# Konzeption und Implementierung eines generischen Grammatikeditors für visuelle Sprachen

Magnus Niemann

Diplomarbeit

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

März 1999

## Zusammenfassung

In dieser Arbeit wird die Entwicklung eines Grammatikeditors für visuelle Sprachen beschrieben. Ausgehend von einem Alphabet mit grafischen Symbolen wird eine einfache Grammatik aufgebaut, mit deren Regeln man weitere komplexe Grammatikregeln definieren kann. Auf diesen Regeln basierend läßt sich dann ein syntaxgesteuerter Editor für Diagramme in der definierten visuellen Sprache erzeugen.

Die logische Grundlage für die Anwendung und den Aufbau der Grammatikregeln bietet die Graphtransformation. Jedes zu erstellende Diagramm hat eine zugrundeliegende Graphstruktur, die durch grafische Informationen zum Layout und der Beziehung von Layoutobjekten (Constraints) untereinander erweitert wird.

Als Werkzeug für die Graphtransformation wird das ebenfalls an der TU Berlin entwickelte *AGG*-System verwendet, für das Lösen der grafischen Constraints benutzen wir den an der Uni Paderborn entwickelten Constraint-Solver *Parcon*.

## **Dank geht an** …

… *Roswitha Bardohl* für stählerne Nerven und ein weites Herz,

… *Manuel Schwarze* für die unerreichbare Perfektion (Co*** Dr** lässt grüßen),

… *Gabriele Taentzer und Thorsten Schultzke* für einige Erleuchtungen im Bezug auf Statecharts und Graphgrammatiken,

… *Michael Rudolf* für die schönste Graphtransformationmaschine, die ich kenne,

… alle Kollegen bei *TFS* für das tolle Klima,

… meine Frau *Uta Niemann* für ihre Engelsgeduld und Unterstützung und

… meinen Sohn (in preparation ☺) für zeitweiliges Geraderücken der Prioritäten – es ist nicht einfach, gleichzeitig Vater und Diplom-Informatiker zu werden.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

Ein Bild sagt mehr als 1000 Worte. *Volksmund*

Das oben genannte Zitat erscheint uns schon ziemlich abgegriffen. Aber ist es nicht so, dass Menschen seit Jahrtausenden eine Zeichnung, und mag sie auch noch so oberflächlich sein, einer sprachlichen Beschreibung vorziehen? Was zeichnet nun Bilder aus, die von vielen Menschen verstanden werden?

Menschen interpretieren die Welt größtenteils in Form von Ähnlichkeiten und Unterschieden, die zwischen Objekten aufgedeckt werden. Auf ähnliche Objekte wird reagiert, als ob sie gleich seien, basierend auf Assoziationen.

Diese Tatsache lässt sich mit der entwicklungspsychologischen Theorie J. Bruners erklären. Demnach repräsentieren Kinder im Laufe ihrer Entwicklung Dinge oder Geschehnisse zunächst *enaktiv* („im Muskel verkörpert"). anschließend auch ikonisch und zuletzt werden Objekte auch symbolisch, beispielsweise als Buchstaben, repräsentiert. Ein Ikon, also ein Bild, hat eine tatsächliche Ähnlichkeit zu dem von ihm bezeichneten Objekt, Symbole dagegen sind abstrakter [Lef94].

Ein Bild mit einem genügend hohen Abstraktionsgrad nennen wir „Zeichnung". Eine Zeichnung dient dazu, Objekte, Beziehungen zwischen Objekten und Sachverhalte darzustellen. So enthält eine Bauzeichnung Informationen über Bauelemente, deren Maße und Position zueinander, ein Stadtplan beispielsweise (siehe Abb. 1.1) beinhaltet Straßen, Gebäude und andere landschaftliche Merkmale und stellt diese in Beziehung. Jede Zeichnung kann einen unterschiedlichen Abstraktionsgrad besitzen, so ist ein Luftbild offensichtlich sehr viel näher an der Realität – jedoch auch unübersichtlicher – als eine Landkarte. Ein Schaltplan für eine elektrische Schaltung besitzt einen noch viel höheren Abstraktionsgrad (siehe Abb. 1.2), wenn man ihn mit der Platine vergleicht, die mit denselben Elementen bestückt ist. Hier muss nicht einmal die Positionierung der Elemente der Wirklichkeit entsprechen, das einzige, was dargestellt werden soll, sind die Verbindungen der Elemente untereinander.


Abbildung 1.1: Stadtplan

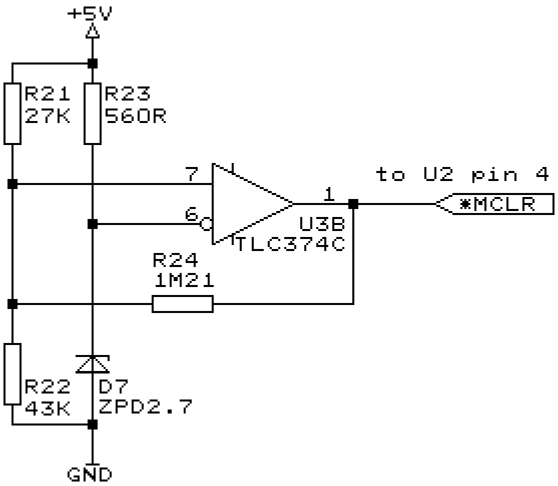



Abbildung 1.2: Schaltplan

Zeichnungen mit einem hohen Abstraktionsgrad, denen wir gerade in der Informatik, aber auch in anderen Zusammenhängen wie der Elektrotechnik oder Mathematik begegnen, sind *Diagramme*. Das Wort *Diagramm* kommt aus dem Griechischen und bedeutet „Umriss“ oder „Figur“. Den ursprünglichen Begriff des mathematischen Diagramms, der uns häufig im Zusammenhang mit Kurven-, Balken- oder Kreisdiagrammen begegnet, wollen wir erweitern. Mit *Diagramm* bezeichnen wir in dieser Arbeit jede Form eines Bildes oder einer Zeichnung, der eine bestimmte formale grafische Struktur zugrunde liegt.

In der Informatik wurden schon vor langer Zeit grafische Diagrammarten verwendet, vor allem um Programmabläufe zu beschreiben. So gibt es Flussdiagramme, die Verzwei-

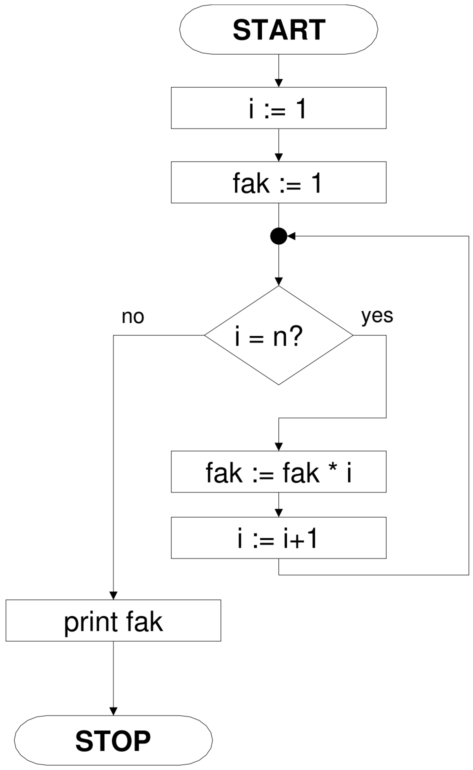



Abbildung 1.3: Flussdiagramm

gungen und Sprünge in einem Programm sichtbar machen (Abb. 1.3), eine verwandte Diagrammart sind Nassi-Shneiderman-Diagramme, die eine andere Art der Darstellung verwenden (Abb. 1.4). Zu Zeiten, in denen die Ausgabemöglichkeiten der Rechner meistens auf pures ASCII beschränkt waren und Zeichnungen nur mit einem Plotter gemalt werden konnten, dem man die Befehle mühsam einprogrammieren musste, waren Papier und Bleistift dann auch die bevorzugten Hilfsmittel, um ein Programm grafisch zu strukturieren.

Die Weiterentwicklung von Diagrammsprachen zur Modellierung von Programmen und ganzen Systemen ging Hand in Hand mit der Verbesserung der Systeme. Mit dem Aufkommen der grafischen Benutzungsoberflächen (GUI = Graphical User Interface) war es immer leichter möglich, auch das Design und die Modellierung der Software auf den Computer selbst zu verlagern. Gleichzeitig war die Informatik in der Lage, anderen Disziplinen eine Möglichkeit zur grafischen Modellierung und Beschreibung von Problemstellungen zu bieten. Besonders stark wird das deutlich an CAD (Computer Aided Design), das das Berufsbild des Technischen Zeichners ähnlich stark beeinflusst hat wie die Entwicklung des Lichtsatzes die Arbeit des Schriftsetzers. Auch die Elektrotechnik hat von der Entwicklung grafischer Tools zum Platinen- und Mikrochipdesign profitiert.

Hand in Hand mit den besseren grafischen Fähigkeiten der Hard- und Softwaresysteme ging aber auch, dass die Produkte und Probleme immer komplexer wurden, so dass die Modellierungsphase beispielsweise von Software immer wichtiger wurde. Gerade in den letzten Jahren hat durch das Aufkommen von objektorientierten Techniken

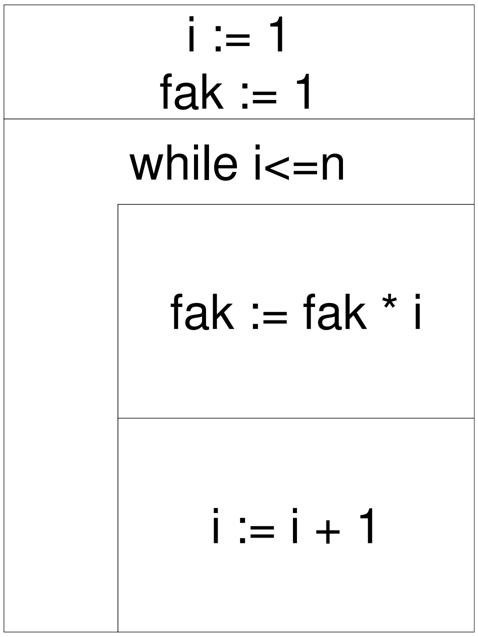



Abbildung 1.4: Nassi-Shneiderman-Diagramm

und Verteiltheit der zu entwickelnden Systeme die Menge der grafischen Diagrammsprachen, mit denen man diese Modellierungen beschreiben kann, geradezu inflationäre Ausmaße angenommen, so dass man heute z.B., wenn man ein System objektorientiert modellieren will, unter Dutzenden von mehr oder weniger formal beschriebenen Sprachen und noch viel mehr Entwicklungstools für verschiedenste Betriebssysteme auswählen kann.

