Schritt entledigen wir uns eines exponentiellen Teilproblems. Dabei fällt auf, daß die Größe des herausfallenden Teilproblems mit fallendem h exponentiell wächst: je früher ein Homomorphietest fehlschlägt, desto größer der Optimierungseffekt. Das bedeutet, daß die Auswahlordnung einen entscheidenden Einfluß auf die Laufzeit des Backtracking-Verfahrens haben kann. Im Bereich der Constraint Satisfaction Algorithmen ist diese Tatsache unter dem Begriff *First-Fail-Prinzip* bekannt. Abbildung 7.2 zeigt ein einfaches, aber drastisches Beispiel für die Wirkung dieses Prinzips. Form und Farbe der Knoten in diesem Beispiel symbolisieren eine Typisierung der Objekte. Offensichtlich kann es keinen Ansatz $m : L \to G$ geben, weil der quadratische Knoten aus L nicht typverträglich abgebildet werden kann. Vergleichen wir nun die Anzahl der Homomorphietests, die der Backtracking-Algorithmus unter Verwendung der Auswahlordnungen A und B benötigt. Ein Homomorphietest wird für jede neu in die Abbildung genommene Zuordnungsvorschrift durchgeführt:

Suchordnung A: Die homomorphe Abbildung der ersten drei Knoten ist in jeder Situation möglich, hier tritt also nie ein Backtracking-Schritt auf. Erst beim Versuch, den vierten, quadratischen Knoten zuzuordnen, schlägt der anschließende Homomorphietest mit Sicherheit fehl und ein Backtracking-Schritt schließt sich an. Die Anzahl der dadurch ausgeschlossenen Abbildungsvarianten beträgt $|G|^{|L|-4} = 5^0 = 1$, was bedeutet, daß der Algorithmus alle 5^4 Varianten testen, also den gesamten Suchraum traversieren wird. Wenn man die schematische Darstellung des Suchraums aus Abb. 7.1 zu einem Baum expandiert, kann man nachvollziehen, daß der Algorithmus mit seiner Depth-First-Strategie dabei jeden Knoten des Suchbaums genau einmal besucht. Jeder besuchte Knoten entspricht aber einer neu hinzuzunehmenden Zuordnungsvorschrift, die einen Homomorphietest nach sich zieht. Bei einer Anzahl von

$$\sum_{i=1}^{|L|} |G|^i = \sum_{i=1}^{4} 5^i$$

Knoten im Suchbaum werden also insgesamt 780 Homomorphietests durchgeführt, ehe der Algorithmus mit dem Ergebnis „es existiert kein Ansatz“ terminiert.

Suchordnung B: Hier wird gleich im ersten Schritt versucht, den quadratischen Knoten abzubilden. Der anschließende Homomorphietest scheitert zwangsläufig, womit gleich $5^3 = 125$ Abbildungsvarianten ausgeschlossen werden. Ein Backtracking-Schritt schließt sich an, und die weiteren vier Möglichkeiten, den quadratischen Knoten abzubilden, scheitern ebenfalls jeweils am Homomorphietest. So ist bereits nach fünf fehlgeschlagenen Tests der gesamte Suchraum von 5^4 Abbildungen durchforstet mit dem Ergebnis, daß kein Ansatz existiert.

Obwohl das Backtracking-Verfahren schon deutlich effizienter arbeitet als Generate-&-Test, ist seine Laufzeit für die meisten nichttrivialen Probleme immer noch exponentiell. Die Einfachheit des Prinzips bewirkt gleichzeitig eine „Sturheit" bei der Traversierung des Suchraums, die das Verfahren immer wieder in Sackgassen geraten läßt, die eigentlich bereits hätten ausgeschlossen werden können. Es bleibt also noch sehr viel Raum für Optimierungen, und in der Fachliteratur finden sich diverse Arbeiten zu diesem Thema. Die meisten dieser Arbeiten behandeln die Optimierung von Backtracking-Algorithmen aber vor dem Hintergrund der *Constraint Satisfaction Probleme*. Aus diesem Grund geben wir im nächsten Abschnitt eine kurze Einführung in dieses Gebiet, bevor wir dann im Rahmen der Constraint Satisfaction Algorithmen auf die Schwächen des einfachen Backtracking-Verfahrens und deren Optimierung zurückkommen.

7.1.4 Constraint Satisfaction Probleme

Viele Probleme der Informatik, insbesondere im Bereich der Künstlichen Intelligenz, lassen sich als Spezialfälle des *Constraint Satisfaction Problems (kurz: CSP)* formulieren und lösen. Maschinelles Sehen, Truth Maintenance, Stundenplanprobleme (Scheduling) und Spieltheorie sind nur einige Beispiele für Bereiche, in denen Constraint-basierte Methoden erfolgreich eingesetzt werden. Wenn ein Problem einmal als CSP formuliert vorliegt, hat das den Vorteil, daß zu seiner Lösung auf die große Auswahl an Methoden zurückgegriffen werden kann, die im Bereich der Constraint Satisfaction Probleme bereits untersucht worden sind. Vom konkreten Ausgangsproblem kann dann weitgehend abstrahiert werden.

Gerade bei schwierigen Problemen, für die keine effizienten exakten Lösungsverfahren bekannt sind, fallen die Vorteile der Darstellung als CSP ins Gewicht, denn dann sind Backtracking-Strategien und Heuristiken gefragt, wie sie zur Lösung von CSPs schon hundertfach entwickelt, erprobt, beschrieben und verglichen wurden, z.B. in [Kon94, GMP$^+$96, Smi97]. In einem späteren Abschnitt dieser Arbeit werden wir aus diesem Grund das Graph-Matching-Problem als CSP formulieren. An dieser Stelle folgt aber zunächst eine kurze allgemeine Einführung in die für uns wichtigsten Begriffe aus diesem Gebiet. Die verwendeten Definitionen folgen weitgehend [DvB97]; das Constraint Satisfaction Problem wird dort aber als *Constraint Network* bezeichnet.

Definition 7.1 (*Constraint Satisfaction Problem*)
Ein *Constraint Satisfaction Problem (CSP)* besteht aus

- einer Menge von *Variablen* $X = \{x_1, \dots, x_n\}$,
- einem Definitionsbereich D_i von möglichen Werten für jede Variable x_i (auch *Domain* genannt) und
- einer Menge von Relationen $\{C_{S_1}, \dots, C_{S_t}\}$, den *Constraints*. Die S_i sind dabei beliebige n-Tupel von Variablen aus X.

△

Üblicherweise wird dabei vorausgesetzt, daß sowohl die Variablenmenge als auch die Domains, die Menge der Constraints und die Relationen selbst endlich sind. Für die meisten Probleme ist dies ausreichend, und so werden auch wir in dieser Arbeit keine unendlichen Mengen benötigen. [DvB97] weisen jedoch darauf hin, daß sich ihre Definitionen und Algorithmen auch auf unendliche Relationen und Domains anwenden lassen.

Vor dem Hintergrund der Definition eines CSPs mit seinem Domain-Begriff definieren wir nun den zentralen Begriff des Constraints:

Definition 7.2 (*Constraint*)
Ein *Constraint*[3] C_S über einem Tupel von Variablen $S = (x_1, \ldots, x_r)$ ist eine Relation auf dem Produkt der Definitionsbereiche seiner Variablen: $C_S \subseteq D_1 \times \cdots \times D_r$.

Die Anzahl r von Variablen, auf die sich ein Constraint bezieht, bezeichnet man als *Stelligkeit (engl.: arity)* des Constraints. △

Ein Constraint schränkt also die möglichen Werte, die eine Variable annehmen kann, in bezug auf andere Variablen ein. Die Literatur befaßt sich aus Gründen der Vereinfachung häufig nur mit zweistelligen, sog. *binären Constraints*. Entsprechend bezeichnet man ein Constraint Satisfaction Problem als *binäres CSP*, wenn es nur ein- bis zweistellige Constraints enthält. Da sich jedes n-stellige CSP in ein äquivalentes binäres konvertieren läßt [RPD89], stellt diese Vereinfachung aus theoretischer Sicht keine Einschränkung dar. Außerdem haben binäre CSPs den Vorteil, daß sie sich einfach und anschaulich als Graphen darstellen lassen: jede Variable wird durch einen Knoten, jedes Constraint durch eine ungerichtete Kante repräsentiert. Unäre Constraints werden als Schlaufen dargestellt. Durch die Richtung der Kante können wir die Reihenfolge der Variablen in dem Tupel darstellen, über denen das Constraint definiert ist. Diese Form der graphischen Darstellung eines CSPs wird *Constraint Graph* genannt.

Nun wollen wir noch definieren, was es bedeutet, ein CSP zu lösen. Dazu benötigen wir den Begriff der Variablenbelegung:

Definition 7.3 (*Variablenbelegung*) Sei $X = \{x_1, \ldots, x_n\}$ eine Menge von Variablen mit den dazugehörigen Domains $D_i, i \in \{1, \ldots, n\}$. Dann bezeichnet ein n-Tupel $\Gamma = (a_1, \ldots, a_n)$ mit $a_i \in D_i$ eine *Belegung (engl.: instantiation)* jeder Variable x_i mit dem entsprechenden Wert a_i. Wir schreiben auch $\Gamma(x_i) = a_i$ für den Wert von x_i unter der Belegung Γ. △

Definition 7.4 (*Lösung eines CSP*) Ein Constraint C_S über einem Tupel von Variablen $S = (x_1, \ldots, x_r)$ heißt *erfüllt (engl.: satisfied)* durch eine Variablenbelegung Γ genau dann, wenn $(\Gamma(x_1), \ldots, \Gamma(x_r)) \in C_S$.

Eine Variablenbelegung Γ heißt *Lösung* eines Constraint Satisfaction Problems, wenn sie alle Constraints des Problems erfüllt. △

Die folgenden Definitionen betreffen Begriffe, die besonders bei der Argumentation über Verfahren zur Lösung von CSPs hilfreich sind:

[3] auf deutsch in etwa: „einschränkende Bedingung“

Definition 7.5 (*Teilbelegung*)
Sei X eine Menge von Variablen mit den dazugehörigen Domains $D_i, i \in \{1, \ldots, n\}$. Weiterhin sei $Y \subset X$ und Γ_Y eine Belegung von Y. Dann heißt Γ_Y auch *Teilbelegung* von X. △

Definition 7.6 (*Konsistenz einer Belegung*) Eine (Teil-)Belegung der Variablenmenge X eines CSPs Z heißt *konsistent*, wenn sie alle Constraints zwischen den von ihr belegten Variablen erfüllt. △

Definition 7.7 (*Konsistenz von Werten*) Sei X die Variablenmenge eines CSPs Z mit den dazugehörigen Domains $D_i, i \in \{1, \ldots, n\}$. Weiterhin sei $Y = \{x_{y_1}, \ldots, x_{y_m}\} \subset X$, $\Gamma_Y = (a_1, \ldots, a_m)$ eine konsistente Belegung von Y und x_k eine Variable aus $X \setminus Y$. Dann heißt ein Wert $a \in D_k$ konsistent zu Γ_Y genau dann, wenn $(a_1, \ldots, a_m, a_{m+1} = a)$ eine konsistente Belegung der Menge $\{x_{y_1}, \ldots, x_{y_m}, x_{y_{m+1}} = x_k\}$ ist. △

7.1.5 Backjumping

Wie das Problem der Ansatzsuche, so lassen sich auch Constraint-Satisfaction Probleme nach dem Generate-&-Test-Prinzip oder mit einem Backtracking-Verfahren lösen. Obwohl Backtracking-Verfahren schon wesentlich effizienter arbeiten als Generate & Test (vgl. Abschnitte 7.1.2 und 7.1.3), sind die Ergebnisse, die sich mit einfachem Backtracking erzielen lassen, in der Praxis selten zufriedenstellend. Das liegt daran, daß das einfache Backtracking anfällig ist für systematische Fehler, die eigentlich vorhersehbar und damit vermeidbar sind. Diese Fehler haben verschiedene Ursachen, werden jedoch häufig unter dem Begriff *Thrashing*[4] zusammengefaßt. Unter dem Sammelbegriff *Intelligent Backtracking* werden in der Literatur deshalb viele Optimierungsmöglichkeiten vorgeschlagen. Einen guten Überblick mit vielen weiteren Literaturhinweisen gibt hier [Kum92]; [Kon94] unternimmt einen theoretischen Vergleich verschiedener Backtracking-Verfahren.

Je aufwendiger aber die Optimierungen werden, desto größer wird auch die Gefahr, daß der entstehende Overhead an Berechnung und Verwaltung von Hilfsstrukturen den eigentlichen Optimierungseffekt zunichte macht. [Kum92] stellt deshalb heraus, daß „ein einfaches intelligentes Backtracking-Verfahren insgesamt durchaus eine geringere Komplexität aufweisen kann als ein aufwendigeres.“

Ein intelligentes Backtracking-Verfahren, das mit relativ wenig zusätzlichem Aufwand eine wichtige Thrashing-Ursache beseitigt, ist das *Backjumping*-Verfahren. Da wir dieses Verfahren als Grundlage für die Ansatzsuche verwenden werden, wollen wir uns mit der zugrundeliegenden Optimierungsidee vertraut machen. Dazu betrachten wir zunächst ein typisches Beispiel für eine Thrashing-Konstellation. Abbildung 7.3 zeigt einen Constraint-Graphen mit den Variablen $x_1, \ldots, x_5$ und ihren jeweiligen Domains. Das Layout ist uns bereits aus Abb. 7.1 auf Seite 82 bekannt, und läßt gut den Suchraum der möglichen Variablenbelegungen erkennen. An die Stelle der Auswahlordnung tritt in diesem Zusammenhang die Variablenordnung, und die Suchordnung wird zur Werteordnung; die Bedeutung dieser Ordnungen für die Effizienz von Lösungsverfahren auf der Basis von Backtracking bleibt jedoch dieselbe.

Wir beginnen nun, nach dem einfachen Backtracking-Paradigma eine Lösung zu suchen, und belegen die ersten drei Variablen der Reihenfolge nach mit v_{11}, v_{12} und v_{13}. Des weiteren

[4] frei übersetzt etwa „blind auf etwas eindreschen“

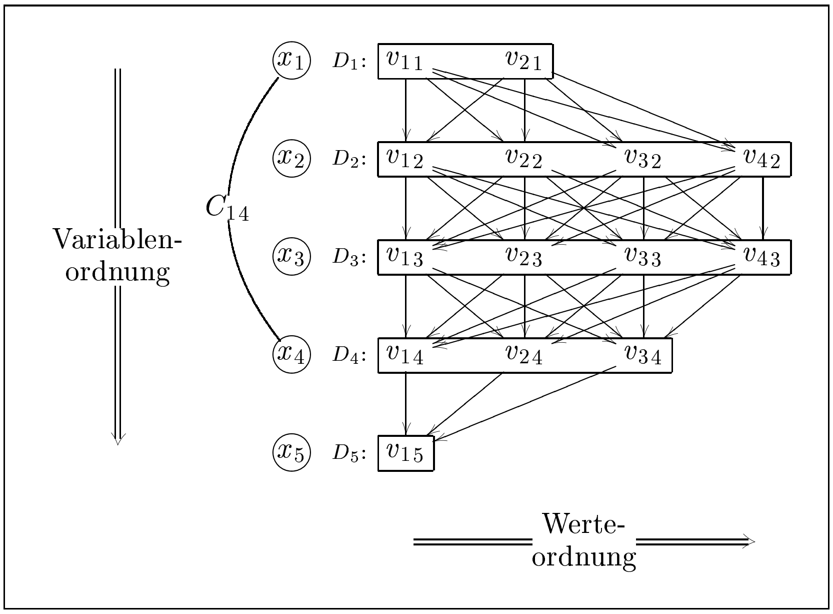


Abbildung 7.3: Eine typische Thrashing-Situation.

gehen wir davon aus, daß das Constraint C_{14} für die gegebene Belegung von x_1 keine der möglichen Werte für x_4 zuläßt: $\forall v \in D_4 : \ (v_{11}, v) \notin C_{14}$. Also schlägt der Versuch, im folgenden Schritt x_4 zu belegen, unbedingt fehl. Folgerichtig schließt sich ein Backtracking-Schritt an, und x_3 wird mit dem nächsten möglichen Wert v_{23} belegt. Das ändert aber nichts daran, daß der nächste Versuch, x_4 zu belegen, erneut scheitert; denn das Constraint C_{14}, das das ursprüngliche Scheitern verursacht hat, ist unabhängig von der Belegung von x_3. Erst eine Änderung in der Belegung von x_1 eröffnet die Chance, die „Sackgasse“ bei x_4 zu durchbrechen. Bis aber das Backtracking-Verfahren die Belegung von x_1 revidiert, hat es auf sämtliche Kombinationsmöglichkeiten der Werte für x_2 und x_3 erfolglos „eingedroschen“, womit sich auch die bildliche Bedeutung des Begriffs „Thrashing“ eröffnet. Es ist unschwer zu erkennen, daß diese Thrashing-Situation einen Aufwand verursacht, der mit der Anzahl derjenigen Variablen exponentiell wächst, die zwischen dem verursachenden Constraint liegen.

In manchen Situationen läßt sich diese Form des Thrashings vermeiden, indem man die Variablenordnung geschickt wählt. In unserem Beispiel wäre das Problem behoben, wenn x_4 direkt hinter x_1 eingeordnet würde. Im allgemeinen Fall existieren aber sehr viele Constraints zwischen den Variablen, die sich in ihren Anforderungen an die Variablenordnung häufig widersprechen. In unserem Beispiel brauchen wir uns nur je ein weiteres Constraint zwischen x_1 und allen übrigen Variablen vorzustellen, und die Thrashing-Situation ist durch die Variablenordnung allein nicht mehr auszuschließen. Hinzu kommt, daß die Berechnung einer optimalen Variablenordnung im Hinblick auf minimalen Abstand von durch Constraints verbundenen Variablen ein NP-vollständiges Problem ist.

Die Lösungsidee des Backjumping-Verfahrens lautet natürlich, die Ursache für eine auftretende Inkonsistenz zu bestimmen und direkt zur entsprechenden Variable zurückzuspringen. Die Zielvariable eines Backjumps zu bestimmen ist im allgemeinen allerdings etwas komplizierter als es das Beispiel vermuten läßt. Deshalb wollen wir den Algorithmus `bj_target` vorstellen, der die Zielvariable berechnet. Das allgemeine Problem lautet wie folgt:

Gegeben sei ein binäres CSP Z mit einer Variablenmenge $X = \{x_1, \ldots, x_n\}$, einer Constraintmenge $R \neq \emptyset$ und einem Definitonsbereich $D_i \neq \emptyset$ für jede Variable $x_i \in X$. Die Variablenordnung $\prec$ sei definiert als $x_i \prec x_j \;\Leftrightarrow\; i < j$. Außerdem gegeben sei eine Teilbelegung Γ_k für alle Variablen $x \in X$ mit $x \prec x_k$. Dann sucht der Algorithmus `bj_target`(x_k) nach einem konsistenten Wert für die Variable x_k. Je nach dem Erfolg der Suche nimmt die Ergebnisvariable `result` nach Beendigung des Algorithmus einen der folgenden drei Zustände ein:

- `result`$= \perp$, wenn ein konsistenter Wert gefunden wurde, die Lösungssuche also mit der nächsten Variable x_{k+1} fortgesetzt werden kann.
- `result`$= x_k$, wenn es ein unäres Constraint $C_{(x_k,x_k)}$ gibt, das von keinem der Werte aus D_k erfüllt wird. Das bedeutet, daß das CSP keine Lösung besitzt.
- `result`$= x_i$ mit $i < k$, wenn kein konsistenter Wert gefunden wurde. x_i ist dann die Zielvariable für den folgenden Backjump.

Für die Formulierung des Algorithmus gelten die folgenden Vereinbarungen:

- $R_k = \{C_{(x_i,x_j)} \in R \mid (i = k \vee j = k) \wedge i \leq k \wedge j \leq k\}$ bezeichnet die Menge aller Constraints zwischen x_k und anderen Variablen, die bezüglich $\prec$ vor x_k liegen, einschließlich x_k selbst.
- Die Funktionen $\max_\prec$ und $\min_\prec$ berechnen das bezüglich einer Variablenordnung $\prec$ größte bzw. kleinste Element einer Menge von Variablen.

Algorithmus 7.8 (*Bestimmung des Backjump-Targets*)

```
bj_target(x_k) :=
begin
    U := ∅
    loop für jeden Wert v ∈ D_k:
    begin
        instanziiere x_k mit v
        T := ∅
        loop für jedes Constraint C_(x_i,x_j) ∈ R_k:
        begin
            if (Constraint nicht erfüllt)
            then  if (x_i = x_k)
                  then T := T ∪ {x_j}
                  else T := T ∪ {x_i}
                  endif
            endif
        end
        if (T = ∅)
        then result:= ⊥    ⇝ terminiere Algorithmus
        endif
        if (x_k ∈ T)
        then U := U ∪ {x_k}
        else U := U ∪ {min_≺(T)}
```

```
            endif
        end
        result:= max≺(U)
    end
```

△

Wir können jedoch nicht jeden Backtracking-Schritt des einfachen Backtracking-Verfahrens durch einen solchen Backjump ersetzen. Backjumps dürfen nur aus einer „Vorwärtsbewegung“ heraus ausgeführt werden, also wenn gerade eine Variable x_i erfolgreich instanziiert wurde und nun eine konsistente Belegung für x_{i+1} aus dem *gesamten* Domain D_{i+1} gesucht wird. Wenn in dieser Situation kein konsistenter Wert gefunden werden kann, handelt es sich um eine echte Sackgasse und ein Backjump kann stattfinden.

Befinden wir uns dagegen in einer „Rückwärtsbewegung“, d.h. wir kommen durch Backtracking oder Backjumping von einer Variable x_j zurück zu einer Variable $x_i \prec x_j$, dann haben wir bereits einen Teil des Domains D_i bearbeitet, und der letztbesuchte Wert v hatte alle Constraints aus R_i erfüllt. Möglicherweise wurde also bereits eine Lösung des Problems gefunden, an der v beteiligt war, und nun suchen wir nach der nächsten Lösung. Selbst wenn im restlichen Domain von x_i kein weiterer konsistenter Wert zu finden ist, würde ein Backjump zu einer Variable $x_h \prec x_{i-1}$ mögliche Lösungskandidaten überspringen. Denn wenn x_{i-1} durch keine Constraints mit x_i verbunden ist, wird sich jeder weitere mögliche Wert von x_{i-1} mit der bereits früher erfolgreichen Belegung von x_i mit v vertragen und entsprechend zu einer möglichen weiteren Lösung führen.

Das Backjumping-Verfahren hat aber noch weitere Schwachpunkte:

- Durch einen Backjump gehen bereits gewonnene Erkenntnisse über konsistente Belegungen der übersprungenen Variablen verloren.
- Die Anzahl der Constraint-Checks pro instanziierter Variable erhöht sich gegenüber dem einfachen Backtracking. Denn um das günstigste Ziel für einen Backjump zu bestimmen, müssen alle Constraints aus der jeweiligen Menge R_i geprüft werden (innere Schleife des `bj_target`-Algorithmus), obwohl nach dem ersten verletzten Constraint bereits feststeht, daß der aktuelle Wert nicht in Frage kommt.

Verfahren, die diese Probleme durch weitere Optimierungen zu beheben versuchen, sind unter der Bezeichnung *Conflict-* bzw. *Dependency-Directed Backjumping* bekannt; siehe z.B. [Bru81, Dec94]. Diese Verfahren sind jedoch entsprechend aufwendig, weshalb wir uns an dieser Stelle nicht näher mit ihnen beschäftigen werden.

7.2 Ansatzsuche als Constraint Satisfaction Problem

Wenn wir im folgenden versuchen, das Problem der Ansatzsuche in ein Constraint Satisfaction Problem zu übersetzen, so verfolgen wir damit vor allem zwei Ziele:

Zugang zur breiten Wissensbasis aus dem Bereich der Constraint Satisfaction.
Schon in der Einführung über Constraint Satisfaction Probleme in Abschnitt 7.1.4

haben wir auf die große Anzahl an Arbeiten hingewiesen, die sich mit effizienten Lösungsverfahren für CSPs befassen. Allgemeine Untersuchungen, Vergleiche und Optimierungen von Backtracking-Verfahren erfolgen fast immer vor dem Hintergrund des Constraint Satisfaction Paradigmas. Besonders vielversprechend ist auch die Möglichkeit, auf die vielen bereits vorliegenden Ergebnisse theoretischer und empirischer Art zum Thema Variablenordnungen zurückgreifen zu können.

Abstraktion von Details des konkreten Problems. Wir haben bereits das Problem angesprochen, daß Ansatzsuche-Algorithmen üblicherweise die speziellen Eigenschaften des jeweils zugrundeliegenden Graphmodells ausnutzen müssen, um effizient zu arbeiten. Das macht die Wiederverwendung eines solchen Algorithmus für ein verändertes Graphmodell nahezu unmöglich. Mit der Formulierung der Ansatzsuche als CSP stellen wir eine Möglichkeit vor, den Algorithmus vom konkreten Graphmodell zu entkoppeln, ohne wesentliche Kompromisse bei der Effizienz eingehen zu müssen. Das erreichen wir, indem wir auf die Eigenschaften des konkreten Graphmodells nur über eine wohldefinierte abstrakte Schnittstelle zugreifen: Die konkreten Eigenschaften spiegeln sich auf der abstrakten Ebene der CSPs in Konzepten wie der Gewichtung von Constraints oder den in Abschnitt 7.2.3 eingeführten Queries wider. Damit wird es möglich, hochoptimierte Lösungsverfahren zu implementieren, die von Änderungen auf dem Graphmodell unbeeinflußt bleiben; lediglich der Übersetzungsschritt vom Graphmodell in die CSP-Repräsentation muß angepaßt werden.

Die folgenden Unterabschnitte haben das Ziel, das Prinzip der Übersetzung des Ansatzsuche-Problems in ein CSP anhand von einfachen Graphen ohne Attribute zu veranschaulichen und so ein Verständnis für das Konzept der Implementierung zu schaffen. In der Implementierung wird der Übersetzungsschritt ganz analog für attributierte ALR-Graphen durchgeführt, es werden lediglich einige zusätzliche Constraints benötigt, um auch die Attributierungs- und Abstraktionsverträglichkeit zu gewährleisten.

7.2.1 Vergleich der Problemstellungen

Um das Problem der Ansatzsuche als Constraint Satisfaction Problem zu fassen, versuchen wir zunächst, Analogien auszumachen, die es uns erleichtern, die beiden Problemklassen zueinander in Beziehung zu setzen. Bei näherem Hinsehen liegt die Analogie auf der Hand: In beiden Fällen geht es grundsätzlich darum, eine Abbildung zwischen zwei Mengen zu finden, die bestimmte Eigenschaften erfüllt. Auf der einen Seite wird nach einer Variablenbelegung gefragt, die sich unschwer als Abbildung von einer Variablenmenge in eine Wertemenge auffassen läßt. Auf der anderen Seite suchen wir einen Morphismus zwischen zwei Graphen L und G, der ja definitionsgemäß nichts weiter ist als eine Abbildung zwischen den Objektmengen zweier Graphen. Von der Variablenbelegung wird gefordert, daß sie alle Constraints des CSPs erfüllt, die Abbildung zwischen den zwei Graphen dagegen wird erst durch die üblichen Verträglichkeitsbedingungen zum Morphismus geadelt.

Tabelle 7.1 zeigt die Analogien zwischen dem Constraint-Satisfaction Problem und der Ansatzsuche im Überblick. Dabei werden die Bezeichnungen aus den Definitionen 7.1 und 2.2 verwendet. Diese Gegenüberstellung legt den Versuch nahe, einfach die Menge der Knoten und Kanten von L als Variablenmenge eines CSPs zu betrachten; als Domain für alle Knoten

	CSP	Ansatzsuche
gesuchte Abbildung:	Variablenbelegung Γ	Paar aus Knoten- und Kantenabbildung: $m = (m_V, m_E)$
Definitionsbereich:	Variablenmenge $X = \{x_1, \ldots, x_n\}$	Menge der Knoten und Kanten von $L : L_V \cup L_E$
Wertebereich:	Vereinigung der Domains: $D = \bigcup_{i=1}^{n} D_i$	Menge der Knoten und Kanten von $G : G_V \cup G_E$
geforderte Eigenschaften:	Γ erfüllt alle Constraints; $\Gamma(x_i) \in D_i$	Operations- und Typverträglichkeit

Tabelle 7.1: Gegenüberstellung von CSP und Ansatzsuche.

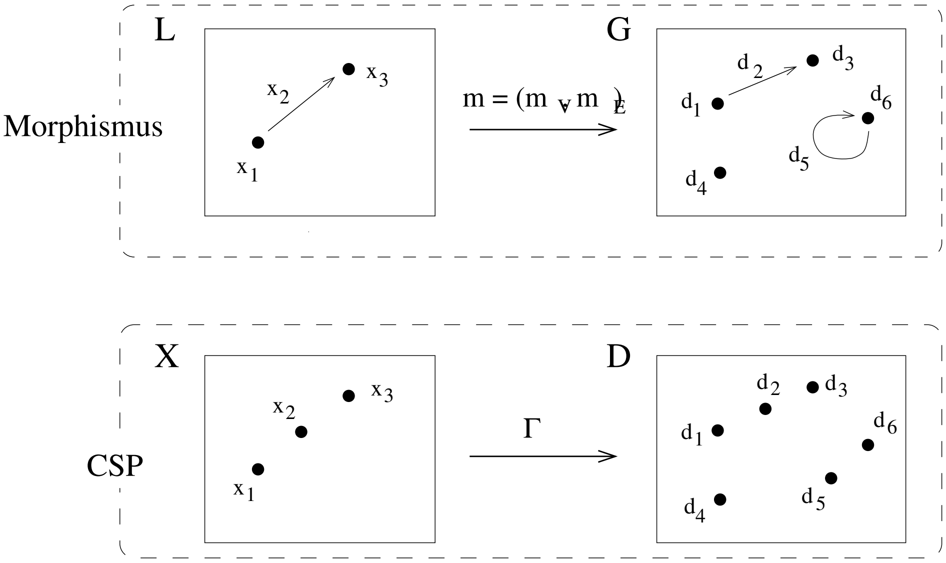


Abbildung 7.4: Beispiel zur Übersetzung des Problems der Ansatzsuche in ein CSP.

wählen wir dann G_V, für alle Kanten G_E. Wenn es uns dann gelingt, eine Menge von Constraints so zu definieren, daß sie die Verträglichkeitsbedingungen eines Morphismus erzwingt, dann hätten wir schon erreicht, was wir suchen: ein CSP, dessen Lösungen den Morphismen zwischen zwei Graphen entsprechen. Wir müßten dann nur noch zeigen, daß auch wirklich alle möglichen Morphismen gefunden werden. Versuchen wir also anhand eines Beispiels, die passenden Constraints zu finden.

Beispiel 7.9 (*CSP zur Ansatzsuche*)
Abbildung 7.4 zeigt zwei Graphen L und G, die der Einfachheit halber keine Typinformation enthalten. Gesucht sind alle totalen Morphismen von L nach G. Unter jedem Graphen steht die ihm im Rahmen der angestrebten CSP-Darstellung entsprechende Menge, wie wir sie aus der Gegenüberstellung in Tabelle 7.1 gefolgert haben. Vorläufig besteht unser Beispiel-CSP also aus:

- Der Variablenmenge $X = \{x_1, x_2, x_3\}$.

- Dem globalen Domain D. Da wir Knoten- und Kantenwerte unterscheiden, ergibt sich dieser Domain zu $D = D_V \cup D_E$ mit $D_V = \{d_1, d_3, d_4, d_6\}$ und $D_E = \{d_2, d_5\}$. Die Domains der einzelnen Variablen lauten dann $D_1 = D_3 = D_V$ und $D_2 = D_E$.

Was uns zu unserem CSP noch fehlt, ist die Menge der Constraints R. Um herauszufinden, welche Constraints wir benötigen, gehen wir einfach einmal von einer leeren Constraint-Menge aus und beobachten, was passiert, wenn wir dieses CSP zu lösen versuchen. Beginnen wir also, indem wir die Variable x_1 mit einem beliebigen Wert aus ihrem Domain D_1 belegen, zum Beispiel mit d_3. Diese Zuordnung verletzt offensichtlich keine der geforderten Morphismuseigenschaften, deshalb fahren wir fort und belegen x_2 mit $d_2 \in D_2$. Zusammen mit der vorherigen Zuordnung ergibt sich aber an dieser Stelle ein Widerspruch zu der von einem Morphismus geforderten Operationsverträglichkeit: $m(s(x_2))) = d_3 \neq s(m(x_2))) = d_1$. Wir brauchen also ein Constraint, das die Verträglichkeit mit der source-Operation erzwingt:

$$C^{\text{src}}_{(x_2,x_1)} = \{(d_E, d_V) \in D_2 \times D_1 \mid s(d_E) = (d_V)\} = \{(d_2, d_1), (d_5, d_6)\}$$

Ganz analog benötigen wir aber auch ein Constraint, um die target-Verträglichkeit für Belegungen der Variable x_2 in Verbindung mit x_3 sicherzustellen:

$$C^{\text{tar}}_{(x_2,x_3)} = \{(d_E, d_V) \in D_2 \times D_3 \mid t(d_E) = d_V\} = \{(d_2, d_3), (d_5, d_6)\}$$

Und damit sind wir mit unserem kleinen Beispiel-CSP auch schon am Ziel, denn mit der Constraint-Menge $R = \left\{C^{\text{src}}_{(x_2,x_1)}, C^{\text{tar}}_{(x_2,x_3)}\right\}$ erhalten wir nur noch Lösungen, die Morphismen repräsentieren:

- $\Gamma_1 = (d_1, d_2, d_3)$ und
- $\Gamma_2 = (d_6, d_5, d_6)$.

Wie man leicht sieht, sind das auch alle Morphismen, die zwischen den Graphen L und G existieren. △

Wären unsere Beispielgraphen L und G typisiert, so müßten wir das entsprechende CSP um Constraints ergänzen, die die Typverträglichkeit der gesuchten Abbildung sicherstellen. Für jedes $x_i \in X$ definieren wir zu diesem Zweck:

$$C^{\text{type}}_{(x_i)} = \{d \in D_i \mid \ell(x_i) = \ell(d)\}$$

Dabei handelt es sich um unäre Constraints, die aber ebenso als binäre Constraints $C^{\text{type}}_{(x_i,x_i)}$ betrachtet werden können. Diese Auffassung liegt der Darstellung unseres CSPs als *Constraint Graph* in Abb. 7.5 zugrunde, in der unäre Constraints als Schlaufen repräsentiert sind.