Was verbindet nun alle diese verschiedenen Diagrammsprachen? Ihnen liegt eine Art Syntax zugrunde. Schon die Bezeichnung „Sprache“ impliziert ja, dass es sich auch bei grafischen Sprachen um solche handelt, die – wie textbasierte Sprachen – bestimmte einfache Regeln besitzen, nach denen man diese Sprache aufbauen kann. Ein Diagramm, das sich nicht an diese Regeln hält, ist dann „falsch“ im Sinne der Sprachregeln. Will man nun ein Programm schreiben, mit dem man Diagramme in einer bestimmten Sprache, beispielsweise Flussdiagramme, erstellen kann[1], muss man die Struktur der Sprache im Programmcode festlegen. Wir müssen beispielsweise definieren, was passiert, wenn der Benutzer[2] ein Element der Diagrammsprache einfügen oder löschen will, in welchem Kontext das geschehen darf und wann gewisse Editieroperationen erlaubt sind. Wenn wir nun einen Editor für eine weitere Diagrammsprache programmieren wollen, können wir den alten wegwerfen oder müssen ihn zumindest in großen Teilen neu schreiben.

Hätten wir die Möglichkeit, die Regeln, nach denen Diagramme in einer Diagrammsprache aufgebaut werden, in einer Art „visuellen Sprache“ zu definieren, müsste es ei-

[1]Ein solches Programm bezeichnen wir als *Editor*, analog zu den Programmen, mit denen man in Text- oder Programmiersprachen arbeiten kann.

[2]Anm. d. A.: Aus Gründen der Vereinfachung wähle ich in der Arbeit die männliche Form, was nicht bedeutet, dass sich Frauen weniger angesprochen fühlen sollten.

gentlich möglich sein, einen Editor mitsamt der Editieroperationen (Einfügen, Löschen, Verschieben, Skalieren etc.) aus der visuellen Sprache zu erzeugen. Und genau an diesem Punkt setzt unser System an. Wir wollen mit Hilfe von mehreren bekannten und wohldefinierten Formalismen aus der theoretischen Informatik (Graphgrammatiken, Algebra und Constraints) ein Art visuelle Metasprache angeben, in der sich eine große Menge der bekannten visuellen Sprachen beschreiben lassen und ein System bauen, dass es möglich macht, eine spezielle visuelle Sprache zu bauen, zu bearbeiten und einen Editor für diese Sprache zu erzeugen. Da es sich um visuelle Sprachen handelt, wollen wir die Definition der Sprache auch – wo immer möglich – auf grafische Weise erlauben.

## 1.2 Struktur der Arbeit

Diese Arbeit soll zunächst eine informelle Einführung in die formalen Konzepte liefern. Zunächst wollen wir den Begriff der *visuellen Sprache* genauer definieren. Dann werden Graphen und Graphtransformationen eingeführt und – im Hinblick auf die Verwendung mit Datentypen – attributierte Graphen als Erweiterung der herkömmlichen Graphtransformation vorgestellt.

Als nächstes werden wir anhand einer Fallstudie (Statecharts) die Konzepte unserer Metasprache genauer anschauen, wobei wir die zwei wichtigsten Konzepte (Graphtransformation und Constraints) verbinden.

Es folgt ein Überblick über das Design und die zum Design verwendeten Tools. Dabei werden wir einerseits kurz die Programme vorstellen, mit denen gearbeitet wurde, andererseits werden wir auf visuelle Sprachen eingehen, die wir zum Design des Systems verwendet haben. Dabei sollen objektorientierte Sprachen und Werkzeuge im Vordergrund stehen.

Nun werden die externen Komponenten beschrieben, auf denen der Grammatikeditor aufbaut. Dazu gehören das Graphtransformationssystem *AGG*, der Constraintsolver *Parcon* und der Alphabeteditor.

Als nächstes werden wir die Implementierung genauer beschreiben. Dafür gehen wir auf die Werkzeuge ein, die wir zur Implementierung verwenden und erläutern dann die Struktur des Systems. Zur Struktur gehört einerseits die grafische Benutzeroberfläche, andererseits die zugrundeliegende Architektur mit den Datenstrukturen, die die visuelle Sprache beschreiben. Außerdem wird die Anbindung des Systems an die Graphtransformationsmaschine und den Constraint-Solver beschrieben. Nicht zuletzt wird auch die Schnittstelle zum Alphabet-Editor erklärt, die es möglich macht, grafische Symbole zu importieren.

Es folgt ein eigenständiger Abschnitt der Diplomarbeit: das Benutzerhandbuch. Es ist als Kapitel der Arbeit gehalten, da es u. a. auch dazu dient, die Funktionalität des Grammatik- und des Diagrammeditors zu beschreiben. Hier wird – an Hand von einfachen Beispielen aus der Fallstudie – der Aufbau und die Arbeit mit dem Programm erklärt. Der Benutzer sollte nach der Lektüre des Handbuchs in der Lage sein, eine eigene visuelle Sprache zu definieren.

Das letzte Kapitel bietet noch einmal eine Zusammenfassung der Arbeit und einen

Ausblick auf Erweiterungen.

Als zweiten Teil und Anhang der Arbeit legen wir die API-Dokumentation der implementierten Java-Klassen vor. Die Klassendokumentation ist eine nach LaTeX konvertierte Version der `javadoc`-Dokumentation. Sie ist vor allem für diejenigen interessant, die das System erweitern oder Teile davon in eigenen Arbeiten verwenden wollen.

# Kapitel 2

# Visuelle Sprachen und Graphtransformation

In diesem Kapitel wird zunächst eine informelle Einführung in visuelle Sprachen gegeben und dann auf die zwei Grundpfeiler unserer Sprachkonzeption eingegangen: Graphtransformation und Constraints.

## 2.1 Von textuellen zu visuellen Sprachen

Wie in der Einleitung beschrieben, liegt jeder Diagrammsprache eine Syntax oder eine Menge von Sprachregeln zugrunde, die es zu formalisieren gilt. Schon die Verwendung des Begriffs „Sprache“ legt nahe, dass wir starke Verbindungen zu textuellen, formalen Sprachen ziehen können. Deren Definition und Beschreibung gehört in der Informatik ja schon zum Standardkanon am Beginn des Studiums. Wir wollen uns also zunächst formale, textuelle Sprachen und deren Konzepte ansehen und dann überblicksweise einige Formalisierungsansätze für visuelle Sprachen betrachten.

### 2.1.1 Textuelle Sprachen

Textuelle Sprachen bestehen aus einem *Alphabet*, also einer nichtleeren endlichen Menge von *Buchstaben* oder auch *Symbolen*. Indem man diese Alphabetsymbole aneinanderhängt, kann man *Wörter* erzeugen. Ein Teilmenge aller Wörter, die sich aus einem Alphabet durch Aneinanderreihung von Wörtern erzeugen lässt, nennen wir *Sprache*.

Natürlich ist diese Beschreibung etwas oberflächlich, da ja den meisten Sprachen wie z.B. Programmiersprachen gewisse *Regeln* zugrundeliegen. Analog zur Beschreibung von natürlichen Sprachen wie Englisch, Deutsch oder Kisuaheli nennen wir diese Regeln *Grammatik*. Eine Grammatik besteht üblicherweise aus einem Alphabet, einer Menge von *Nonterminalsymbolen*[1] und einer Menge von Produktionsregeln mit einer linken und einer rechten Seite, die die Symbole links durch die Symbole rechts ersetzt.

[1]Diese Symbole sind solche, die im Laufe der Regelanwendung durch andere ersetzt werden können.

Ein Wort ist in einer Sprache, wenn es aus einem Startsymbol durch wiederholte Anwendung der Regeln erzeugt werden kann. Im Laufe der Anwendung werden alle Nonterminalsymbole durch Alphabetsymbole ersetzt.

Startsymbol ist $A$, Nonterminalsymbole sind $A$ und $B$, die erzeugte Sprache ist $a^n b^m, n, m \in \mathbb{N}$:
$A \rightarrow aA$
$A \rightarrow aB$
$B \rightarrow Bb$
$B \rightarrow b$

Abbildung 2.1: Eine textuelle Grammatik

Diese Grammatiken haben nun verschiedene Eigenschaften, von denen einige in der Chomsky-Hierarchie [Sal73] gefasst werden. Die allgemeinste Form sind Typ-0-Grammatiken, die alle Grammatiken umfassen. Die Typ-1-Grammatiken oder *kontextsensitiven* Grammatiken sind solche, bei denen es vom Kontext auf der linken Seite abhängt, ob eine Regel anwendbar ist. Der nächste Typ sind die Typ-2- oder *kontextfreien* Grammatiken, bei denen auf der linken Seite nur ein Nonterminalsymbol auftaucht. Die größte Einschränkung stellen Typ-3- bzw. *reguläre* Grammatiken dar, deren Regeln alle entweder die Form $A \rightarrow aB$, $A \rightarrow a$ oder $A \rightarrow \epsilon$ (leeres Symbol) haben.

Diese Hierarchie ist vor allem dann interessant, wenn man prüfen will, ob ein gegebenes Wort von einer Grammatik erzeugt werden kann, es sich also um ein „richtiges“ Wort handelt. Ebenso gibt es Beweisverfahren, die zeigen, dass eine gegebene Sprache **nicht** von einer Grammatik erzeugt werden kann (die „Pumping Lemmata“).

Eine Erweiterung der Wortsprachen sind *Termsprachen*. Hierbei werden statt Alphabetsymbolen *Terme* über einer Signatur verwendet. Diese Sprachen besitzen schon sehr viel mehr Struktur als reine Wortsprachen und wir können sie als Zwischenschritt auf dem Weg zu visuellen Sprachen betrachten.

### 2.1.2 Visuelle Sprachen

Wenn wir gegebene Wortsprachen, wie z.B. Programmiersprachen, mit den in Abschnitt 1.1 vorgestellten Beispielen für visuelle Diagrammsprachen vergleichen, fällt uns zunächst ein großer Unterschied auf: Wortsprachen sind **eindimensional**, d.h. die einzige räumliche Relation zwischen zwei Symbolen ist diejenige, dass ein Symbol einem zweiten *folgt*. Diagramme hingegen enthalten viel mehr Beziehungen zwischen den Symbolen, und gerade diese grafischen[2] Beziehungen machen den Großteil einer visuellen Sprache aus. Alle diese Beziehungen innerhalb eines Diagramms bezeichnen wir als *Layout* des Diagramms.

Es gibt im Bereich der visuellen Sprachen nun grob gesagt zwei Ansätze: Der erste Ansatz liefert eine *textuelle* Definition der räumlichen Beziehungen grafischer Symbole

[2] Im Gegensatz zum Englischen, wo es nur *graphical* gibt, haben wir in der deutschen Sprache die Möglichkeit, zwischen *grafisch*, also zeichnerisch und *graphisch*, also graphbezogen zu unterscheiden.

untereinander. Zu diesem Ansatz lassen sich Positional-Grammatiken [MM97], Picture-Layout-Grammatiken [Gol91a, Gol91b], Constraint-Multiset-Grammatiken [Mar94] oder relationale Grammatiken zählen. Der zweite Ansatz bietet eine *visuelle* Definition der Sprache. Dazu gehören Hypergraphgrammatiken und attributierte Graphgrammatiken. Um sich über die aktuellen Entwicklungen auf dem immer größer werdenden Gebiet der visuellen Sprachen zu informieren, lese man [VL95, VL96, VL97].