7.2.2 Konstruktion

Nachdem wir an einem Beispiel gesehen haben, wie die Suche nach einem Morphismus zwischen zwei Graphen als Constraint Satisfaction Problem formuliert werden kann, wollen wir nun ein allgemeines Konstruktionsverfahren für diesen Übersetzungsschritt angeben. Anschließend beweisen wir die Korrektheit dieser Konstruktion in dem Sinne, daß jede Lösung des

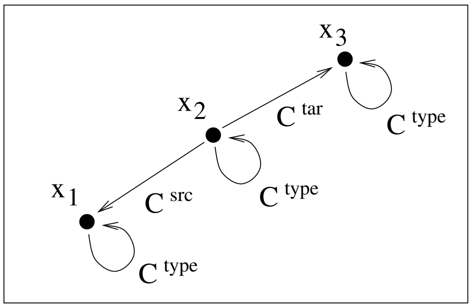


Abbildung 7.5: Der Constraint Graph zum Beispiel 7.9.

konstruierten CSPs einen Morphismus zwischen den Ausgangsgraphen repräsentiert, und daß umgekehrt zu jedem Morphismus eine Lösung des CSPs existiert.

Konstruktion 7.10 (*CSP zur Ansatzsuche*)
Gegeben seien zwei Graphen $L = (L_V, L_E, L_L, s_L, t_L, \ell_L)$ und $G = (G_V, G_E, L_G, s_G, t_G, \ell_G)$ gemäß Def. 2.1. Dann konstruieren wir ein CSP $Z = (X, D, R)$ wie folgt:

- $X = L_V \cup L_E = \{x_1, \ldots, x_n\},\ n = |X|$.
- $D = G_V \cup G_E = \bigcup_{i=1}^{n} D_i \quad \text{mit} \quad D_i = \begin{cases} G_V, & \text{wenn } x_i \in L_V \\ G_E, & \text{sonst.} \end{cases}$
- Die folgende Tabelle gibt an, unter welchen Bedingungen ein bestimmtes Constraint zwischen zwei Variablen x_i, x_k in die Constraint-Menge R aufgenommen wird. Die Bedingungen sind dabei als notwendig und hinreichend zu verstehen, es besteht also eine „genau dann, wenn"-Beziehung.

Bedingung $\longleftrightarrow$	Constraint $\in R$
$x_i = x_k$	$C^{\text{type}}_{(x_i)} = \{d \in D_i \mid \ell_L(x_i) = \ell_G(d)\}$
$x_i \in L_E,\ x_k \in L_V,\ s_L(x_i) = x_k$	$C^{\text{src}}_{(x_i,x_k)} = \{(d_i, d_k) \in D_i \times D_k \mid s_G(d_i) = d_k\}$
$x_i \in L_E,\ x_k \in L_V,\ t_L(x_i) = x_k$	$C^{\text{tar}}_{(x_i,x_k)} = \{(d_i, d_k) \in D_i \times D_k \mid t_G(d_i) = d_k\}$

$\triangle$

Satz 7.11 (*Korrektheit und Vollständigkeit von Konstruktion 7.10*)
Sei $Z = (X, D, R)$ mit $X = \{x_1, \ldots, x_n\}$ ein CSP, das nach dem Verfahren 7.10 aus den beiden Graphen $L = (L_V, L_E, L_L, s_L, t_L, \ell_L)$ und $G = (G_V, G_E, L_G, s_G, t_G, \ell_G)$ konstruiert worden ist. Dann gilt:

(Korrektheit)
Wenn eine Belegung Γ Lösung von Z ist, dann ist $m = (m_V, m_E)$ mit

$$\begin{aligned} m_V(x) &= \Gamma(x) \quad \text{für alle} \quad x \in L_V \\ m_E(x) &= \Gamma(x) \quad \text{für alle} \quad x \in L_E \end{aligned}$$

ein Graphmorphismus von L nach G.

(Vollständigkeit)
Wenn $m = (m_v, m_E)$ ein Morphismus $m : L \to G$ ist, dann ist $\Gamma = (d_1, \ldots, d_n)$ mit

$$\Gamma(x_i) = d_i = \begin{cases} m_V(x_i), & \text{wenn } x_i \in L_V \\ m_E(x_i), & \text{wenn } x_i \in L_E \end{cases}$$

Lösung von Z.

△

Beweis: Der Einfachheit halber vernachlässigen wir im folgenden die Indizierung der Operationen s, t und ℓ; sie ergibt sich jeweils eindeutig aus dem Kontext.

(Korrektheit) Wir müssen zeigen, daß m operations- und typverträglich ist. Die *Operationsverträglichkeit* zeigen wir am Beispiel der source-Operation; der Beweis für die target-Operation erfolgt analog. Weil L ein Graph ist, gilt

$$\forall x \in L_E : \exists n \in L_V : \; s(x) = n. \tag{7.1}$$

Dann gibt es gemäß Konstruktion ein Constraint $C^{\text{src}}_{(x,n)} \in R$, und weil Γ Lösung von Z ist, folgt

$$(\Gamma(x), \Gamma(n)) \in C^{\text{src}}_{(x,n)}. \tag{7.2}$$

Aus (7.2) und der Definiton von $C^{\text{src}}_{(x,n)}$ folgt aber $s(\Gamma(x)) = \Gamma(n)$, und wegen (7.1) gilt dann $\Gamma(s(x)) = s(\Gamma(x))$. Mit der Definition von m folgt schließlich

$$m_V(s(x)) = s(m_E(x)).$$

Nun zeigen wir noch die *Typverträglichkeit*: Gemäß Konstruktion gilt

$$\forall x \in L_V : \exists C^{\text{type}}_{(x)} \in R$$

Weil Γ Lösung von Z ist, gilt außerdem $\Gamma(x) \in C^{\text{type}}_{(x)}$. Aus der Defintion des Typ-Constraints folgt $\ell(x) = \ell(\Gamma(x))$ und mit der Definition von m schließlich

$$\ell(x) = \ell(m_V(x)).$$

Dasselbe gilt für alle Kanten $x \in L_E$.

(Vollständigkeit) Wir müssen zeigen, daß die aus dem Morphismus m konstruierte Belegung Γ alle Constraints aus R erfüllt. Wir betrachten zunächst die source-Constraints; der Beweis für die target-Constraints verläuft analog. Sei $C^{\text{src}}_{(x_i,x_k)}$ ein beliebiges source-Constraint aus R. Dann gilt nach Konstruktion

$$x_i \in L_E, \; x_k \in L_V, \; s(x_i) = x_k. \tag{7.3}$$

Weil m Morphismus ist, ist außerdem $m_V(s(x_i)) = s(m_E(x_i))$, und mit (7.3) folgt $m_V(x_k) = s(m_E(x_i))$. Wegen $m_E(x_i) \in D_i$ und $m_V(x_k) \in D_k$ ist dann

$$(m_E(x_i), m_V(x_k)) \in C^{\text{src}}_{(x_i,x_k)}$$

nach Definition des source-Constraints. Mit der Konstruktion von Γ aus m folgt schließlich

$$(\Gamma(x_i), \Gamma(x_k)) \in C^{\text{src}}_{(x_i,x_k)}.$$

Bleibt noch zu zeigen, daß Γ auch die Type-Constraints erfüllt. Sei $C^{\text{type}}_{(x_i)}$ ein beliebiges type-Constraint aus R. Dann ist entweder $x_i \in L_V$ oder $x_i \in L_E$. Wir nehmen an, daß $x_i \in L_V$ ist; der andere Fall ist analog. Weil m Morphismus ist, gilt dann

$$\ell(x_i) = \ell(m_V(x_i)).$$

Wegen $m_V(x_i) \in D_i$ folgt aus der Definition des type-Constraints, daß

$$(m_V(x_i)) \in C^{\text{type}}_{(x_i)},$$

und mit der Konstruktion von Γ gilt schließlich auch

$$(\Gamma(x_i)) \in C^{\text{type}}_{(x_i)}.$$

□

7.2.3 Domainreduktion durch Queries

Der *worst-case*-Aufwand für ein CSP-Lösungsverfahren beträgt n^k, wobei k die Anzahl der Variablen bezeichnet und n die durchschnittliche Größe ihrer Domains. Bisher hat sich unsere Aufmerksamkeit bezüglich der Optimierungsmöglichkeiten zu recht auf den bestimmenden Exponenten k gerichtet: First-Fail-Variablenordnung, Backtracking und Backjumping zielten alle auf ein möglichst effizientes Durchforsten des Variablendomains. Oder anders formuliert: wir haben bisher nur die *Tiefe* des Suchraums zu optimieren versucht. Im folgenden wollen wir uns nun mit der Optimierung bezüglich der *Breite* des Suchraums beschäftigen, gegeben durch den Faktor n, der die Größe der Wertedomains angibt. Auch wenn die Breitenoptimierung auf den ersten Blick im Verhältnis zu den Optimierungsmöglichkeiten von k wenig erfolgversprechend erscheinen mag, darf man die tatsächliche Wirkung nicht unterschätzen. Denn häufig ist insbesondere die dynamische Domainreduktion, die wir später einführen werden, in der Lage, den aktuell zu durchsuchenden Domain um Größenordnungen zu reduzieren.

Unser Ziel ist es also, den Aufwand zur Lösung eines gegebenen CSPs durch Reduktion seiner Domains zu verringern. Natürlich dürfen nur solche Werte aus einem Domain entfernt werden, die mit Sicherheit nicht zu einer Lösung des CSPs gehören können. Als Ausschlußkriterium suchen wir deshalb nach notwendigen Bedingungen für die Werte einer Lösung, und wir finden diese natürlich in den Constraints. Wenn wir eine Variable x mit einem Wert v aus dem Domain D_x von x belegen, dann lautet die Bedingung: v muß alle Constraints erfüllen, die zwischen x und anderen bereits belegten Variablen – x selbst eingeschlossen – bestehen.

Auf der Prüfung genau dieser Bedingung beruhen die Lösungsverfahren für CSPs. Wir bezeichnen diese Bedingung als *dynamisch*, weil sie vom aktuellen Belegungszustand anderer Variablen abhängig ist. Wir können aber einen Teil dieser Bedingung herauslösen, der statisch ist: v muß nämlich insbesondere die unären Constraints von x erfüllen. Das bedeutet, wir können alle Werte aus dem Domain einer Variable entfernen, die die unären Constraints der Variable nicht erfüllen. Dies können wir bereits vor dem Start eines Lösungsalgorithmus tun, womit wir verhindern, daß der Algorithmus anschließend bei jedem Besuch der Variable erneut feststellt, daß der bewußte Wert inkonsistent ist. Diese Erkenntnis ist nicht neu, und ein CSP, dessen Domains einer solchen *statischen Domainreduktion* unterzogen wurden, wird als *knotenkonsistent (engl.: node consistent; vgl. z.B. [Kum92])* bezeichnet.

Tatsächlich können wir jedoch auch die mit einer Variable verbundenen echt binären Constraints in statische Bedingungen an ihre Werte übersetzen. Betrachten wir dazu das Beispiel aus Abb. 7.4 auf Seite 91 mit dem dazugehörigen Constraint-Graphen aus Abb. 7.5. Der Domain $D_1 = \{d_1, d_3, d_4, d_6\}$ von x_1 ist knotenkonsistent. Offensichtlich kommen aber nur solche Knoten[5] als Wert für x_1 in Frage, die mindestens eine ausgehende Kante haben. Auf das entsprechende Constraint $C^{\text{src}}_{(x_2,x_1)}$ bezogen können wir daraus die folgende statische notwendige Bedingung an einen Wert v von x_1 formulieren: $\exists e \in D_2 : \ (e, v) \in C^{\text{src}}_{(x_2,x_1)}$. Aufgrund dieser Bedingung können wir den Domain D_1 von ursprünglich vier Elementen auf die beiden Elemente $\{d_1, d_6\}$ reduzieren. Die Reduktion aller Domains aufgrund derartiger Bedingungen führt zum Begriff der *Kantenkonsistenz (engl. arc consistency)*, und mit der Berücksichtigung ganzer Constraint-Pfade schließlich zur allgemeinen *k-Konsistenz*. Wir wollen uns mit diesen Konsistenzbegriffen nicht näher beschäftigen, es sei jedoch abschließend angemerkt, daß ein Algorithmus existiert, der ein gegebenes CSP mit einem zur Größe der Domains quadratischen Aufwand kantenkonsistent macht [MH86], während das Erreichen von k-Konsistenz im allgemeinen mit exponentiellem Aufwand verbunden ist.

Kommen wir nun zurück zu den eingangs bereits erwähnten dynamischen Bedingungen. Wir betrachten wieder das Beispiel aus Abb. 7.4. Nehmen wir an, bisher sei als einzige Variable x_2 mit dem Wert d_2 belegt, und als nächstes soll nun ein konsistenter Wert für x_3 gefunden werden. Der Domain von x_3 ist $D_3 = D_V$, das ist die Menge aller Werte, die den Knoten im Graphen G entsprechen. Nach der üblichen Vorgehensweise wären also vier Kandidaten auszuprobieren und auf Konsistenz zu den bisher belegten Variablen zu testen. Aus dem Target-Constraint $C^{\text{tar}}_{(x_2,x_3)}$ ergibt sich aber in dieser Situation die folgende Bedingung an den gesuchten Wert $\Gamma(x_3)$: $\ (d_2, \Gamma(x_3)) \in C^{\text{tar}}_{(x_2,x_3)}$. Daraus folgt über die Definition des Target-Constraints, daß $\Gamma(x_3) = t(d_2)$ gelten muß, und diese Bedingung erfüllt nur noch d_3. Dieses Ergebnis ist an sich natürlich nicht überraschend; auch das übliche Backtracking-Lösungsverfahren hätte den ursprünglichen Domain durch Constraint-Checks Schritt für Schritt schließlich auf den Wert d_3 reduziert. Interessant ist aber, daß wir den linearen Aufwand dieses üblichen Verfahrens auf den konstanten Aufwand reduzieren konnten, einmal die Target-Operation anzuwenden.

Betrachten wir dasselbe Beispiel unter leicht veränderten Vorzeichen: Diesmal sei x_3 bereits mit d_3 belegt, gesucht eine konsistente Belegung für x_2. Das Target-Constraint liefert uns die Bedingung $d_3 = t(\Gamma(x_2))$, und durch „scharfes Hinsehen“ erkennen wir natürlich sofort, daß nur d_2 als Belegung in Frage kommt. Dennoch liegt der Fall hier anders als in der zuvor betrachteten Situation:

- Es könnte mehrere Werte geben, die die Bedingungsgleichung erfüllen, nämlich alle am Knoten d_3 einlaufenden Kanten. Falls $\Gamma(x_2)$ also außer $C^{\text{tar}}_{(x_2,x_3)}$ noch weitere Constraints erfüllen muß, ist immer noch eine Suche mit linearem Aufwand nach einem konsistenten

[5]hier sind die Knoten aus dem Graphen G gemeint, wohingegen sich der Begriff der Knotenkonsistenz auf die Knoten eines Constraint-Graphen bezieht, d.h. auf die Variablen des CSPs

Wert erforderlich.

- In unserer Graphsignatur existiert keine Umkehrrelation zur Target-Operation, wir können die Bedingungsgleichung $d_3 = t(\Gamma(x_2))$ nicht nach $\Gamma(x_2)$ auflösen. In einem solchen Modell können wir nur alle Elemente des Domains auf die vorgegebene Bedingung testen; das aber entspricht dem normalen Vorgehen zum Lösen eines CSP und bringt keine Effizienzvorteile.

Wir sehen also, daß uns bei der dynamischen Domainreduktion eine notwendige Bedingung für die gesuchten Werte allein nicht weiterhilft. Was wir benötigen, um einen Effizienzgewinn zu erzielen, sind Operationen, oder allgemeiner gesagt: Strukturen im konkreten Domain, die den direkten Zugriff auf die Objekte mit den gewünschten Eigenschaften ermöglichen. Und der „konkrete Domain“ ist dabei letztlich erst auf Implementierungsebene zu finden. So wissen wir z.B. aus Kapitel 5, daß in unserer Implementierung von Graphen durchaus Operationen für den direkten Zugriff auf die ein- und auslaufenden Kanten eines Objekts vorgesehen sind, während sie in unserem abstrakten Modell eines Graphen fehlen.

An dieser Stelle haben wir ein Problem: Einerseits wollen wir konkretes Wissen über die Strukturierung auf den Domains für Optimierungszwecke ausnutzen, andererseits wollen wir die Lösungsverfahren, die dieses konkrete Wissen nutzen sollen, auf der abstrakten Ebene eines allgemeinen CSPs formulieren. Glücklicherweise ist dieses Problem nicht neu. Das Zauberwort heißt „Abstraktion“, und das beste Beispiel sind die Constraints selbst, die ja ihre konkrete Definition hinter dem abstrakten Begriff des Erfülltseins verbergen. Nach diesem Vorbild ergänzen wir auf der Ebene der allgemeinen CSPs den Begriff des *Queries* als Abstraktion von Anfrageoperationen auf dem konkreten Domain. Der Name „Query“ wurde in Anlehnung an die englische Bezeichnung für Anfrageoperationen in Datenbanken gewählt.

Definition 7.12 (*Query*) Gegeben sei ein CSP Z mit einer Variablenmenge $X = \{x_1, \ldots, x_n\}$ und den dazugehörigen Domains $D_i,\ i \in \{1, \ldots, n\}$. Weiterhin sei $Y \subset X$, Γ eine konsistente Belegung von Y und x_t eine Variable aus $X \setminus Y$. Dann ist ein *Query* $Q_{Y,x_t}(\Gamma)$ eine Teilmenge von D_t mit der Eigenschaft, daß:

$$\forall d \in D_t \setminus Q_{Y,x_t}(\Gamma) : \quad \nexists \Gamma_X \ \text{ mit } \ \Gamma_X(x_t) = d \ \text{ und } \ \Gamma_X \text{ ist Lösung von } Z.$$

Die Variablenmenge Y bezeichnen wir als *Vorbereich, Voraussetzungen* oder *Quellvariablen* von $Q_{Y,x_t}(\Gamma)$, x_t heißt *Zielvariable* des Queries. Wenn bereits aus dem Kontext hervorgeht, auf welche Variablenbelegung sich ein Query bezieht, schreiben wir auch einfach Q_{Y,x_t} statt $Q_{Y,x_t}(\Gamma)$.

Ein Query $Q_{Y,x_t}(\Gamma)$ heißt *konstant* genau dann, wenn $Y = \emptyset$. Wir schreiben dann auch vereinfachend Q_{x_t}. △

Wir können durch Queries also sowohl dynamische als auch statische Domainreduktion ausdrücken; letzteres gerade durch konstante Queries. Im Constraint-Graphen stellen wir ein Query Q_{Y,x_t} als Hyperkante zwischen den Quellvariablen aus Y und der Zielvariable x_t dar. Abbildung 7.6 zeigt den Constraint-Graphen aus Abb. 7.5, ergänzt um einige Beispiele für einfache Queries; in diesem Fall sind die Vorbereiche immer kleiner oder gleich eins, so daß keine echten Hyperkanten zu sehen sind. Als ein Beispiel für die konkrete Definition eines

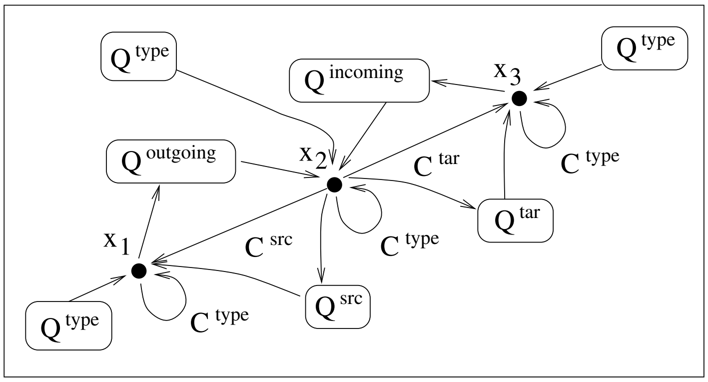


Abbildung 7.6: Der Constraint Graph aus Abb. 7.5, ergänzt um einige Queries.

Queries betrachten wir $Q^{\text{tar}}_{\{x_2\},x_3}$:

$$Q^{\text{tar}}_{\{x_2\},x_3}(\Gamma) = \{t\,(\Gamma(x_2))\}.$$

7.2.4 Verwandte Arbeiten

Die Idee für das im vorigen Abschnitt vorgestellte Query-Konzept geht auf [Zün96] zurück, der die Ansatzsuche in PROGRES beschreibt. Dort spielen Queries eine große Rolle, was insofern nicht verwunderlich ist, als dieses System auf einer Datenbank basiert, in der die Graphstrukturen verwaltet werden. Das Verfahren von PROGRES arbeitet jedoch nicht mit Constraint-Satisfaction-Methoden. In Abschnitt 7.3.4 kommen wir noch einmal bezüglich der Variablenordnung auf PROGRES zurück.

Eine weitere Arbeit, auf die in diesem Zusammenhang hingewiesen werden soll, ist [Bie97], wo ebenfalls ein Algorithmus für die Ansatzsuche auf einem konkreten Graphmodell entwickelt wird. Diese Arbeit beschäftigt sich weder mit Constraint Satisfaction noch werden explizit Query-Konzepte angewandt; die dort eingeführten Konstruktionen der *Possible-Mengen* bzw. *Situation dependent Possible-Mengen* entsprechen aber im wesentlichen den Ideen der statischen bzw. dynamischen Domainreduktion. Auf einer konkreten Ebene werden die für das spezielle Graphmodell zugeschnittenen notwendigen Bedingungen aufgestellt. In einem weiteren Schritt werden dann die aus verschiedenen Bedingungen für dasselbe Graphobjekt gewonnenen reduzierten Domains durch Schnittmengenbildung weiter reduziert. Diese Idee läßt sich auch auf das Konzept der Queries übertragen; so könnte es im Beispiel aus Abb. 7.6 durchaus Sinn machen, etwa die Schnittmenge eines Outgoing- und eines Type-Queries mit derselben Zielvariable zu berechnen, um den Domain weiter einzuschränken. Häufig bedeutet aber eine Schnittmengenbildung einen größeren Aufwand als das direkte Constraint-Checking auf einem ohnehin schon sehr kleinen Query. So haben die Source- und Target-Queries z.B. grundsätzlich eine maximale Mächtigkeit von eins. Der Einsatz der Schnittmengentechnik erfordert also eine sehr sorgfältige Steuerung, die wir zukünftigen Erweiterungen überlassen wollen.

7.3 Entwurf

Dieser Abschnitt setzt die theoretischen Konzepte aus Abschnitt 7.2 in den Entwurf eines Morphismusvervollständigungsverfahrens auf der Basis von Constraint Satisfaction um. Nach der Definition der allgemeinen Anforderungen an die Ansatzsuche und der Vorstellung der entsprechenden Anwendungsschnittstelle beschreiben die Unterabschnitte 7.3.3 und 7.3.4 dabei zunächst den Entwurf eines Frameworks für die Lösung von CSPs mit einem Lösungsalgorithmus nach dem Backjumping-Verfahren. Die letzten beiden Unterabschnitte gelten dann der Anwendung dieses Frameworks für die Realisierung des eigentlichen Vervollständigungsverfahrens für ALR-Morphismen.

7.3.1 Anforderungsdefinition

Funktionale Anforderungen

Teilansätze vervollständigen.
Wir wollen die Möglichkeit haben, partielle Ansatzmorphismen vorzugeben und vervollständigen zu lassen. Zum einen ergibt dies die Chance, eine Einschränkung des Suchraums und dadurch eine deutliche Effizienzverbesserung der Ansatzsuche zu erreichen, zum anderen läßt sich durch diese Funktionalität eine Parameterübergabe an Graphregeln ausdrücken.

Allgemeine Morphismen vervollständigen.
Wir wollen nicht nur Ansätze, also Morphismen von der linken Seite einer Regel in einen Arbeitsgraphen, sondern allgemein beliebige Morphismen vervollständigen können. Diese Funktionalität ist z.B. für die Überprüfung von negativen Anwendungsbedingungen [HHT96] von Bedeutung, wird aber auch für die Expansion von Interface-Morphismen auf lokale Graphen im Zusammenhang mit verteilter Graphtransformation [Tae96] benötigt.

Alle Ansätze bzw. alle Vervollständigungen finden.
Natürlich wollen wir alle möglichen Ansätze einer Regel in einen Arbeitsgraphen bzw. alle Vervollständigungen eines partiellen Morphismus bestimmen können. Dies soll nur auf Anforderung geschehen, denn in den meisten Fällen wird nur ein Ansatz gefordert, der beliebig sein kann, aber möglichst schnell gefunden werden soll.

Auch nicht-injektive Morphismen finden.
Das AGG-System soll u.a. zur Demonstration von theoretischen Konzepten und zum Test ihrer praktischen Anwendbarkeit dienen. Deshalb soll die Basisfunktionalität des Systems möglichst allgemein gehalten werden. Für die Morphismen bedeutet das, daß wir beliebige partielle ALR-Morphismen nach Def. 6.7 darstellen wollen. Optional wollen wir aber die Möglichkeit haben, die Suche auf injektive Morphismen einzuschränken, was in der Praxis häufig gewünscht ist. In einer Erweiterungsstufe ist auch die Einschränkung auf weitere Ansatzeigenschaften vorzusehen (vgl. Def. 2.11), insbesondere bezüglich der *gluing-condition* für die Simulation von DPO-Ableitungen.

Attribute matchen und Variablen belegen.
Die Möglichkeit, die Objekte eines Graphen mit Elementen aus einer Datenalgebra

zu attributieren und bei der Graphtransformation mit diesen Attributen zu rechnen, hat sich für den praktischen Einsatz als unverzichtbar erwiesen. Für die Ansatzsuche bedeutet das, daß nicht nur Graphobjekte, sondern auch deren Attribute „gematcht“ bzw. Variablen belegt werden müssen.

Softwaretechnische Anforderungen

Effizienz. Die zentrale Anforderung an die Ansatzsuche überhaupt betrifft die Zeit-Effizienz der Implementierung, da das zugrundeliegende Problem NP-vollständig ist und damit den Gesamtaufwand eines Transformationsschrittes im System bestimmt. Praktische Beispiele haben aber gezeigt, daß durch geschickte Algorithmen der durchschnittliche Aufwand durchaus auf ein praxistaugliches Maß gedrückt werden kann [Zün96, BGT91, Dör95].

Abstraktion. Es ist wünschenswert, den eigentlichen Algorithmus auf einer Ebene zu formulieren, die von dem Graphmodell eines konkreten Graphtransformationssystems abstrahiert. Nur auf diese Weise können wir vermeiden, daß der Algorithmus nach jeder Änderung des Graphmodells angepaßt und verifiziert oder gar neu geschrieben werden muß. Diese Anforderung steht allerdings potentiell in Konflikt zur Anforderung an die Effizienz, die unter Abstraktion im allgemeinen leidet.

Wartbarkeit, Austauschbarkeit, Modularität. Die Standardanforderungen an jedes Softwaresystem treffen auch und gerade auf die Ansatzsuche zu; um so mehr, als die Ansatzsuche von zentraler Bedeutung für das Gesamtsystem ist und aufgrund der NP-Vollständigkeit des Problems Heuristiken eine große Rolle spielen. Im Falle der Ansatzsuche ist es durchaus angebracht, die Anforderung an die Austauschbarkeit dynamisch zu verstehen, um den direkten empirischen Vergleich verschiedener Implementierungen und die Auswahl der günstigsten Heuristik für eine spezielle Graphklasse zur Laufzeit zu ermöglichen.

7.3.2 Anwendungsschnittstelle

In Abschnitt 7.1.1 haben wir die Morphismusvervollständigung als allgemeine Form der Ansatzsuche erkannt. Dementsprechend steht die Klasse `ALR_Morphism` im Mittelpunkt des objektorientierten Entwurfs der Ansatzsuche. Die wichtigste Anforderung an den Entwurf zielt auf die modulare Entkopplung und Austauschbarkeit des Ansatzsuche-Algorithmus. Dieser Anforderung werden wir gerecht, indem wir den Algorithmus in einer eigenständigen Klasse implementieren, auf die von außerhalb nur über ein extrem schlankes Interface zugegriffen werden kann. Das Interface heißt `ALR_MorphCompletionStrategy`, um darauf hinzuweisen, daß es sich hier um eine Instanz des *Strategy-Pattern* aus [GHJV95] handelt. Neben Entkopplung und Austauschbarkeit der Algorithmus-Implementierung wird durch dieses Pattern auch erreicht, daß beliebig viele Implementierungsvarianten zur Verfügung gestellt werden können, unter denen nach Bedarf auch zur Laufzeit ausgewählt werden kann. Damit erfüllt der Entwurf auch die Anforderung, spezialisierte Varianten des Algorithmus für eingeschränkte Ansatzbegriffe (vgl. Abschnitt 7.1.1) zu ermöglichen.

Abb. 7.7 veranschaulicht die Integration der Morphismus-Vervollständigung in die Morphismus-Klasse. `ALR_Morphism` wird um die Methode `nextCompletion()` ergänzt, de-

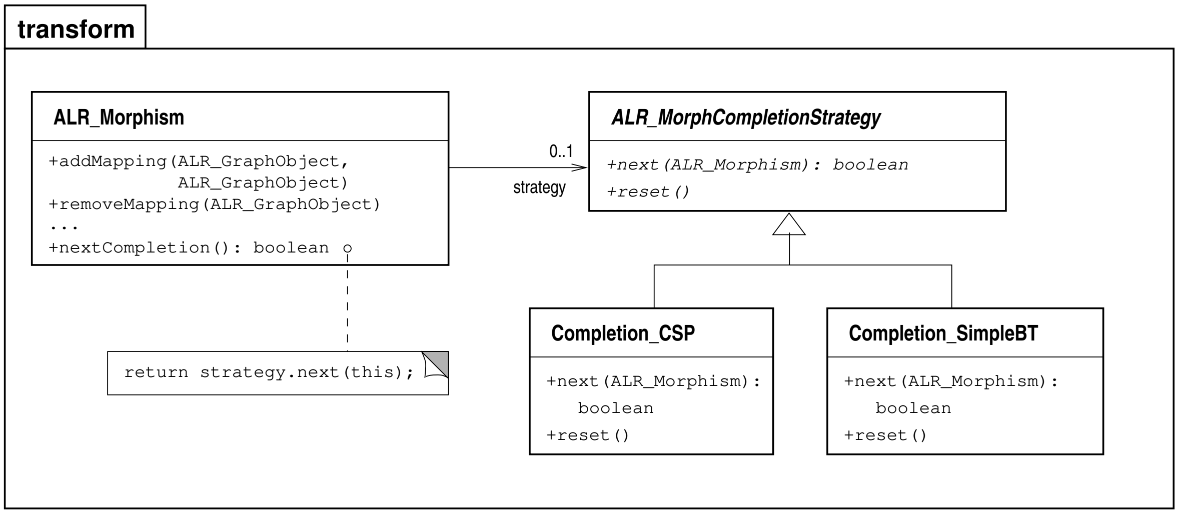


Abbildung 7.7: Modulare Integration der Morphismusvervollständigung mittels Strategy-Pattern.

ren Funktionalität für einen Aufruf auf einem Morphismus-Objekt `morph` wie folgt definiert wird:

- Wenn `morph` eine Vervollständigung besitzt, dann wird diese *in place* berechnet, d.h., `morph` selbst wird um entsprechende Mappings ergänzt. Der Rückgabewert ist `true`.
- Wenn keine Vervollständigung möglich ist, bleibt `morph` unverändert, und der Rückgabewert ist `false`.

Solange zwischen zwei Aufrufen von `nextCompletion()` keine Veränderungen am Morphismus bzw. seinen Quell- und Zielgraphen vorgenommen werden, wird immer die nächste mögliche Vervollständigung der Ausgangskonstellation berechnet. Auf diese Weise können alle möglichen Totalisierungen eines partiellen Morphismus aufgezählt werden.

Tatsächlich delegiert `nextCompletion()` die Implementierung der gerade beschriebenen Funktionalität aber an die Methode `next()` einer assoziierten Instanz von `ALR_MorphCompletionStrategy`. Dabei übergibt der Morphismus als Parameter eine Referenz auf sich selbst, so daß die Implementierung des Totalisierungs-Algorithmus auf alle öffentlichen Methoden von `ALR_Morphism` zugreifen kann. Insbesondere benötigt der Algorithmus die Methode `ALR_Morphism.addMapping()`, um die in-place Berechnung auf dem Morphismus durchzuführen. Die beiden Klassen `Completion_CSP` und `Completion_SimpleBT` in Abb. 7.7 stehen beispielhaft für mögliche Implementierungen des Strategy-Interfaces; `Completion_CSP` implementiert die Ansatzsuche als Constraint-Satisfaction Problem und wird uns in Abschnitt 7.3.5 noch genauer beschäftigen, während `Completion_SimpleBT` zu Vergleichszwecken ein naives Backtracking-Verfahren realisiert.

Jede Implementierung eines Vervollständigungs-Algorithmus muß zwischen zwei Aufrufen von `next()` mit derselben Morphismus-Referenz `ms` ihren internen Berechnungszustand bewahren, um die oben geforderte Aufzählung aller existierenden Vervollständigungen zu ermöglichen. Wenn aber der Morphismus zwischen den Aufrufen modifiziert wird, z.B. durch eine interaktive Benutzereingabe, dann wird der gespeicherte Berechnungszustand mit großer Wahrscheinlichkeit inkonsistent. Deshalb verlangen wir vom Morphismus, daß er die Strategie über relevante Modifikationen informiert. Dies geschieht durch den Aufruf der Methode `reset()` aus dem `ALR_MorphCompletionStrategy`-Interface, die den Berechnungszustand des Algorithmus zurücksetzt. Der nächste Aufruf von `next()` mit einem Argument `morph` bewirkt dann eine Initialisierung des Berechnungszustands, wobei insbesondere der aktuelle Zustand von `morph` festgehalten wird. Alle folgenden Aufrufe mit derselben Referenz ergeben dann nicht mehr Vervollständigungen des aktuellen Zustands von `morph`, denn der wurde ja bereits beim ersten Aufruf in-place vervollständigt, wenn das möglich war. Stattdessen werden dann wie gewünscht alle weiteren Vervollständigungen desjenigen Zustands berechnet, der beim ersten Aufruf vorgefunden wurde. Vor einem Aufruf von `next()` mit einer anderen Morphismus-Referenz muß nicht explizit ein Reset durchgeführt werden, weil die `next()`-Methode selbst die Gleichheit der aktuell übergebenen Referenz mit der vorherigen vergleicht und ggf. den internen Zustand initialisiert.