Ein Diagramm besteht hauptsächlich aus Symbolen (Kreis, Rechteck, Text, Bitmap etc.), räumlichen Relationen (neben, über, angrenzend an, etc.) und Attributen für die Symbole (Farbe, Maße, Linienbreite etc.). Alle diese Bestandteile müssen bei der Definition einer Grammatik für eine visuelle Sprache berücksichtigt werden, wobei einige Formalismen mehr Wert auf die grafischen Attribute legen, andere weniger. Vor allem visuelle Sprachen, die textuell definiert sind, lassen oft offen, in welcher Weise sich z.B. Größenänderungen von Symbolen auf die Sprache selbst auswirken.

Wir wollen in unserem System, wie schon in der Einleitung dargestellt, Möglichkeiten bieten, eine visuelle Sprache auch visuell zu definieren. Unsere visuellen Sprachen benötigen dazu Bestandteile:

- Ein Alphabet, d.h. eine Menge von grafischen Symbolen mit einer entsprechenden Beschreibung ihrer Attribute,
- eine Möglichkeit, das Layout eines Diagramms zu beschreiben,
- einen Rahmen für die Definition der Grammatikregeln.

Die ersten beiden Bestandteile werden im nächsten Abschnitt erklärt, in dem es um die Beschreibung von grafischen Layouts durch *Constraints* geht. Den Rahmen für die Definition und Anwendung von Regeln der Grammatik liefert die Graphtransformation, die in Abschnitt 2.3 eingeführt wird. Die Grammatikregeln sollen dabei nicht nur in der Lage sein, ein Diagramm in der visuellen Sprache aufzubauen, wir wollen – im Hinblick auf einen zu generierenden Editor – auch die Möglichkeit bieten, Symbole mit Hilfe von Regeln zu verändern und zu löschen.

## 2.2 Layout und Constraints

Zunächst wollen wir eine Möglichkeit vorstellen, Attribute von grafischen Symbolen aufzuschreiben. Dabei wollen wir uns auf die Attribute beschränken, die später in der Beziehung zu anderen Objekten eine Rolle spielen – Farbe und Linienstärke z.B. gehören nicht dazu. Ein Rechteck bekommt z.B. folgende Attribute (siehe Abb. 2.2):

- Einen Referenzpunkt `rp` mit den Koordinaten `rp.x` und `rp.y`, der die rechte obere Ecke des Symbols auf einer kartesischen Fläche bezeichnet,
- eine Breite `w` (engl. width) und
- eine Höhe `h` (engl. height).

Für andere Symbole sind andere Attribute denkbar, so könnte eine Linie statt Breite und Höhe einen Anfangspunkt `p1` und einen Endpunkt `p2` in kartesischen Koordinaten erhalten.

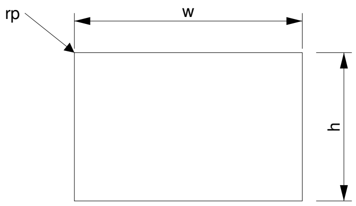



Abbildung 2.2: Rechteck mit Attributen

Die Beziehung mehrerer grafischer Symbole untereinander auf einer kartesischen Fläche lässt sich jetzt durch *Constraints* beschreiben. Constraint bedeutet im Deutschen soviel wie „Zwang“. Das impliziert, dass Symbole zueinander in Relation gestellt werden. Da sie sich nun nicht mehr irgendwo auf der Fläche befinden dürfen, sind sie „gezwungen“, sich einander anzupassen.

Ein solcher Constraint könnte z.B. heißen: „Das Rechteck `rectangle1` befindet sich oberhalb des Rechtecks `rectangle2`“. Wie kann man diese Einschränkung jetzt genau spezifizieren? Das geschieht über Relationen zwischen den Attributen der Symbole. Meistens werden diese Relationen über Gleichungen und Ungleichungen beschrieben. Der eben genannte Constraint ließe sich mit Hilfe der Rechteck-Attribute wie folgt beschreiben[3] (siehe Abb. 2.3):

```
rectangle2.rp.y > rectangle1.rp.y + rectangle1.h
```

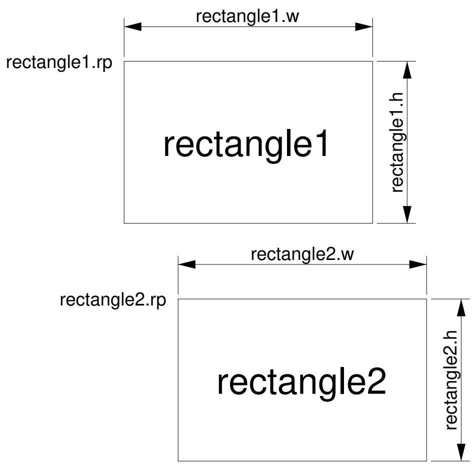



Abbildung 2.3: Der Constraint *above*

Im Klartext: „Die Y-Koordinate der linken oberen Ecke des zweiten Rechtecks soll größer sein als die Summe der Y-Koordinate des ersten Rechtecks und seiner Höhe.“ Bei

[3]Wir nehmen an, dass die Y-Achse im Koordinatensystem von oben nach unten verläuft, so dass ein größerer Y-Wert auch eine Positionierung weiter unten auf der Fläche bedeutet.

geeigneter Festlegung der Attribute eines Symbols lassen sich mit diesen Beschreibungen beliebig komplexe grafische Relationen definieren. Es sind im Übrigen auch Constraints zwischen mehr als einem Symbol denkbar, ebenso lassen sich auch Constraints definieren, die die Attribute genau eines Symbols festlegen, z.B. die Mindestbreite oder -höhe eines Rechtecks.

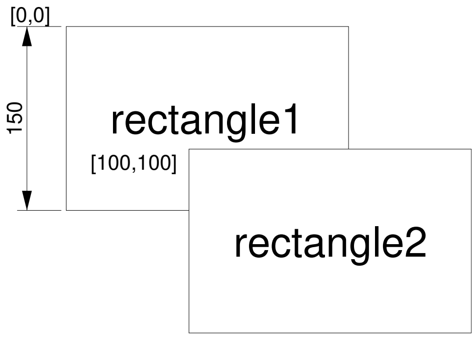



Abbildung 2.4: *above* ist nicht erfüllt

Ein Diagramm enthält jetzt einerseits eine Menge von grafischen Symbolen, andererseits eine Menge von grafischen Constraints, diese definieren zusammen das *Layout* des Diagramms. Ein Layout erfüllt eine Constraintmenge, wenn die (Un-)Gleichungen in den Constraints für die Attribute aller Symbole im Layout erfüllt sind. In Abb. 2.4 ist der Constraint *above* nicht erfüllt (oder auch „verletzt"), da die Ungleichung für die angegebenen Werte nicht erfüllt ist.

Wenn wir nun ein gültiges Layout vorzuliegen haben und ein Objekt beispielsweise so verschieben oder skalieren, dass das Layout ungültig wird, also einige Constraints verletzt sind, haben wir ein Problem, ein sogenanntes *Constraint Satisfaction Problem* (CSP). Unser Ziel ist es, durch geeignete Änderungen an den Attributwerten das Layout eines Diagramms so zu ändern, dass alle Constraints wieder erfüllt sind, also das CSP zu lösen[4]. Bei der Lösung eines CSP werden die Attributwerte so angepasst, dass das Layout mit den geänderten Werten alle Constraints erfüllt. Zum Lösen eines CSP gibt es verschiedene Strategien und Algorithmen, mehr darüber findet sich in [Gri96] und [PSG95].

Ebenso wird das Layout verändert, wenn Regeln der Grammatik Symbole hinzufügen oder löschen, denn das Hinzufügen von Symbolen ist fast immer mit dem Einfügen neuer Constraints zwischen den neuen Symbolen und einem alten Kontext verbunden. Auch nach dem Anwenden einer Regel muss daher das Layout überprüft und ein Constraintsolving durchgeführt werden.

## 2.3 Attributierte Graphtransformation

Wir werden im Folgenden einen Rahmen vorstellen, den wir dazu verwenden, einerseits die einem Diagramm zugrunde liegende logische Struktur zu beschreiben und an-

[4] Im weiteren Text wird in diesem Fall meist in Kurzform vom „Lösen der Constraints" gesprochen, gemeint ist aber, dass das CSP gelöst wird.

dererseits die Anwendung von visuellen Grammatikregeln zur Erstellung und Änderung von Diagrammen über die Anwendung von Regeln auf der logischen Struktur eines Diagramms zu definieren. Eine genaue und formalisierte Beschreibung von Graphen und Graphtransformation, wie wir sie in unserem System verwenden, findet sich in [CL95, Rud97].

### 2.3.1 Graphen und Graphmorphismen

Die logische Struktur eines Diagramms kann man mit Hilfe von Graphen beschreiben. Ein *Graph* ist eine mathematische Struktur, die aus einer Menge von Knoten $V$ (engl. vertex), einer Menge von Kanten $E$, zwei Abbildungen $s : E \longrightarrow V, t : E \longrightarrow V$, einer Labelmenge $L$ und einer totalen Abbildung $\ell : V \cup E \longrightarrow L$. Die Abbildungen $s$ und $t$ („source" und „target") ordnen jeder Kante ihren Ausgangs- bzw. Zielknoten zu und die Abbildung $\ell$ markiert Knoten und Kanten in geeigneter Weise.

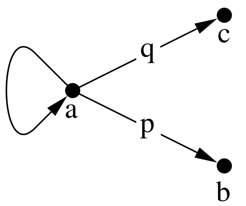



Abbildung 2.5: Ein einfacher Graph

Beziehungen zwischen Graphen werden durch *Morphismen* ausgedrückt. Ein Morphismus ist eine Abbildung von einem Graphen in einen zweiten, wobei Knoten auf Knoten, Kanten auf Kanten abgebildet werden und durch eine Homomorphiebedingung festgelegt ist, dass einerseits Labels erhalten bleiben und andererseits die Struktur des Graphen erhalten bleibt, wenn also eine Kante abgebildet wird, so muss deren Quelle und Ziel erhalten bleiben (siehe Abb. 2.6).

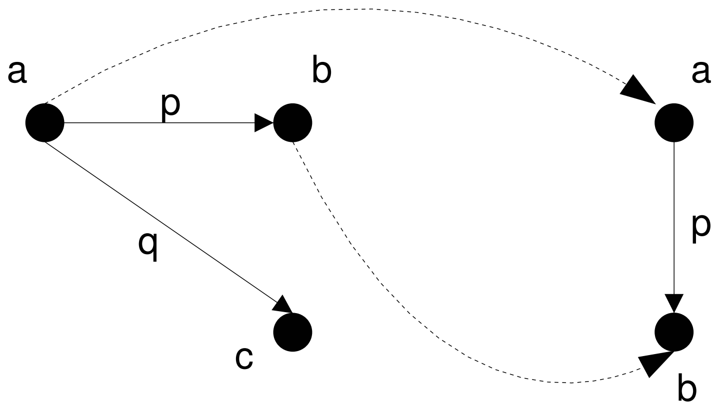



Abbildung 2.6: Ein Graphmorphismus

Mit diesen einfachen Graphen lässt sich die logische Struktur eines Diagramms schon recht gut beschreiben, ein Beispiel ist in Abb. 2.7 für das Flussdiagramm aus der Einleitung gegeben.

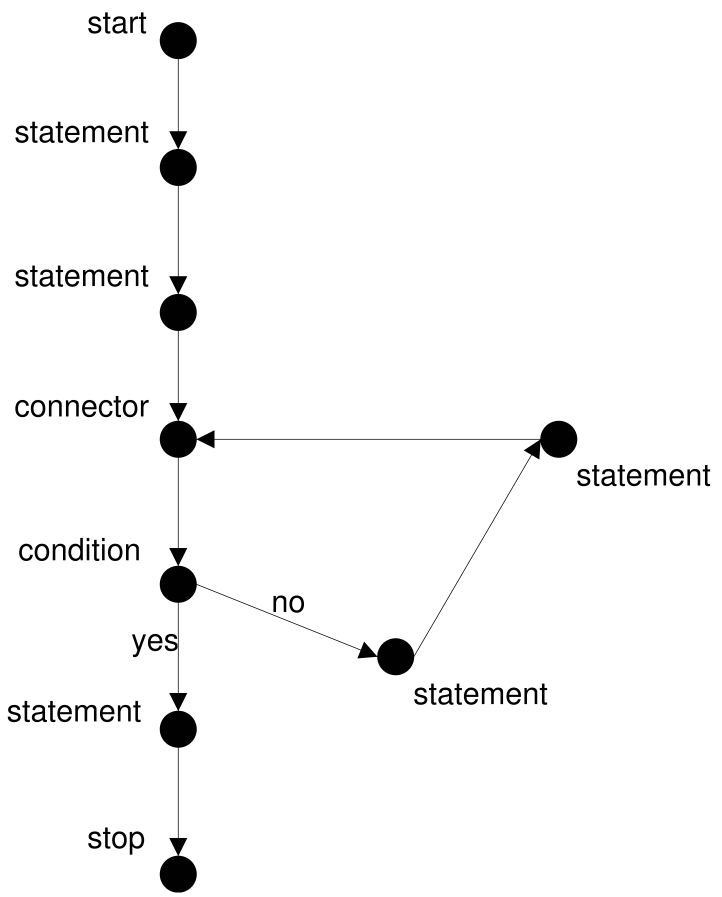



Abbildung 2.7: Die logische Struktur eines Flussdiagramms

### 2.3.2 Attributierte Graphen

Bis jetzt werden Knoten und Kanten in einem Graphen nur mit Labels bezeichnet. Wir wollen eine Erweiterung vorstellen, die die Knoten und Kanten eines Graphen mit Attributen über einer geeigneten Algebra zu einer Signatur versieht (zur Definition der Begriffe Signatur und Algebra siehe [EMC$^+$99]). Die Signatur gibt die Syntax für eine Reihe von Algebren vor, diese Algebren „implementieren" die Syntax. Diese Attributierung werden wir später dazu verwenden, nicht-grafische Attribute in einem Diagramm (wie z.B. Namen oder Nummern von Symbolen) in den Attributen eines Graphen festzuhalten.

Ein *attributierter Graph* ist ein Graph, wie oben beschrieben, mit zwei Erweiterungen:

- einer totalen Abbildung $at : L \longrightarrow \Sigma$ (für „attribute type"), die jedem Label eine

Sorte der Signatur $\Sigma$ zuordnet und

- einer totalen Abbildung $av : V \cup E \longrightarrow AV$ (für „attribute value"), wobei $AV = \biguplus_{s \in \Sigma} \mathcal{A}_s$ die disjunkte Vereinigung aller Trägermengen von $\mathcal{A}$ ist. Diese Abbildung ordnet jedem Knoten und jeder Kante genau einen Attributwert zu.

Die beiden neuen Abbildungen sind so definiert, dass jedem Knoten und jeder Kante mit einem bestimmten Label und einem Attributtyp auch nur ein Attributwert zu diesem Typ zugeordnet werden kann.

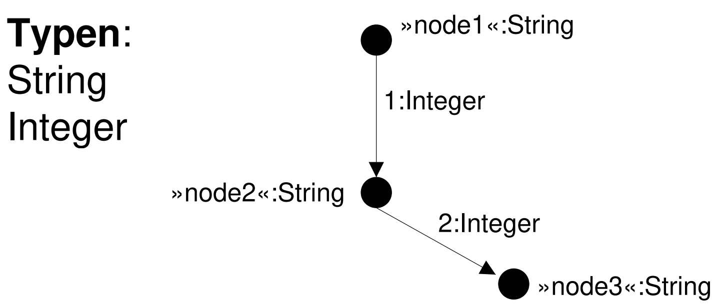



Abbildung 2.8: Ein attributierter Graph

Jedem Label wird in dieser Definition genau ein Attributtyp, also eine Sorte der Signatur zugeordnet. Den Knoten und Kanten eines bestimmten Typs werden jetzt Werte zugeordnet, die aus der Attributalgebra $\mathcal{A}$ stammen. Im Allgemeinen möchten wir einem Graphobjekt mehrere Attribute zuordnen, das geschieht dann dadurch, dass wir in der Trägermenge $\mathcal{A}_s$ mehrere Datenstrukturen in einem Tupel zusammenfassen. So könnte ein Attribut beispielsweise aus einem Tupel mit einem Wort (String) und einer Zahl (Integer) bestehen.

Als weiteres werden wir noch Morphismen für attributierte Graphen betrachten. Ein Morphismus zwischen zwei attributierten Graphen entspricht einem Morphismus zwischen zwei einfachen Graphen, erweitert durch einen totalen $\Sigma$-Homomorphismus zwischen zwei Graphen zur gleichen Signatur $\Sigma$. Der Homomorphismus ist dafür zuständig, bei der Abbildung die Attribut*werte* zu erhalten.

### 2.3.3 Graphtransformation

Wir haben nun das Handwerkszeug, um die logische Struktur eines Diagramms durch Graphen zu beschreiben und Abbildungen zwischen diesen Strukturen zu definieren. Diese Abbildungen (Morphismen) können wir jetzt dazu verwenden, Graphen mit bestimmten *Graphtransformationsregeln* ineinander überzuführen. Die Transformation von Graphen baut bei uns auf dem Single-Pushout-Ansatz auf, auf den hier nicht weiter eingegangen werden soll, näheres findet sich in [Löw93, LKW93].

Zunächst müssen wir klären, wie eine Regel auf Graphen aussieht. Eine Graphregel ist ein Morphismus von einem Graphen $L$ in einen Graphen $R$.

Damit wir auch die Attributwerte von Graphobjekten mit unseren Graphregeln ändern können, verzichten wir darauf, dass ein Regelmorphismus die Homomorphiebedingung erfüllen muss, es soll also auch möglich sein, den Wert eines Attributs im Verlauf der Regelanwendung zu ändern. Die Theorie löst das Problem der statischen Attribute durch die Einführung von sogenannten *Attributträgern* [Rud97].

Die linke Seite der Regel enthält einen Graphen, den wir verändern wollen, indem wir die Regel anwenden. Objekte auf der linken Seite, die durch den Regelmorphismus auf den Graphen der rechten Seite abgebildet werden, bleiben erhalten, Objekte auf der linken Seite, die nicht abgebildet werden, werden gelöscht und Objekte, die rechts, aber nicht links auftauchen, kommen neu hinzu. Alle Veränderungen an den Graphobjekten werden also durch den Regelmorphismus beschrieben. Da der Regelmorphismus – um löschen zu können – nicht alle Objekte auf der linken Seite abbilden muss, kann er partiell sein.

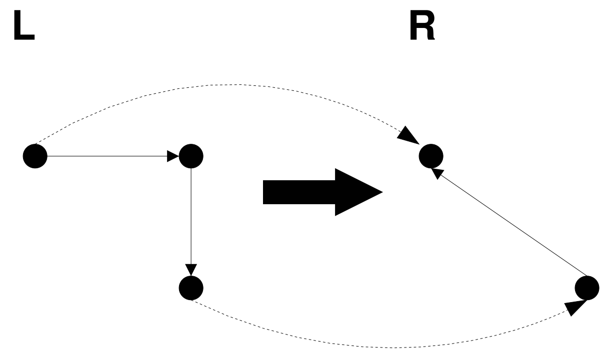



Abbildung 2.9: Eine Graphregel

Um mit Hilfe einer Graphregel einen Graphen zu transformieren, müssen wir nun die linke Regelseite in einem *Arbeitsgraphen* wiederfinden. Das ist das Konzept des *Regelansatzes* oder *Matches*. Ein Ansatz ist ein totaler Morphismus der linken Regelseite in einen Arbeitsgraphen. Meistens ist dieser Morphismus eine Inklusion, so dass es darum geht, die linke Regelseite als Teilgraphen in einem größeren Graphen zu identifizieren.

Wie wir leicht sehen, kann der Arbeitsgraph $G$ sehr viel größer als die linke Regelseite sein. Alle Graphobjekte, die nicht im Wertebereich des Ansatzmorphismus $m$ liegen, werden *Kontext* genannt. Eine Regel kann mehrere Ansätze in einem Graphen haben.

Was jetzt noch bleibt, ist die Transformation eines Arbeitsgraphen mit einer Regel.

Wir wollen an dieser Stelle nicht näher auf den kategorientheoretischen Hintergrund eingehen (dieser ist in [Löw93, EMC$^+$99]) ausführlich beschrieben), sondern stattdessen eine Konstruktion für den Fall „nichtattributierte Graphen“ angeben. Weitere Beispiele werden dann in Kapitel 3 ausführlich behandelt.

**Konstruktion 2.1** (*Graphtransformationsschritt*) Gegeben ist eine Graphregel $r : L \longrightarrow R$ und ein Ansatz $m : L \longrightarrow G$. Der Ergebnisgraph $H$ wird wie folgt konstruiert:

1. **Verkleben und Hinzufügen**

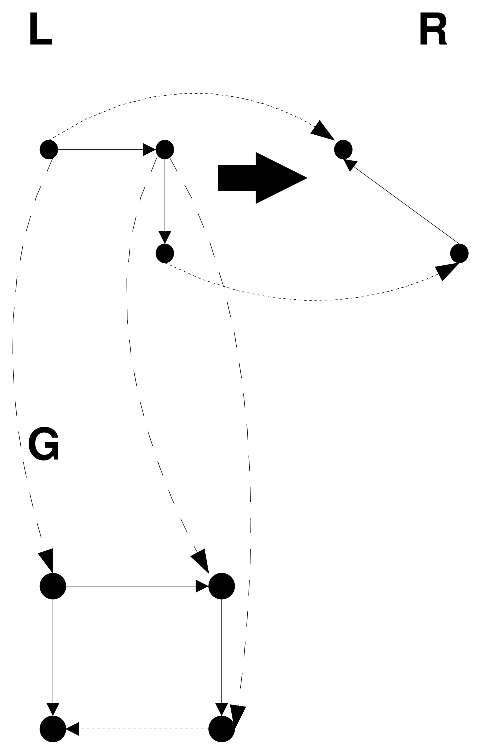



Abbildung 2.10: Ein Ansatz

Verklebe zunächst alle Graphobjekte im Arbeitsgraphen, die von $r$ abgebildet und identifiziert werden. Füge dem Arbeitsgraphen dann alle Objekte hinzu, die in $R \backslash L$ liegen. Das Zwischenergebnis ist ein Graph $Z$, in dem $G$ als Teilgraph enthalten ist.