Alternativ zum expliziten Aufruf der `reset()`-Methode durch den Morphismus könnten wir auch eine Observer/Observable-Beziehung zwischen Strategie und Morphismus etablieren. Die Strategie könnte dann selbständig entscheiden, wann sie ihren Zustand initialisieren muß; dementsprechend wäre `reset()` nicht mehr Teil des öffentlichen Interfaces. Üblicherweise ist der Einsatz des Oberver-Patterns aber vor allem dadurch motiviert, daß auf diesem Weg eine maximale Entkopplung des Observers vom Observable erreicht werden kann. In unserer Situation ist jedoch eine weitere Entkopplung nicht möglich, da der Morphismus in jedem Fall den Zugriff auf das minimale Strategy-Interface benötigt. Deshalb ersparen wir uns hier den zusätzlichen Implementierungsaufwand und den potentiellen Laufzeit-Overhead, den das Observer-Pattern mit sich bringt.

7.3.3 Ein Framework für Constraint Satisfaction Probleme

Um prinzipiell eine Wiederverwendung zu ermöglichen, vor allem aber, um Abhängigkeiten zwischen den Systemteilen zu minimieren und klar zu definieren, entwerfen wir ein Framework für die Repräsentation und Lösung von binären Constraint Satisfaction Problemen. Unter einem Framework verstehen wir ein Softwaresystem, das einen vorimplementierten Rahmen für eine Klasse von Anwendungen bereitstellt, indem es deren Gemeinsamkeiten analysiert und von den Unterschieden abstrahiert. Im Entwurf eines Frameworks spiegelt sich diese Abstraktion in Interfaces und abstrakten Klassen wieder, die erst in einem konkreten System implementiert werden, welches das Framework benutzt. Das Framework für sich ist im allgemeinen kein lauffähiges System.

Bei den Constraint Satisfaction Problemen fällt es uns nicht schwer, die allgemeinen von den problemspezifischen Aspekten zu trennen; schließlich wurde schon die theoretische Definition eines CSPs im Geiste eines Frameworks erdacht, um nämlich viele verschiedene Probleme mit einer gemeinsamen Methode lösen zu können. Die problemspezifischen Teile eines CSPs sind offensichtlich in der Definition der konkreten Constraints und Queries gekapselt. Nur Constraints und Queries benötigen das Wissen über die konkreten Variablen- und Werte-Domains eines Problems, darüber hinaus sind einfache Mengen von identifizierbaren Elementen ausreichend. Das gesamte Lösungsverfahren schließlich läßt sich mit dem abstrakten Wissen über das Erfülltsein von Constraints implementieren, und natürlich liegt in der Wiederverwendung dieses allgemeinen Lösungsverfahrens für viele konkrete Ausprägungen des

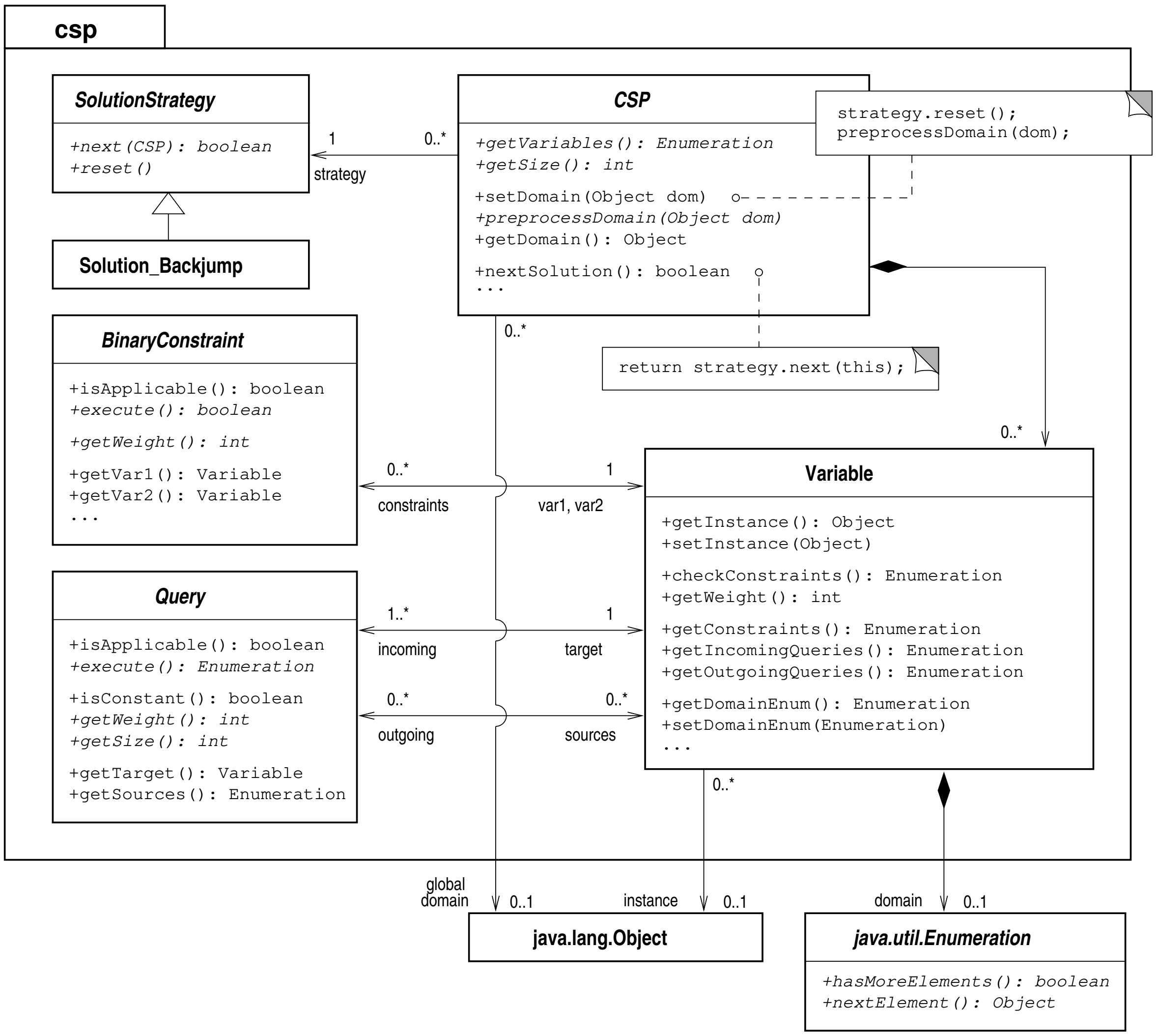


Abbildung 7.8: Die wichtigsten Klassen des CSP Frameworks.

Problems der eigentliche Sinn des Frameworks.

Im Entwurf des Frameworks – Abb. 7.8 zeigt die wichtigsten Klassen und deren Beziehungen – finden wir die problemspezifischen Aspekte als abstrakte Klassen repräsentiert: `BinaryConstraint`, `Query` und `CSP`. Ein Anwendungssystem, das das Framework benutzt, muß diese Klassen mit Hilfe des Vererbungsmechanismus konkretisieren. Die Klasse `Variable` und das nach dem Strategy-Pattern entkoppelte Lösungsverfahren `Solution_Backjump` (Abschnitt 7.3.4) lassen sich dagegen auf den abstrakten Interfaces der übrigen Klassen vollständig definieren.

Wir beschreiben nun kurz die Funktionalität der einzelnen Klassen des Frameworks:

`BinaryConstraint`
: Die wichtigste Methode der Klasse `BinaryConstraint` ist `execute()`, mit der geprüft werden kann, ob ein Constraint durch die aktuelle Belegung seiner Variablen erfüllt ist. Diese Methode ist abstrakt, denn die eigentliche Definition des Constraints, die sich da-

hinter verbirgt, kann natürlich erst in einem konkreten Constraint erfolgen. Vor einem Aufruf von `execute()` müssen wir sicherstellen, daß beide Variablen des Constraints überhaupt belegt sind; diese Vorbedingung kann über `isApplicable()` abgefragt werden.

`Query`

Eine ähnliche Aufgabe hat `isApplicable()` auch in der Klasse `Query`: ein Query ist nur anwendbar, wenn alle Variablen im Vorbereich des Queries – also die Source-Knoten in der Hyperkantendarstellung – belegt sind. Zusätzlich fordern wir, daß die Zielvariable noch nicht belegt ist, denn sonst ist die Anfrage überflüssig. Die „Anwendung“ erfolgt dann über die Methode `execute()`; als Ergebnis liefert sie eine Aufzählung der in Frage kommenden Werte für die Zielvariable. Die Implementierung von `execute()` setzt die eigentliche Definition eines konkreten Queries um und kann deshalb ganz analog zu den Constraint-Definitionen erst in den Subklassen von `Query` erfolgen.

Die Methode `isConstant()` prüft, ob ein `Query` unabhängig von der Variablenbelegung ist. Diese Eigenschaft kann in einem Lösungsverfahren ggf. zu Optimierungszwecken genutzt werden. In der vorgegebenen Implementierung der Methode gilt ein Query genau dann als konstant, wenn sein Vorbereich leer ist.

`Variable`

Mit der Methode `setInstance()` kann eine Variable mit einem Wert belegt werden, mit `getInstance()` wird auf den aktuellen Wert zugegriffen. Typisiert ist der Wert einer Variable dabei als `Object`, so daß in einem konkreten CSP jede beliebige Klasse als Werte-Implementierung eingesetzt werden kann[6]. Sehr wichtig für die Implementierung eines Lösungsverfahrens ist die Methode `checkConstraints()`, mit der geprüft wird, ob die aktuelle Belegung einer Variable konsistent ist, ob sie also alle anwendbaren Constraints erfüllt, an denen die Variable beteiligt ist. Ist das nicht der Fall, so gibt die Methode eine Aufzählung aller Variablen zurück, deren aktuelle Belegung die Verletzung eines Constraints verursacht haben. Dazu implementiert `checkConstraints()` die innere Schleife des Algorithmus `bj_target` von Seite 88, nämlich die Berechnung der Menge T bezüglich der aktuellen Belegung der Variable.

Der Suffix „`Enum`“ der beiden Methoden `get-` und `setDomainEnum()` der Klasse `Variable` soll darauf hinweisen, daß hier entgegen der üblichen Praxis das konsumtive Verhalten eines Objektes vom Typ `Enumeration` exponiert wird. Ein einmal erreichter Zustand der Iteration in einem Wertebereich wird also auch durch einen erneuten Aufruf von `getDomainEnum()` nicht zurückgesetzt. Dieses Verhalten kann bei der Implementierung eines auf Backtracking basierenden Lösungsverfahrens ausgenutzt werden, wo nach einem Rückschritt auf den nächsten noch nicht besuchten Wert des Domains zugegriffen werden muß.

`CSP`

Die Klasse `CSP` übernimmt die Funktion eines Containers für die Variablen: Mit `getVariables()` wird auf die Variablen zugegriffen, `getSize()` liefert ihre Anzahl. `setDomain()` setzt den *globalen Wertebereich* des Problems; im einfachsten Fall ist das die Vereinigungsmenge der Domains für die einzelnen Variablen. Als Typ für den Wertebereich wählen wir wiederum `Object`, so daß die Wahl der Implementierung in ei-

[6] In Java erbt jede Klasse implizit von `java.lang.Object`.

nem konkreten CSP nicht eingeschränkt wird. Durch die `setDomain()`-Methode wird es möglich, dieselbe Problemstellung für verschiedene Wertebereiche zu lösen, ohne das gesamte CSP neu aufbauen zu müssen. Bezogen auf das Beispiel der Ansatzsuche bedeutet das etwa, daß wir Ansätze derselben Regel in verschiedene Arbeitsgraphen berechnen können, indem wir ein konkretes `CSP` einfach für verschiedene Domains – sprich: Arbeitsgraphen – lösen. `preprocessDomain()` ist eine *Template Method* nach [GHJV95] und ermöglicht einer konkreten `CSP`-Implementierung, zu Optimierungszwecken einen eben gesetzten Domain zu analysieren. `nextSolution()` schließlich startet das Lösungsverfahren, das versucht, die Variablen des CSPs so zu belegen, daß alle Constraints erfüllt werden. Dabei wird die Belegung von Variablen, die bereits zu Beginn des Verfahrens belegt waren, nicht verändert. Somit wird es möglich, partiell vorgegebene Lösungen zu vervollständigen. Ganz analog zu `nextCompletion()` in der Klasse `ALR_Morphism` (vgl. Abschnitt 7.3.2) kann auch `nextSolution()` wiederholt aufgerufen werden, um alle Lösungen eines CSPs bzw. alle Vervollständigungen einer partiellen Lösung zu berechnen.

`SolutionStrategy`

`SolutionStrategy` definiert das Interface für die Implementierung von Lösungsverfahren im Strategy-Pattern, die von `nextSolution()` in `CSP` aufgerufen werden. Mit der Methode `reset()` wird der Berechnungszustand des Algorithmus initialisiert. Nach einem `reset()` ist sichergestellt, daß durch den nächsten Aufruf von `next()` die *erste* lösende Erweiterung der Teilbelegung des übergebenen CSPs berechnet wird, und daß durch wiederholtes Aufrufen tatsächlich *alle* weiteren Lösungen gefunden werden. Das CSP veranlaßt automatisch ein `reset()`, wenn sein globaler Wertebereich neu gesetzt wird.

Abschnitt 7.3.4 beschreibt die Implementierung der konkreten Lösungsstrategie `Solution_Backjump`.

`InstantiationHook` (Abb. 7.14, S. 118)

Die Klasse `InstantiationHook` dient dazu, Seiteneffekte bei der Instanziierung und Deinstanziierung von Variablen zu erzielen, die für die Implementierung bestimmter Queries und Constraints benötigt werden. In Abschnitt 7.3.6 wird das Prinzip dazu am Beispiel der Implementierung von Attribut-Constraints demonstriert.

Bemerkung 7.13 (*Domains*) Konstante Queries werden in unserem Framework auch zur Darstellung der Domains für die einzelnen Variablen verwendet. Deshalb fordern wir, daß es in einem Anwendungssystem zu jeder Variable mindestens ein konstantes Query geben muß. Der Aufruf von `execute()` auf einem Query liefert uns dann eine Wertemenge, die anschließend mit der Methode `setDomainEnum()` zum Wertebereich der entsprechenden Zielvariable des Queries erklärt werden kann. Diese Vorgehensweise mag zunächst umständlich erscheinen; schließlich könnte man das Ergebnis einer Anfrage auch automatisch der jeweiligen Zielvariable als Domain zuweisen. Oder wir könnten eine Methode `getDomain()` in der Klasse `Variable` definieren, die sich aus den eingehenden Queries selbst einen geeigneten Domain auswählt. Tatsächlich ist aber die Bestimmung der Domains über Queries ein weites Feld, das reichlich Raum für Optimierungen läßt. So sind wir z.B. natürlich daran interessiert, von mehreren anwendbaren Queries dasjenige als Domain auszuwählen, das die wenigsten Kandidaten liefert. Durch Schnittmengenbildung zwischen mehreren Queries können wir die

Kandidatenliste oftmals weiter reduzieren, was aber andererseits einen Berechnungsoverhead mit sich bringen kann, der die Vorteile deutlich überwiegt. Und schließlich ist es auch für gewisse Optimierungen an den Lösungsverfahren wichtig, auf Informationen über die Konstruktion der Domains zugreifen zu können. Deshalb entscheiden wir uns an dieser Stelle bewußt gegen einen festverdrahteten Automatismus, stattdessen für die flexible Lösung, die die Verantwortung für das Bestimmen der Domains an konkrete Lösungsstrategien überträgt. △

Bemerkung 7.14 (*Variablenordnung*) Bereits aus den Einführungsabschnitten 7.1.3 und 7.1.5 über Backtracking-basierte Lösungsverfahren wissen wir, daß die Variablenordnung, die einem Lösungsalgorithmus zugrundeliegt, eine ganz wesentliche Auswirkung auf dessen Laufzeitverhalten hat. Nach dem *First-Fail-Prinzip* sollten diejenigen Variablen zuerst instanziiert werden, die an besonders vielen oder besonders „strengen“ Constraints beteiligt sind. Als ein Kriterium für eine Variablenordnung bietet das `Variable`-Interface die Methode `getWeight()`, mit der eine Gewichtung der Variable abgefragt werden kann. Diese Gewichtung berechnet sich aus der Summe der Gewichte aller Constraints, an denen die Variable beteiligt ist, plus der Summe der Gewichte der ausgehenden Queries. Die Gewichte von Constraints und Queries werden durch Implementierung der jeweiligen `getWeight()`-Methoden erst im konkreten Anwendungsumfeld definiert. Für alle Gewichtungen gilt: Die Priorität steigt mit dem Gewicht.

Diese starre Gewichtung ist aber bei weitem nicht das einzige Kriterium für eine Variablenordnung. Ein großer Teil der Literatur über Constraint Satisfaction Probleme beschäftigt sich mit Heuristiken zur Variablenordnung, von der MinWidth-Ordnung [Fre82] bis hin zu dynamischen Ordnungsverfahren, z.B. [Pur83, GMP+96]. Aus denselben Überlegungen wie oben für die Konstruktion der Domains übertragen wir deshalb die Verantwortung für eine konkrete Variablenordnung an die Implementierung der Lösungsverfahren. △

Nachdem wir uns nun einen Überblick über das Framework für binäre CSPs verschafft haben, werden wir es in Abschnitt 7.3.5 auf das Problem der Morphismusvervollständigung hin konkretisieren. Dies wird uns als Beispiel für die Anwendung des Frameworks helfen, die Funktionsweise des `csp`-Packages besser zu verstehen. Im folgenden Abschnitt werfen wir aber zunächst noch einen Blick auf den Entwurf des Lösungsverfahrens `Solution_Backjump`.

7.3.4 Das Lösungsverfahren `Solution_Backjump`

Mit `Solution_Backjump` stellen wir in unserem CSP-Framework ein Lösungsverfahren zur Verfügung, das auf dem in Abschnitt 7.1.5 vorgestellten Backjumping-Prinzip beruht. Das Konzept der Domainreduktion durch Queries (vgl. Abschnitt 7.2.3) erzwingt jedoch eine geringfügige Anpassung des Algorithmus `bj_target` von Seite 88. Die Idee des Backjumping besteht darin, im Falle eines Backtracking nach einer Sackgasse bei einer Variable x_k alle diejenigen Vorgängervariablen zu überspringen, deren Belegung mit Sicherheit keine Auswirkungen auf die Konsistenz der Werte aus D_k hat. Da wir uns dafür entschieden haben, die Domains der Variablen durch Queries zu repräsentieren, ist $D_k = Q_{Y,x_k}$ mit $Y \subseteq \{x_1, \dots, x_{k-1}\}$, und dieses Query ist definitionsgemäß abhängig von der Belegung Γ_Y der Variablenmenge Y. Wenn aber die Belegung der Variablen aus Y den Domain D_k verändert, dann ist auch nicht mehr sicher, daß D_k keine konsistenten Werte enthält. Deshalb muß der Vorbereich Y

des Queries, das den Domain der aktuellen Variable bestimmt hat, in die Menge der möglichen Backjumptargets U aus dem Algorithmus `bj_target` aufgenommen werden. Wir passen die ersten Zeilen des Algorithmus entsprechend an, und nennen die Query-taugliche Version „`bjq_target`“:

Algorithmus 7.15 (*Query-Erweiterung von Algorithmus 7.8*)

`bjq_target`$(x_k, Q_{Y,x_k}) :=$
`begin`
  $U := Y$
  $D_k := Q_{Y,x_k}$
  `loop` für jeden Wert $v \in D_k$:
  `begin`
    $\vdots$

△

In Abschnitt 7.2.3 haben wir auf die Möglichkeit hingewiesen, den Domain für eine Variable als Schnittmenge aus mehreren Queries zu bestimmen. In diesem Fall müßte selbstverständlich die Vereinigungsmenge aller Vorbereiche der beteiligten Queries in die Menge U der möglichen Backjumptargets eingehen. Je größer also die Anzahl der beteiligten Queries wird, desto größer wird im allgemeinen auch die Menge der potentiellen Backjumptargets, und damit sinkt die zu erwartende Größe eines ggf. auszuführenden Backjumps. Dies ist ein weiterer Grund, weshalb mit dem Einsatz der Schnittmengentechnik vorsichtig umgegangen werden sollte.

Das in der Klasse `Solution_Backjump` implementierte Lösungsverfahren findet alle Lösungen eines CSPs, injektive sowie nicht-injektive. Gemäß einer Anforderung an die Ansatzsuche (vgl. Abschnitt 7.3.1) wollen wir aber die Möglichkeit haben, nicht-injektive Lösungen auszuschließen. `Solution_InjBackjump` ist eine Spezialisierung des `Solution_Backjump`-Verfahrens, die ausschließlich injektive Lösungen betrachtet. Die Implementierung von `Solution_InjBackjump` ist dabei trivial, es wird lediglich ein Schalter in der Superklasse gesetzt, der das Lösungsverfahren in einen „Injektivitätsmodus“ versetzt. In diesem Modus wird nach jeder Instanziierung einer Variable geprüft, ob der aktuelle Wert bereits einer anderen Variable zugewiesen worden ist. Nur wenn das nicht der Fall ist, wird mit der Überprüfung der Constraints fortgefahren. Die bereits vergebenen Werte werden in einer Hashtabelle gesammelt, so daß jeder einzelne Injektivitätstest lediglich konstanten Aufwand verursacht.

Außer der Realisierung der eigentlichen Lösungsverfahren haben wir beim Entwurf des `csp`-Frameworks in Abschnitt 7.3.3 noch zwei weitere Kompetenzen an die Implementierungen des Interfaces `SolutionStrategy` übertragen, nämlich

- die Berechnung einer günstigen Variablenordnung und
- die Bestimmung der einzelnen Domains für die Variablen.

Auf die wesentliche Bedeutung der Variablenordnung für die Effizienz von Backtracking-basierten Lösungsalgorithmen, zu denen auch das Backjumping-Verfahren zählt, haben wir

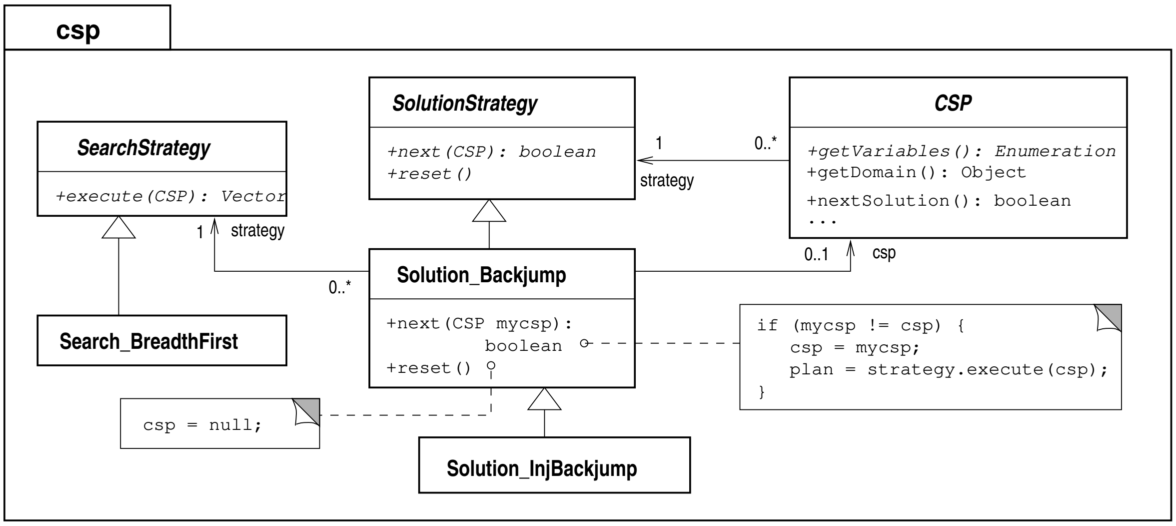


Abbildung 7.9: Das Lösungsverfahren `Solution_Backjump` und das Interface zur Variablenordnung `SearchStrategy`.

bereits verschiedentlich hingewiesen. Deshalb setzen wir an dieser Stelle erneut das bewährte Strategy-Pattern ein, um die Funktionalität für die Berechnung einer Variablenordnung auszugliedern und damit ein komfortables Experimentieren mit verschiedenen Ordnungen zu ermöglichen. In Abbildung 7.9 ist das Interface `SearchStrategy` für Variablenordnungen mit seiner Strategy-Beziehung zum Lösungsverfahren `Solution_Backjump` zu sehen. Das Interface `SearchStrategy` besteht aus einer einzigen Methode: `execute()` bekommt das CSP als Parameter, dessen Variablen geordnet werden sollen, und liefert einen `Vector` zurück. Dieser Vektor enthält für jede Variable des CSPs, die beim Aufruf von `execute()` noch nicht belegt war, genau ein Query mit dieser Variable als Zielvariable. Durch die Ordnung der Queries im Vektor ist die Variablenordnung gegeben, und jedes Query gibt gleichzeitig den Domain für seine Zielvariable an. Damit haben wir sozusagen zwei Fliegen mit einer Klappe geschlagen und die beiden zusätzlichen Implementierungskompetenzen aus der eigentlichen Implementierung des Verfahrens in eine eigene Strategie verbannt. Aus der Idee, die Variablenordnung an den Queries zu orientieren, rührt auch der Name `SearchStrategy`, denn dieses Vorgehen erinnert an die Berechnung der sog. *search plans* in PROGRES [Zün96].

In PROGRES wird der *search plan* für eine Regel berechnet, sobald die Regel fertig eingegeben wurde, also unabhängig von einem konkreten Arbeitsgraphen. Später wird immer derselbe Plan verwendet, unabhängig davon, auf welchen Arbeitsgraphen die Regel angewendet wird. Bezüglich des durch Graphregeln in PROGRES spezifizierten Systems heißt das, daß die Berechnung des *search plans* zur Compilezeit stattfindet. Das hat den großen Vorteil, daß man sich einen aufwendigen Algorithmus zur Bestimmung eines günstigen Plans leisten kann, denn dieser Aufwand schlägt sich nicht in der Laufzeit des spezifizierten Systems nieder, sondern lediglich in der Antwortzeit des Editors in der Spezifikationsphase. Der Nachteil ist jedoch, daß auf die Gegebenheiten eines konkreten Arbeitsgraphen nicht mehr eingegangen werden kann. Das betrifft insbesondere so wichtige Faktoren wie die konkrete Häufigkeitsverteilung der Objekttypen im Arbeitsgraphen. Auch partielle Ansatzvorgaben können für die Suchplanerstellung nicht mehr ausgenutzt werden. Letzteres ist für das PROGRES-System kein Kriterium, denn partielle Ansätze liegen hier nur in Form von Regelparametern vor, diese wiederum sind aber als formale Parameter zur Compilezeit bereits bekannt und werden bei der Berechnung des *search plans* mit höchster Priorität berücksichtigt.

Im AGG-System können wir aber jederzeit durch die Vorgabe beliebiger partieller Ansätze in die Ausführung einer Spezifikation eingreifen. Auf der Ebene der CSPs bedeutet das, daß das zu lösende CSP bereits eine beliebige Teilbelegung seiner Variablen haben kann. Um das entscheidende Optimierungspotential nutzen zu können, das in dieser Teilbelegung steckt, bleibt uns keine andere Wahl, als die Berechnung des Suchplans auf die Laufzeit zu verschieben. In dem Codefragment zur Methode `next()` der Klasse `Solution_Backjump` in Abb. 7.9 können wir erkennen, wie die Suchplanberechnung immer dann angestoßen wird, wenn ein neues CSP zur Lösung übergeben wird. Auch nach einem `reset()` wird grundsätzlich ein neuer Plan berechnet; so ruft ein `CSP` z.B. explizit `reset()` auf seiner `SolutionStrategy` auf, wenn ein neuer globaler Domain gesetzt wird.

`Search_BreadthFirst` ist eine konkrete Strategie zur Variablenordnung, die aus der Umgebung aller bereits instanziierten Variablen diejenige als nächste instanziiert, deren `getWeight()`-Methode das höchste Gewicht ausweist. Unter der Umgebung einer Variable verstehen wir dabei alle anderen Variablen, die durch Constraints mit ihr verbunden sind. Nachdem die Variable mit dem höchsten Gewicht bestimmt wurde, wird unter allen Queries, die diese Variable als Ziel haben, das als Domain am besten geeignete ausgesucht. Dazu sind zwei Kriterien zu überprüfen:

1. Ist das Query überhaupt anwendbar, d.h., sind alle seine Quellvariablen bereits instanziiert?

2. Wie groß ist die zu erwartende Ergebnismenge des Queries?

Um die Größe der Ergebnismenge abzuschätzen, gibt es zum einen die `Query`-Methode `getWeight()`. Durch sie wird jeder konkreten Query-Klasse ein festes Gewicht zugeteilt, das umgekehrt proportional zu der durchschnittlich zu erwartenden Anzahl von Ergebniswerten definiert wird. Greifen wir für ein Beispiel kurz auf den nächsten Abschnitt vor: Die Klasse `Query_Incoming` aus Abb. 7.11 bekommt ein größeres Gewicht zugewiesen als die Klasse `Query_Type`, weil anzunehmen ist, daß in einem konkreten Graphen eine Anfrage nach allen Objekten eines bestimmten Typs mehr Elemente liefern wird als eine Anfrage nach den bei einem bestimmten Graphobjekt eingehenden Kanten. Im Einzelfall kann diese Annahme aber durchaus fehl gehen. Deshalb bietet die Klasse `Query` eine weitere Bewertungsmethode namens `getSize()`. Diese Methode darf nur unter der Bedingung angewandt werden, daß der konkrete globale Domain des CSPs bekannt ist, und liefert eine von diesem Domain abhängige Schätzung für die Anzahl der Ergebniswerte des Queries. Diese Schätzung basiert auf Analysen des Domains; so könnte z.B. für ein `Query_Incoming` die durchschnittliche Anzahl eingehender Kanten bestimmt werden. Für konstante Queries kann `getSize()` sogar die exakte Größe der Ergebnismenge liefern, zum Beispiel die Anzahl der Graphobjekte eines bestimmten Typs. Derartige Analysen werden von der Methode `preprocessDomain()` eines konkreten CSPs vorgenommen.

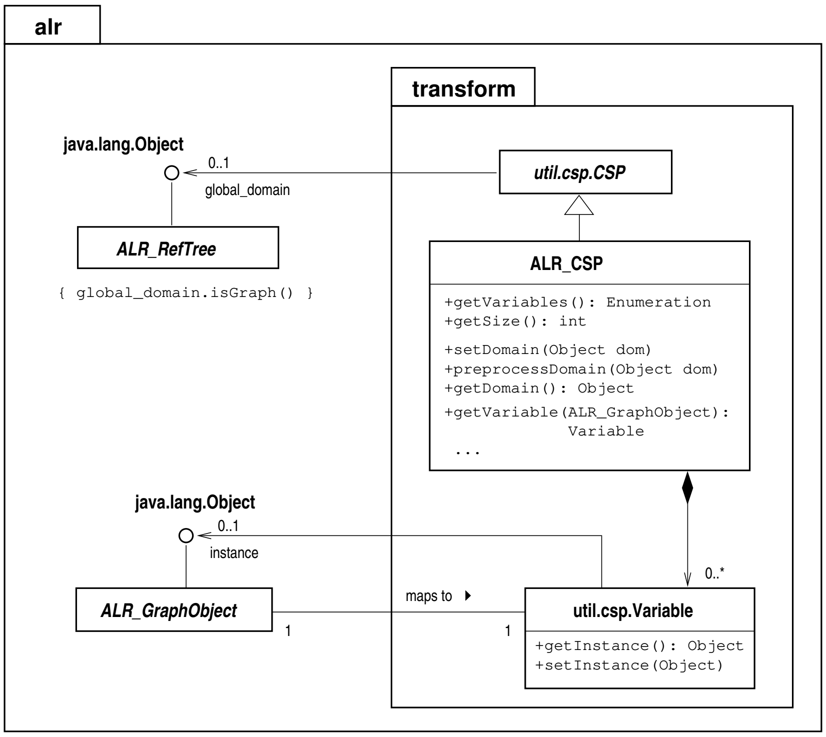


Abbildung 7.10: Variablen, Werte und Domains des CSPs zur Vervollständigung von ALR-Morphismen.

7.3.5 Das Verfahren `Completion_CSP` zur Morphismusvervollständigung

Im vorigen Abschnitt haben wird den objektorientierten Entwurf für ein Framework beschrieben, das allgemeine binäre CSPs lösen kann. In diesem Abschnitt geht es nun darum, den in Abschnitt 7.2.1 theoretisch untersuchten Übersetzungsschritt vom Problem der Ansatzsuche in ein CSP softwaretechnisch umzusetzen, um dieses dann mit Hilfe der allgemeinen Verfahren zu lösen, die das Framework zur Verfügung stellt. Obwohl wir im AGG-System attributierte ALR-Morphismen vervollständigen wollen, während wir in Abschnitt 7.2.1 den Übersetzungsschritt nur am Beispiel einfacher Graphen durchgeführt haben, verläuft die Konstruktion absolut analog; es müssen lediglich einige Constraints mehr berücksichtigt werden.

In Abschnitt 7.2.1 haben wir einfach die Objekte des Originalgraphen des Morphismus als Variablenmenge definiert. Da in unserem Systementwurf aber Variablen und Graphobjekte bereits durch unabhängige Klassen implementiert werden, tritt an die Stelle der Identität eine bijektive Abbildung zwischen den Graphobjekten aus dem Originalgraphen und den Variablen.