2. **Löschen**
   Lösche alle Objekte in $Z$, die nicht in $dom(r)$ liegen, also nicht abgebildet werden. Ergeben sich dabei *hängende Kanten*, dh. Kanten, deren Quelle oder Ziel gelöscht wurde, lösche auch diese Kanten aus $Z$.

$\triangle$

Wenn wir uns diese Konstruktion betrachten, erkennen wir zwei Konfliktfälle. Zum einen kann es vorkommen, dass der Ansatzmorphismus zwei Objekte im Arbeitsgraphen identifiziert und dann eines der Objekte durch die Transformation gelöscht wird. Der andere Konflikt entsteht durch hängende Kanten, deren Quelle bzw. Ziel gelöscht wurde.

Beide Konflikte werden im Single-Pushout-Ansatz ziemlich rigide gelöst: Was ein Problem bereitet, wird einfach gelöscht. Um diese Probleme zu vermeiden, kann man an den Ansatzmorphismus $m$ weitere Einschränkungen stellen:

- Der Ansatz ist **injektiv**, wenn die Abbildung keine Graphobjekte in der linken Regelseite auf dasselbe Objekt im Arbeitsgraphen abbildet.

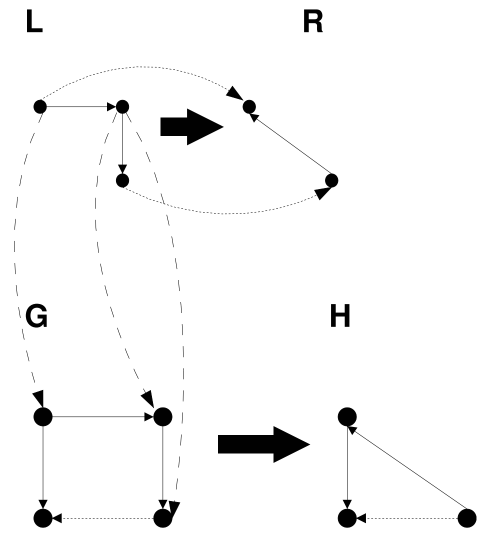



Abbildung 2.11: Eine Graphtransformation

- Der Ansatz ist **löschinjektiv**, wenn die Abbildung nur Graphobjekte identifiziert, die durch die Regelanwendung gelöscht werden. Diese Bedingung an den Ansatz heißt auch *identification condition*.
- Der Ansatz ist **konfliktfrei**, wenn zwei Graphobjekte, die im Arbeitsgraphen identifiziert werden, entweder gelöscht werden oder beide erhalten bleiben.
- Der Ansatz ist **löschvollständig**, kurz: **l-vollständig**, wenn jede Kante, deren Quelle oder Ziel durch die Regelanwendung gelöscht wird, ebenfalls gelöscht wird. Das verhindert hängende Kanten nach der Transformation – diese Bedingung heißt deshalb auch **dangling edge condition**.
- Der Ansatz ist **vorsichtig**, wenn $m$ l-vollständig und l-injektiv ist. Diese Bedingung heißt auch **gluing condition**.

Wir lassen für unsere Graphtransformation allerdings alle Arten von Ansätzen zu, da Kanten ohnehin eine besondere Bedeutung erhalten.

### 2.3.4 Negative Anwendungsbedingungen

Bislang haben wir in der linken Seite einer Graphregel alle Objekte aufgelistet, die in unserem Arbeitsgraphen auftauchen sollen. Vor allem beim Design einer visuellen Sprache aber werden wir feststellen, dass wir beim Ansatz einer Regel auch festlegen wollen, dass bestimmte Objekte und Konstellationen **nicht** im Ansatz auftauchen sollen. So soll beispielsweise nur dann eine Kante zwischen zwei Knoten eingefügt werden, wenn noch keine Kante vorhanden ist.

Eine solche Bedingung nennen wir *negative Anwendungsbedingung*, kurz: *NAC*[5]

[5] *Anm. d. A.* Auch auf die Gefahr hin, mich mit der gesamten deutschen Graphgrammatik-Gemeinde zu überwerfen, ist eine **NAC** bei mir weiblich.

(engl. negative application condition).

Eine *negative Anwendungsbedingung* oder kurz *NAC* ist ein Graph $N$ zusammen mit einem (auch partiellen) Morphismus von der linken Regelseite nach $N$. Die Anwendungsbedingung heißt in einem Match $m$ *erfüllt*, wenn es *keinen* totalen Morphismus $n : L \longrightarrow G$ gibt, so dass $n \circ l = m|_{dom(L)}$.

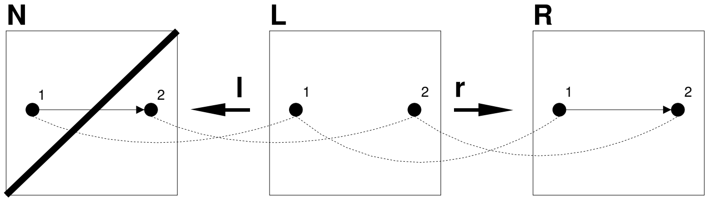



Abbildung 2.12: Eine negative Anwendungsbedingung

Ein Transformationsschritt gelingt jetzt nur, wenn *alle* negativen Anwendungsbedingungen der Regel erfüllt sind. Wir werden später die Anwendungsbedingungen dahingehend einschränken, dass der Morphismus $n$ in den Arbeitsgraphen auf dem Teil, der nicht aus $L$ stammt, injektiv sein muss.

Eine Regel kann $0 \ldots n$ Anwendungsbedingungen umfassen, von denen dann zum Zeitpunkt der Transformation *jede* erfüllt sein muss. Negative Anwendungsbedingungen können ebenso die Attribute umfassen, die Definition wird dann kanonisch erweitert.

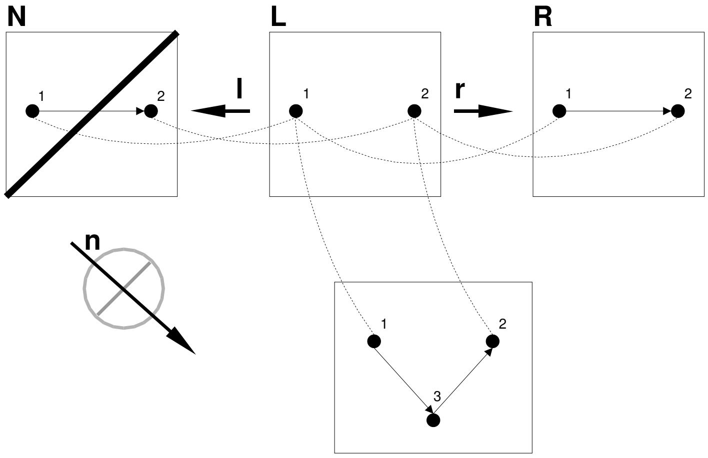



Abbildung 2.13: Ansatz einer NAC

# Kapitel 3

# Fallstudie und Sprachkonzepte

In diesem Kapitel sollen die im letzten Kapitel vorgestellten Einzelbestandteile einer visuellen Sprache, beschrieben durch Constraints und Graphtransformationen, zusammengeführt und das Konzept unserer visuellen Metasprache vorgestellt werden. Wir zeigen zunächst anhand einer zunächst einfach erscheinenden Fallstudie, welche Falltüren, aber auch welche Möglichkeiten die Definition einer visuellen Sprache bereithält.

## 3.1 Fallstudie: Statecharts

Statecharts sind eine Erweiterung einfacher Zustandsautomaten. Ein Zustandsautomat ist nichts weiter als ein Graph, dessen Knoten Zuständen und dessen Kanten den Zustandsübergängen entsprechen.

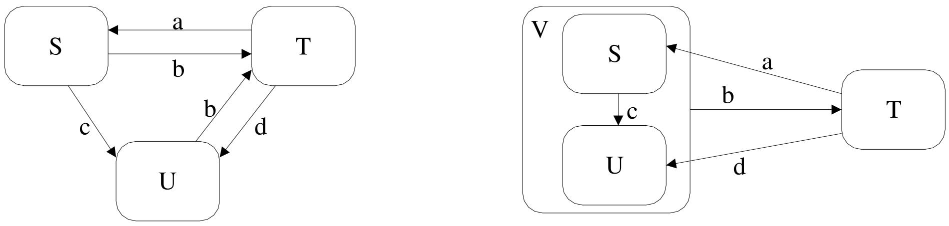

Abbildung 3.1: Ein einfacher Zustandsautomat

*E* bezeichnet hierbei das Ereignis, das den Zustandsübergang auslöst, *C* ist die Bedingung (engl. condition), die dazu führt, dass das Ereignis ausgelöst wird und *A* ist die (meist externe) Aktion, die durch das Ereignis ausgelöst wird.

In einer Pumpensteuerung könnte beispielsweise Zustand **S** bedeuten, dass die Pumpen in Betrieb sind, **T** wäre dann der Zustand „Pumpen außer Betrieb“, *E* wäre das Ereignis „Wasserstand wird abgelesen“, *C* die Bedingung „Wasserstand überschreitet gesetztes Limit“ und *A* wäre ein Abschaltsignal, das an die Pumpen geschickt wird.

Die Notation für Statecharts wurde im Laufe der achtziger Jahre von David Harel entwickelt. Das Grundprinzip der Zustandsautomaten bleibt erhalten und wird um Konstrukte erweitert, die die Strukturierung von Zustandssystemen erlauben:

**Hierarchische Komposition** ermöglicht es, mehrere Unterzustände zu einem Ge-




Einfacher Zustandsautomat

Äquivalenter Zustandsautomat mit XOR−Komposition

Abbildung 3.2: Hierarchische Komposition

samtzustand zusammenzufassen (siehe Abb. 3.2). Der Gesamtzustand kann dann, um die Darstellung zu vereinfachen, ohne die Unterzustände dargestellt werden.