Im Zentrum des Entwurfs steht die Klasse `ALR_CSP`, die die Variablen und ihre Bijektion zu den Graphobjekten verwaltet. In Abbildung 7.10 ist diese Bijektion als 1:1-Beziehung mit dem Namen „maps to“ gut zu erkennen. `ALR_CSP` implementiert die abstrakte Methoden `getVariables()` und `getSize()` aus der Klasse `CSP` des Frameworks und erlaubt damit den Zugriff auf die Variablen sowie auf ihre Anzahl. Darüber hinaus ergänzt `ALR_CSP` das Interface von `CSP` um die Funktion `getVariable()`, die die einem Graphobjekt bijektiv zugeordnete

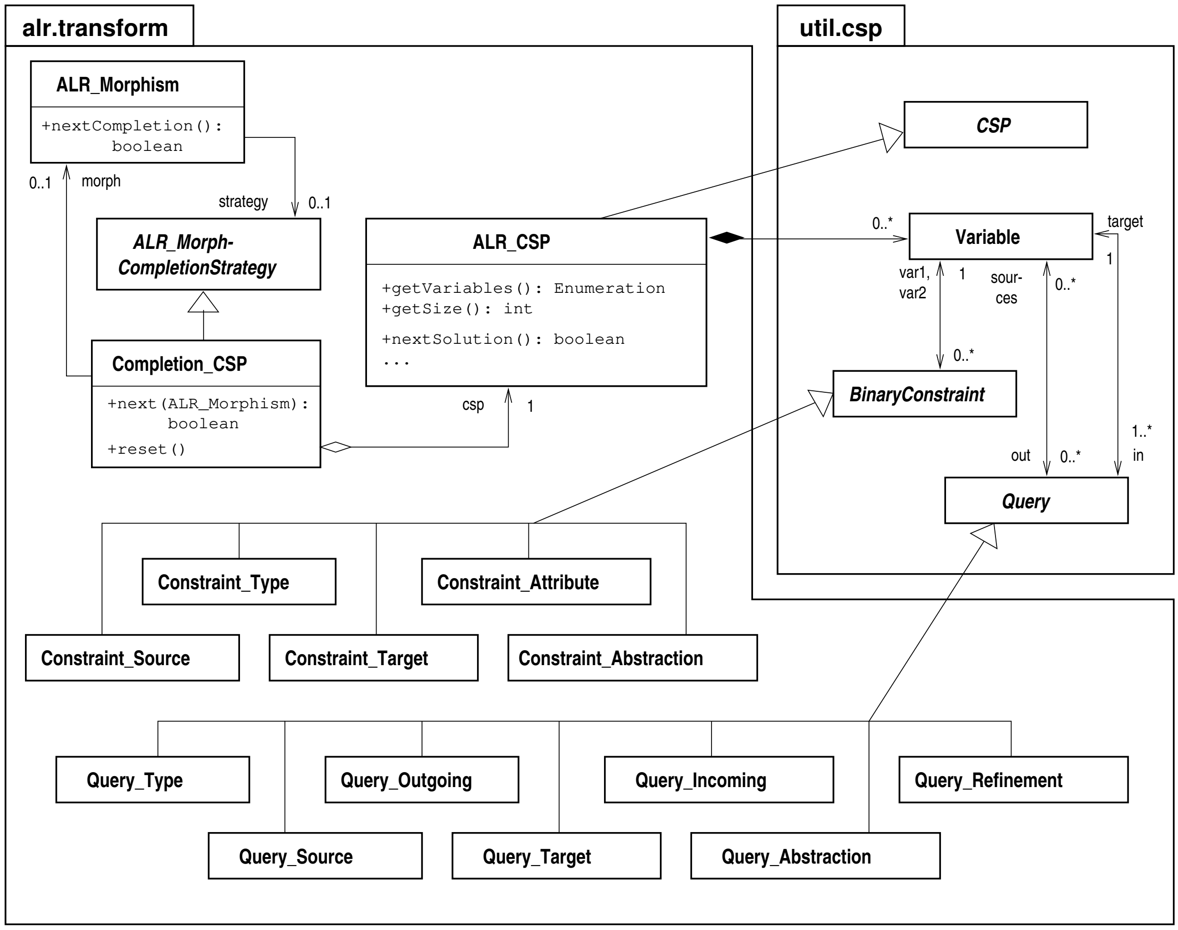


Abbildung 7.11: Das Verfahren `Completion_CSP` zur Vervollständigung von ALR-Morphismen.

Variable liefert.

So wie der Originalgraph eines ALR-Morphismus der Variablenmenge des zugehörigen CSPs entspricht, so ergibt der Bildgraph den globalen Domain für die Belegung der Variablen. Der Bildgraph wird im Entwurf repräsentiert durch eine Instanz `global_domain` vom Typ `ALR_RefTree`, wobei die Zusicherung gilt, daß `global_domain.isGraph()` wahr ist. Die Assoziation zu und die Zugriffsoperationen auf den Domain erbt `ALR_CSP` von `CSP`, wo der Domain nur als abstraktes `Object` bekannt ist. Für die konkrete Implementierung `ALR_CSP` gilt aber natürlich die Zusicherung, daß der Domain ein ALR-Graph ist, und deshalb kann auch auf das vollständige Interface von `ALR_RefTree` zugegriffen werden. Analoges gilt für die Werte der Variablen mit ihren Zugriffsoperationen `get-` und `setInstance()`: Die Werte sind im Interface als `Object` typisiert, aber es gilt die Zusicherung, daß es sich um Instanzen von `ALR_GraphObject` handelt. Mehr noch: Natürlich sind die Werte auch Elemente des Domains, also Graphobjekte im Bildgraphen des Morphismus, der vervollständigt werden soll.

In Abschnitt 7.3.2 haben wir `ALR_MorphCompletion` als Schnittstelle für beliebige Implementierungen von Vervollständigungsverfahren definiert. Unsere Implementierung auf Basis von Constraint-Satisfaction nennen wir `Completion_CSP`. Abbildung 7.11 zeigt die am Entwurf dieses Verfahrens beteiligten Klassen der Packages `alr.transform` und `util.csp` im Überblick. Gut zu erkennen sind hier die konkreten Klassen für das CSP und für alle Constraints und Queries sowie deren Vererbungsbeziehungen zu den abstrakten Klassen des `csp`-Frameworks. `Completion_CSP` kann auf eine Instanz von `ALR_CSP` zugreifen, um einen Aufruf von `next()`, der die nächste Vervollständigung des übergebenen Morphismus anfordert, durch den Aufruf von `nextSolution()` auf die nächste Lösung des entsprechenden CSPs abzubilden.

Um die dynamischen Vorgänge bei einem Aufruf des Vervollständigungsverfahrens besser zu verstehen und dabei insbesondere zu klären, bei welcher Gelegenheit die einzelnen Klassen instanziiert werden, betrachten wir in Abb. 7.12 ein Sequenzdiagramm des Aufrufs von `nextCompletion()` auf einem Objekt `morph` vom Typ `ALR_Morphism`. Das Bild zeigt den ersten Aufruf, es sind also noch keine Vervollständigungen desselben Morphismus berechnet worden. Zur besseren Übersicht kann der Gesamtablauf in drei Phasen untergliedert werden:

- In der *Init-Phase* wird das CSP mit seinen Constraints und Queries, seinen Lösungs- und Suchstrategien erzeugt und entsprechend eines ggf. vorgegebenen partiellen Morphismus initialisiert.
- Die *Computation*- oder *Berechnungsphase* beinhaltet die Berechnung der Variablenordnung und der lösenden Variablenbelegung.
- In der *Finit-Phase* wird eine ggf. gefundene Lösung des CSPs auf den ursprünglichen Morphismus übertragen.

Im folgenden wollen wir uns etwas ausführlicher mit dem Diagramm beschäftigen.

Der Morphismus delegiert den Aufruf von `nextCompletion()` direkt an seine `ALR_MorphCompletionStrategy`, in diesem Fall eine Instanz `comp` von `Completion_CSP`, und übergibt dabei eine Referenz auf sich selbst. `comp` bestimmt zunächst den Attributkontext, in dem die Belegung der Attributvariablen erfolgen soll. Auf diesen Schritt wollen wir hier nicht näher eingehen; den Problemen mit den Attributen bei der Morphismusvervollständigung wird später ein eigener Abschnitt gewidmet. Nachdem vom aufrufenden Morphismus der Originalgraph abgefragt wurde, erzeugt `comp` eine Instanz von `ALR_CSP` und übergibt ihr den Graphen und den Attributkontext. Das CSP instanziiert dann seine `SolutionStrategy`, diese wiederum ihre Strategie zur Variablenordnung. Die hier angegebenen konkreten Implementierungsklassen sind als Beispiel anzusehen, sie sind natürlich austauschbar; das ist der Sinn des Strategy-Patterns.

Nun erfolgt die Konstruktion des eigentlichen Constraint-Graphen: Das CSP erzeugt je eine Variable zu jedem Graphobjekt des Originalgraphen, den sein Konstruktor als aktuellen Parameter `vargraph` übergeben bekam. Die bijektive Beziehung zwischen Graphobjekten und Variablen wird dabei in einer Hash-Tabelle umgesetzt. Dann wird `vargraph` analysiert, und entsprechend werden konkrete Constraints zwischen den Variablen installiert. Diese Analyse erfolgt anhand von Bedingungen, die ganz entsprechend formuliert werden können, wie wir das in der Constraint-Tabelle von Konstruktion 7.10 auf Seite 93 bereits getan haben. Auch für die Queries lassen sich entsprechende Bedingungen aufstellen. Üblicherweise korrespondieren Queries mit bestimmten Constraints; existiert beispielsweise zwischen zwei Variablen ein Source-Constraint, so muß zwischen diesen Variablen in der einen Richtung ein `Query_Source`, in der anderen ein `Query_Outgoing` eingesetzt werden. Die gesamte Konstruktions- und Ana-

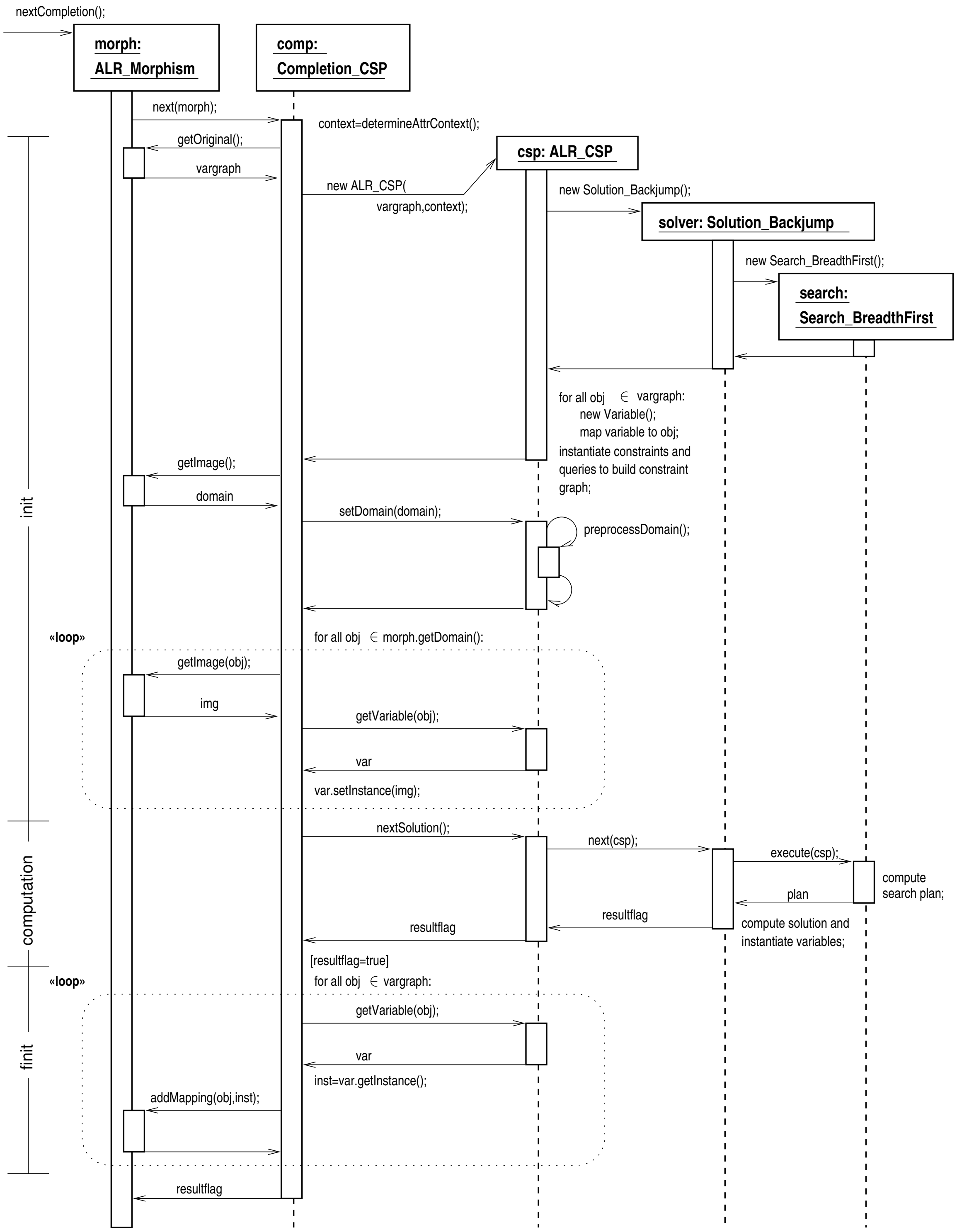


Abbildung 7.12: Sequenzdiagramm zur Vervollständigung von ALR-Morphismen.

lysephase benötigt lediglich zwei Durchläufe durch den Graphen `vargraph`, hat also linearen Aufwand.

Nachdem der Kontrollfluß zu `comp` zurückgekehrt ist, wird von dort der Bildgraph des Morphismus abgefragt und als Domain[7] des CSPs gesetzt. Dabei wird die Methode `preprocess-Domain()` angestoßen, die in unserer konkreten CSP-Implementierung eine Hash-Tabelle der Graphobjekte anlegt, in der Markierung der Objekte als Schlüssel dient. Damit wird den Type-Queries der effiziente Zugriff auf alle Graphobjekte eines Typs ermöglicht, was das Interface `ALR_RefTree` der ALR-Graphen nicht unterstützt. Dies ist ein Beispiel dafür, wie man Queries auch ohne direkte Unterstützung des zugrundeliegenden Datentyps verwirklichen kann. Die Type-Queries sind darüber hinaus konstant und für jedes Graphobjekt verfügbar, womit wir die vom `csp`-Framework geforderte Voraussetzung auch gleich erfüllt haben.

Im nächsten Schritt wird unserem Anspruch Rechnung getragen, partielle Morphismen vervollständigen zu können. Für alle Graphobjekte `obj`, die schon ein Bild `img` unter dem partiellen Morphismus `morph` haben, wird die entsprechende Variable bereits mit dem Wert `img` instanziiert. Die Belegung dieser vorinstanziierten Variablen wird im Laufe des nachfolgend mit `nextSolution()` gestarteten Lösungsverfahrens nicht mehr verändert. Das mit der Lösung beauftragte CSP gibt eine Referenz auf sich selbst an seine Strategie `solver` weiter, und diese berechnet zunächst einmal eine günstige Variablenordnung. Dazu kann das Wissen über die vorinstanziierten Variablen auf vielfältige Weise ausgenutzt werden. Dann wird versucht, eine Variablenbelegung zu finden, die die Vorgaben ergänzt und alle Constraints erfüllt. Hat die Suche keinen Erfolg, dann wird die Finit-Phase übersprungen und die negative Rückmeldung sogleich bis zum Morphismus weitergereicht. Ein positives Ergebnis, wie im Sequenzdiagramm aus Abb. 7.12 angenommen, wird jedoch vom Vervollständigungsverfahren `comp` zunächst abgefangen, um die berechnete Variablenbelegung in Abbildungsvorschriften für den Ausgangsmorphismus `morph` zu übersetzen. Danach terminiert die Methode `comp.next()` und meldet den Erfolg an den Morphismus zurück, der nun totalisiert ist.

Alle folgenden Aufrufe von `nextSolution()` auf demselben Morphismus `morph` berechnen die weiteren möglichen Vervollständigungen des Zustands vor dem ersten Aufruf. Dabei kann die gesamte Init-Phase übersprungen werden, und auch die Berechnung der Variablenordnung wird natürlich nicht wiederholt.

7.3.6 Attribut-Constraints

In diesem Abschnitt wollen wir uns den speziellen Problemen widmen, die bei der Definition und bei der Implementierung der Attribut-Constraints entstehen. Die Attribut-Constraints sollen dafür sorgen, daß in einem Morphismus nur solche Graphobjekte aufeinander abgebildet werden, deren Attributwerte zueinander passen. Was dieses „Passen“ genau bedeutet, hängt von verschiedenen Umständen ab, vgl. dazu Abschnitt 6.2.5; für unsere folgenden Betrachtungen gehen wir von dem üblichen Fall aus, daß in einem Ansatzmorphismus die Attributwerte zweier aufeinander abgebildeter Graphobjekte gleich sein müssen, ggf. unter der Berücksichtigung einer durch den bisherigen Ansatz induzierten Variablenbelegung.

Wir versuchen nun anhand des Beispiels in Abbildung 7.13, eine Definition für die

[7] Der globale Domain des CSPs entspricht gerade dem Codomain, also dem Bildbereich des Morphismus; diese Begriffsverschiebung liegt in den unterschiedlichen Sichtweisen begründet, die sich aus den Bereichen Kategorientheorie bzw. Constraint-Satisfaction ergeben.

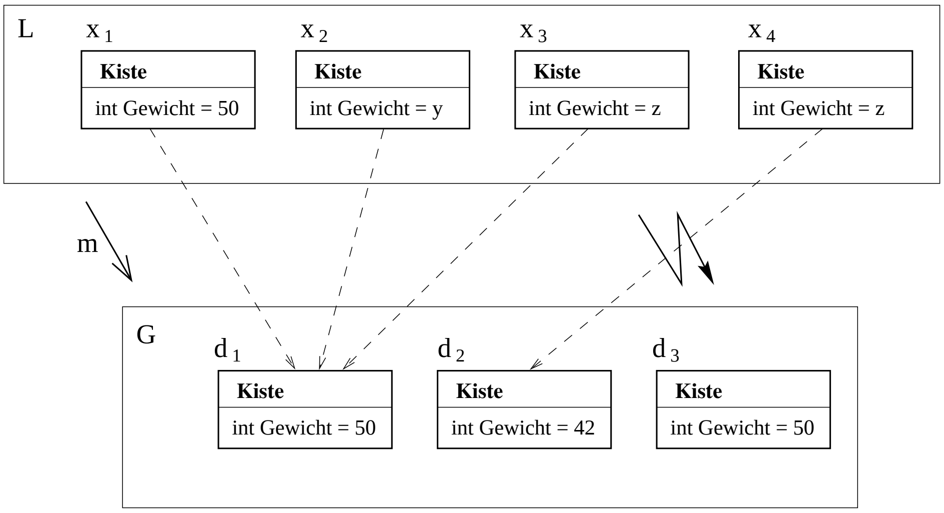


Abbildung 7.13: Implizite Anwendungsbedingung durch mehrmaliges Verwenden derselben Variable.

Attribut-Constraints C^{attr} zu finden, so wie wir es schon in Beispiel 7.9 (S. 91) für die Source- und Target-Constraints getan haben. Die Abbildung zeigt zwei über der Algebra der natürlichen Zahlen attributierte Graphen (Def. 2.3) $L = (L_V, \emptyset, \{Kiste\}, s_L, t_L, l_L, Nat_{\text{var}}, av_L, ae_L)$ und $G = (G_V, \emptyset, \{Kiste\}, s_G, t_G, l_G, Nat, av_G, ae_G)$ mit den Knotenmengen $L_V = \{x_1, x_2, x_3, x_4\}$ bzw. $G_V = \{d_1, d_2, d_3\}$ und einen Ansatzmorphismus $m : L \to G$. Dabei steht Nat_{var} für die Algebra der natürlichen Zahlen mit Variablen. Die Variablenmenge X des entsprechenden CSPs besteht dann gemäß Konstruktion 7.10 (S. 93) aus $X = \{x_1, x_2, x_3, x_4\}$, und die Domains ergeben sich zu $D = D_1 = D_2 = D_3 = D_4 = \{d_1, d_2, d_3\}$.

Von einem Ansatzmorphismus erwarten wir, daß aufgrund der geforderten Attributgleichheit z.B. x_1 auf d_1 abgebildet werden kann, nicht aber auf d_2. Im entsprechenden CSP können wir dies recht einfach durch das folgende unäre Constraint ausdrücken:

$$C^{\text{attr}}_{(x_1)} = \{d \in D_1 \mid av_L(x_1) = av_G(d)\} \tag{7.4}$$

Diese Definition erinnert sehr an die Definition des Typ-Constraints in Konstruktion 7.10, und tatsächlich kann man die Attributierung auch als eine Art dynamische Typisierung auffassen.

Betrachten wir als nächstes den Knoten x_2. Hier ist die Situation anders: da das Attribut von x_2 eine Variable[8] ist, wollen wir diesen Knoten auf jeden beliebigen Knoten des gleichen Typs in G abbilden können. Wie drücken wir das durch ein Constraint im entsprechenden CSP aus? Nun, wir brauchen gar kein Attribut-Constraint, weil wir die Abbildungsmöglichkeiten durch die Attributierung in diesem Fall nicht einschränken wollen. Alternativ können wir das triviale Constraint $C^{\text{attr}}_{(x_2)} = D_2$ definieren.

Nun bleiben noch die beiden Knoten x_3 und x_4 zu untersuchen. Beide sind mit derselben Variable z attributiert, womit ausgedrückt wird, daß die beiden Bilder dieser Knoten unter dem Morphismus m gleiche Attributwerte haben müssen. Wenn also wie in Abb. 7.13 $m(x_3) = d_1$ ist, dann darf x_4 nicht mehr auf d_2 abgebildet werden, wohl aber auf d_3. Diese Bedingung

[8] Wir haben es hier auf zwei verschiedenen Ebenen mit Variablen zu tun: Die x_i sind Variablen im CSP, also Graphobjekte in L, während y, z Attributvariablen aus Nat_{var} sind.

läßt sich durch die beiden folgenden binären Constraints formulieren:

$$C^{\text{attr}}_{(x_3,x_4)} = \{(d_a, d_b) \in D_3 \times D_4 \mid av_G(d_a) = av_G(d_b)\},$$
$$C^{\text{attr}}_{(x_4,x_3)} \text{ symmetrisch.}$$

Bis hierher verlief die Definition der Attribut-Constraints recht einfach. Allerdings haben wir drei verschiedene Constraintdefinitionen für drei verschiedene Attributierungssituationen unterscheiden müssen:

1. Attributierung mit einem Wert.
2. Attributierung mit einer Variable, die nur einmal vorkommt.
3. Attributierung mit einer Variable, die mehrfach verwendet wird.

Die Probleme beginnen jedoch bei dem Versuch, eine allgemeine Konstruktion anzugeben, wie wir es für die übrigen Constraints in Konstruktion 7.10 getan haben. Dazu müssen wir nämlich Bedingungen finden, um die drei unterschiedlichen Attributierungssituationen eindeutig zu charakterisieren. Das wird aber spätestens dann sehr aufwendig, wenn wir berücksichtigen wollen, daß die Attribute, die einem Graphobjekt über die Attributierungsoperation *av* zugeordnet werden, im allgemeinen Tupel sind, wobei jeder Tupeleintrag bezüglich der drei Attributierungssituationen unabhängig ist. Das führt dazu, daß Attribut-Constraints pro Tupeleintrag definiert werden müßten. In jedem Fall aber ist ein hohes Maß an Detailwissen über die Attributspezifikation erforderlich, um die entsprechende allgemeine Konstruktion für die Definition der Attribut-Constraints durchzuführen.

Die Implementierung verfolgt einen anderen Weg. Die öffentliche Attributrepräsentation der Attributkomponente ist opaque, sie folgt also gerade der Philosophie, möglichst *wenig* Informationen über die Attribute preiszugeben. Insbesondere ist es nicht möglich festzustellen, ob ein Attribut eine Variable ist oder nicht. Deshalb haben wir auch keine Chance, die Abhängigkeiten zwischen Graphobjekten mit gleichen Attributvariablen durch binäre Constraints darzustellen. Stattdessen bietet uns die Attributkomponente eine abstrakte Funktionalität nach dem Motto: „Gib mir zwei Attribute, und ich sage dir, ob sie passen.“ Die Information über die beiden Attribute allein reicht aber noch nicht ganz aus. Nehmen wir an, wir wollen in Abb. 7.13 x_4 auf d_2 abbilden, und fragen dazu die Attributkomponente: „Paßt z zu 42?“, dann hängt die Antwort davon ab, ob bereits vorher x_3 auf d_1 abgebildet wurde. Genauer gesagt ist entscheidend, ob z bereits belegt wurde, und wenn ja, mit welchem Wert. Deshalb erwartet die Attributkomponente außer den beiden Attributen auch die aktuelle Variablenbelegung als Eingabe der Anfrage. Woher nehmen wir aber diese Variablenbelegung, wenn wir noch nicht einmal wissen, welche Attribute Variablen sind? – Die Attributkomponente erzeugt sie implizit selbst. Wenn wir z.B. fragen, ob z zu 42 paßt, und dabei eine leere Variablenbelegung übergeben, dann wird die Antwort der Attributkomponente positiv sein, und gleichzeitig wird die übergebene Variablenbelegung um $z \mapsto 42$ erweitert. Diese erweiterte Belegung erhalten wir zurück und geben sie bei der nächsten Anfrage wieder mit. So wird die Belegung induktiv erzeugt. Außerdem bietet die Attributkomponente noch die Funktionalität, die Belegung einzelner Variablen zurückzunehmen, was z.B. im Rahmen der Ansatzsuche nach einem Backtracking-Verfahren unbedingt erforderlich ist.

Bemerkung 7.16 (*Entsprechungen in der konkreten* `attribute`*-Schnittstelle*) In der Terminologie der implementierten Schnittstelle der Attributkomponente (s. Abb. 6.6 auf S. 70) entspricht die Variablenbelegung einer Instanz der Klasse `AttrContext`, und die Anfrage, ob zwei Attribute zueinander passen, entspricht einem Aufruf der Methode `newMapping()` von `AttrManager`, die neben den Attributen einen Attributkontext als Parameter erwartet. Der Rückgabewert von `newMapping()` ist vom Typ `AttrMapping`. Dieser Wert repräsentiert genau die Änderung an der Variablenbelegung, die während des Aufrufs implizit durchgeführt wurde. Durch den Aufruf von `remove()` auf diesem `AttrMapping()` wird die entsprechende Änderung wieder zurückgenommen. △

Mit dieser Funktionalität läßt sich ein unäres Attribut-Constraint implementieren, das ganz ähnlich aussieht wie unsere erste Constraint-Definition (7.4) für die Attributierung ohne Variable:

$$C^{\text{attr}}_{(x_i)}(\sigma) = \{d \in D_i \mid av(x_i) =_\sigma av(d)\}$$

Dabei steht σ für die aktuelle Attributvariablenbelegung, und „$a =_\sigma b$“ bedeutet „a paßt unter der Belegung σ zu b“. In dieser Definition gehen wir aber davon aus, daß σ fest bleibt, daß also „$=_\sigma$“ seiteneffektfrei ist. Um mit diesem unären Constraint, das die Abhängigkeiten zwischen den CSP-Variablen nur implizit formuliert, dasselbe Lösungsverhalten zu realisieren wie das entsprechende explizite CSP, müssen wir dafür sorgen, daß σ immer gerade dann erweitert wird, wenn die CSP-Variable x_i belegt wird, und daß diese Erweiterung genau dann wieder zurückgenommen wird, wenn auch die Instanziierung von x_i aufgehoben wird.

Genau das ist aber das Problem bei der Implementierung: Wir müssen bei der Instanziierung einer CSP-Variable eine für das Attribut-Constraint spezifische Methode aufrufen, die uns die Attributvariablenbelegung erweitert, und entsprechend eine zweite, die die Erweiterung zurücknimmt, sobald die Instanziierung der CSP-Variable aufgehoben wird. Da die Klasse `Variable` aber Bestandteil des abstrakten CSP-Frameworks ist, das von konkreten Constraints keine Kenntnis haben soll, verallgemeinern wir die Situation und führen im Framework das Konzept der `InstantiationHooks` ein (Abbildung 7.14). Eine konkrete Implementierung des Interfaces `InstantiationHook` stellt mit `instantiate()` und `uninstantiate()` zwei Methoden bereit, die von einer Variable bei jeder Instanziierung respektive Deinstanziierung aufgerufen werden sollen, wobei die Variable eine Selbstreferenz an die Hook-Methoden übergibt. Die Schnittstelle von `Variable` wird in diesem Zusammenhang um die Methode `addInstantiationHook()` erweitert, mit der Instanzen von `InstantiationHook` in einer Variable installiert werden können. Daraus ergibt sich, daß eine Variable beliebig viele Hooks haben kann.

Abbildung 7.14 zeigt als Beispiel für die Verwendung eines Hooks die Implementierung der Klasse `Constraint_Attribute`. Diese Klasse implementiert selbst das `InstantiationHook`-Interface, um zu erreichen, daß die Routinen zur Manipulation des Attributkontextes zur Instanziierungs- bzw. Deinstanziierungszeit der entsprechenden CSP-Variable ausgeführt werden.

Durch die Verlagerung des Aufrufs von `newMapping()` in die Hook-Methode `instantiate()` erreichen wir, daß der eigentliche Constraint-Check-Aufruf `execute()` seiteneffektfrei bleibt. Damit ist sichergestellt, daß der Check auch bei mehrmaligem Aufruf immer dasselbe Ergebnis liefert.

Bemerkung 7.17 (*Auswirkungen auf Backjump*) Die Backjumping-Optimierung des einfachen Backtrackingverfahrens beruht auf der Analyse der Fehlerursache von Constraint-Checks; die Fehlerursache ergibt sich dabei aus den Abhängigkeiten, die durch die binären

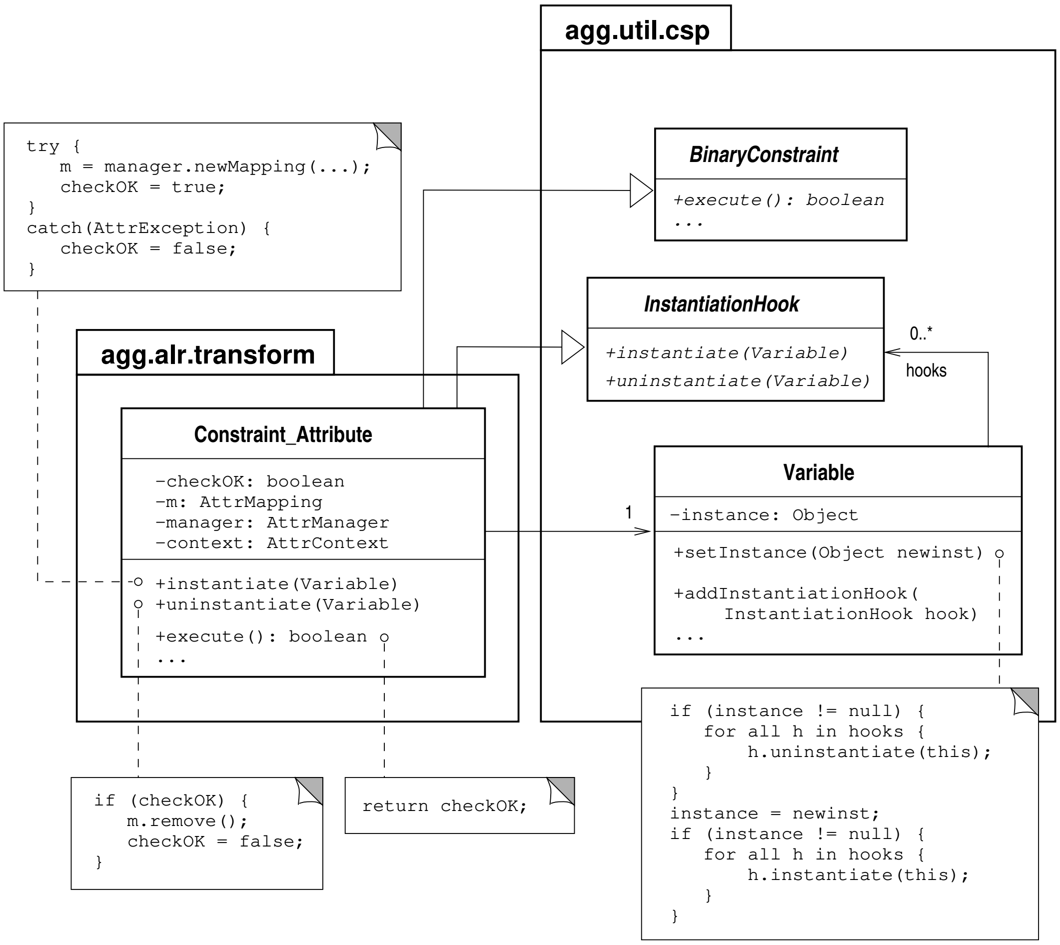


Abbildung 7.14: Implementierung von `Constraint_Attribute` als `InstantiationHook`.

Constraints ausgedrückt werden. Das Verfahren verläßt sich dabei auf die Vollständigkeit der angegebenen Abhängigkeiten: wenn zwei Variablen x_i, x_k nicht durch ein Constraint verbunden sind, wird davon ausgegangen, daß ein Scheitern der Belegung von x_i nicht durch eine Änderung der Belegung von x_k beeinflußt werden kann, und umgekehrt. Die implizite Formulierung von Abhängigkeiten in dem oben angegebenen unären Attribut-Constraint entzieht dem Backjumping-Verfahren jedoch die Grundlage. Um dennoch einen Einsatz der Backjumping-basierten Ansatzsuche in Verbindung mit dem `attribute`-Package zu ermöglichen, ist eine Erweiterung der Attributkomponente vorgesehen: Da der Komponente intern alle Informationen über die Abhängigkeiten vorliegen, kann sie selbst die Ursache eines Scheiterns von `newMapping()` bestimmen, und die entsprechende Information als zusätzliches Datum in der ausgelösten `AttrException` zur Verfügung stellen. In der Klasse `BinaryConstraint` gibt es eine geschützte Methode `getCause()`, die in der Standardimplementierung die Ursache eines fehlgeschlagenen Constraint-Checks aus den Constraint-Verbindungen bestimmt. Diese Methode wird in der Implementierung von `Constraint_Attribute` dann so überschrieben, daß die Fehlerursache stattdessen aus der entsprechenden `AttrException` bestimmt wird. Auf diese Weise können wir von der Art der Bestimmung der Fehlerursache abstrahieren, und die Lösungsverfahren arbeiten problemlos.

$\triangle$

Kapitel 8

Systemschicht für die Transformation einfacher Graphen

Wenn in der Praxis mit Graphen modelliert wird, dann wird im allgemeinen ein einfacher Graphbegriff zugrundegelegt, so wie wir ihn in der Einführung in Kapitel 2 verwendet haben. Dieser einfache Graphbegriff läßt sich als Spezialfall auf die ALR-Graphen zurückführen. Um aber Anwendungskomponenten, die auf einem einfachen Graphbegriff beruhen, den Modellierungsaufwand und den Umgang mit dem komplexeren Graphbegriff zu ersparen, bietet das AGG-System eine Implementierungsschicht zum Arbeiten mit einfachen Graphen, die von der Modellierung durch ALR-Graphen abstrahiert. Diese Schicht spiegelt die vollständige Funktionalität der ALR-Ebene wider, also insbesondere das Attributierungskonzept, die automatische Ansatzsuche und die Berechnung von Transformationsschritten. Abschnitt 8.1 beschreibt, wie die Implementierung der Systemschicht für einfache Graphen auf die ALR-Ebene zurückgeführt wird, und geht dabei besonders auf die Modellierung des Graphgrammatik-Begriffs durch einen ALR-Graphen ein. Dieses Wissen über die Konzeption der Implementierung ist jedoch keine Voraussetzung für das Verständnis der Anwendungsschnittstelle, die Abschnitt 8.2 in einer kompakten Übersicht präsentiert.

8.1 Konzeption

Die Implementierung der Graphtransformationsfunktionalität für einfache Graphen beruht auf der Tatsache, daß jeder einfache Graph äquivalent durch einen ALR-Graphen dargestellt werden kann. In dieser Repräsentation können wir die Graphen dann mit den in den bisherigen Kapiteln entwickelten Methoden bearbeiten.

Die Übersetzung eines einfachen Graphen in einen ALR-Graphen ist einfach. Jede Ebene eines ALR-Graphen ist für sich genommen bereits ein einfacher Graph, solange wir davon absehen, Kanten zwischen Kanten zu verwenden. Haben wir also einen einfachen Graphen G gegeben, so brauchen wir diesen nur als unterste Ebene eines ALR-Graphen G_{ALR} anzusehen. Über dieser Ebene fügen wir dann eine weitere Ebene ein, die nur einen einzigen Knoten h enthält, und abstrahieren alle Graphobjekte von G auf diesen Knoten h, der uns als das für einen ALR-Graphen obligatorische Top-Objekt dient. Jeder Morphismus zwischen einfachen Graphen induziert nun eindeutig einen ALR-Morphismus zwischen den entsprechenden

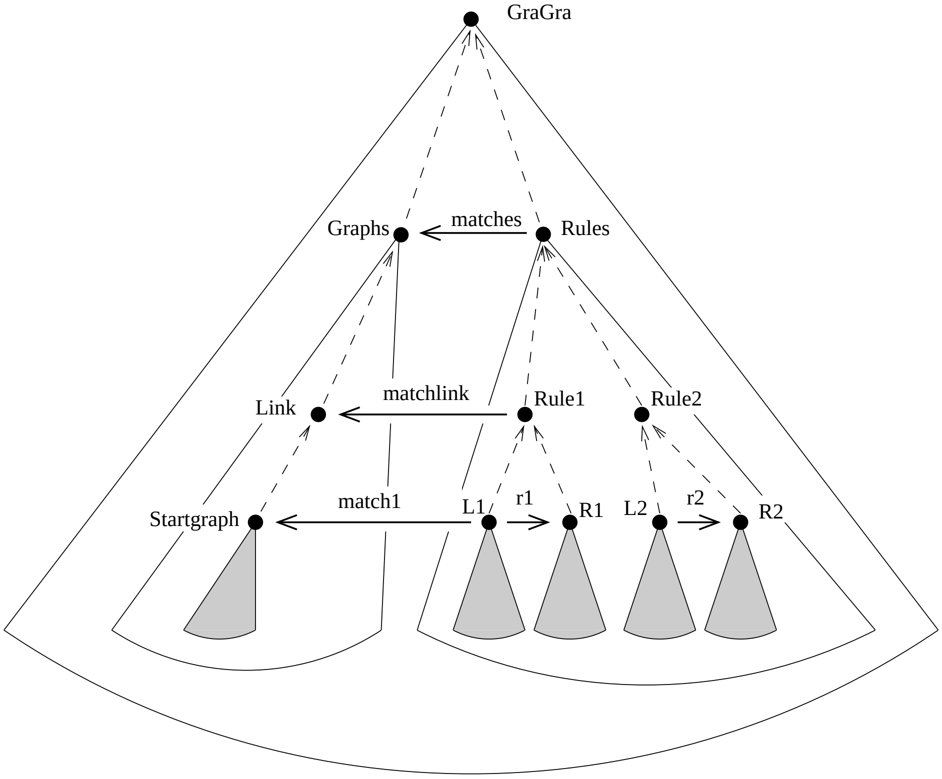


Abbildung 8.1: Modellierung einer Graphgrammatik als ALR-Graph.

ALR-Graph-Repräsentationen, und solange wir die Top-Objekte der beiden ALR-Graphen auf derselben Ebene eines umfassenden ALR-Graphen ansiedeln, können wir diesen ALR-Morphismus natürlich wieder als Verfeinerungsbaum einer Kante zwischen den Top-Objekten modellieren (vgl. Abschnitt 6.1.2). Daß diese Darstellungen von Graphen und Morphismen jeweils äquivalent sind, ist offensichtlich; es findet sich aber auch ein Beweis in [AR89].

Wir gehen jedoch noch einen Schritt weiter und nutzen die Meta-Modellierungsmöglichkeiten der ALR-Graphen, um eine ganze Graphgrammatik, bestehend aus einer Menge von Regeln und einem Startgraphen, als ALR-Graph zu modellieren. Abbildung 8.1 zeigt ein Beispiel für eine solche Graphgrammatik in Verfeinerungsbaumdarstellung. Die Graphgrammatik enthält zwei Regeln $Rule1$ und $Rule2$ und einen Startgraphen. Die gestrichelten Pfeile zeigen dabei auf die Abstraktionsobjekte von Knoten, die Abstraktionen der Kanten ergeben sich implizit. Die grau unterlegten Verfeinerungsbäume repräsentieren die ALR-Modellierungen der konkreten einfachen Graphen; diese Verfeinerungsbäume haben also unterhalb ihrer Top-Objekte jeweils nur noch eine Ebene. Diese Ebene ist gleichzeitig die unterste Ebene des umfassenden ALR-Graphen mit dem Top-Objekt $GraGra$, der damit aus insgesamt fünf Ebenen besteht. Jede Ebene kann dabei als diagrammatische Zusammenfassung der nächsttieferen Ebene angesehen werden. Besonders interessant ist dies für die Ebene der Top-Objekte $Startgraph, match1, L1, \ldots$, denn hier ergibt sich die aus der kategorientheoretischen Sicht bekannte Darstellung eines Diagramms über der Kategorie einfacher Graphen mit partiellen Morphismen.

Um eine Graphregel, bestehend aus zwei Graphen $L1, R1$ und einem Morphismus $r1$, ein-

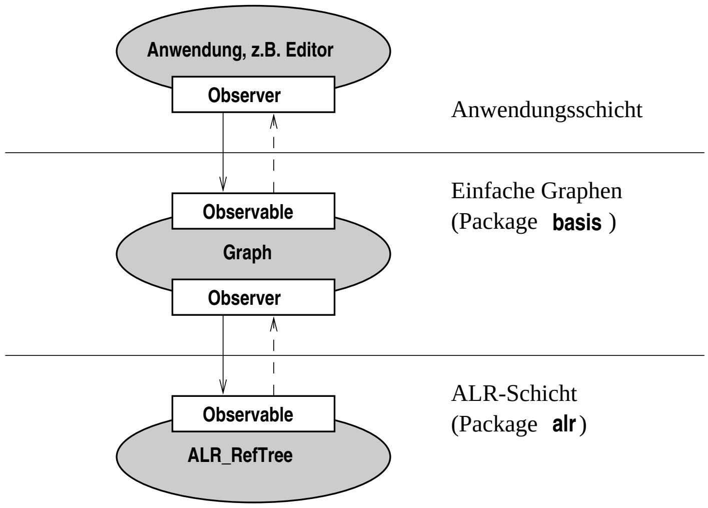


Abbildung 8.2: Schichtung von Observer-Pattern.

deutig durch einen ALR-Graphen darstellen zu können, werden die Top-Objekte der Graph- bzw. Morphismusrepräsentationen auf ein gemeinsames Top-Objekt $Rule1$ in der nächsthöheren Ebene abstrahiert, welches nun die Regel als Ganzes repräsentiert. Alle Regelrepräsentanten werden dann wiederum auf einen Knoten *Rules* abstrahiert, der die Regelmenge der Graphgrammatik repräsentiert. Auf derselben Ebene gibt es einen Knoten *Graphs* als Top-Objekt für die ALR-Repräsentation einer Menge von Graphen. Wir verwenden in der gegenwärtigen Implementierung nur eine einelementige Graphmenge, die aus dem Startgraphen besteht, der durch Transformationsschritte „in place“ manipuliert wird. Da wir Kanten in einem ALR-Graphen nicht zwischen verschiedenen Ebenen ziehen können und weil wir Morphismen als Verfeinerungsbäume von Kanten modellieren, müssen wir dafür sorgen, daß sich die Top-Objekte von Graphen, die wir durch Morphismen in Beziehung setzen wollen, immer auf derselben Ebene befinden. Insbesondere müssen wir Ansatzmorphismen zwischen der linken Seite einer Regel und dem Startgraphen angeben können, wie z.B. $match1 : L1 \rightarrow Startgraph$ in Abb. 8.1. Da wir aber in der Regelmodellierung einen zusätzlichen Abstraktionsschritt für die Zusammenfassung von Regelgraphen und Regelmorphismus zu einer Regel benötigt haben, müssen wir in der Modellierung der Graphen einen entsprechenden Knoten *Link* einfügen, damit das Ebenengefüge nicht in Schieflage gerät.

Die Implementierung der Systemschicht für einfache Graphen setzt die hier beschriebene Graphgrammatik-Modellierung in dem Java-Package `basis` 1:1 um. Das bedeutet, daß jede Klasse aus diesem Package, seien es Regeln, Graphen, Graphgrammatiken oder Morphismen, direkt oder indirekt durch einen Verfeinerungsbaum in der ALR-Schicht implementiert wird, also durch Instanzen von `ALR_RefTree`. Eine indirekte Modellierung ergibt sich z.B. für die Implementierung einfacher Morphismen, die zunächst auf die Klasse `ALR_Morphism` zurückgeführt werden, welche selbst aber wiederum als Verfeinerungsbaum implementiert ist. Die Klassen des `basis`-Packages übernehmen dabei die Rolle von Observern ihrer ALR-Repräsentationen. Da sie jedoch keine lokalen Zustandsdaten zu verwalten haben – ihr Zustand ist durch ihre jeweilige ALR-Modellierung vollständig definiert –, dient diese Rolle nur dazu, die von der ALR-Schicht generierten Änderungsinformationen abzufangen und zu filtern, um sodann nur solche Informationen an die eigenen Observer weiterzugeben, die in Bezug auf deren

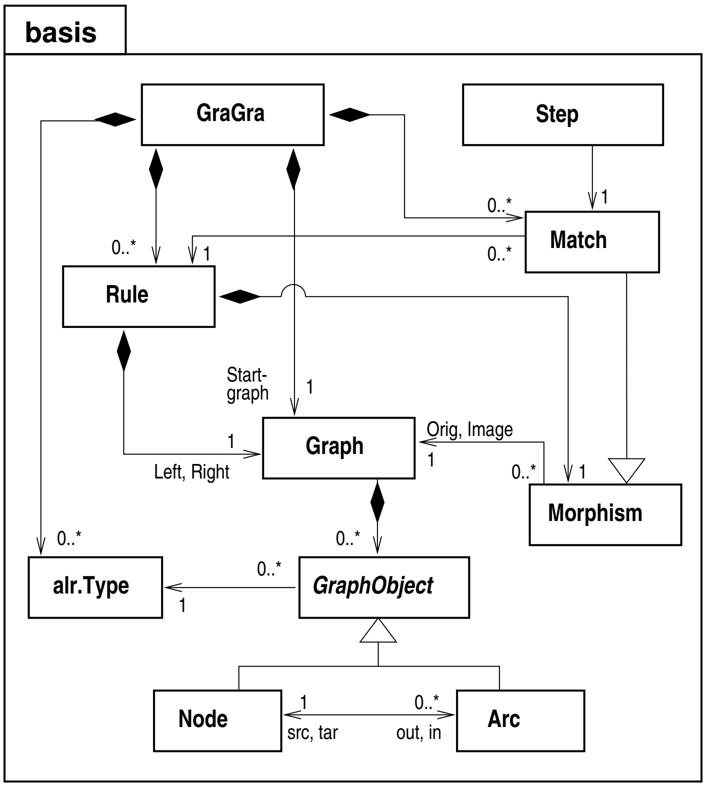


Abbildung 8.3: Überblick über die Klassen des `basis`-Packages.

einfache Graphsicht relevant sind. Abbildung 8.2 verdeutlicht schematisch die sich ergebende Schichtung von Observer-Pattern am Beispiel der `basis`-Klasse `Graph`.

8.2 Anwendungsschnittstelle

In diesem Abschnitt beschreiben wir die Schnittstellen für die Repräsentation von einfachen Graphen und deren Morphismen gemäß den Definitionen aus Abschnitt 2.1 sowie für die Ansatzsuche und die Transformation auf diesem Graphmodell. Diese Schnittstellen spiegeln im wesentlichen die Funktionalität und die Philosophie ihrer Entsprechungen auf der ALR-Ebene, wie sie in den jeweiligen Abschnitten zur Anwendungsschnittstelle in den Kapiteln 5, 6 und 7 ausführlich dargestellt wurde. Auch die in diesen Kapiteln formulierten Entwurfsanforderungen gelten, auf das Modell der einfachen Graphen übertragen, natürlich entsprechend. Aufgrund der vorhandenen Analogien wollen wir uns bei der Erklärung der Schnittstellen an dieser Stelle relativ kurz fassen und verweisen für nähere Einzelheiten ggf. auf die Schnittstellenbeschreibungen der ALR-Ebene bzw. auf die API-Dokumentation im Anhang A.

Abbildung 8.3 gibt einen Überblick über die wichtigsten Klassen des `basis`-Packages, das die Systemschicht der einfachen Graphen im AGG-System realisiert. Eine zentrale Rolle spielt hier die Klasse `GraGra`, die eine Graphgrammatik repräsentiert, bestehend aus einem Startgraphen und einer Menge von Regeln. `GraGra` ist für die zentrale Instanziierung und Verwaltung von Typen, Regeln und Ansätzen zuständig. Etwas aus dem Rahmen fällt die Klasse `Type`, die

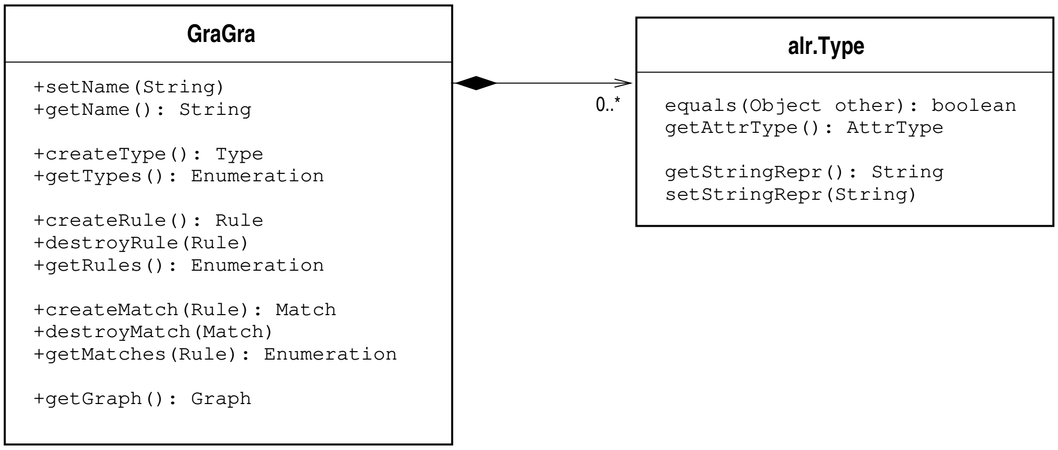


Abbildung 8.4: Die Interfaces der Klassen `GraGra` und `Type`.

für die Typisierung von Graphobjekten genutzt wird; sie ist die einzige Klasse, deren Interface direkt aus dem `alr`-Package übernommen wird. Ansonsten wurde bei dem Entwurf der `basis`-Klassen konsequent der Grundsatz befolgt, die zugrundeliegende ALR-Implementierung nicht an die Anwendungsschnittstelle durchscheinen zu lassen. Die Ausnahme bezüglich der Klasse `Type` ist unbedeutend, weil deren Interface keine ALR-spezifischen Aspekte enthält; da eine Adaption des Interfaces einen unverhältnismäßig hohen technischen Aufwand erfordert hätte, wurde an dieser Stelle die Entscheidung für die direkte Übernahme getroffen.

Wir beschreiben nun die Interfaces der einzelnen `basis`-Klassen.

`GraGra` (Abb. 8.4)

Über die Methoden `get`- und `setName()` kann eine Graphgrammatik mit einem Namen versehen werden. Wie bei allen Klassen, die eine solche Benennung erlauben, hat der Name keine Bedeutung im System, sondern dient ausschließlich der Orientierung eines menschlichen Anwenders. Um sicherzustellen, daß alle Regeln und Graphen einer Graphgrammatik über derselben Typmenge typisiert werden, wird diese Typmenge von der Graphgrammatik zentral verwaltet. `createType()` erzeugt neue Typen, `getTypes()` zählt alle Elemente der aktuellen Typmenge auf. Das Löschen von Typen wird aus technischen Gründen derzeit nicht unterstützt.

Neben dem Startgraphen, der mit der Methode `getGraph()` abgefragt werden kann, besteht eine Graphgrammatik nach der gebräuchlichen Vorstellung aus einer Menge von Graphtransformationsregeln. `createRule()` erzeugt eine neue Regel, die implizit in die Regelmenge aufgenommen wird. Linke und rechte Seite einer neu erzeugten Regel sind leer, entsprechend auch ihr Morphismus. `destroyRule()` entfernt eine bestimmte Regel aus der Regelmenge, und `getRules()` iteriert über alle vorhandenen Regeln.

In Abweichung von der üblichen Vorstellung, daß eine Graphgrammatik nur den statischen Ausgangszustand für eine Graphtransformation beschreibt, dient der Startgraph in der Klasse `GraGra` gleichzeitig als Arbeitsgraph für Inplace-Graphtransformationsschritte. Deshalb erzeugt und verwaltet `GraGra` auch die Ansatzmorphismen von ihren Regeln in ihren Startgraphen. `createMatch()` erzeugt einen neuen Ansatz für eine gegebene Regel. Dieser Ansatz ist zunächst leer, also kein Ansatz im

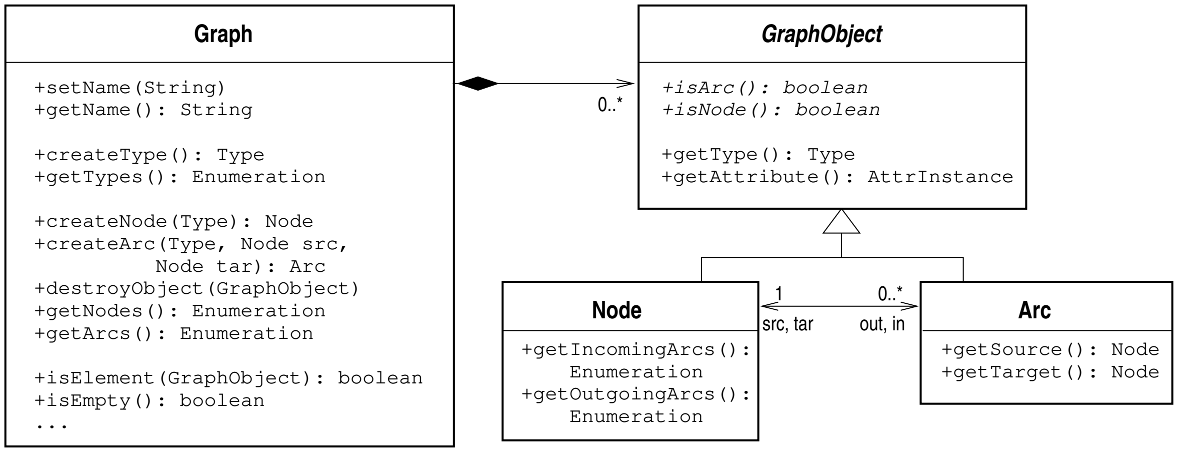


Abbildung 8.5: Die Interfaces für Graphen, Knoten und Kanten.

Sinne der Theorie, die Totalität verlangt. Der eigentliche Ansatz wird also erst anschließend mit Hilfe des Interfaces von `Morphism` aufgebaut. `getMatches()` liefert die Menge aller bisher erzeugten Ansätze zu einer gegebenen Regel; mit `destroyRule()` kann ein Ansatz aus dieser Menge entfernt werden.

`Type` (Abb. 8.4)
`equals()` prüft zwei Typen auf Gleichheit. `getAttrType()` liefert den Attributtyp, der dem vorliegenden Graphobjekttyp zugeordnet ist. Die Methoden `get-` und `setStringRepr()` erlauben die Definition eines Namens, der für die Repräsentation des Typs in einem Editor verwendet werden kann. Diese Namen müssen nicht eindeutig sein, d.h., die Gleichheit von Typen wird nicht über die Gleichheit ihrer Stringrepräsentation definiert.

`Graph` (Abb. 8.5)
Neben der Funktionalität zum Setzen und Lesen eines Namens bietet `Graph` aus Gründen der Bequemlichkeit dieselben Methoden zur Verwaltung von Typen wie `GraGra`. Die Aufrufe dieser Methoden werden intern an die Graphgrammatik delegiert, zu der der Graph gehört. Die zentrale Aufgabe eines Graphen ist jedoch die Instanziierung und Verwaltung von Knoten und Kanten. Mit den Methoden `createNode()` und `createArc()` werden Knoten und Kanten erzeugt und automatisch in die Knoten- bzw. Kantenmenge des Graphen aufgenommen. Als Parameter erwarten beide `create`-Methoden den gewünschten Typ des zu erzeugenden Objekts; für eine Kante müssen außerdem ihre Source- und Targetknoten angegeben werden. `getNodes()` und `getArcs()` liefern Aufzählungen der Knoten- und Kantenmengen; aus dem Graphen gelöscht werden Knoten und Kanten über dieselbe Methode `destroyObject()`. Beim Löschen von Knoten werden hängende Kanten implizit ebenfalls entfernt. Das Prädikat `isElement()` testet, ob ein gegebenes Graphobjekt zum Graphen gehört. `isEmpty()` ist wahr, wenn die Knoten- und Kantenmengen des Graphen leer sind.

`GraphObject`, `Node` **und** `Arc` (Abb. 8.5)
`GraphObject` definiert die gemeinsame Schnittstelle für Knoten und Kanten. `GraphObject` ist eine abstrakte Klasse, weil die Prädikate `isNode()` und `isArc()` erst

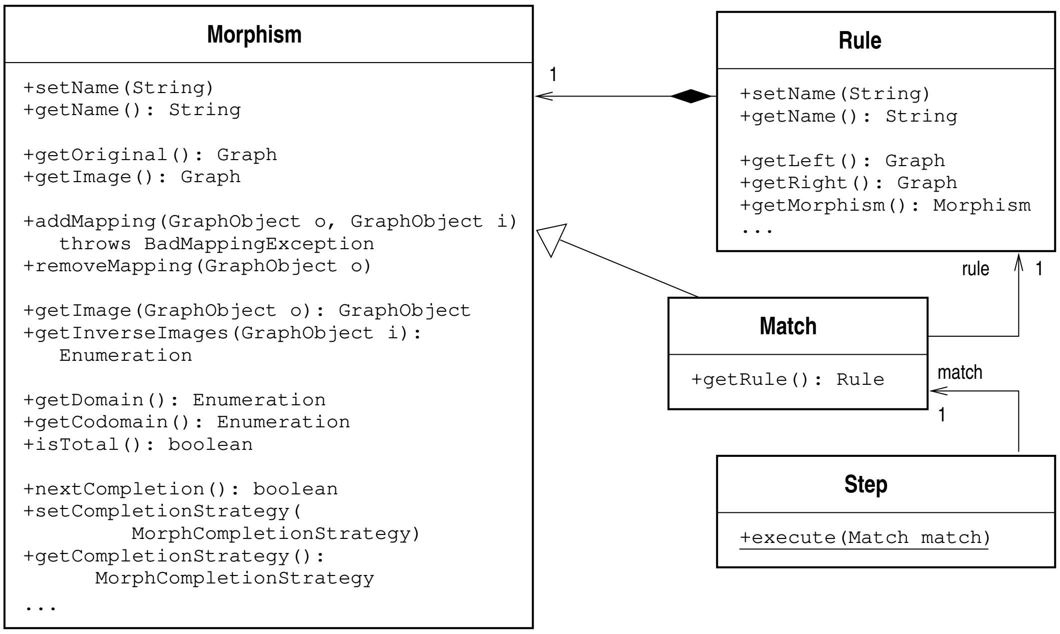


Abbildung 8.6: Die Interfaces für Morphismen, Regeln und Transformation.

von den Subklassen implementiert werden. Außer dem Test, ob es sich um einen Knoten oder eine Kante handelt, kann von jedem Graphobjekt sein Typ und sein Attributwert abgefragt werden. Mit den Methoden der Klasse `AttrInstance` kann das entsprechende Attribut dann z.B. editiert oder als Stringrepräsentation angezeigt werden. Für eine ausführliche Beschreibung der Attributschnittstellen siehe [Mel97].

Die Subklassen `Node` und `Arc` von `GraphObject` definieren jeweils die für Knoten bzw. Kanten spezifischen Methoden. So macht es nur für eine Kante Sinn, mittels `getSource()` bzw. `getTarget()` nach dem Source- oder Targetknoten zu fragen. Umgekehrt kann man nur von einem Knoten die ein- und ausgehenden Kanten erfragen.

`Morphism` (Abb. 8.6)

Ein Morphismus bietet zunächst die üblichen Methoden für die Verwaltung eines Namens. Die Methoden `getOriginal()` und `getImage()` erlauben den Zugriff auf Original- und Bildgraphen des Morphismus. Die eigentlichen Zuordnungsvorschriften zwischen Objekten aus dem Original- und dem Bildgraphen werden mit Hilfe der Methode `addMapping()` definiert. Wenn ein angegebenes Graphobjektpaar die Morphismuseigenschaften verletzt, dann wird die entsprechende Zuordnungsvorschrift nicht aufgenommen, und stattdessen eine Exception ausgelöst; die Exception enthält einen kurzen Klartexthinweis auf die genaue Fehlerursache. `Morphism` erlaubt also nur den Aufbau von konsistenten Morphismen.

Mit `removeMapping()` können einzelne Zuordnungsvorschriften aus einem Morphismus wieder entfernt werden. Als Parameter wird nur das Originalobjekt der zu entfernenden Zuordnungsvorschrift übergeben; aufgrund der Rechtseindeutigkeit einer Abbildung ist das gewünschte Graphobjektpaar dadurch eindeutig identifiziert.

`m.getImage(o)` liefert das Bild eines Graphobjekts `o` unter dem Morphismus `m`. Umgekehrt ergibt `m.getInverseImages(i)` eine Aufzählung aller Urbilder von `i` unter `m`. Die Methoden `getDomain()` und `getCodomain()` ermöglichen den Zugriff auf den Definitions- und Wertebereich eines Morphismus; die aufgezählten Graphobjekte sind also eine Teilmenge der Objektmenge des Original- oder des Bildgraphen. Die allgemeinen Morphismuseigenschaften sichern dabei zu, daß diese Teilmengen abgeschlossen sind bezüglich der Source- und Target-Operation, sie induzieren also Teilgraphen. Das Prädikat `isTotal()` prüft die Totalität eines Morphismus.

Mit `nextCompletion()` kann ein Morphismus totalisiert werden; damit bildet die Methode die Schnittstelle zur Ansatzsuche, vgl. 7.1.1. Wenn also der Rückgabewert eines Aufrufs `m.nextCompletion() == true` ist, dann gilt die Zusicherung, daß `m.isTotal()` wahr ist. Wird die Methode wiederholt aufgerufen, so werden nacheinander alle Vervollständigungen desjenigen Ausgangszustands Z berechnet, der vor dem ersten Aufruf von `nextCompletion()` bestand. Dieser Ausgangszustand Z wird erst durch den nächsten Aufruf der Methoden `add`- oder `removeMapping()` oder durch Modifikationen an den Original- und Bildgraphen des Morphismus aktualisiert.

Die Methoden `set`- und `getCompletionStrategy()` erlauben es, den konkreten Algorithmus zur Morphismusvervollständigung auszuwählen. Außerdem können über das Interface der Klasse `MorphCompletionStrategy` bestimmte Parameter gesetzt werden, die die Arbeitsweise der Algorithmen beeinflussen.

Bemerkung 8.1 (*Konsistenzsicherung von Morphismen*) Verletzungen der Morphismuseigenschaften durch Modifikationen an Original- und Bildgraphen des Morphismus werden automatisch repariert, indem die für die Verletzung verantwortlichen Zuordnungsvorschriften entfernt werden. Dadurch ist die Konsistenz einer Instanz von `Morphism` unter allen Umständen sichergestellt. Für weitere Erläuterungen siehe Abschnitt 6.2.6. △

`Rule`, `Match` und `Step` (Abb. 8.6)

`Rule` modelliert eine Graphregel, bestehend aus zwei Graphen für die beiden Regelseiten und einem Regelmorphismus. Die Methoden der Klasse ermöglichen den Zugriff auf diese Komponenten und bieten darüber hinaus die übliche Funktionalität zur Namensgebung.

`Match` ist eine Spezialisierung von `Morphism` und ergänzt deren Funktionalität lediglich um den Zugriff auf die zu einem Ansatz gehörige Regel. `Match`- und `Rule`-Morphismen unterscheiden sich bezüglich ihres Attributierungsverhaltens; vgl. Abschnitt 6.1.3. Die entsprechenden Attributkontexte werden von beiden Klassen automatisch erzeugt und gesetzt.

`Step` definiert die Schnittstelle zur Graphtransformation. Ein Aufruf der Klassenmethode `execute(match)` berechnet das Ergebnis der Anwendung der Graphregel `match.getRule()` auf den Arbeitsgraphen `match.getImage()` unter dem Ansatz `match`. Die Berechnung erfolgt „in place“, d.h. der Arbeitsgraph wird direkt manipuliert.

8.3 Änderungsinformationen

Von den Klassen des `basis`-Packages implementieren `Graph` und `Morphism` die Observable-Schnittstelle `ExtObservable` (vgl. Abschnitt 3.2). Das bedeutet, daß die Klassen einer Anwen-

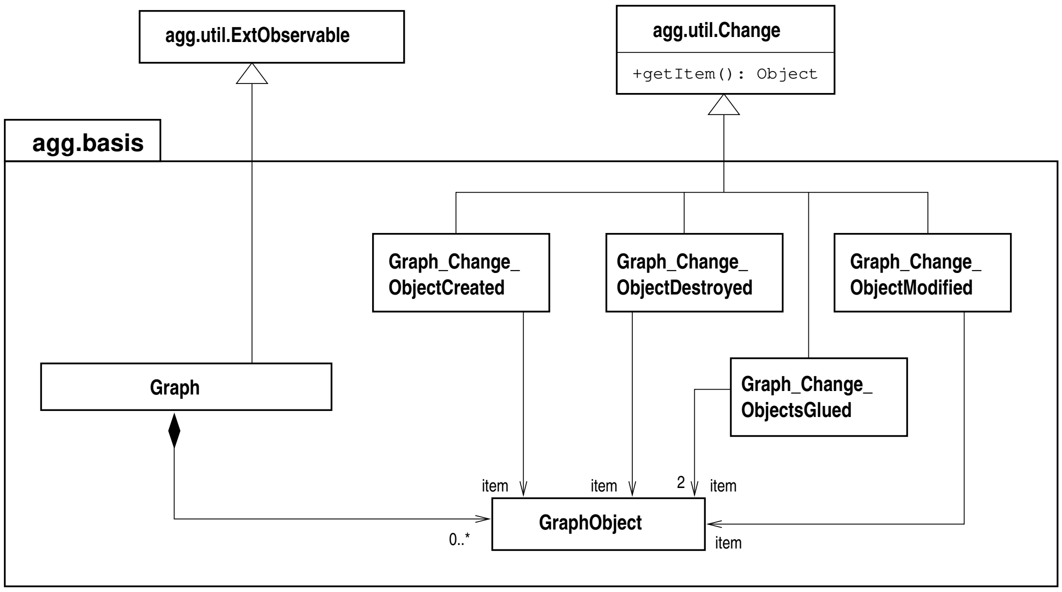


Abbildung 8.7: Änderungsinformationen des Observables `Graph`.

`Change`	Item: Itemtyp	Bedeutung
`Graph_Change_ObjectCreated`	`x: GraphObject`	`x` wurde erzeugt.
`Graph_Change_ObjectDestroyed`	`x: GraphObject`	`x` wurde gelöscht.
`Graph_Change_ObjectModified`	`x: GraphObject`	Die Attributierung von `x` wurde modifiziert.
`Graph_Change_ObjectsGlued`	`(x,y): Pair` (von `GraphObject`)	`x` und `y` wurden verklebt.
`Change_ObservableGone`	`g: Graph`	`g` wurde gelöscht (vgl. Abschnitt 3.2).
`null`	–	Undefinierte Zustandsänderung. Vom Observer wird eine vollständige Resynchronisation mit dem Zustand des Observables erwartet.

Tabelle 8.1: Semantik der Änderungsinformationen, die eine Instanz `g` von `Graph` an ihre Observer schicken kann.

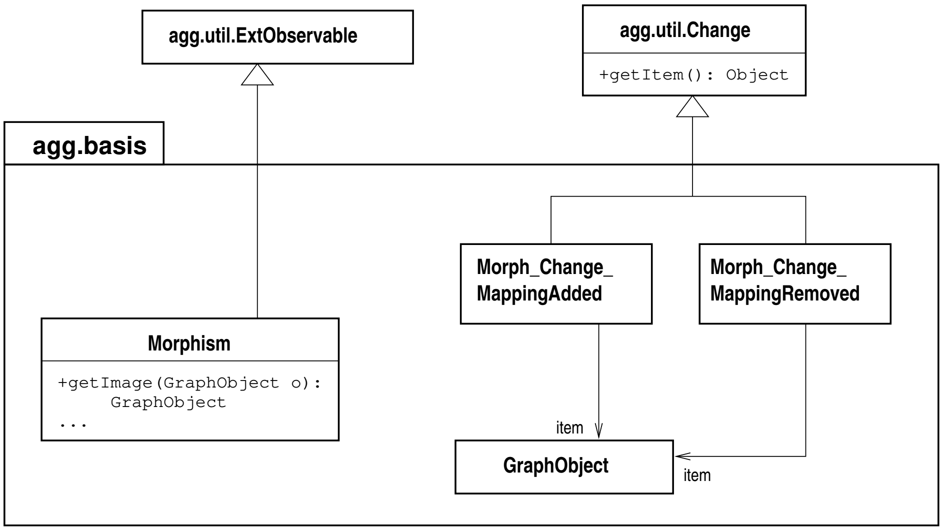


Abbildung 8.8: Änderungsinformationen des Observables `Morphism`.

`Change`	Item: Itemtyp	Bedeutung
`Morph_Change_ MappingAdded`	`x: GraphObject`	Die Zuordnungsvorschrift `x` $\mapsto$ `m.getImage(x)` wurde aufgenommen.
`Morph_Change_ MappingRemoved`	`x: GraphObject`	Die Zuordnungsvorschrift für `x` wurde entfernt.
`Change_ ObservableGone`	`m: Morphism`	`m` wurde gelöscht (vgl. Abschnitt 3.2).
`null`	–	Undefinierte Zustandsänderung. Vom Observer wird eine vollständige Resynchronisation mit dem Zustand des Observables erwartet.