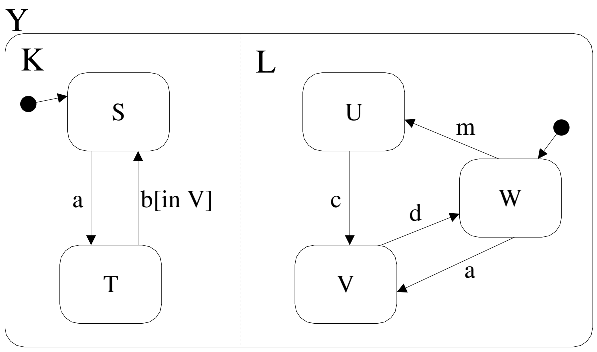



Abbildung 3.3: Parallele Komposition

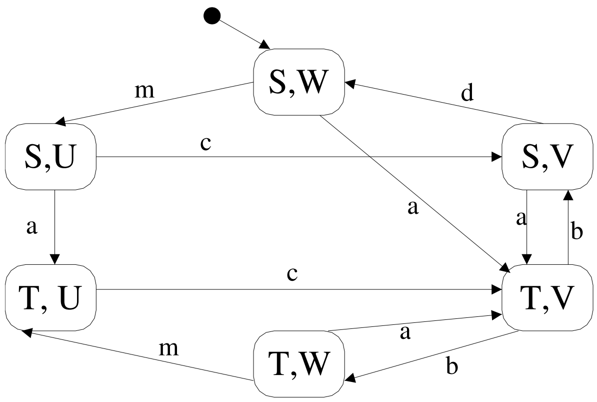



Abbildung 3.4: Äquivalente Beschreibung von $Y$ ohne parallele Komposition

**Parallele Komposition** ermöglicht es, mehrere Zustände durch Konjunktion zu einem Oberzustand zu verknüpfen. Innerhalb einer parallelen Komposition befindet sich das System gleichzeitig in allen verknüpften Zuständen (siehe Abb. 3.3 und 3.4).

**Default-Zustand** : Der initiale Zustand im Gesamtsystem bzw. in einem Oberzustand wird durch einen Pfeil visualisiert (siehe Abb. 3.5). Dieser Pfeil legt bei Eintritt in

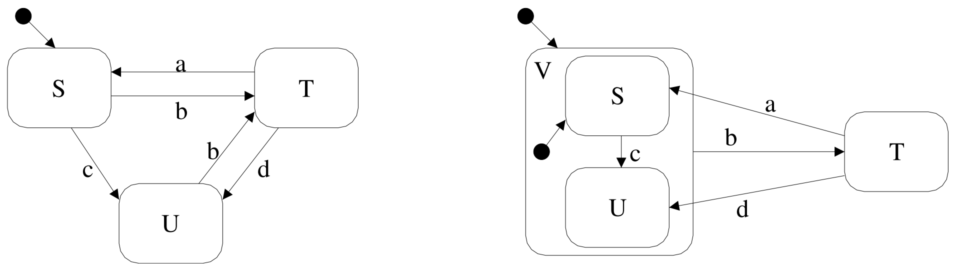



Abbildung 3.5: Default-Zustände

einen Gesamtzustand fest, in welchen Unterzustand das System wechselt, wenn der Gesamtzustand eintritt.

Es gibt noch weitere Zusatznotationen für Statecharts, die wir aber nicht mehr betrachten wollen. Eine ausführliche Beschreibung findet sich in [Har87] und [Har90].

### 3.1.1 Alphabet

Wie wir in Kapitel 2 gesehen haben, entspricht der Aufbau von visuellen Sprachen im Allgemeinen dem von textuellen Sprachen. Als erstes benötigen wir also ein *Alphabet* für unsere Statechart-Notationssprache. Das Alphabet muss alle grafischen Symbole umfassen, die in der oben vorgestellten Notation vorkommen. Alle „visuellen Sätze“ oder Diagramme bestehen aus diesen Alphabetsymbolen.

Ein grafisches Symbol ist eine Gruppe aus sogenannten „Primitiva“[1] wie z.B. Rechtecken, Linien, Pfeilen etc., die durch geeignete grafische Constraints zusammengehalten werden. Wir verwenden also auch hier – anders als viele Editoren – Constraints schon als Mörtel für den Bau ganzer Symbole. Jedes grafische Symbol hat bestimmte Eigenschaften, die in die Constraints Eingang finden, außerdem müssen wir für ein Symbol ein Default-Layout definieren.

Verschiedene grafische Symbole wiederum lassen sich dann auf bestimmte Weise miteinander verbinden, so z.B. Zustände und Zustandsübergänge. Auch diese Verbindungen werden durch grafische Constraints entweder zwischen den Symbolen als Ganzen oder zwischen Primitiva in Symbolen gegeben.

Constraints finden also in zwei Formen Eingang in unsere visuelle Sprache: Zum einen „kleben“ sie die Primitiva in einem grafischen Symbol zusammen, zum anderen beschreiben sie grafisch die Relation zwischen verschiedenen Symbolen.

#### 3.1.1.1 Konventionen

Wir legen zunächst einige Konventionen fest, um leichter über Grafiken sprechen zu können:

[1] lat. Primitivum, Primitiva

- Die Maßeinheit für Längen und Koordinaten ist unbestimmt. So lassen sich Grafiken problemlos in ihrer Gesamtheit vergrößern und verkleinern.

- Das Koordinatensystem (zweidimensional) der „Welt“, in der sich unsere Objekte aufhalten, hat seinen Ursprung in der linken oberen Ecke (siehe Abb. 3.6), der Y-Wert bewegt sich von oben nach unten vom Negativen ins Positive.

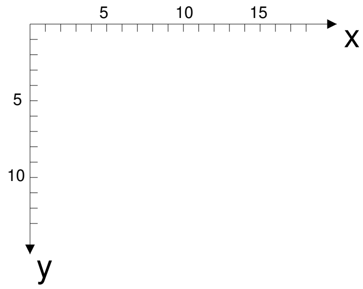


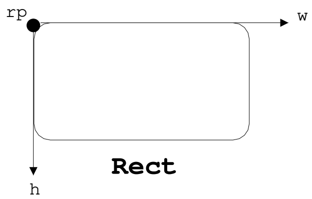

Abbildung 3.6: Koordinatensystem

#### 3.1.1.2 Grafische Primitiva

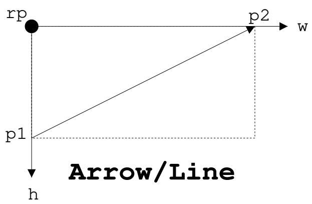
Ein grafisches Primitivum hat einen Namen und eine Reihe von grafischen Attributen wie Breite, Höhe etc.; außerdem besitzt jedes Primitivum ein umschließendes Rechteck (*Bounding Box*). Grafische Attribute werden in einer Art Pfadnotation geschrieben, so referenziert z.B. der Name `rect1.rp.x` die X-Koordinate des Referenzpunkts `rp` vom Objekt `rect1`.

Wir benötigen folgende grafische Primitiva:

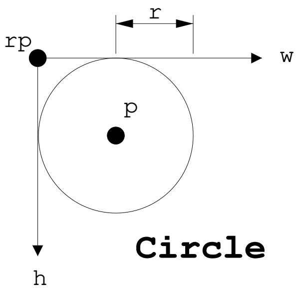
Ein *Rechteck* hat einen Referenzpunkt `rp` sowie eine Breite `w` und eine Höhe `h`. Rechtecke können eckig oder abgerundet sein.



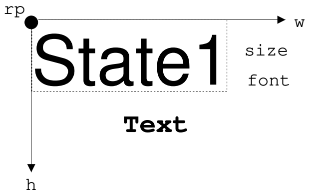

Eine *Linie* wird durch zwei Punkte `p1` und `p2` beschrieben. Die Linie ist von einer unsichtbaren Box umgeben, die auch wieder einen Referenzpunkt sowie Breite und Höhe besitzt. Eine Linie kann eine Pfeilspitze besitzen, diese Spitze befindet sich bei Punkt `p2`.



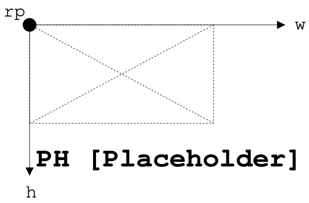
Ein *Kreis* wird durch einen Mittelpunkt `p` und einen Radius `r` beschrieben. Auch der Kreis ist von einer unsichtbaren Box umgeben.




*Text* wird durch Fontgröße und -art beschrieben sowie durch den dargestellten String. Dieser Text ist statisch und sollte nicht mit Datentypen, die als Text dargestellt werden, verwechselt werden.




Ein *Platzhalter* ist eine unsichtbare Box, die jedoch genau wie die umgebenden Boxen aller anderen Primitiva Breite, Höhe und Referenzpunkt besitzt.




#### 3.1.1.3 Anknüpfungspunkte

Durch das *Bounding-Box*-Konzept ist es – wie in herkömmlichen Vektorgrafikeditoren – möglich, jedes Objekt mit neun initialen Anknüpfungspunkten zu versehen (siehe Abb. 3.7). Diese bezeichnen wir mit den Himmelsrichtungen (*.n*= „north“, *.e*=„east“ etc.). Für Linien und Pfeile verwenden wir zwei zusätzliche Anknüpfungspunkte, namentlich *.begin* und *.end*. Das erleichtert später die genaue Positionierung von Kopf und Ende einer Linie.

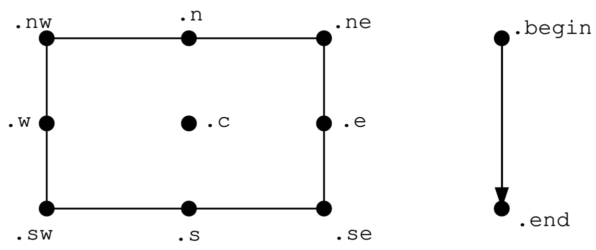



Abbildung 3.7: Anknüpfungspunkte

#### 3.1.1.4 Grafische Constraints

Um nun mehrere Primitiva zu einem grafischen Symbol zu verkleben, benötigen wir Constraints, die grafische Beziehungen zwischen den Subobjekten herstellen. Da es mehrere Constraint-Sprachen gibt, wir jedoch von der verwendeten Sprache unabhängig sein wollen, geben wir Constraints in einer Art Metasprache an. Diese Constraints nennen wir *High-Level-Constraints* (HLC).

Ein HLC besteht aus einem Namen und einer Reihe von Parametern. Diese Parameter können primitive grafische Objekte, Punkte und Ganzzahlen sein. Es ist möglich, die Parameter eines Constraints in bestimmter Weise zu typisieren. Das soll die Möglichkeiten einschränken, diesen Constraint auf bestimmte Objekte zu verwenden. Wenn im Folgenden von Parametern der Form `obj` die Rede ist, handelt es sich immer um ein beliebiges grafisches Objekt, bei Parametern, die `line`, `rect`, `p` etc. heißen, ist dagegen gemeint, dass dieser Constraint nur auf Linien, Rechtecke, Punkte etc. angewendet werden darf. Da wir im Unterabschnitt 3.1.1.3 Anknüpfungspunkte definiert haben, lassen sich die meisten der im Folgenden vorgestellten Constraints auf eine Kombination von einfachen Constraints dieser Art zurückführen:

```
equal[p1, p2]:
  p1.x = p2.x
  p1.y = p2.y
```

Hier sehen wir schon, wie die Constraints intern als Gleichungen bzw. Ungleichungen auf Koordinaten definiert werden können, genau wie in Abschnitt 2.2 beschrieben. Diesen Constraint werden wir – da er immer wieder auftauchen wird – so schreiben:

```
p1 = p2
```

Als nächstes definieren wir Constraints, die Objekte räumlich in einer bestimmten Beziehung zueinander positionieren. Ein Constraint dieser Art wäre z.B. das in Abschnitt 2.2 vorgestellte `above[obj1, obj2]`, das einfach besagt, dass sich Objekt `obj1` räumlich über Objekt `obj2` befinden soll.

Hierbei sind `.x` und `.y` wieder Punktkoordinaten, während `obj1.h` die Höhe der Bounding-Box des Objekts bezeichnet. Allgemein werden die Maße genau durch die in Abschnitt 3.1.1.2 angegebenen Bezeichner in den Primitiva referenziert.

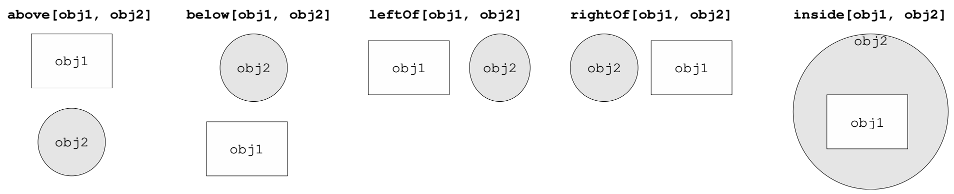



Abbildung 3.8: Constraints für die Anordnung von Objekten

Wir haben alle solche Anordnungs-Constraints in Abb. 3.8 zusammengefasst. Außerdem benötigen wir Constraints, um Objekte aneinander auszurichten. Ein Beispiel dafür ist der Constraint

```
alignTop[obj1, obj2]:
  obj1.rp.y = obj2.rp.y
```

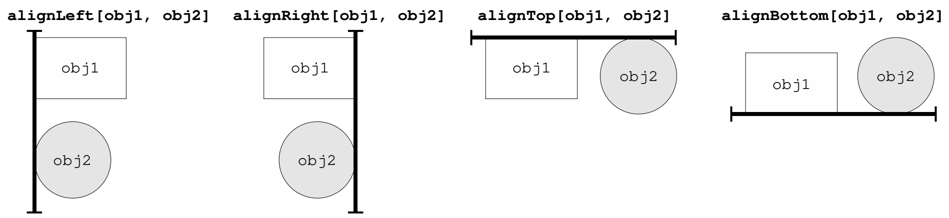



Abbildung 3.9: Constraints für die Ausrichtung von Objekten

Alle Ausrichtungsconstraints befinden sich in Abb. 3.9. Um nun Linienanfänge bzw. -enden mit anderen Objekten wie Rechtecken oder Kreisen zu verbinden, wie das bei unserer Statechart-Notation oft auftauchen wird, definieren wir Verankerungs-Constraints (siehe Abb. 3.10). Hier erkennen wir ein weiteres Konzept für Constraints: Den Constraint `anchorBorder[obj, line]` können wir durch Konjunktion der vier anderen `anchor`-Constraints bilden.

```
anchorBorder[obj, line]:
  anchorLeft[obj, line] OR
  anchorRight[obj, line] OR
  anchorTop[obj, line] OR
  anchorBottom[obj, line]
```

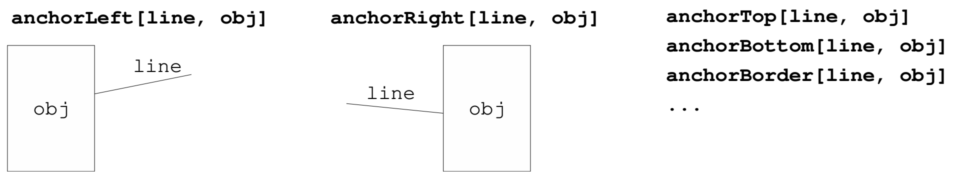



Abbildung 3.10: Constraints für die Verankerung von Linien

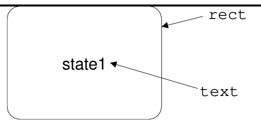
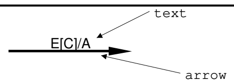
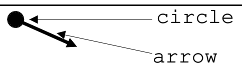
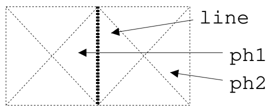
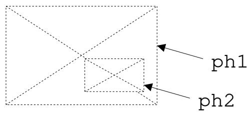
Eine besondere Klasse von Constraints umfasst solche, die nur ein Objekt als Parameter bekommen. Alle weiteren Parameter (soweit vorhanden) sind Ganzzahlen. Dazu gehören beispielsweise Constraints wie `fixedWidth[obj, w]` oder `fixed-Height[obj, h]`, die bestimmen, dass ein Objekt eine bestimmte Höhe oder Breite haben soll. In dieselbe Kategorie fallen Constraints wie `maxWidth[obj, w]`, die Objektattributen einen Maximal- bzw. Minimalwert zuweisen. Einer der wichtigsten Constraints in dieser Klasse ist der Constraint `fixed[obj]`, der nichts weiter besagt, als dass keines der Attribute dieses Objekts von den initialen (Default-)Werten abweichen soll. Mit diesem Constraint lassen sich, wie wir weiterhin sehen werden, konstante Symbole sehr elegant definieren.

Alle weiteren Constraints wollen wir nicht näher vorstellen, da auf dieses Thema in [Sch99] detaillierter eingegangen wird.

#### 3.1.1.5 Grafische Symbole

Nun haben wir das Handwerkszeug, um unsere grafischen Primitiva zu Statechart-Symbolen zusammenzusetzen, vergleichbar mit einem Hausbau: Die Ziegel sind unsere grafischen Symbole, der Mörtel sind die Constraints. Unsere Symbole werden in Tabelle 3.1 definiert.

Tabelle 3.1: Grafische Symbole

| Name | Abbildung | Constraints |
|---|---|---|
| Zustand (`State`) |  | `minSize[rect, 50, 40]` |
| Zustandsname (`SN`) | **State1** | |
| Zustandsübergang (`Transition`) |  | `minLength[arrow, 10]` |
| Transitionsname (`TN`) | **E[C]/A** | |
| Default-Knoten (`Default`) |  | `circle.p = arrow.begin`<br>`minLength[line, 1]`<br>`fixed[circle]` |
| Parallel-Verbinder (`Parallel`) |  | `vertical[line]`<br>`touchLeft[p1, line]`<br>`touchRight[p2, line]` |
| Hierarchie-Verbinder (`Hierarchy`) |  | `inside[ph2, ph1]` |

Ein Zustand besteht also aus einem abgerundeten Rechteck mit einem Text. Der Text ist dynamisch, wir fassen ihn deshalb in ein gesondertes Symbol ein, wobei der Text selbst als Datentyp dargestellt wird (siehe Abschnitt 3.1.1.7). Ein Zustandsrechteck soll eine Mindestbreite von 50 sowie eine Mindesthöhe von 40 Einheiten haben.

Ein Zustandsübergang ist ein einfacher Pfeil mit einem Text, der genau wie der Zustandsname in einem Datentypen gespeichert wird. Der Pfeil hat eine Mindestlänge von 20 Einheiten. Die Verbindungen zwischen einem Zustand bzw. einer Transition und den jeweiligen Namen wird, da die Namen externe Symbole sind, als grafischer Link definiert.

Der Default-Knoten besteht aus einem konstanten Kreis und einem Pfeil, dessen Ursprung im Mittelpunkt des Kreises liegt. Zur besseren Darstellung legen wir fest, dass der Pfeil eine Mindestlänge von 10 Einheiten haben soll.

Die beiden letzten Symbole haben besondere Aufgaben – ähnlich wie eine Transition verbinden sie zwei Zustände, um sie zu strukturieren. Dabei wird der Parallelverbinder als senkrechte gestrichelte Linie mit zwei angrenzenden Platzhaltern dargestellt. Diese Platzhalter nehmen, wie wir sehen werden, den rechten und linken Unterzustand des parallelisierten Zustands auf. Der Hierarchie-Verbinder besteht aus zwei Platzhaltern, die ineinander geschachtelt sind. Diese Schachtelung wird durch den `inside`-Contraint ausgedrückt.

Diese Platzhalter sind nötig, da wir sonst keine Verbindung zwischen den tatsächlichen Zuständen herstellen können. Später werden die Zustände an die Platzhalter gebunden und die Verbinder sorgen dafür, dass die räumlichen Beziehungen `inside` bzw. `touchLeft` hergestellt werden.

#### 3.1.1.6 Grafische Links

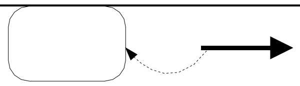
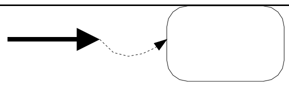
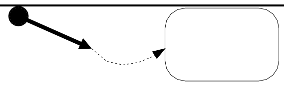
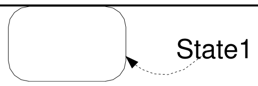
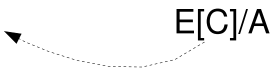
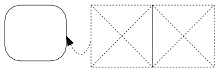
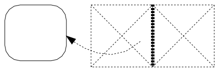
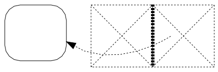
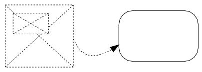
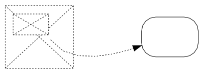
Als letzte Aktion legen wir die möglichen grafischen Verbindungen (Links) zwischen den unterschiedlichen Symbolen fest. Ein Link gibt an, an welchen Stellen (Anknüpfungspunkte bzw. Objekte) ein grafisches Symbol des Alphabets mit einem anderen verknüpft werden kann. Jeder Link hat einen Namen, ein Quell- und ein Zielsymbol. So gibt es z.B. den Link begin: Transition $\longrightarrow$ State, der einen Zustandsübergang mit einem Ausgangszustand verknüpft.

Links werden natürlich nicht nur dafür verwendet, die logische Struktur unserer Sprache zu erzeugen, sie steuern gleichzeitig auch den grafischen Aufbau eines Satzes oder Diagramms. So erhält jeder Link einen oder mehrere Constraints, die angeben, wie die grafischen Subsymbole oder Anknüpfungspunkte im Quell- und Zielsymbol des Links miteinander verbunden werden. Für unsere Sprache ergeben sich folgende Links:

Hierbei definieren die from- und to-Links die Verbindung einer Transition mit ihrem Anfangs- bzw. Endzustand. Die beiden Links sName und tName sorgen für die Verbindung zwischen Zustand und Zustandsnamen bzw. Transition und Namen. Hier wird auch festgelegt, dass der Zustandsname sich innerhalb des Zustands befinden muss, ebenso, dass der Transitionsname oberhalb der Transition sitzt und links mit der Transition abschließt. Der Link link_default_state verbindet einen Default-Konnektor mit einem Zustand.

Die Links common, left, right, super und sub sollen im Folgenden genauer erläutert werden. Der Link common nimmt einen zu parallelisierenden Oberzustand auf, der Parallelverbinder wird genau so groß, dass er in den Zustand passt. Die Links left und right sorgen für die Platzierung des linken bzw. rechten Unterzustands. Hier ist zu beachten, dass wir die Grammatik so einschränken, dass nur jeweils zwei parallele Unterzustände für einen Oberzustand existieren können. Die Links super und sub liefern auf ähnliche Weise die Verbindung zwischen Ober- und Unterzustand.