Tabelle 8.2: Semantik der Änderungsinformationen, die eine Instanz `m` von `Morphism` an ihre Observer schicken kann.

dungsschicht die Möglichkeit haben, sich im Rahmen eines Observer-Patterns bei Instanzen von `Graph` oder `Morphism` anzumelden und sich über Änderungen auf deren Zustand automatisch informieren zu lassen. Als Subklasse von `Morphism` bietet natürlich auch `Match` diese Möglichkeit, und wenn eine Anwendung über die Zustandsänderungen einer Instanz von `Rule` informiert werden will, dann meldet sie sich als Observer bei deren Komponenten an, also je nach Bedarf bei ihrem linken oder rechten Regelgraphen und/oder ihrem Regelmorphismus.

Abbildung 8.7 zeigt die möglichen Typen von Änderungsinformationen, die das Observable `Graph` erzeugen kann, zusammen mit ihren *Item*-Beziehungen, die bei einer konkreten Instanz einer Änderungsinformation auf das von der Änderung betroffene Objekt verweist. Tabelle 8.1 erläutert die Bedeutung der Änderungsinformationen von `Graph` aus der Sicht eines Observers. Die Verklebung von Graphobjekten, die durch die Änderungsinformation `Graph_Change_ObjectsGlued` gemeldet wird, kann durch Transformationsschritte mit nichtinjektiven Regeln verursacht werden, oder aber durch einen direkten Aufruf der entsprechenden `glue()`-Methode auf der ALR-Graph-Repräsentation einer `Graph`-Instanz, vgl. Abschnitt 5.2.2.

Abbildung 8.8 und Tabelle 8.2 beschreiben nach dem gleichen Schema die Änderungsinformationen der Observable-Klasse `Morphism`.

Kapitel 9

Anmerkungen zur Implementierung

In diesem Kapitel fassen wir den Stand der Implementierung zusammen und geben Hinweise auf einige Abweichungen zwischen Entwurf und Implementierung. Außerdem werden die bei der Implementierung verfolgten Namenskonventionen zusammengestellt.

9.1 Stand der Implementierung

Im wesentlichen entspricht der Stand der Implementierung zur Zeit der Abgabe dieser Arbeit dem beschriebenen Entwurf. Es gibt jedoch einige wenige Merkmale, die im Entwurf beschrieben werden, deren vollständige Implementierung aber noch aussteht:

- Die Änderungsinformationen sind noch unvollständig. Von der Klasse `basis.Graph` fehlt die explizite Nachricht für die Verklebung von zwei Graphobjekten. Die entsprechende Nachricht `ALR_Change_ObjectsGlued` von `ALR_RefTree` wird derzeit umgesetzt in eine `Graph_Change_ObjectModified`-Nachricht für das erhaltene und eine `Graph_Change_ObjectDestroyed`-Information für das durch die Verklebung gelöschte Objekt. Außerdem sind bisher weder auf der ALR-Ebene noch für die einfachen Graphen die spezifizierten Informationen über Attributmodifikationen verfügbar. Für die Realisierung dieser Informationen muß eine Observer-Beziehung zwischen einem Graphobjekt und seinem Attribut etabliert werden, die entsprechenden Observable-Schnittstellen der Attributkomponente fehlen aber derzeit noch.

- Die Klasse `alr.transform.ALR_ColimDiagram` unterstützt derzeit ausschließlich die Inplace-Berechnung des Colimes-Objekts, entsprechend kann auch ein Graphtransformationsschritt nur *in place* ausgeführt werden.

- Die in Kapitel 8 beschriebene Schnittstelle für einfache Graphen im Package `basis` ist vollständig implementiert, aber bisher ungetestet. Es gibt aber ein erweitertes Package `xt_basis`, das eine Obermenge der `basis`-Schnittstellen implementiert und lauffähig ist, siehe unten.

Auf der anderen Seite gibt es zwei interessante Funktionalitäten der Implementierung, die im Entwurf nicht beschrieben werden:

- Als zusätzliches Package in der Schnittstellenschicht des Basissystems bietet `xt_basis` (für „extended basis") die Möglichkeit, die ALR-Graphen der Implementierungsschicht ebenenweise als einfache Graphen zu bearbeiten, wobei gegenüber den einfachen Graphen die Möglichkeit hinzukommt, Kanten auch zwischen Kanten zu ziehen. Die Entwicklung der Erweiterungsfunktionalität ist noch nicht abgeschlossen und wird fortgesetzt. Die den beschriebenen `basis`-Schnittstellen entsprechende Grundfunktionalität ist jedoch vollständig und getestet. Die Editorkomponente setzt derzeit auf dem `xt_basis`-Package auf, ohne jedoch den Erweiterungsanteil zu nutzen.

- Für die Auswahl von Morphismusvervollständigungsverfahren im Strategy-Pattern und für die Konfiguration der Verfahren bezüglich optionaler Ansatzeinschränkungen (vgl. Def. 2.11, S. 13) bieten die Packages `alr.transform` und `basis` entsprechend flexible Schnittstellen (`ALR_CompletionStrategySelector`, `CompletionPropertyBits` und `ALR_MorphCompletionStrategy` auf ALR-Ebene, im `basis`-Package analog). Diese Schnittstellen sind im Anhang dokumentiert. In diesem Zusammenhang sei daran erinnert, daß das Vervollständigungsverfahren `Completion_CSP` bisher nur die Einschränkung auf injektive Morphismen unterstützt.

Abschließend noch zwei allgemeine Anmerkungen zur Implementierung des Systems:

- Zu Beginn des Entwurfs wurde die Entscheidung getroffen, die Implementierungsklassen in speziellen Sub-Packages mit Namen `impl` streng von den Schnittstellen zu trennen. Diese Trennung führte jedoch im Verlauf der Entwicklung vermehrt zu technischen Schwierigkeiten, die darin begründet sind, daß Java diese Art der Auslagerung von Schnittstellen nicht speziell unterstützt, sondern die `impl`-Packages als eigenständige Packages betrachtet. Die Tatsache, daß die Schnittstellen in Java oft als abstrakte Klassen formuliert werden müssen – `java.util.Observable` etwa ist eine abstrakte Klasse, kein Interface –, hat in Verbindung mit den Schutzmechanismen von Java, die sich mit dem Schlüsselwort `protected` auf Package-Grenzen beziehen, und dem Fehlen der Möglichkeit für Mehrfachvererbung u.a. zu massiven Problemen mit parallelen Vererbungshierarchien von Schnittstellen und Implementierungen geführt. Aus diesen Gründen wurde die Trennung an vielen Stellen aufgeweicht und wird in Zukunft konsequent aufgehoben werden. Das vermindert aber die Übersichtlichkeit der Packages und erhöht damit den Stellenwert einer angemessenen Dokumentation.

- Die derzeitige Implementierung der ALR-Schicht dient zur Modellierung der `basis`-Schnittstellen nach dem Baukastenprinzip. Dabei werden die ALR-Strukturen (z.B. die Modellierung einer Graphgrammatik, vgl. Abb. 8.1, S. 121) durch explizite Konstruktoraufrufe der ALR-Klassen innerhalb der `basis`-Implementierungen aufgebaut. Prinzipiell ist aber auch vorgesehen, daß Anwendungskomponenten direkt auf die ALR-Schnittstellen zugreifen können. Deshalb sollte die Erzeugung von konkreten Instanzen auch in der ALR-Schicht, wie in den anderen Systemteilen üblich, in Zukunft durch Factory-Methoden gekapselt werden. Um den Baukastencharakter zu erhalten und keine strenge Strukturierung vorzuschreiben, ist die Einführung einer speziellen Factory-Klasse vorgesehen.

9.2 Persistenz

Zum Abspeichern des Systemzustands, insbesondere dem Speichern und Laden von Graphgrammatiken, wird vom AGG-System der *Serializable*-Mechanismus genutzt (vgl. [Sun97]). Dies hat den Vorteil, daß Persistenz nahezu ohne jeden Implementierungsaufwand erreicht werden kann. Dies ist zunächst einmal als großer Vorteil anzusehen, denn noch in [Rud96] hat die Implementierung des Dateiformats beträchtlichen Arbeitsaufwand erfordert. Dabei geht das eingebaute Verfahren intelligent vor, so daß auch aufgrund von mehrfach vorhandenen Referenzen auf dasselbe Objekt oder durch zirkuläre Referenzen niemals der Zustand desselben Objekts mehrfach gespeichert wird. Der große Nachteil ist jedoch, daß der automatische Sequentialisierungsmechanismus im Prinzip einem Speicher-Dump gleichkommt, weshalb jede Änderung an der Implementierung potentiell zur Inkompatibilität mit allen zuvor gespeicherten Zuständen führt. Man kann jedoch in den Sequentialisierungsvorgang für eine Klasse gezielt eingreifen, indem man eine bestimmte Sequentialisierungsmethode implementiert, die dann anstelle der vorgegebenen Default-Sequentialisierung verwendet wird. Es ist zu hoffen, daß auf diese Weise in Zukunft ein ausreichender Abstraktionsgrad bei der Sequentialisierung erreicht werden kann.

9.3 Konventionen

Die Implementierung richtet sich nach den folgenden Namenskonventionen:

Klassen beginnen mit einem Großbuchstaben. Beispiel: `Morphism`.

Variablen beginnen mit einem Kleinbuchstaben:

- Lokale Variablen beginnen mit „`a`“. Beispiel: `anObject`.
- Instanzvariablen von Klassen beginnen mit „`its`“. Beispiel: `itsObject`.
- Klassenvariablen beginnen mit „`their`“. Beispiel: `theirTotal`.
- Methodenparameter werden vollständig klein geschrieben. Beispiel: `object`.

Konstanten von Klassentypen beginnen mit einem kleinen „`c`“. Beispiel: `cDefaultStrategy`.

Konstanten von Standarddatentypen werden vollständig groß geschrieben. Beispiel: `MATCH_MAP`.

Methoden beginnen mit einem Kleinbuchstaben. Beispiel: `nextCompletion()`. Weiterhin:

- Methoden, die den Zustand eines Objektes ändern bzw. abfragen, beginnen mit „`set`“ bzw. „`get`“. Beispiel: `setName()`.
- Factory-Methoden beginnen mit „`create`“. Beispiel: `createObject()`.
- Methoden, die Objekte löschen, die mit `create`-Methoden erzeugt wurden, beginnen mit „`destroy`“. Beispiel: `destroyObject()`.
- Prädikate beginnen mit „`is`“ oder „`has`“. Beispiel: `isEmpty()`.

Diese Konventionen sind eine angepaßte Version der Vorgaben aus [Rud96] und orientieren sich an *Taligent's guide to designing programs: well-mannered object-oriented design in C++* (Taligent Press/Addison-Wesley, 1994) und an *A Coding Convention for C++ Code* von Dan Wallach (CS Department, Princeton University).

Kapitel 10

Zusammenfassung und Ausblick

Inzwischen sind zehn Minuten in der zweiten Halbzeit gespielt. Die Graphen haben sich bemüht, doch durch viele Ballverluste im Mittelfeld sind sie noch zu keinem weiteren Torerfolg gekommen. In den vergangenen Kapiteln haben wir uns aber auf einen Auswechselspieler konzentriert, der sich an der Seitenlinie warmläuft: das neue AGG-System.

Der in dieser Arbeit entwickelte Systemkern bietet mit der Repräsentation von Graphen, der automatischen Suche nach Ansätzen einer Regel in einen Arbeitsgraphen und der Berechnung eines direkten Transformationsschritts die vollständige Basisfunktionalität für ein Graphtransformationssystem. Fassen wir noch einmal die wichtigsten Eigenschaften des Systems zusammen:

Nähe zur Theorie. Die Berechnung eines SPO-Graphtransformationsschrittes durch die Colimit-Bibliothek sichert die Übereinstimmung mit der Theorie durch die nachgewiesene Korrektheit des Bibliotheksalgorithmus. Die theorienahe Implementierung des Graphmodells, die die Verantwortung für Layout und Visualisierung konsequent an die jeweiligen Anwendungskomponenten überträgt, erleichtert außerdem die praktische Umsetzung weiterer theoretischer Ergebnisse auf der Grundlage des Basissystems. Durch die zusätzliche Systemschicht für einfache Graphen ist dabei die Erweiterung der Theorien auf ALR-Graphen nicht mehr zwingend erforderlich.

Ausdrucksstärke. Die Integration einer Attributkomponente zur Attributierung von Graphobjekten mit beliebigen Java-Ausdrücken in Verbindung mit dem Graphmodell der ALR-Graphen eröffnet schon auf dem gegenwärtigen Entwicklungsstand neue Ausdrucksmöglichkeiten. Eine wesentliche Steigerung der Ausdruckskraft kann durch das Konzept der negativen Anwendungsbedingungen [HHT96] erreicht werden. Die Implementierung dieses Konzeptes ist innerhalb kurzer Zeit zu erwarten, da bereits alle Voraussetzungen dafür geschaffen wurden.

Flexible, modulare Softwarearchitektur. Zur flexiblen Entkopplung von Teilsystemen und Kapselung von Algorithmen spielen Design Pattern eine wichtige Rolle. Insbesondere wurde mit Hilfe des Observer-Patterns eine Systemstruktur verwirklicht, die die Konsistenzerhaltung und Kooperation von mehreren Anwendungskomponenten über demselben Basissystem ermöglicht, ohne eine Kopplung der Anwendungskomponenten selbst zu verlangen.

Automatische Ansatzsuche. Durch die Realisierung der Ansatzsuche als Constraint Satisfaction Problem eröffnet sich der direkte Zugang zu den Forschungsergebnissen aus dem Gebiet der Constraint Satisfaction, das sich u.a. intensiv mit der Optimierung von Backtracking-Algorithmen beschäftigt. Gleichzeitig wird so eine Abstraktion des zur Ansatzsuche verwendeten Algorithmus vom konkreten Graphmodell erreicht, so daß das Modell variieren kann, ohne den Algorithmus zu beeinflussen. Mit dem Query-Konzept haben wir dabei eine Möglichkeit vorgestellt, konkrete Eigenschaften des Graphmodells abstrakt zu repräsentieren und für Optimierungen auf der abstrakten Ebene der CSPs zu nutzen.

In Verbindung mit der separaten Editorkomponente, die parallel zu dem in dieser Arbeit beschriebenen Systemkern von der Programmiererin Olga Runge entwickelt wurde, kann das Nahziel, das wir uns in der Einleitung gesteckt haben, als erreicht gelten: Eine Basis wurde geschaffen, auf der weitere theoretische Ergebnisse erprobt und veranschaulicht werden können. In nächster Zeit kommen hierfür z.B. die Konzepte zur Konsistenzsicherung und -analyse [HW95] und die verteilte Graphtransformation [Tae96] in Frage. Einen Schritt weiter in Richtung praktischer Anwendungsmöglichkeiten geht derzeit ein Projekt zur Generierung von Graphikeditoren, das ebenfalls auf dem hier beschriebenen Basissystem aufsetzen soll. Eine Instanz für einen generierten Graphikeditor könnte etwa ein spezieller Editor für UML-Klassendiagramme sein, wie sie auch in dieser Arbeit vielfach verwendet wurden. Die möglichen Editoroperationen sollen dabei durch Graphtransformationen spezifiziert und implementiert werden.

Um jedoch unser Fernziel zu erreichen, daß nämlich der Experte aus Definition 1.1 die Probleme seines Fachgebiets mit Graphtransformationen besser und anschaulicher lösen kann als mit herkömmlichen Methoden, ist auch von seiten der Theorie noch weitere Arbeit zu leisten. Dabei fehlt es insbesondere noch an geeigneten Konzepten zur Strukturierung und Modularisierung von großen Spezifikationen, was sich in größeren Fallbeispielen immer wieder als Problem erweist [CGHB+95, EB94, Rib96]. Einige Ansätze zu diesem Thema sind in [EE96] und [ES95] zu finden. Aber auch mögliche Techniken für die programmierte Regelanwendung etwa mittels Transaktionen, die es ermöglichen sollten, die für Graphtransformation typische nichtdeterministische Regelanwendung mit deterministischen Aspekten zu verbinden, sind bisher nicht ausreichend theoretisch untersucht worden. Einen Vorschlag für ein einfaches Konzept machen hier [AEH+96] mit den *Transformation Units*, deren Einsatz im AGG-System vorgesehen ist.

Während wir nun eine Auswechslung vornehmen und das neue AGG-System für das erschöpfte alte aufs Feld schicken, können wir die Situation auf dem Rasen wie folgt analysieren: Während die Abwehr theoretisch gut fundiert ist und sicher steht, muß die Theorie vermehrt auch die Regie im Mittelfeld übernehmen, damit die praktischen Anwendungen in der Spitze nicht im Abseits stehen. Das Spiel ist noch nicht entschieden.

Anhang A

API - Application Programming Interface

Die folgenden Abschnitte enthalten eine Zusammenstellung der Dokumentation zu den wichtigsten Schnittstellen aus den Packages des Basissystems. Die vollständige Schnittstellendokumentation liegt im Hypertextformat auf dem Rechner vor.

Nach den Schlüsselwörtern **Pre:** und **Post:** werden für viele Methoden Vor- und Nachbedingungen spezifiziert. Die Notation der Bedingungen erfolgt in Java-Syntax oder umgangssprachlich. Das Symbol „`->`“ steht für die logische Implikation.

A.1 Package `util`

A.1.1 Class agg.util.Change

```
java.lang.Object
  |
  +----agg.util.Change
```

public class Change

extends Object

Encapsulation of change information sent out by observable classes using the method `notifyObservers(change)` of the class `java.util.Observable`. Observer objects then receive such information as the second argument to their `update()` method.

Subclasses may specifiy the detailed semantics.

Constructors

- **Change**

```
public Change(Object item)
```

Construct myself to be a change information with the given item.

Methods

- **getItem**

```
public final Object getItem()
```

Return my item.

A.1.2 Class agg.util.Change_ObservableGone

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.util.Change_ObservableGone
```

public class Change_ObservableGone

extends Change (cf. Section A.1.1)

This change information may be sent out by an observable object when it wants its Observers to dispose their reference to it. This is necessary to break the circular reference inherent to the observer pattern which makes it inaccessible for the garbage collector.

Its item is of type `java.util.Observable` and denotes the observable that should no longer be referenced.

Constructors

- **Change_ObservableGone**

```
public Change_ObservableGone(Observable item)
```

A.1.3 Interface agg.util.Disposable

public interface Disposable

`Disposable` is an interface that is implemented by classes that need explicit finalization in order to become accessible to garbage collection. The contract is as follows: The caller of the constructor of a disposable object is responsible for calling its `dispose()` method when the object is no longer needed.

Methods

- **dispose**

```
public abstract void dispose()
```

Prepare myself for garbage collection.

A.1.4 Class agg.util.ExtObservable

```
java.lang.Object
 |
 +----java.util.Observable
         |
         +----agg.util.ExtObservable
```

public class ExtObservable

extends Observable

implements Disposable (cf. Section A.1.3)

An extension of the native Java `Observable` class that addresses the need for explicit disposure arising from the circular references inherent to the observer pattern (observer knows observable and vice versa). In a multi-layer observer architecture, the disposure command has to be passed way up the observer hierarchy to break the circular references in every layer and thus make the participants amenable for garbage collection.

Constructors

- **ExtObservable**

  `public ExtObservable()`

Methods

- **dispose**

  `public void dispose()`

  Prepare myself for garbage collection. A change message `Change_ObservableGone` with myself as the item is sent out to all of my observers. Subclasses may override this to break their individual circular references, but they should always include a call to this original implementation.

  See Also: Change_ObservableGone (cf. Section A.1.2)

- **finalize**

  `protected void finalize()`

  Overrides: finalize in class Object

A.1.5 Interface agg.util.StrategyProperties

public interface StrategyProperties

This interface may be implemented by abstract strategies to provide support for special properties that a concrete strategy may have. A property is represented as a bit in a `BitSet`. Symbolic names for the property bits of a category of strategies may be defined in separate interfaces.

Methods

- **getSupportedProperties**

```
public abstract BitSet getSupportedProperties()
```

Return information about what properties I support. A property is supported if its corresponding bit is set.

- **getProperties**

```
public abstract BitSet getProperties()
```

Return information about what properties are currently activated. Properties can be activated or deactivated by setting or clearing their respective bits via the `BitSet` interface.

A.2 Package `util.csp`

A.2.1 Class agg.util.csp.CSP

```
java.lang.Object
 |
 +----agg.util.csp.CSP
```

public abstract class CSP

extends Object

An abstract class for Constraint Satisfaction Problems with only binary constraints.

Constructors

- **CSP**

```
public CSP(SolutionStrategy solver)
```

Construct myself with an initial SolutionStrategy.

Post: `getDomain() == null`.

Methods

- **getVariables**

```
public abstract Enumeration getVariables()
```

Return an Enumeration of all my variables. Enumeration elements are of type `Variable`.

- **getSize**

```
public abstract int getSize()
```

Return the number of variables in the CSP.

- **setDomain**

```
public final void setDomain(Object domain)
```

Set the global domain of values for the variables, and call `preprocessDomain()` with the given `domain`.

Post: `getDomain() == domain`.

See Also: preprocessDomain (cf. Section A.2.1)

- **preprocessDomain**

```
protected abstract void preprocessDomain(Object domain)
```

Preprocess the given domain for optimization purposes, e.g. to get more accurate data for Constraint weights, or to initialize Query databases. This is a template method to be implemented in subclasses, and is invoked out of `setDomain()`.

See Also: setDomain (cf. Section A.2.1)

- **getDomain**

```
public final Object getDomain()
```

Return the current global domain of values.

- **nextSolution**

```
public final boolean nextSolution()
```

Compute my next solution, and instantiate my variables appropriately. Variables already instantiated will not be altered, so this method can be used to complete partial solutions. Invoke this method repeatedly to get all solutions.

Pre: `getDomain() != null`.

Returns: `false` if there are no more solutions.

- **setSolutionStrategy**

```
public final void setSolutionStrategy(SolutionStrategy solver)
```

Set the algorithm which is used to compute my solutions.

A.2.2 Class agg.util.csp.Variable

```
java.lang.Object
 |
 +----agg.util.csp.Variable
```

public class Variable

extends Object

A class for the variables of a CSP.

Constructors

- **Variable**

  `public Variable()`

Methods

- **getInstance**

  `public final Object getInstance()`

  Return my current value, `null` if uninstantiated.

- **setInstance**

  `public final void setInstance(Object value)`

  Instantiate me with the given value. To uninstantiate, use `setInstance()` with `value == null`.

- **checkConstraints**

  `public final Enumeration checkConstraints()`

  Check all my applicable constraints, i.e., check the consistency of my current instantiation with all previously instantiated variables.

  Pre: `getInstance() != null`.

  Returns: An Enumeration of all the Variables whose instantiations conflict with my current instantiation. If all applicable constraints are satisfied, the Enumeration is empty. Enumeration elements are of type `Variable`.

- **addInstantiationHook**

  `public final void addInstantiationHook(InstantiationHook hook)`

  Add `hook` to the set of my InstantiationHooks. I will call the encapsulated operations at the respective times of instantiation/uninstantiation, with myself as an argument.

  See Also: InstantiationHook (cf. Section A.2.3)

- **getDomainEnum**

  `public final Enumeration getDomainEnum()`

  Return the enumeration of my domain. This enumeration continues at the position where a previous access left off. The type of enumeration elements is dependent on the concrete domain.

- **setDomainEnum**

  `public final void setDomainEnum(Enumeration dom)`

Set my domain in an enumeration representation. This very same enumeration is returned from a subsequent call of `getDomainEnum()`.

- **getWeight**

```
public final int getWeight()
```

Return my weight. It is computed as the sum of the weights of all constraints attached and of all outgoing queries.

- **getConstraints**

```
public final Enumeration getConstraints()
```

Return an enumeration of all the constraints I'm involved in. Enumeration elements are of type `BinaryConstraint`.

See Also: BinaryConstraint (cf. Section A.2.4)

- **getOutgoingQueries**

```
public final Enumeration getOutgoingQueries()
```

Return an enumeration of all my outgoing queries. Enumeration elements are of type `Query`.

See Also: Query (cf. Section A.2.5)

- **getIncomingQueries**

```
public final Enumeration getIncomingQueries()
```

Return an enumeration of all my incoming queries. Enumeration elements are of type `Query`.

See Also: Query (cf. Section A.2.5)

- **addConstraint**

```
protected final void addConstraint(BinaryConstraint c)
```

Let me know of a new constraint which I'm involved in.

- **addOutgoingQuery**

```
protected final void addOutgoingQuery(Query q)
```

Let me know of a query for which I am a source variable.

- **addIncomingQuery**

```
protected final void addIncomingQuery(Query q)
```

Let me know of a query for which I am the target variable.

A.2.3 Interface agg.util.csp.InstantiationHook

public interface InstantiationHook

An interface for the realization of side effects which are to take place at instantiation/uninstantiation time of a CSP variable.

See Also: Variable (cf. Section A.2.2)

Methods

- **instantiate**

  ```
  public abstract void instantiate(Variable var)
  ```

  This method is called whenever the variable I'm hooked to gets instantiated. It is called *after* the new value has been set, with the variable as an argument.

- **uninstantiate**

  ```
  public abstract void uninstantiate(Variable var)
  ```

  This method is called whenever the variable I'm hooked to gets uninstantiated, or when it is set to a new value. It is called *before* the value is re- or unset, with the variable as an argument.

A.2.4 Class agg.util.csp.BinaryConstraint

```
java.lang.Object
 |
 +----agg.util.csp.BinaryConstraint
```

public abstract class BinaryConstraint

extends Object

An abstract class for binary constraints.

Constructors

- **BinaryConstraint**

  ```
  public BinaryConstraint(Variable v1,
                          Variable v2,
                          int weight)
  ```

  Construct myself to be a binary constraint between variables `v1` and `v2`, with the specified `weight`.

- **BinaryConstraint**

  ```
  public BinaryConstraint(Variable v,
                          int weight)
  ```

  Construct myself to be a ¨unary¨ constraint on `v`. Actually, this is a BinaryConstraint with both its variables being `v`.

Methods

- **isApplicable**

  `public boolean isApplicable()`

  Return `true` iff all variables involved are instantiated.

- **execute**

  `public abstract boolean execute()`

  Check if the constraint is satisfied.
  Pre: `isApplicable()`.

- **getCause**

  `public Variable getCause(Variable rvar)`

  When `execute()` failed, this returns the variable that is supposed to have caused the failure.

  Parameters: rvar - the variable (of the two involved) that has been instantiated most recently.

- **getVar1**

  `public Variable getVar1()`

  Return my first variable.

- **getVar2**

  `public Variable getVar2()`

  Return my second variable.

- **getWeight**

  `public int getWeight()`

  Return my weight. The higher the value, the higher the significance of the constraint.

A.2.5 Class agg.util.csp.Query

```
java.lang.Object
  |
  +----agg.util.csp.Query
```

public abstract class Query

extends Object

An abstract class that represents a query for a variable domain.

Constructors

- **Query**

```
public Query(Variable tar,
             int weight)
```

Construct myself to be a constant query.

- **Query**

```
public Query(Variable src,
             Variable tar,
             int weight)
```

Construct myself to be a unary query.

- **Query**

```
public Query(Variable src1,
             Variable src2,
             Variable tar,
             int weight)
```

Construct myself to be a binary query.

Methods

- **isApplicable**

```
public boolean isApplicable()
```

Return `true` iff all my source variables are instantiated, while my target variable is not.

- **isConstant**

```
public boolean isConstant()
```

Return `true` iff I am a constant query. That means, the result of `execute()` is the same for any variable instantiation configuration.

- **execute**

```
public abstract Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Pre: `isApplicable()`.

- **getTarget**

```
public Variable getTarget()
```

Return the variable that I'm determining the domain for.

- **getSources**

  `public final Enumeration getSources()`

  Return an enumeration of the variables that need to be instantiated for the query to work. Enumeration elements are of type `Variable`.

- **getSize**

  `public abstract int getSize()`

  Return the number of candidate values `execute()` will provide. For non-constant queries, this will most probably be based on estimation. The value may change in response to re-setting the CSP domains with the `setDomain()` method.

  Pre: `csp.getDomain() != null`.

  See Also: setDomain (cf. Section A.2.1)

- **getWeight**

  `public final int getWeight()`

  Return my weight. This is a constant integer usually chosen inversely proportional to the estimated size of the candidate set returned by a query execution.

- **getSourceInstance**

  `protected final Object getSourceInstance(int i)`

  Return the current instance of the source variable given by the index `i`.

  Pre: `i` < `itsSources.size()`.

A.2.6 Interface agg.util.csp.SolutionStrategy

public interface SolutionStrategy

An interface for solution strategies for Constraint Satisfaction Problems.

See Also: CSP (cf. Section A.2.1)

Methods

- **next**

  `public abstract boolean next(CSP csp)`

  Find the next solution of `csp`, and instantiate its variables accordingly. Variables already instantiated will not be altered, so this method can be used to complete partial solutions. Invoke this method successively with the same argument to get all solutions (or all completions of a given partial solution).

  Parameters: csp - The CSP to solve.

Returns: `false` if there are no more solutions.

- **reset**

```
public abstract void reset()
```

Reset my internal state, so that the forthcoming invocation of `next()` returns the first solution of the given CSP.

A.2.7 Class agg.util.csp.Solution_Backjump

```
java.lang.Object
 |
 +----agg.util.csp.Solution_Backjump
```

public class Solution_Backjump

extends Object

implements SolutionStrategy (cf. Section A.2.6)

A CSP solution strategy using the backjumping technique.

Constructors

- **Solution_Backjump**

```
public Solution_Backjump()
```

- **Solution_Backjump**

```
public Solution_Backjump(boolean injective)
```

Methods

- **reset**

```
public final void reset()
```

- **next**

```
public final boolean next(CSP csp)
```

A.2.8 Class agg.util.csp.Solution_InjBackjump

```
java.lang.Object
 |
 +----agg.util.csp.Solution_Backjump
          |
          +----agg.util.csp.Solution_InjBackjump
```

public class Solution_InjBackjump

extends Solution_Backjump (cf. Section A.2.7)

A CSP solution strategy using the backjumping technique. Only injective solutions are considered.

Constructors

- **Solution_InjBackjump**

```
public Solution_InjBackjump()
```

A.2.9 Class agg.util.csp.QueryOrder

```
java.lang.Object
 |
 +----agg.util.csp.QueryOrder
```

public class QueryOrder

extends Object

implements BinaryPredicate

Constructors

- **QueryOrder**

```
public QueryOrder()
```

Methods

- **execute**

```
public final boolean execute(Object q1,
                              Object q2)
```

Return true iff the weight of `q1` is greater than the weight of `q2`.

Pre: `q1,q2 instanceof Query`.

See Also: Query (cf. Section A.2.5)

A.2.10 Interface agg.util.csp.SearchStrategy

public interface SearchStrategy

An interface for algorithms calculating search plans, i.e. variable orderings given by a list of queries.

Methods

- **execute**

```
public abstract Vector execute(CSP csp)
```

Return a list of queries representing a search plan. A variable ordering is given by the target variables of the queries, and the domain for such a target variable is given by its query. Vector elements are of type `Query`.

See Also: Query (cf. Section A.2.5)

A.2.11 Class agg.util.csp.SimpleVariableOrder

```
java.lang.Object
  |
  +----agg.util.csp.SimpleVariableOrder
```

public class SimpleVariableOrder

extends Object

implements BinaryPredicate

Constructors

- **SimpleVariableOrder**

```
public SimpleVariableOrder()
```

Methods

- **execute**

```
public final boolean execute(Object var1,
                             Object var2)
```

Return true iff the weight of `var1` is greater than the weight of `var2`.

Pre: `var1,var2 instanceof Variable`.

A.2.12 Class agg.util.csp.Search_BreadthFirst

```
java.lang.Object
  |
  +----agg.util.csp.Search_BreadthFirst
```

public class Search_BreadthFirst

extends Object

implements SearchStrategy (cf. Section A.2.10)

A search strategy that traverses the constraint graph breadth first.

Constructors

- **Search_BreadthFirst**

```
public Search_BreadthFirst()
```

Methods

- **execute**

```
public final Vector execute(CSP csp)
```

A.3 Package `alr`

A.3.1 Interface agg.alr.ALR_GraphObject

public interface ALR_GraphObject

extends Serializable, Disposable (cf. Section A.1.3)

ALR_GraphObject is the common interface for nodes and arcs in an ALR graph.

Methods

- **isNode**

  `public abstract boolean isNode()`

  Return `true` iff I am a node.

- **isArc**

  `public abstract boolean isArc()`

  Return `true` iff I am an arc.

- **getType**

  `public abstract Type getType()`

  Return my type.

- **setType**

  `public abstract void setType(Type type)`

  Note: setType() is deprecated.

- **getAttribute**

  `public abstract AttrInstance getAttribute()`

  Return my attribute value.

- **getLevel**

  `public abstract Level getLevel()`

  Return the level I belong to.

- **getSource**

  `public abstract ALR_GraphObject getSource()`

  Return my source object.

- **getTarget**

  `public abstract ALR_GraphObject getTarget()`

  Return my target object.

- **getAbstraction**

  `public abstract ALR_GraphObject getAbstraction()`

  Return my abstraction object.

- **getIncomingArcs**

  `public abstract Enumeration getIncomingArcs(ALR_GraphObject top)`

  Iterate through my incoming Arcs. Enumeration elements are of type `ALR_GraphObject`.

  Parameters: top - the Top-Object of the context to consider.

- **getOutgoingArcs**

  `public abstract Enumeration getOutgoingArcs(ALR_GraphObject top)`

  Iterate through my outgoing Arcs. Enumeration elements are of type `ALR_GraphObject`.

  Parameters: top - the Top-Object of the context to consider.

- **getRefinementObjects**

  `public abstract Enumeration getRefinementObjects()`

  Iterate through all of my refinement Objects. Enumeration elements are of type `ALR_GraphObject`.

A.3.2 Class agg.alr.ALR_RefTree

```
java.lang.Object
 |
 +----java.util.Observable
         |
         +----agg.util.ExtObservable
                 |
                 +----agg.alr.ALR_RefTree
```

public abstract class ALR_RefTree

extends ExtObservable (cf. Section A.1.4)

implements Serializable, Disposable (cf. Section A.1.3)

ALR_RefTree is the common interface for refinement trees of ALR graph objects, i.e. ALR-graphs (as given by the refinement of an ALR node) and ALR-relations (as given by the refinement of an ALR arc).

Constructors

- **ALR_RefTree**

```
public ALR_RefTree()
```

Methods

- **setName**

```
public abstract void setName(String n)
```

Set my name.

- **getName**

```
public abstract String getName()
```

Return my name.

- **getTopObject**

```
public abstract ALR_GraphObject getTopObject()
```

Return my top object.

- **getFocus**

```
public abstract ALR_RefTree getFocus(ALR_GraphObject obj)
```

Return the refinement tree of the given `obj` as an `ALR_RefTree`.

- **hasFocus**

```
public abstract boolean hasFocus(ALR_GraphObject obj)
```

Return `true` iff an internal `ALR_RefTree`-representation for the refinement tree of `obj` has already been created.

- **getFocuses**

```
public abstract Enumeration getFocuses()
```

Iterate through all of my subfocuses, including myself. Enumeration elements are of type `ALR_RefTree`.

- **createType**

```
public abstract Type createType()
```

Create and return a new type for GraphObjects.

- **destroyType**

```
public abstract void destroyType(Type type)
```

Remove the given `type` from the set of available types.

- **getTypes**

```
public abstract Enumeration getTypes()
```

Iterate through the set of available types that may be given to a graphobject. Enumeration elements are of type `Type`.

See Also: Type (cf. Section A.3.4)

- **getDefaultType**

```
public abstract Type getDefaultType()
```

Return the default type.

- **createObject**

```
public abstract ALR_GraphObject createObject(Type type,
                                             ALR_GraphObject abs,
                                             ALR_GraphObject src,
                                             ALR_GraphObject tar)
```

Create a new object with the given type and structural context.

Pre:

1. `this.isElement(src) && this.isElement(tar)` or `src == null && tar == null`.
2. `this.isElement(abs)`.
3. `src == null` -> `abs.isNode()`.
4. `(src != null && abs.isNode())` -> `(src.getAbstraction() == abs && tar.getAbstraction() == abs)`.
5. `(src != null && abs.isArc())` -> `(src.getAbstraction() == abs.getSource() && tar.getAbstraction() == abs.getTarget())`.

- **createObject**

```
public abstract ALR_GraphObject createObject(ALR_GraphObject orig,
                                             ALR_GraphObject abs,
                                             ALR_GraphObject src,
                                             ALR_GraphObject tar)
```

Create a GraphObject as a copy of `orig`. Only type and attributes are copied, the structural context has to be specified explicitly by means of the parameters `abs, src,` and `tar`.

- **destroyObject**

```
public abstract void destroyObject(ALR_GraphObject obj)
```

Delete the specified object. Dangling arcs get deleted recursively, as well as the object's refinement.

- **glue**

```
public abstract void glue(ALR_GraphObject keep,
                          ALR_GraphObject glue)
```

Merge the two objects `keep` and `glue` into one. This means, let `keep` inherit all the properties of `glue`, incl. its structural context, and delete `glue` afterwards.

Pre:

1. `keep != null, glue != null`.
2. `keep.getLevel().equals( glue.getLevel() )`.
3. `keep.getType().equals( glue.getType() )`.
4. `keep.isNode() && glue.isNode()` or `keep.isArc() && glue.isArc()`.
5. `this.isElement(keep) && this.isElement(glue)`.

- **getElements**

```
public abstract Enumeration getElements()
```

Iterate through all of my nodes and arcs. Enumeration elements are of type `ALR_GraphObject`.

See Also: ALR_GraphObject (cf. Section A.3.1)

- **isGraph**

```
public abstract boolean isGraph()
```

Return `true` if I am an ALR graph, i.e., my top object is a node.

- **isRelation**

```
public abstract boolean isRelation()
```

Return `true` if I am an ALR relation, i.e., my top object is an arc.

- **isEmpty**

```
public abstract boolean isEmpty()
```

Return `true` if I contain no objects apart from my top object.

- **isElement**

```
public abstract boolean isElement(ALR_GraphObject obj)
```

Return `true` if I contain `obj`.

- **clear**

```
public abstract void clear()
```

Delete all of my objects except for the top object.

- **setAttrContext**

```
public abstract void setAttrContext(AttrContext context)
```

Set my attribute context. In subsequent calls to my `createObject()` method, this context will be used as the context for the new object's `AttrInstance`.

The attribute context is needed for the handling of variables in attributes. See documentation of attribute package for details.

- **getAttrContext**

```
public abstract AttrContext getAttrContext()
```

Return my attribute context.

- **getAttrManager**

```
public abstract AttrManager getAttrManager()
```

Return my attribute manager. This serves as an AttrContext factory.

A.3.3 Interface agg.alr.Level

public interface Level

extends Serializable

Methods

- **nextLevel**

```
public abstract Level nextLevel()
```

Return the level below me.

- **prevLevel**

```
public abstract Level prevLevel()
```

Return the level above me.

A.3.4 Interface agg.alr.Type

public interface Type

extends Serializable

Instances of this class are used for dynamic typing of graphobjects. Each type is associated with a name (also called *string representation*). Note that two types with the same name need not be equal.

Methods

- **getStringRepr**

```
public abstract String getStringRepr()
```

Return my string representation.

- **setStringRepr**

```
public abstract void setStringRepr(String n)
```

Set my string representation.

- **getAttrType**

```
public abstract AttrType getAttrType()
```

Return the associated attribute type.

A.3.5 Class agg.alr.ALR_Change_FocusMoved

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.alr.ALR_Change_FocusMoved
```

public class ALR_Change_FocusMoved

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR reftree `g` if its top object `t` has been deleted due to a call of the form `glue(some,t)`. with

The item is of type `ALR_RefTree` and references the refinement tree of the object `some` which is the reftree `g` has been glued to.

See Also: ALR_RefTree (cf. Section A.3.2)

Constructors

- **ALR_Change_FocusMoved**

```
public ALR_Change_FocusMoved(ALR_RefTree new_focus)
```

A.3.6 Class agg.alr.ALR_Change_ObjectCreated

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.alr.ALR_Change_ObjectCreated
```

public class ALR_Change_ObjectCreated

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR reftree if a new graphobject has been created.
The item is of type `ALR_GraphObject` and references the newly created object.

See Also: ALR_RefTree (cf. Section A.3.2)

Constructors

- **ALR_Change_ObjectCreated**

```
public ALR_Change_ObjectCreated(ALR_GraphObject item)
```

A.3.7 Class agg.alr.ALR_Change_ObjectDestroyed

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.alr.ALR_Change_ObjectDestroyed
```

public class ALR_Change_ObjectDestroyed

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR reftree if one of its graphobjects has been deleted.

The item is of type `ALR_GraphObject` and references the object which has just been deleted. The object is fully accessible, because the garbage collector will not get hold of it as long as we hold a reference.

See Also: ALR_RefTree (cf. Section A.3.2)

Constructors

- **ALR_Change_ObjectDestroyed**

```
public ALR_Change_ObjectDestroyed(ALR_GraphObject item)
```

A.3.8 Class agg.alr.ALR_Change_ObjectModified

```
java.lang.Object
  |
  +----agg.util.Change
          |
          +----agg.alr.ALR_Change_ObjectModified
```

public class ALR_Change_ObjectModified

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR reftree if the attributes of of one of its graphobjects have been modified.

The item is of type `ALR_GraphObject` and references the object whose attributes have just been modified.

See Also: ALR_RefTree (cf. Section A.3.2)

Constructors

- **ALR_Change_ObjectModified**

```
public ALR_Change_ObjectModified(ALR_GraphObject item)
```

A.3.9 Class agg.alr.ALR_Change_ObjectsGlued

```
java.lang.Object
  |
  +----agg.util.Change
          |
          +----agg.alr.ALR_Change_ObjectsGlued
```

public class ALR_Change_ObjectsGlued

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR reftree if two of its graphobjects have been glued.

The item is of type `COM.objectspace.jgl.Pair`. `pair.first` and `pair.second` are of type `ALR_GraphObject`. The first element of the pair represents the glue result, the second one has been deleted.

See Also: ALR_RefTree (cf. Section A.3.2)

Constructors

- **ALR_Change_ObjectsGlued**

```
public ALR_Change_ObjectsGlued(ALR_GraphObject keep,
                               ALR_GraphObject glue)
```

A.3.10 Class agg.alr.ALR_Change_RefTreeModified

```
java.lang.Object
  |
  +----agg.util.Change
          |
          +----agg.alr.ALR_Change_RefTreeModified
```

public class ALR_Change_RefTreeModified

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR reftree if its name has changed.

The item is of type `ALR_RefTree` and references the refinement tree whose name was changed.

See Also: ALR_RefTree (cf. Section A.3.2)

Constructors

- **ALR_Change_RefTreeModified**

```
public ALR_Change_RefTreeModified(ALR_RefTree item)
```

A.4 Package `alr.transform`

A.4.1 Class agg.alr.transform.ALR_Morphism

```
java.lang.Object
  |
  +----java.util.Observable
          |
          +----agg.util.ExtObservable
                  |
                  +----agg.alr.transform.ALR_Morphism
```

public class ALR_Morphism

extends ExtObservable (cf. Section A.1.4)

implements Observer, Disposable, Serializable (cf. Section A.1.3)

This class implements the notion of an ALR graph morphism. It uses an ALR refinement tree (class `ALR_RefTree`) of an arc as its internal representation. It attaches itself as an observer not only to the representing refinement tree of an arc, but also to the ALR graphs which are the original and image graphs of the morphism. This way, we can assure that the implementation keeps morphism properties even if graphobjects in the original or image graph are created, deleted or modified. Note that morphism properties can *not* be guaranteed if the underlying ALR relation representing the morphism is modified directly, bypassing the `ALR_Morphism` interface. Another reaction to changes in the original and image graphs is that the state of a morphism completion session is reset.

This class is capable of passing the following change informations as an argument to its observers' `update()` methods:

`ALR_Change_MappingAdded, ALR_Change_MappingRemoved.`

Additionally, `Change_ObservableGone` may be generated by its base class `ExtObservable`. If, however, the value of the change information is given as `null`, the observer is recommended to perform a full rescan of the morphism.

See Also: ALR_Change_MappingAdded (cf. Section A.4.3), ALR_Change_MappingRemoved (cf. Section A.4.4), Change_ObservableGone (cf. Section A.1.2)

Constructors

- **ALR_Morphism**

```
public ALR_Morphism(ALR_RefTree rt,
                    ALR_MorphCompletionStrategy sg)

throws ALR_BadMappingException
```

Construct myself out of a given refinement tree of an Arc.

Pre: `rt.isRelation()`.

Parameters: rt - my ALR refinement tree representation. sg - the strategy to use for completion.

Throws: ALR_BadMappingException (cf. Section A.4.2) when the given refinement tree does not represent a valid ALR morphism.

- **ALR_Morphism**

```
public ALR_Morphism(ALR_RefTree rt)

throws ALR_BadMappingException
```

Construct myself out of a given refinement tree of an Arc with a default completion strategy.

Pre: `rt.isRelation()`.

Parameters: rt - my ALR refinement tree representation.

Throws: ALR_BadMappingException (cf. Section A.4.2) when the given refinement tree does not represent a valid ALR morphism.

Methods

- **dispose**

```
public final void dispose()
```

Prepare myself for garbage collection.

Overrides: dispose (cf. Section A.1.4) in class ExtObservable (cf. Section A.1.4)

- **update**

```
public final void update(Observable obs,
                         Object change)
```

- **setName**

```
public final void setName(String n)
```

Set my name.

- **getName**

```
public final String getName()
```

Return my name.

- **addMapping**

```
public final void addMapping(ALR_GraphObject o,
                             ALR_GraphObject i)
```

```
throws ALR_BadMappingException
```

Map an object of my source graph to an object of my target graph.

Pre:

1. `o` is an element of `getOriginal()`.
2. `i` is an element of `getImage()`.

Parameters: o - the source object of the mapping. i - the target object of the mapping.

Throws: ALR_BadMappingException (cf. Section A.4.2) if the given mapping violates morphism properties.

- **removeMapping**

```
public final void removeMapping(ALR_GraphObject o)
```

Remove the mapping of a given ALR_GraphObject. Any mappings of incoming or outgoing arcs will be removed as well.

Pre: `getImage(o) != null`

- **clear**

  `public final void clear()`

  Remove all of my mappings except for the mapping of the top objects.

- **getOriginal**

  `public final ALR_RefTree getOriginal()`

  Return the ALR graph which is the source graph of the morphism.

  Post: `r.isGraph()`, where `r` is the return value.

- **getImage**

  `public final ALR_RefTree getImage()`

  Return the ALR graph which is the target graph of the morphism.

  Post: `r.isGraph()`, where `r` is the return value.

- **getDomain**

  `public final Enumeration getDomain()`

  Return an Enumeration of the graphobjects out of the source graph which are actually taking part in one of my mappings. Enumeration elements are of type `ALR_GraphObject`.

  See Also: ALR_GraphObject (cf. Section A.3.1)

- **getCodomain**

  `public final Enumeration getCodomain()`

  Return an Enumeration of the graphobjects out of the target graph which are actually taking part in one of my mappings. Enumeration elements are of type `ALR_GraphObject`.

  See Also: ALR_GraphObject (cf. Section A.3.1)

- **getImage**

  `public final ALR_GraphObject getImage(ALR_GraphObject o)`

  Return the image of the specified object.

  Returns: `null` if the object is not in domain.

- **getInverseImage**

  `public final Enumeration getInverseImage(ALR_GraphObject o)`

Return an Enumeration of the inverse images of the specified object. Enumeration will be empty when the object is not in my codomain. Enumeration elements are of type `ALR_GraphObject`.

- **getAttrContext**

```
public final AttrContext getAttrContext()
```

Return the attribute context that is used for attribute matching.

See Also: AttrContext

- **setAttrContext**

```
public final void setAttrContext(AttrContext ac)
```

Set the attribute context that is used for attribute matching.

See Also: AttrContext

- **setCompletionStrategy**

```
public final void setCompletionStrategy(ALR_MorphCompletionStrategy sg)
```

Set the algorithm to use for morphism completion. The given strategy is internally cloned to prevent undesired side effects. Class `ALR_CompletionStrategySelector` provides a way to present and obtain available algortihms.

See Also: ALR_CompletionStrategySelector (cf. Section A.4.7)

- **getCompletionStrategy**

```
public final ALR_MorphCompletionStrategy getCompletionStrategy()
```

Return the strategy currently used by `nextCompletion()`.

See Also: nextCompletion (cf. Section A.4.1)

- **nextCompletion**

```
public final boolean nextCompletion()
```

Compute my next completion. Invoke this method successively to get all my completions.

Returns: `false` if there are no more completions.

- **isTotal**

```
public final boolean isTotal()
```

Return `true` iff I am a total morphism.

- **getRefTree**

```
public final ALR_RefTree getRefTree()
```

Return my ALR refinement tree representation.

- **setChanged**

```
protected synchronized void setChanged()
```

Overrides: setChanged in class Observable

A.4.2 Class agg.alr.transform.ALR_BadMappingException

```
java.lang.Object
  |
  +----java.lang.Throwable
          |
          +----java.lang.Exception
                  |
                  +----java.lang.RuntimeException
                          |
                          +----agg.alr.transform.ALR_BadMappingException
```

public class ALR_BadMappingException

extends RuntimeException

This Exception is thrown by methods of the class `ALR_Morphism`, especially by `addMapping()`, to indicate a violation of some morphism property. A more detailed description of the violation cause will be given in the exception's detail message.

See Also: ALR_Morphism (cf. Section A.4.1), addMapping (cf. Section A.4.1)

Constructors

- **ALR_BadMappingException**

```
public ALR_BadMappingException()
```

Construct myself as an exception without any detail message.

- **ALR_BadMappingException**

```
public ALR_BadMappingException(String n)
```

Construct myself with the specified detail message.

A.4.3 Class agg.alr.transform.ALR_Change_MappingAdded

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.alr.transform.ALR_Change_MappingAdded
```

public class ALR_Change_MappingAdded

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR morphism if a mapping has been added to it.

The item is of type `ALR_GraphObject` and references the object which has been assigned a new image. The image object can be obtained via the `getImage(ALR_GraphObject)` method on the morphism.

See Also: ALR_Morphism (cf. Section A.4.1), getImage (cf. Section A.4.1)

Constructors

- **ALR_Change_MappingAdded**

```
public ALR_Change_MappingAdded(ALR_GraphObject item)
```

A.4.4 Class agg.alr.transform.ALR_Change_MappingRemoved

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.alr.transform.ALR_Change_MappingRemoved
```

public class ALR_Change_MappingRemoved

extends Change (cf. Section A.1.1)

This change information is sent out by an ALR morphism if a mapping has been removed from it.

The item is of type `ALR_GraphObject` and references the object which is no longer in the morphism's domain.

See Also: ALR_Morphism (cf. Section A.4.1)

Constructors

- **ALR_Change_MappingRemoved**

```
public ALR_Change_MappingRemoved(ALR_GraphObject item)
```

A.4.5 Class agg.alr.transform.ALR_MorphCompletionStrategy

```
java.lang.Object
 |
 +----agg.alr.transform.ALR_MorphCompletionStrategy
```

public abstract class ALR_MorphCompletionStrategy

extends Object

implements StrategyProperties, CompletionPropertyBits, Cloneable (cf. Section A.4.6)

Abstract class with preimplemented property support for the implementation of morphism completion algorithms.

Constructors

- **ALR_MorphCompletionStrategy**

  `public ALR_MorphCompletionStrategy(BitSet supported_properties)`

  Construct myself to be a completion strategy with support for the properties whose corresponding bits are set in the given `BitSet`.

Methods

- **next**

  `protected abstract boolean next(ALR_Morphism morph)`

  Compute the next completion of `morph`. Invoke this method successively with the same argument to get all completions of a morphism.

  Parameters: morph - the morphism to totalize.

  Returns: `false` if there are no more completions.

- **reset**

  `protected abstract void reset()`

  Reset my internal state, so that the forthcoming invocation of `next()` computes the first completion of the given morphism.

- **getSupportedProperties**

  `public BitSet getSupportedProperties()`

  Return information about what properties I support. A property is supported if its corresponding bit is set.

- **getProperties**

```
public BitSet getProperties()
```

Return information about what properties are currently activated. Properties can be activated or deactivated by setting or clearing their respective bits via the `BitSet` interface.

- **clone**

```
public Object clone()
```

Return a clone of myself. My property bitset is cloned as well.

Overrides: clone in class Object

- **equals**

```
public boolean equals(Object other)
```

Return `true` iff the given object is an instance of the same concrete strategy class as me and has the same property bits set.

Overrides: equals in class Object

A.4.6 Interface agg.alr.transform.CompletionPropertyBits

public interface CompletionPropertyBits

This interface defines symbolic names for the properties that can be asked from a morphism completion strategy. These are names for the bits of a `BitSet` that is used to define which properties are supported by a concrete completion strategy and which properties are actually set.

Example 1: When you want to know if a given strategy `strat` supports restriction to completions that satisfy the dangling condition, you would ask `strat.getSupportedProperties().get(DANGLING)`.

Example 2: When you want a strategy to find only injective completions, you would do `strat.getProperties.set(INJECTIVE)`. Setting a property that is not supported will have no effect.

See Also: BitSet, StrategyProperties, ALR_MorphCompletionStrategy (cf. Section A.4.5)

Variables

- **INJECTIVE**

```
public final static int INJECTIVE
```

If this bit is set to `true`, only injective completions of a morphism will be found.

• **DANGLING**

```
public final static int DANGLING
```

If this bit is set to `true`, only matches that satisfy the dangling condition will be found. (This bit only applies to the completion of match morphisms.)

• **IDENTIFICATION**

```
public final static int IDENTIFICATION
```

If this bit is set to `true`, only matches that satisfy the identification condition will be found. (This bit only applies to the completion of match morphisms.)

• **BITNAME**

```
public final static String BITNAME[]
```

A short descriptive name for each of my bits. This may be used e.g. for GUI button labels.

A.4.7 Class agg.alr.transform.ALR_CompletionStrategySelector

```
java.lang.Object
 |
 +----agg.alr.transform.ALR_CompletionStrategySelector
```

public class ALR_CompletionStrategySelector

extends Object

This class contains an instance of every available completion strategy associated with a minimal description. It provides the basic functionality to implement interactive strategy selection via a GUI.

Constructors

• **ALR_CompletionStrategySelector**

```
public ALR_CompletionStrategySelector()
```

Methods

• **getStrategies**

```
public static Enumeration getStrategies()
```

Return an enumeration of available strategies. Enumeration elements are of type `ALR_MorphCompletionStrategy`.

See Also: ALR_MorphCompletionStrategy (cf. Section A.4.5)

- **getDefault**

```
public static ALR_MorphCompletionStrategy getDefault()
```

Return the default strategy.

- **getName**

```
public static String getName(ALR_MorphCompletionStrategy strat)
```

Return a short descriptive name for the given strategy. This name is intended for use e.g. as a label in a GUI listbox.

A.4.8 Class agg.alr.transform.ALR_Match

```
java.lang.Object
 |
 +----java.util.Observable
         |
         +----agg.util.ExtObservable
                 |
                 +----agg.alr.transform.ALR_Morphism
                         |
                         +----agg.alr.transform.ALR_Match
```

public class ALR_Match

extends ALR_Morphism (cf. Section A.4.1)

This class is used to represent ALR matches, i.e. ALR morphisms from the left side ALR graph of a rule into a host ALR graph. Note that not every instance of this class is a valid match in terms of theory, because in theory a match has to be a total morphism. The `isTotal()` method can be used to check for this additional property dynamically.

See Also: isTotal (cf. Section A.4.1)

Constructors

- **ALR_Match**

```
public ALR_Match(ALR_RefTree match,
               ALR_Morphism rule)

throws ALR_BadMappingException
```

Construct myself out of a given ALR relation.

Pre:

1. `match.isRelation()`.
2. `match.getTopObject().getSource().equals( rule.getRefTree().get-TopObject().getSource() )`.

Parameters: match - the ALR relation that is used to model the match morphism. rule - the rule morphism I am a match for.

Throws: ALR_BadMappingException (cf. Section A.4.2) when the given refinement tree does not represent a valid ALR morphism.

Methods

- **getRuleMorphism**

```
public final ALR_Morphism getRuleMorphism()
```

Return the rule morphism that I am a match for.

A.4.9 Class agg.alr.transform.ALR_Step

```
java.lang.Object
 |
 +----agg.alr.transform.ALR_Step
```

public class ALR_Step

extends Object

This class implements a direct graph transformation step in the single pushout (SPO) approach to algebraic graph transformation. The transformation is performed *in place*, i.e. the host graph is modified according to the rule's instructions.

The transformation result is computed in two steps: First the pushout of the graph part is computed using the colimit library, and then the attributes are calculated using the attribute package.

Constructors

- **ALR_Step**

```
public ALR_Step()
```

Methods

- **execute**

```
public final static void execute(ALR_Match match)
```

Perform an inplace graph transformation step: apply the rule given by `match.getRuleMorphism()` via `match` on the host graph given by `match.getImage()`. The host graph is modified to represent the result of the application.

See Also: getImage (cf. Section A.4.1)

- **finalize**

```
protected final void finalize()
```

Overrides: finalize in class Object

A.4.10 Class agg.alr.transform.ALR_ColimDiagram

```
java.lang.Object
 |
 +----agg.alr.transform.ALR_ColimDiagram
```

public class ALR_ColimDiagram

extends Object

implements COLIM_DEFS

This class allows for representation of general diagrams of ALR-graphs and for computation of their colimit. It has capabilities for optional in-place computation of the colimit object in one of the diagram nodes. Attributes are ignored for colimit computation. The colimit computation itself is implemented using the colimit library from Dietmar Wolz.

Constructors

- **ALR_ColimDiagram**

  `public ALR_ColimDiagram(ALR_RefTree result)`

  Construct myself to be an empty diagram where the colimit object is to be computed into the given ALR-Graph `result`. By adding `result` as an ordinary diagram node via `addNode` as well, in-place computation can be achieved.

  Pre: `result.isGraph()`.

Methods

- **addNode**

  `public void addNode(ALR_RefTree graph)`

  Add an ALR-Graph as a node to the diagram.

  Pre: `graph.isGraph()`.

- **addEdge**

  `public void addEdge(ALR_Morphism morph)`

  Add an ALR-Morphism as an edge to the diagram.

  Pre: `morph.getOriginal()` and `morph.getImage()` have been added to the diagram with `addNode()` before.

  See Also: getOriginal (cf. Section A.4.1), getImage (cf. Section A.4.1), addNode (cf. Section A.4.10)

- **computeColimit**

  `public final void computeColimit()`

Perform the colimit computation for the diagram I'm representing. The ALR-Graph `result` which has been passed to my constructor becomes the colimit object, and the colimit morphisms requested by `requestEdge()` are built accordingly.

See Also: requestEdge (cf. Section A.4.10)

- **requestEdge**

```
public final void requestEdge(ALR_Morphism morph)
```

Request the computation of the given empty morphism as a colimit morphism.

Pre:

1. The domain of `morph` is empty.
2. `morph.getOriginal()` has been added to the diagram via `addNode()`.
3. `morph.getImage()` is the `result` object that has been passed to my constructor.

See Also: getOriginal (cf. Section A.4.1), getImage (cf. Section A.4.1), addNode (cf. Section A.4.10)

A.4.11 Class agg.alr.transform.impl.ALR_CSP

```
java.lang.Object
 |
 +----agg.util.csp.CSP
         |
         +----agg.alr.transform.impl.ALR_CSP
```

public class ALR_CSP

extends CSP (cf. Section A.2.1)

A CSP whose solutions represent morphisms between two ALR graphs.

See Also: CSP (cf. Section A.2.1)

Constructors

- **ALR_CSP**

```
public ALR_CSP(ALR_RefTree vargraph,
               AttrContext ac,
               boolean injective)
```

Construct myself to be a CSP where every GraphObject of `vargraph` corresponds to exactly one of my variables.

Pre: `vargraph.isGraph()`.

Post: `getDomain() == null`.

Parameters: vargraph - The ALR graph whose elements represent the variables of the CSP. ac - The attribute context to map attributes in. injective - If set to `true`, only injective solutions will be considered.

Methods

- **preprocessDomain**

```
protected final void preprocessDomain(Object domaingraph)
```

Preprocess the given domain for optimization purposes, e.g.

Overrides: preprocessDomain (cf. Section A.2.1) in class CSP (cf. Section A.2.1)

- **getVariables**

```
public final Enumeration getVariables()
```

Return an Enumeration of all my variables.

Overrides: getVariables (cf. Section A.2.1) in class CSP (cf. Section A.2.1)

- **getSize**

```
public final int getSize()
```

Return the number of variables in the CSP.

Overrides: getSize (cf. Section A.2.1) in class CSP (cf. Section A.2.1)

- **getVariable**

```
public final Variable getVariable(ALR_GraphObject obj)
```

A.4.12 Class agg.alr.transform.impl.Completion_CSP

```
java.lang.Object
  |
  +----agg.alr.transform.ALR_MorphCompletionStrategy
          |
          +----agg.alr.transform.impl.Completion_CSP
```

public class Completion_CSP

extends ALR_MorphCompletionStrategy (cf. Section A.4.5)

An implementation of ALR morphism completion as a Constraint Satisfaction Problem (CSP).

Constructors

- **Completion_CSP**

```
public Completion_CSP()
```

Methods

- **reset**

```
protected final void reset()
```

Reset my internal state, so that the forthcoming invocation of `next()` computes the first completion of the given morphism.

Overrides: reset (cf. Section A.4.5) in class ALR_MorphCompletionStrategy (cf. Section A.4.5)

- **next**

```
protected final boolean next(ALR_Morphism morph)
```

Compute the next completion of `morph`.

Overrides: next (cf. Section A.4.5) in class ALR_MorphCompletionStrategy (cf. Section A.4.5)

A.4.13 Class agg.alr.transform.impl.Completion_InjCSP

```
java.lang.Object
 |
 +----agg.alr.transform.ALR_MorphCompletionStrategy
         |
         +----agg.alr.transform.impl.Completion_CSP
                 |
                 +----agg.alr.transform.impl.Completion_InjCSP
```

public class Completion_InjCSP

extends Completion_CSP (cf. Section A.4.12)

An implementation of ALR morphism completion as a Constraint Satisfaction Problem (CSP), considering injective solutions only.

Constructors

- **Completion_InjCSP**

```
public Completion_InjCSP()
```

A.4.14 Class agg.alr.transform.impl.Completion_SimpleBT

```
java.lang.Object
 |
 +----agg.alr.transform.ALR_MorphCompletionStrategy
         |
         +----agg.alr.transform.impl.Completion_SimpleBT
```

public class Completion_SimpleBT

extends ALR_MorphCompletionStrategy (cf. Section A.4.5)

Simple Backtracking implementation of ALR morphism completion.

Constructors

- **Completion_SimpleBT**

```
public Completion_SimpleBT()
```

Methods

- **reset**

```
protected final void reset()
```

Reset my internal state, so that the forthcoming invocation of `next()` computes the first completion of the given morphism.

Overrides: reset (cf. Section A.4.5) in class ALR_MorphCompletionStrategy (cf. Section A.4.5)

- **next**

```
protected final boolean next(ALR_Morphism morph)
```

Compute the next completion of `morph`.

Overrides: next (cf. Section A.4.5) in class ALR_MorphCompletionStrategy (cf. Section A.4.5)

A.4.15 Class agg.alr.transform.impl.Constraint_Abstraction

```
java.lang.Object
 |
 +----agg.util.csp.BinaryConstraint
         |
         +----agg.alr.transform.impl.Constraint_Abstraction
```

public class Constraint_Abstraction

extends BinaryConstraint (cf. Section A.2.4)

Constructors

- **Constraint_Abstraction**

```
public Constraint_Abstraction(Variable abs,
                              Variable obj)
```

Methods

- **execute**

```
public final boolean execute()
```

Return true iff the current instance of `abs` is the abstraction object of `obj`'s instance. Pre: (1) abs.getInstance(), obj.getInstance() instanceof ALR_GraphObject.

Overrides: execute (cf. Section A.2.4) in class BinaryConstraint (cf. Section A.2.4)

A.4.16 Class agg.alr.transform.impl.Constraint_Attribute

```
java.lang.Object
  |
  +----agg.util.csp.BinaryConstraint
          |
          +----agg.alr.transform.impl.Constraint_Attribute
```

public class Constraint_Attribute

extends BinaryConstraint (cf. Section A.2.4)

implements InstantiationHook (cf. Section A.2.3)

Constructors

- **Constraint_Attribute**

```
public Constraint_Attribute(ALR_GraphObject graphobj,
                            Variable var,
                            AttrContext ac,
                            AttrManager man)
```

Methods

- **execute**

```
public final boolean execute()
```

Return true iff the attributes of `graphobj` and of the current instance of `var` match. (The names correspond to my constructor arguments.)

Pre: (1) var.getInstance() instanceof ALR_GraphObject.

Overrides: execute (cf. Section A.2.4) in class BinaryConstraint (cf. Section A.2.4)

- **instantiate**

```
public final void instantiate(Variable var)
```

- **uninstantiate**

```
public final void uninstantiate(Variable var)
```

A.4.17 Class agg.alr.transform.impl.Constraint_Source

```
java.lang.Object
  |
  +----agg.util.csp.BinaryConstraint
          |
          +----agg.alr.transform.impl.Constraint_Source
```

public class Constraint_Source

extends BinaryConstraint (cf. Section A.2.4)

Constructors

- **Constraint_Source**

```
public Constraint_Source(Variable src,
                         Variable arc)
```

Methods

- **execute**

```
public final boolean execute()
```

Return true iff the current instance of `src` is the source object of the instance of `arc`.

Pre: (1) src.getInstance(), arc.getInstance() instanceof ALR_GraphObject.

Overrides: execute (cf. Section A.2.4) in class BinaryConstraint (cf. Section A.2.4)

A.4.18 Class agg.alr.transform.impl.Constraint_Target

```
java.lang.Object
  |
  +----agg.util.csp.BinaryConstraint
          |
          +----agg.alr.transform.impl.Constraint_Target
```

public class Constraint_Target

extends BinaryConstraint (cf. Section A.2.4)

Constructors

- **Constraint_Target**

```
public Constraint_Target(Variable tar,
                         Variable arc)
```

Methods

- **execute**

```
public final boolean execute()
```

Return true iff the current instance of `tar` is the target object of the instance of `arc`.

Pre: (1) tar.getInstance(), arc.getInstance() instanceof ALR_GraphObject.

Overrides: execute (cf. Section A.2.4) in class BinaryConstraint (cf. Section A.2.4)

A.4.19 Class agg.alr.transform.impl.Constraint_Type

```
java.lang.Object
 |
 +----agg.util.csp.BinaryConstraint
         |
         +----agg.alr.transform.impl.Constraint_Type
```

public class Constraint_Type

extends BinaryConstraint (cf. Section A.2.4)

Constructors

- **Constraint_Type**

```
public Constraint_Type(ALR_GraphObject graphobj,
                       Variable obj)
```

Methods

- **execute**

```
public final boolean execute()
```

Return true iff the current instance of `obj` is type compatible with the GraphObject that has been passed to my constructor. In this case, *type* means an element of the cartesian product of `Type` x {Node,Arc}.

Pre: (1) obj.getInstance() instanceof ALR_GraphObject.

Overrides: execute (cf. Section A.2.4) in class BinaryConstraint (cf. Section A.2.4)

A.4.20 Class agg.alr.transform.impl.Query_Abstraction

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Abstraction
```

public class Query_Abstraction

extends Query (cf. Section A.2.5)

Constructors

• **Query_Abstraction**

```
public Query_Abstraction(Variable obj,
                         Variable querytar)
```

Construct myself to be a unary query for the abstraction of `obj`.

Methods

• **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

• **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.4.21 Class agg.alr.transform.impl.Query_Incoming

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Incoming
```

public class Query_Incoming

extends Query (cf. Section A.2.5)

Constructors

• **Query_Incoming**

```
public Query_Incoming(Variable obj,
                      Variable abs,
                      Variable tar)
```

Construct myself to be a binary query for incoming arcs of `obj` with abstraction `abs`.

Methods

- **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

- **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.4.22 Class agg.alr.transform.impl.Query_Outgoing

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Outgoing
```

public class Query_Outgoing

extends Query (cf. Section A.2.5)

Constructors

- **Query_Outgoing**

```
public Query_Outgoing(Variable obj,
                      Variable abs,
                      Variable tar)
```

Construct myself to be a binary query for outgoing arcs of `obj` with abstraction `abs`.

Methods

- **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

- **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.4.23 Class agg.alr.transform.impl.Query_Refinement

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Refinement
```

public class Query_Refinement

extends Query (cf. Section A.2.5)

Constructors

- **Query_Refinement**

```
public Query_Refinement(Variable obj,
                        Variable querytar)
```

Construct myself to be a unary query for refinement Objects of `obj`.

Methods

- **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

- **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.4.24 Class agg.alr.transform.impl.Query_Source

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Source
```

public class Query_Source

extends Query (cf. Section A.2.5)

Constructors

- **Query_Source**

```
public Query_Source(Variable arc,
                    Variable querytar)
```

Construct myself to be a unary query for the source of `arc`.

Methods

- **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

- **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.4.25 Class agg.alr.transform.impl.Query_Target

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Target
```

public class Query_Target

extends Query (cf. Section A.2.5)

Constructors

- **Query_Target**

```
public Query_Target(Variable arc,
                    Variable querytar)
```

Construct myself to be a unary query for the target of `arc`.

Methods

- **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

- **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.4.26 Class agg.alr.transform.impl.Query_Type

```
java.lang.Object
 |
 +----agg.util.csp.Query
         |
         +----agg.alr.transform.impl.Query_Type
```

public class Query_Type

extends Query (cf. Section A.2.5)

Constructors

- **Query_Type**

```
public Query_Type(Vector objects,
                  Variable querytar)
```

Construct myself to be a constant query for the objects given in the parameter `objects`.

Methods

- **execute**

```
public final Enumeration execute()
```

Return an Enumeration of candidate values for the target variable.

Overrides: execute (cf. Section A.2.5) in class Query (cf. Section A.2.5)

- **getSize**

```
public final int getSize()
```

Return the number of candidate values `execute()` will provide.

Overrides: getSize (cf. Section A.2.5) in class Query (cf. Section A.2.5)

A.5 Package basis

A.5.1 Class agg.basis.GraGra

```
java.lang.Object
  |
  +----agg.basis.GraGra
```

public class GraGra

extends Object

implements Serializable

Constructors

- **GraGra**

  `public GraGra()`

Methods

- **setName**

  `public final void setName(String n)`

  Set my name.

- **getName**

  `public final String getName()`

  Return my name.

- **getGraph**

  `public final Graph getGraph()`

  Return my start graph.

- **getRules**

  `public final Enumeration getRules()`

  Iterate through all of my rules. Enumeration elements are of type `Rule`.

  See Also: Rule (cf. Section A.5.16)

- **createRule**

  `public Rule createRule()`

  Create an empty rule.

- **destroyRule**

```
public final void destroyRule(Rule rule)
```

Dispose the specified rule.

- **getMatches**

```
public final Enumeration getMatches(Rule rule)
```

Iterate through all of the matches existing between the given rule and the start graph. Enumeration elements are of type `Match`.

See Also: Match (cf. Section A.5.15)

- **createMatch**

```
public final Match createMatch(Rule rule)
```

Create an empty match morphism between the left side of the given rule and my start graph. Note that this does not yield a valid match (unless the left side of the given rule is empty), because matches have to be total morphisms.

- **destroyMatch**

```
public final void destroyMatch(Match match)
```

Dispose the specified match.

- **destroyMatch**

```
public final void destroyMatch(Morphism match)
```

Note: destroyMatch() is deprecated. *Use destroyMatch(Match) instead.*

Dispose the specified match.

- **createType**

```
public final Type createType()
```

Create a new type for typing of GraphObjects.

- **getTypes**

```
public final Enumeration getTypes()
```

Iterate through all of the types that may be assigned to GraphObjects. Enumeration elements are of type `Type`.

See Also: Type (cf. Section A.3.4)

A.5.2 Class agg.basis.Graph

```
java.lang.Object
 |
 +----java.util.Observable
         |
         +----agg.util.ExtObservable
                 |
                 +----agg.basis.Graph
```

public class Graph

extends ExtObservable (cf. Section A.1.4)

implements Serializable, Observer

This class implements the concept of a simple graph.

Class Graph is capable of passing the following change informations as an argument to its observers' `update()` methods:

`Graph_Change_ObjectCreated`, `Graph_Change_ObjectDestroyed`, `Graph_Change_ObjectModified`.

Additionally, `Change_ObservableGone` may be generated by its base class `ExtObservable`. If, however, the value of the change information is given as `null`, the observer is recommended to perform a full rescan of the graph.

See Also: Graph_Change_ObjectCreated (cf. Section A.5.3), Graph_Change_ObjectDestroyed (cf. Section A.5.4), Graph_Change_ObjectModified (cf. Section A.5.5), Change_ObservableGone (cf. Section A.1.2)

Constructors

- **Graph**

```
protected Graph()
```

Note: Graph() is deprecated.

Construct myself to be an independent graph. The underlying ALR refinement tree is implicitly created.

- **Graph**

```
protected Graph(ALR_RefTree alrgraph)
```

Attach myself to a given ALR refinement tree representation.

Methods

- **dispose**

```
public final void dispose()
```

Prepare myself for garbage collection.

Overrides: dispose (cf. Section A.1.4) in class ExtObservable (cf. Section A.1.4)

- **update**

```
public final void update(Observable obs,
                         Object change)
```

Process change information.

- **setName**

```
public final void setName(String n)
```

Set my name.

- **getName**

```
public final String getName()
```

Return my name.

- **createType**

```
public final Type createType()
```

Create and return a new type for GraphObjects.

- **getTypes**

```
public final Enumeration getTypes()
```

Iterate through all the valid types that may be given to a GraphObject. Enumeration elements are of type `Type`.

See Also: Type (cf. Section A.3.4)

- **createNode**

```
public final Node createNode(Type type)
```

Create a new Node with given Type.

- **createNode**

```
public final Node createNode(Node orig)
```

Create a new Node as a copy of `orig`. Only the type and the attributes are copied, the structural context (incoming/outgoing arcs) is not.

- **destroyNode**

```
public final void destroyNode(Node node)
```

Delete a Node. Dangling Arcs are deleted implicitly.

- **createArc**

```
public final Arc createArc(Type type,
                           GraphObject src,
                           GraphObject tar)
```

Create a new Arc with given Type, source and target objects.

- **createArc**

```
public final Arc createArc(Arc orig,
                           GraphObject src,
                           GraphObject tar)
```

Create a new Arc as a copy of `orig`. Only the type and the attributes are copied, the structural context (source, target) is not.

- **destroyArc**

```
public final void destroyArc(Arc arc)
```

Delete an Arc.

- **getNodes**

```
public final Enumeration getNodes()
```

Iterate through my Nodes. Enumeration elements are of type `Node`.

See Also: Node (cf. Section A.5.7)

- **getArcs**

```
public final Enumeration getArcs()
```

Iterate through my Arcs. Enumeration elements are of type `Arc`.

See Also: Arc (cf. Section A.5.8)

- **getElements**

```
public final Enumeration getElements()
```

Iterate through my Nodes and Arcs. Enumeration elements are of type `GraphObject`.

See Also: GraphObject (cf. Section A.5.6)

- **destroyObject**

```
public final void destroyObject(GraphObject obj)
```

Delete Arc or Node.

- **isEmpty**

```
public final boolean isEmpty()
```

Return `true` iff I contain no graphobjects.

- **isElement**

```
public final boolean isElement(GraphObject obj)
```

Return `true` iff I contain the specified graphobject.

- **clear**

```
public final void clear()
```

Delete all of my graphobjects.
Post: `isEmpty()`.

- **getALR_Repr**

```
protected final ALR_RefTree getALR_Repr()
```

Return my ALR refinement tree representation.

A.5.3 Class agg.basis.Graph_Change_ObjectCreated

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.basis.Graph_Change_ObjectCreated
```

public class Graph_Change_ObjectCreated

extends Change (cf. Section A.1.1)

This change information is sent out by a graph if a new graphobject has been created. The item is of type `GraphObject` and references the newly created object.

See Also: Graph (cf. Section A.5.2)

Constructors

- **Graph_Change_ObjectCreated**

```
public Graph_Change_ObjectCreated(GraphObject item)
```

A.5.4 Class agg.basis.Graph_Change_ObjectDestroyed

```
java.lang.Object
  |
  +----agg.util.Change
          |
          +----agg.basis.Graph_Change_ObjectDestroyed
```

public class Graph_Change_ObjectDestroyed

extends Change (cf. Section A.1.1)

This change information is sent out by a graph if one of its graphobjects has been deleted.

The item is of type `GraphObject` and references the object which has just been deleted. The object is fully accessible, because the garbage collector will not get hold of it as long as we hold a reference.

See Also: Graph (cf. Section A.5.2)

Constructors

- **Graph_Change_ObjectDestroyed**

```
public Graph_Change_ObjectDestroyed(GraphObject item)
```

A.5.5 Class agg.basis.Graph_Change_ObjectModified

```
java.lang.Object
  |
  +----agg.util.Change
          |
          +----agg.basis.Graph_Change_ObjectModified
```

public class Graph_Change_ObjectModified

extends Change (cf. Section A.1.1)

This change information is sent out by a graph if the attributes of of one of its graphobjects have been modified.

The item is of type `GraphObject` and references the object whose attributes have just been modified.

See Also: Graph (cf. Section A.5.2)

Constructors

- **Graph_Change_ObjectModified**

  `public Graph_Change_ObjectModified(GraphObject item)`

A.5.6 Class agg.basis.GraphObject

```
java.lang.Object
 |
 +----agg.basis.GraphObject
```

public abstract class GraphObject

extends Object

implements Serializable

GraphObject defines the common interface and implementations for Nodes and Arcs. It is implemented by delegation to an underlying ALR_GraphObject.

Variables

- **itsALR_Repr**

  `protected ALR_GraphObject itsALR_Repr`

Constructors

- **GraphObject**

  `protected GraphObject(ALR_GraphObject alrobj)`

  Attach to an existing underlying ALR-representation. Intended for use in subclasses only.

Methods

- **equals**

  `public final boolean equals(Object obj)`

  Test myself for equality with another graphobject.

  Overrides: equals in class Object

- **isArc**

  `public abstract boolean isArc()`

  Return `true` iff I am an arc.

- **isNode**

  `public abstract boolean isNode()`

Return `true` iff I am a node.

- **getType**

  ```
  public final Type getType()
  ```

  Return my type.

- **getAttribute**

  ```
  public final AttrInstance getAttribute()
  ```

  Return my attribute value.

- **getALR_Repr**

  ```
  protected final ALR_GraphObject getALR_Repr()
  ```

  Return my ALR graphobject representation.

A.5.7 Class agg.basis.Node

```
java.lang.Object
 |
 +----agg.basis.GraphObject
         |
         +----agg.basis.Node
```

public class Node

extends GraphObject (cf. Section A.5.6)

A subclass of `GraphObject` that represents nodes.

Constructors

- **Node**

  ```
  public Node(ALR_GraphObject alrobj)
  ```

  Attach myself to an existing underlying ALR-representation.

Methods

- **isArc**

  ```
  public final boolean isArc()
  ```

  Return `false`.

  Overrides: isArc (cf. Section A.5.6) in class GraphObject (cf. Section A.5.6)

- **isNode**

```
public final boolean isNode()
```

Return `true`.

Overrides: isNode (cf. Section A.5.6) in class GraphObject (cf. Section A.5.6)

- **getIncomingArcs**

```
public final Enumeration getIncomingArcs()
```

Iterate through all the arcs that I am the target of. Enumeration elements are of type `Arc`.

See Also: Arc (cf. Section A.5.8)

- **getOutgoingArcs**

```
public Enumeration getOutgoingArcs()
```

Iterate through all the arcs that I am the source of. Enumeration elements are of type `Arc`.

See Also: Arc (cf. Section A.5.8)

A.5.8 Class agg.basis.Arc

```
java.lang.Object
 |
 +----agg.basis.GraphObject
         |
         +----agg.basis.Arc
```

public class Arc

extends GraphObject (cf. Section A.5.6)

A subclass of `GraphObject` that represents arcs.

Constructors

- **Arc**

```
public Arc(ALR_GraphObject alrobj)
```

Attach myself to an existing underlying ALR-representation.

Methods

- **isArc**

  `public final boolean isArc()`

  Return `true`.

  Overrides: isArc (cf. Section A.5.6) in class GraphObject (cf. Section A.5.6)

- **isNode**

  `public final boolean isNode()`

  Return `false`.

  Overrides: isNode (cf. Section A.5.6) in class GraphObject (cf. Section A.5.6)

- **getSource**

  `public final GraphObject getSource()`

  Return my source object.

- **getTarget**

  `public final GraphObject getTarget()`

  Return my target object.

A.5.9 Class agg.basis.Morphism

```
java.lang.Object
 |
 +----java.util.Observable
         |
         +----agg.util.ExtObservable
                 |
                 +----agg.basis.Morphism
```

public class Morphism

extends ExtObservable (cf. Section A.1.4)

implements Serializable, Observer

Implementation of a graph morphism, modeled by an ALR relation. Consistency is achieved by letting this class be an Observer of its modelling ALR_Morphism, which in turn wraps the underlying ALR_RefTree structure.

Class Morphism is capable of passing the following change informations as an argument to its observers' `update()` methods:

`Morph_Change_MappingAdded, Morph_Change_MappingRemoved.`

Additionally, the information `Change_ObservableGone` may be generated by its base class `ExtObservable`. If, however, the value of the change information is given as `null`, the observer is recommended to perform a full rescan of the morphism.

Note: This implementation is guaranteed to keep morphism properties when objects are deleted/created/modified in the original or image graphs. It is NOT, however, guaranteed to keep morphism properties if arcs are added to the underlying ALR relation from outside of this class.

See Also: Morph_Change_MappingAdded (cf. Section A.5.11), Morph_Change_MappingRemoved (cf. Section A.5.12), Change_ObservableGone (cf. Section A.1.2)

Constructors

- **Morphism**

```
protected Morphism(ALR_RefTree alr_rt,
                   Graph orig,
                   Graph image)
```

Attach myself to an existing underlying ALR refinement tree representation. The ALR morphism that wraps this refinement tree is newly created.

Parameters: alr_rt - the ALR relation that represents the morphism. orig - the graph where the morphism starts. image - the graph where the morphism ends.

Pre: `alr_repr.isRelation()`.

- **Morphism**

```
protected Morphism(ALR_Morphism alr_morph,
                   Graph orig,
                   Graph image)
```

Attach myself to an existing underlying ALR morphism representation.

Parameters: alr_morph - the underlying ALR morphism representation to attach to. orig - the graph where the morphism starts. image - the graph where the morphism ends.

Methods

- **dispose**

```
public final void dispose()
```

Prepare myself for garbage collection.

Overrides: dispose (cf. Section A.1.4) in class ExtObservable (cf. Section A.1.4)

- **update**

```
public final void update(Observable obs,
                         Object change)
```

Process change information.

- **setName**

```
public final void setName(String n)
```

Set my name.

- **getName**

```
public final String getName()
```

Return my name.

- **addMapping**

```
public void addMapping(GraphObject o,
                       GraphObject i)
```

```
throws BadMappingException
```

Map an object of my source graph to an object of my target graph.

Pre:

1. `o` is an element of `getOriginal()`.
2. `i` is an element of `getImage()`.

Parameters: o - the source object of the mapping. i - the target object of the mapping.

Throws: BadMappingException (cf. Section A.5.10) if the given mapping violates morphism properties.

- **removeMapping**

```
public final void removeMapping(GraphObject o)
```

Remove the mapping of a given GraphObject. Any mappings of incoming or outgoing arcs will be removed as well.

Pre: `getImage(o) != null`

- **clear**

```
public final void clear()
```

Remove all of my mappings.

- **getOriginal**

```
public final Graph getOriginal()
```

Return my source graph.

- **getImage**

```
public final Graph getImage()
```

Return my target graph.

- **getDomain**

```
public final Enumeration getDomain()
```

Return an Enumeration of the graphobjects out of my source graph which are actually taking part in one of my mappings. Enumeration elements are of type `GraphObject`.

See Also: GraphObject (cf. Section A.5.6)

- **getCodomain**

```
public final Enumeration getCodomain()
```

Return an Enumeration of the graphobjects out of my target graph which are actually taking part in one of my mappings. Enumeration elements are of type `GraphObject`.

See Also: GraphObject (cf. Section A.5.6)

- **getImage**

```
public final GraphObject getImage(GraphObject o)
```

Return the image of the specified object.

Returns: `null` if the object is not in domain.

- **getInverseImage**

```
public final Enumeration getInverseImage(GraphObject o)
```

Return an Enumeration of the inverse images of the specified object. Enumeration will be empty when the object is not in codomain. Enumeration elements are of type `GraphObject`.

See Also: GraphObject (cf. Section A.5.6)

- **setCompletionStrategy**

```
public final void setCompletionStrategy(MorphCompletionStrategy strat)
```

Set the algorithm to use for morphism completion. The given strategy is internally cloned to prevent undesired side effects. Class `CompletionStrategySelector` provides a way to present and obtain available algorithms.

See Also: CompletionStrategySelector (cf. Section A.5.14)

- **getCompletionStrategy**

```
public final MorphCompletionStrategy getCompletionStrategy()
```

Return the strategy currently used by `nextCompletion()`.

See Also: next (cf. Section A.4.5)

- **nextCompletion**

```
public final boolean nextCompletion()
```

Compute my next completion. Invoke this method successively to get all my completions.

Returns: `false` if there are no more completions.

- **isTotal**

```
public final boolean isTotal()
```

Return `true` iff I am a total morphism.

- **getAttrContext**

```
public final AttrContext getAttrContext()
```

Return the attribute context that is used for attribute matching.

See Also: AttrContext

- **getALR_Repr**

```
protected final ALR_RefTree getALR_Repr()
```

Return my ALR refinement tree representation. This is the first level implementation.

- **getALR_Morph**

```
protected final ALR_Morphism getALR_Morph()
```

Return my ALR morphism representation. This is the second level implementation.

A.5.10 Class agg.basis.BadMappingException

```
java.lang.Object
 |
 +----java.lang.Throwable
         |
         +----java.lang.Exception
                 |
                 +----java.lang.RuntimeException
                         |
                         +----agg.basis.BadMappingException
```

public class BadMappingException

extends RuntimeException

This Exception is thrown by methods of the class `Morphism`, especially by `addMapping()`, to indicate a violation of some morphism property. A more detailed description of the violation cause will be given in the exception's detail message.

See Also: Morphism (cf. Section A.5.9), addMapping (cf. Section A.5.9)

Constructors

- **BadMappingException**

  `public BadMappingException()`

  Construct myself as an exception without any detail message.

- **BadMappingException**

  `public BadMappingException(String n)`

  Construct myself with the specified detail message.

A.5.11 Class agg.basis.Morph_Change_MappingAdded

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.basis.Morph_Change_MappingAdded
```

public class Morph_Change_MappingAdded

extends Change (cf. Section A.1.1)

This change information is sent out by a morphism if a mapping has been added to it.

The item is of type `GraphObject` and references the object which has been assigned a new image. The image object can be obtained via the `getImage(GraphObject)` method on the morphism.

See Also: Morphism (cf. Section A.5.9), getImage (cf. Section A.5.9)

Constructors

- **Morph_Change_MappingAdded**

```
public Morph_Change_MappingAdded(GraphObject item)
```

A.5.12 Class agg.basis.Morph_Change_MappingRemoved

```
java.lang.Object
 |
 +----agg.util.Change
         |
         +----agg.basis.Morph_Change_MappingRemoved
```

public class Morph_Change_MappingRemoved

extends Change (cf. Section A.1.1)

This change information is sent out by a morphism if a mapping has been removed from it.

The item is of type `GraphObject` and references the object which is no longer in the morphism's domain.

See Also: Morphism (cf. Section A.5.9)

Constructors

- **Morph_Change_MappingRemoved**

```
public Morph_Change_MappingRemoved(GraphObject item)
```

A.5.13 Class agg.basis.MorphCompletionStrategy

```
java.lang.Object
 |
 +----agg.basis.MorphCompletionStrategy
```

public class MorphCompletionStrategy

extends Object

implements StrategyProperties, CompletionPropertyBits (cf. Section A.4.6)

Adapter class for the ALR implementations of morphism completion algorithms.

See Also: ALR_MorphCompletionStrategy (cf. Section A.4.5)

Constructors

- **MorphCompletionStrategy**

```
protected MorphCompletionStrategy(ALR_MorphCompletionStrategy alrstrat)
```

Attach myself to given ALR morphism completion strategy.

Methods

- **getSupportedProperties**

```
public final BitSet getSupportedProperties()
```

Return information about what properties I support. A property is supported if its corresponding bit is set.

- **getProperties**

```
public final BitSet getProperties()
```

Return information about what properties are currently activated. Properties can be activated or deactivated by setting or clearing their respective bits via the `BitSet` interface.

- **equals**

```
public final boolean equals(Object other)
```

Return `true` iff the given object is an instance of the same concrete strategy class as me and has the same property bits set.

Overrides: equals in class Object

- **getALR_Repr**

```
protected final ALR_MorphCompletionStrategy getALR_Repr()
```

Return my ALR representation.

A.5.14 Class agg.basis.CompletionStrategySelector

```
java.lang.Object
 |
 +----agg.basis.CompletionStrategySelector
```

public class CompletionStrategySelector

extends Object

This class contains an instance of every available completion strategy associated with a minimal description. It provides the basic functionality to implement interactive strategy selection via a GUI.

Constructors

- **CompletionStrategySelector**

```
public CompletionStrategySelector()
```

Methods

- **getStrategies**

```
public final static Enumeration getStrategies()
```

Return an enumeration of available strategies. Enumeration elements are of type `MorphCompletionStrategy`.

See Also: MorphCompletionStrategy (cf. Section A.5.13)

- **getDefault**

```
public final static MorphCompletionStrategy getDefault()
```

Return the default strategy.

- **getName**

```
public final static String getName(MorphCompletionStrategy strat)
```

Return a short descriptive name for the given strategy. This name is intended for use e.g. as a label in a GUI listbox.

A.5.15 Class agg.basis.Match

```
java.lang.Object
 |
 +----java.util.Observable
         |
         +----agg.util.ExtObservable
                 |
                 +----agg.basis.Morphism
                         |
                         +----agg.basis.Match
```

public class Match

extends Morphism (cf. Section A.5.9)

This class is used to represent matches, i.e. morphisms from the left side graph of a rule into a host graph. Note that not every instance of this class is a valid match in terms of theory, because in theory a match has to be a total morphism. The `isTotal()` method can be used to check for this additional property.

See Also: isTotal (cf. Section A.5.9)

Constructors

- **Match**

```
protected Match(ALR_RefTree rt,
                Rule rule,
                Graph host)
```

Construct myself to be an empty match from the left graph of `rule` into `host`.

Pre: `rt.isRelation()`.

Parameters: rt - an empty ALR relation that represents the match morphism. rule - the rule to find a match for. host - the target graph of the match morphism.

Methods

- **getRule**

```
public final Rule getRule()
```

Return the rule that I am a match for.

A.5.16 Class agg.basis.Rule

```
java.lang.Object
 |
 +----agg.basis.Rule
```

public class Rule

extends Object

implements Serializable

Simple implementation of an SPO graph transformation rule modeled by an ALR graph.

Constructors

- **Rule**

```
public Rule(ALR_RefTree alrGraph,
            GraGra myGraGra)
```

Construct myself to be an empty graph rule.

Pre: `alrGraph.isGraph()`.

Parameters: alrGraph - an empty ALR graph that is given the internal structure to represent a rule. myGraGra - the graph grammar I belong to.

Methods

- **setName**

```
public final void setName(String n)
```

Set my name.

- **getName**

```
public final String getName()
```

Return my name.

- **getLeft**

```
public final Graph getLeft()
```

Return my left graph.

- **getRight**

```
public final Graph getRight()
```

Return my right graph.

- **getMorphism**

```
public final Morphism getMorphism()
```

Return the rule morphism between my left and right graphs.

- **getMatches**

```
public final Enumeration getMatches()
```

Iterate through all the matches that are currently available between my left graph and the start graph of the grammar I belong to. Enumeration elements are of type `Match`.

Note that the morphisms returned by this enumeration are not guaranteed to be valid matches; see comment to `createMatch()`.

See Also: createMatch (cf. Section A.5.16), Match (cf. Section A.5.15)

- **createMatch**

```
public final Match createMatch()
```

Create an empty match morphism between my left graph and the start graph of the graph grammar I belong to.

Note that this does not yield a valid match (unless the left side of the given rule is empty), because matches have to be total morphisms.

- **destroyMatch**

```
public final void destroyMatch(Morphism match)
```

Note: destroyMatch() is deprecated. *Use destroyMatch(Match) instead.*

Dispose the specified match.

- **destroyMatch**

```
public final void destroyMatch(Match match)
```

Dispose the specified match.

- **getAttrContext**

```
public AttrContext getAttrContext()
```

Return the attribute context in which my variables are declared.

- **getALR_Repr**

```
protected final ALR_RefTree getALR_Repr()
```

Return my ALR refinement tree representation.

Post: `r.isGraph()`, where `r` is the return value.

- **getALR_Morph**

```
protected final ALR_Morphism getALR_Morph()
```

Return the ALR morphism representation of my rule morphism.

A.5.17 Class agg.basis.Step

```
java.lang.Object
 |
 +----agg.basis.Step
```

public class Step

extends Object

This class implements a direct graph transformation step in the single pushout (SPO) approach to algebraic graph transformation. The transformation is performed *in place*, i.e. the host graph is modified according to the rule's instructions.

Constructors

- **Step**

```
public Step(Morphism rule,
            Morphism match)
```

Note: Step() is deprecated. *use class method* `execute()` *instead.*

Perform an inplace graph transformation step.

Methods

- **execute**

```
public final static void execute(Match match)
```

Perform an inplace graph transformation step: apply the rule given by `match.getRule()` via `match` on the host graph given by `match.getImage()`. The host graph is modified to represent the result of the rule application.

See Also: getImage (cf. Section A.5.9)

- **execute**

```
public final static void execute(Morphism rule,
                                 Morphism match)
```

Note: execute() is deprecated. *use* `execute(Match)` *instead.*

Perform an inplace graph transformation step: apply `rule` via `match` on the host graph given by `match.getImage()`. The host graph is modified to represent the result of the application.

See Also: getImage (cf. Section A.5.9)

Literaturverzeichnis

[AEH$^+$96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr und G. Taentzer. *Graph Transformation for Specification and Programming.* Technischer Bericht 7/96, University of Bremen, 1996.

[AHS90] J. Adamek, H. Herrlich und G. Strecker. *Abstract and Concrete Categories.* Series in Pure and Applied Mathematics. John Wiley and Sons, 1990.

[AM75] M. Arbib und E. Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, New York, 1975.

[AR89] R. Arlt und M. Röder. *Grundlegende Datenstrukturen und Algorithmen zur Implementierung von algebraischen Graph Grammatiken.* Studienarbeit, Fachbereich 20, Technische Universität Berlin, 1989.

[Arl90] R. Arlt. *Effiziente Konstruktion von Regelansätzen für algebraische Graphersetzung: Konzeption und Implementierung.* Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1990.

[Bey91] M. Beyer. *GAG: Ein graphischer Editor für algebraische Graphgrammatiksysteme.* Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1991.

[BGT91] H. Bunke, T. Glauser und T.-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In: *Proc. 4th Intl. Workshop on Graph-Grammars and their Application to Computer Science and Biology*, Band 532 der *Lecture Notes in Computer Science*, Seiten 174–189. Springer-Verlag, 1991.

[Bie97] C. Bierans. *Ansatzsuche in Graphersetzungssystemen.* Diplomarbeit, Universität Hildesheim, Institut für Informatik, 1997.

[Boo91] G. Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Bru81] M. Bruynooghe. Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[CGHB+95] M. Conrad, M. Gajewsky, R. Holl-Biniasz, M. Rudolf, J. Demuth, S. Weber, R. Heckel, J. Müller, G. Taentzer und A. Wagner. *Graphische Spezifikation ausgewählter Teile von AGG - einem algebraischen Graphgrammatiksystem.* Technischer Bericht 95-7, TU-Berlin, 1995.

[CL95] I. Classen und M. Löwe. Scheme evolution in object oriented models: A graph transformation approach. In: *Proc. Workshop on Formal Methods at the ISCE'95, Seattle (U.S.A.)*, 1995.

[Dec94] R. Dechter. *Backtracking algorithms for constraint satisfaction problems - a survey.* Technischer Bericht, University of California, Irvine, September 1994. `ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/backtracking.ps`.

[Dör95] H. Dörr. *Efficient graph rewriting and its implementation*, Band 922 der *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

[DvB97] R. Dechter und P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997. `ftp://ftp.cs.ualberta.ca/pub/vanbeek/papers/tcs97.ps`.

[EB94] H. Ehrig und R. Bardohl. Specification Techniques using Dynamic Abstract Data Types and Application to Shipping Software. In: *Proc. of the International Workshop on Advanced Software Technology*, Seiten 70–85, 1994.

[EC96] S. Erdmann und I. Classen. Object-oriented Design of a Class Library for a Metamodel based on Algebraic Graph Theory. In: *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science. Springer-Verlag, 1996. `http://cis.cs.tu-berlin.de/~serdmann/alpha/download/alpha.ps`.

[EE96] H. Ehrig und G. Engels. Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems. In: J. Cuny, H. Ehrig, G. Engels und G. Rozenberg (Hrsg.), *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, Seiten 137 – 154. 5th International Workshop Williamsburg, VA, USA, November 1994, Springer LNCS 1073, 1996.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In: V. Claus, H. Ehrig und G. Rozenberg (Hrsg.), *1st Graph Grammar Workshop, Lecture Notes in Computer Science 73*, Seiten 1–69. Springer-Verlag, 1979.

[EM85] H. Ehrig und B. Mahr. *Fundamentals of algebraic specifications 1: Equations and initial semantics*, Band 6 der *EACTS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1985.

[EPS73] H. Ehrig, M. Pfender und H. Schneider. Graph grammars: an algebraic approach. In: *14th Annual IEEE Symposium on Switching and Automata Theory*, Seiten 167–180. IEEE, 1973.

[Erd95] S. Erdmann. *Konzeption und Realisierung einer Basisklassenbibliothek für Modellierungs- und Restrukturierungswerkzeuge.* Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1995.

[ES95] G. Engels und A. Schürr. Hierarchical Graphs, Graph Types and Meta Types. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html .

[Fre82] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 21(11):958–965, 1982.

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

[GMP+96] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith und T. Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In: E. C. Freuder (Hrsg.), *Proc. 2nd Intl. Conf. on Principles and Practice of Constraint Programming*, Band 1118 der *Lecture Notes in Computer Science*, Seiten 179–193. Springer-Verlag, 1996. http://www.cs.strath.ac.uk/~apes/papers/CRCcp96gmpsw.ps.gz.

[HHT96] A. Habel, R. Heckel und G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[HW95] R. Heckel und A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[Joh92] H. Johnen. *Ansatzsuche in Hypergraph-Grammatiken*. Diplomarbeit, Universität Bremen, 1992.

[Kon94] G. Kondrak. *A Theoretical Evaluation of Selected Backtracking Algorithms*. Technischer Bericht TR94-10, University of Alberta, 1994. ftp://ftp.cs.ualberta.ca/pub/TechReports/TR94-10.ps.Z.

[Kum92] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992. http://www.cirl.uoregon.edu/constraints/archive/kumar.ps.

[LB93] M. Löwe und M. Beyer. AGG — An Implementation of Algebraic Graph Rewriting. In: *Proc. Fifth Int. Conf. Rewriting Techniques and Applications, '93, LNCS 690*, Seiten 451–456. Springer-Verlag, 1993.

[LKW93] M. Löwe, M. Korff und A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In: M. Sleep, M. Plasmeijer und M. van Eekelen (Hrsg.), *Term Graph Rewriting: Theory and Practice*, Kapitel 14, Seiten 185–199. John Wiley & Sons Ltd, 1993.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[Mel97] B. Melamed. *Grundkonzeption und -implementierung einer Attributkomponente für ein Graphtransformationssystem.* Studienarbeit, Technische Universität Berlin, Fachbereich Informatik, 1997.

[MH86] R. Mohr und T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Plu95] D. Plump. On Termination of Graph Rewriting. In: *Proc. 21st Workshop on Graph-Theoretic Concepts in Computer Science (WG '95).* Lecture Notes in Computer Science, 1995. to appear.

[PR69] J. L. Pfaltz und A. Rosenfeld. Web Grammars. *Int. Joint Conference on Artificial Intelligence*, Seiten 609–619, 1969.

[Pro93] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Pur83] P. W. Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21:117–133, 1983.

[Rat97] Rational Software Corporation, http://www.rational.com/uml. *Unified Modeling Language Notation Guide*, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Eddy und W. Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[Rib96] L. Ribeiro. *A Telephone System's Specification using Graph Grammars.* Technischer Bericht 96-23, Technical University of Berlin, 1996.

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

[RPD89] F. Rossi, C. Petrie und V. Dhar. *On the Equivalence of Constraint Satisfaction Problems.* Technischer Bericht ACT-AI-222-89, MCC, Austin, Texas, 1989.

[Rud96] M. Rudolf. *Konzeption und Implementierung grundlegender Aspekte der Ableitungskomponente eines Graphtransformationssystems.* Studienarbeit, Technische Universität Berlin, Fachbereich Informatik, 1996.

[Sch97] A. Schürr. Programmed Graph Replacement Systems. In: G. Rozenberg (Hrsg.), *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[Smi97] B. M. Smith. Succeed-first or Fail-first: A Case Study in Variable and Value Ordering Heuristics. In: G. Smolka (Hrsg.), *Proc. 3rd Intl. Conf. on Principles and Practice of Constraint Programming*, Band 1330 der *Lecture Notes in Computer Science*, Seiten 321–330. Springer-Verlag, 1997. `file://www.scs.leeds.ac.uk/scs/doc/reports/1996/96_26.ps.Z`.

[Sun97] Sun Microsystems, http://java.sun.com:80/products/jdk/. *The Java Development Kit (JDK)*, 1997.

[Tae96] G. TAENTZER. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* Dissertation, TU Berlin, 1996. Shaker Verlag.

[TK97] G. TAENTZER und M. KOCH. Distributing Attributed Graph Transformation. In: *First European Workshop on "General Theory of Graph Transformation Systems", Bordeaux, Oct. 8-10,1997*, 1997.

[Wal91] R. F. C. WALTERS. *Categories and Computer Science.* Cambridge University Press, 1991.

[Win95] B. WINDOLPH. *Vom Grapheditor AGG zu einem Graph-Grammatik-System.* Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1995.

[Wol96] D. WOLZ. Tool Design for Structuring Mechanisms for Algebraic Specification Languages with Initial Semantics. In: *Recent Trends in Data Type Specifications, Proc. 11th Intl. Workshop on the Specification of Abstract Data Types*, Band 1130 der *Lecture Notes in Computer Science*, Seiten 536–550. Springer-Verlag, 1996.

[Wol97] D. WOLZ. *Colimit Library for Graph Transformations and Algebraic Development Techniques.* Dissertation, Technische Universität Berlin, Fachbereich Informatik, 1997. Eingereicht im Dezember 1997.

[Zün95] A. ZÜNDORF. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme.* Dissertation, RWTH Aachen, 1995.

[Zün96] A. ZÜNDORF. Graph Pattern Matching in PROGRES. In: *Proc. 5th Intl. Workshop on Graph Grammars and their Application to Computer Science*, Band 1073 der *Lecture Notes in Computer Science*, Seiten 454–468. Springer-Verlag, 1996.