Tabelle 3.2: Grafische Links

| **Name** | **Abbildung** | **Constraints** |
|---|---|---|
| from: Transition ⟶ State |  | `anchorBorder[tr.begin, st]` `outside[Transition, State]` |
| to: Transition ⟶ State |  | `anchorBorder[tr.end, st]` `outside[Transition, State]` |
| link_default_state: Default ⟶ State |  | `anchorBorder[df.end, st]` `outside[Default, State]` |
| sName: SN ⟶ State |  | `inside[sn, st]` |
| tName: TN ⟶ Transition |  | `touchTop[tn, tr]` `alignLeft[tn, tr]` `alignRight[tn, tr]` |
| common: Parallel ⟶ State |  | `pa = st` |
| left: Parallel ⟶ State |  | `pa.ph1 = st` |
| right: Parallel ⟶ State |  | `pa.ph2 = st` |
| super: Hierarchy ⟶ State |  | `hi.ph1 = st` |
| sub: Hierarchy ⟶ State |  | `hi.ph2 = st` |

Wollen wir alle möglichen Links visualisieren, bietet es sich an, einen *Typgraphen* für Statecharts anzugeben, so geschehen in Abb. 3.11. In diesem Graphen stellen die Knoten die grafischen Symbole und die Kanten die grafischen Links dar.

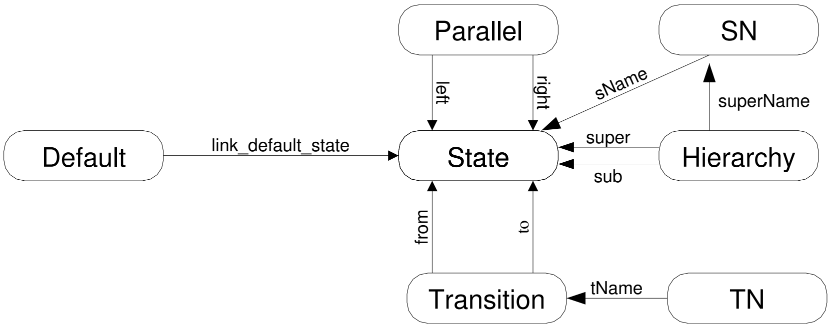



Abbildung 3.11: Visualisierung von Links mit Typgraphen

#### 3.1.1.7 Datentypen

Wir werden später auf ein Problem stoßen, das wir jetzt schon lösen können: Wir werden bei der Anwendung von Regeln und der Darstellung von Diagrammen sehen, dass wir Text dynamisch halten wollen. So soll beispielsweise der Name eines Zustands, den wir eigentlich in einem Grafikprimitivum vom Typ „Text" sichern könnten, zwar immer Text bleiben, aber je Objekt individuelle Werte haben. Bei Primitiva wie Rechteck und Linie lassen sich Änderungen grafisch gut ausdrücken, bei Text oder Zahlen haben wir dagegen das Problem, dass wir so etwas wie Texteingaben (wenn z.B. der Name eines Zustands in einer Regel geändert werden soll) schlecht über unsere grafischen Constraints ausdrücken können, denn das einzige, was sich am Text fassen lässt, sind solche Attribute wie Textgröße und -stil.

Wir führen deshalb *Datentypen* ein. Ein Datentyp ist eine Klasse von Objekten, die sich grafisch darstellen lässt, die aber auch durch Operationen geändert werden kann. Der Datentyp, den wir sehr oft verwenden werden, ist `String`. Ein String ist eine Zeichenkette, eine Operation auf einem String wäre z.B. `concat: String String -> String`. Diese Operation verknüpft zwei Strings zu einem dritten. Wo immer Datentypen einen beliebigen Wert annehmen können, verwenden wir *Variablen*.

Für unser Diagramm ist ein Datentyp nichts weiter als ein Rechteck, dessen Inhalt durch den Wert bestimmt wird. Bei einem String z.B. wird die beinhaltete Zeichenkette mit einem bestimmten Font geschrieben, die Ausmaße des umgebenden Rechtecks ergeben sich dann durch die Größe der Zeichenkette.

Für die oben vorgestellten Symbole ergeben sich folgende Datentypen:

```
State.text : String
Transition.text : String
```

Natürlich müssen wir für die Symbole für den Zustands-/Transitionsnamen ebenso wie für alle anderen Symbole Default- oder initiale Werte angeben, nur, dass die initialen Werte jetzt Konstanten des Datentyps sind. Für den Zustandsnamen wählen wir die Konstante „`state`\, für den Transitionsnamen die Konstante „`transition`\.

### 3.1.2 Grammatik

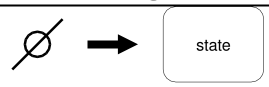
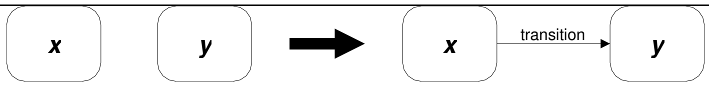
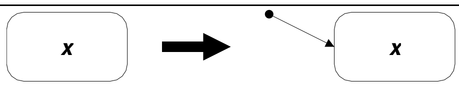
Als nächstes werden wir die Regeln einer Grammatik angeben, die die Sprache der Statecharts erzeugt. Regeln bestehen aus einer linken Seite und einer rechten Seite, wobei die linke Seite ein Ausgangsdiagramm enthält, das durch die Regelanwendung in die rechte Seite transformiert wird (wir erkennen schon ansatzweise die Graphtransformation). Das Ausgangsdiagramm stellt einen Ausschnitt eines Arbeitsdiagramms dar, der geändert werden soll. Ist das Ausgangsdiagramm leer, schreiben wir stattdessen $\emptyset$.

#### 3.1.2.1 Einfache Regeln

Aus den möglichen Links erkennen wir schon die einfachen Regeln zum Einfügen der Grundelemente (Zustand, Transition, Default). Allerdings müssen wir aufpassen: Die Elemente `Transition` und `Default` dürfen beim Einfügen nicht einfach „in der Luft hängen“, sie müssen immer mit schon existierenden Zuständen verbunden werden. Dieses Problem lösen wir wie folgt: Wir erzwingen, dass, wenn wir ein Symbol in ein Diagramm einfügen, das als Quellsymbol in einem grafischen Link auftaucht, dieser Link definiert sein muss. Das eingefügte Symbol muss also komplett mit Link eingefügt und an ein schon vorhandenes Zielsymbol gebunden werden.

Wir erhalten folgende Regeln (die Namen sollen die Aktion der Regel widerspiegeln):

Tabelle 3.3: Regeln für das Einfügen von Objekten

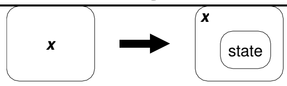
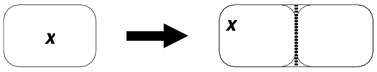
| **Name** | **Abbildung** |
|---|---|
| InsertState |   |
| InsertTransition |   |
| InsertDefault |   |

Die Datentypen `State.name` und `Transition.name` werden hier mit den initialen Werten belegt. Dort, wo beliebige Werte für die Namen stehen, verwenden wir Variablen $(x, y)$.

Die einzufügenden Objekte werden mit Hilfe der grafischen Links positioniert, so sehen wir z.B., dass eine Transition mit Hilfe der from- und to-Links an den jeweiligen Anfangs- bzw. Endzuständen befestigt wird. Nach einer Regelanwendung sind also alle grafischen Constraints erfüllt.

Mit Hilfe der oben eingeführten Regeln lassen sich schon einfache Zustandsautomaten bauen, wir führen jetzt aber weitere Regeln auf, mit denen die Komposition von Zuständen beschrieben wird. Diese Regeln kreieren wir gleich so, dass nicht zwei Zustände in eine Hierarchie-Beziehung gebracht werden, sondern ein Unterzustand in einen Oberzustand eingefügt wird. Ebenso werden nicht zwei Zustände zu einem parallelen verknüpft, sondern ein Zustand wird genommen und parallelisiert.

Tabelle 3.4: Regeln für die Komposition von Objekten

| **Name** | **Abbildung** |
|---|---|
| InsertSubstate |  |
| ParallelizeState |  |

#### 3.1.2.2 Negative Anwendungsbedingungen

Mit Hilfe dieser Grammatik können wir jetzt Statecharts aufbauen. Doch sind diese Statecharts richtige Diagramme im Sinne der Fallstudie? Werfen wir einen Blick auf einige Problemfälle:

1. Die Hierarchie-Beziehung darf nur einmal für je zwei Zustände auftreten. Außerdem darf ein Zustand, der schon parallelisiert ist, keinen Unterzustand haben (außer natürlich den zwei parallelen).

2. Ein Zustand darf nur einmal parallelisiert werden und – analog zum vorhergehenden Problem – darf keinen Unterzustand haben, wenn er parallelisiert werden soll.

3. Eine Transition darf keinen Unterzustand mit einem seiner Oberzustände verbinden.

4. Eine Transition darf keine parallelen Zustände miteinander verbinden.

5. Nur ein Zustand auf einer Ebene darf Default-Zustand sein.

6. Ein Zustand, der parallel zu einem zweiten steht, darf kein Default-Zustand sein.

Wir lösen diese Probleme, indem wir bei Anwendung einer Regel bestimmte Konstellationen von Grafiken unterbinden. Das geschieht, indem man *negative Anwendungsbedingungen* angibt (analog zu Abschnitt 2.3.4).

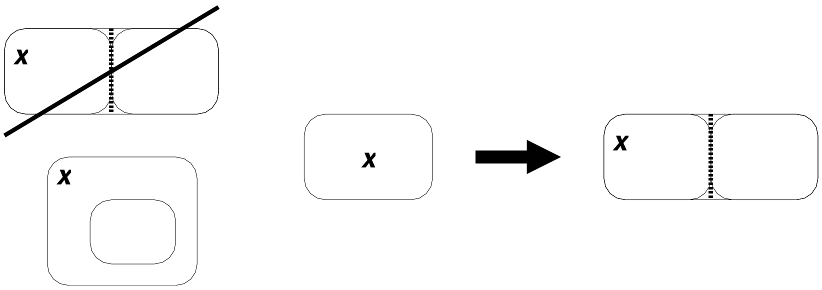


Abbildung 3.12: Zu Problem 1: Einfügen eines Unterzustands

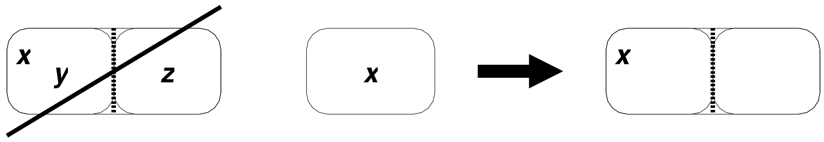



Abbildung 3.13: Zu Problem 2: Parallelisieren eines Zustands

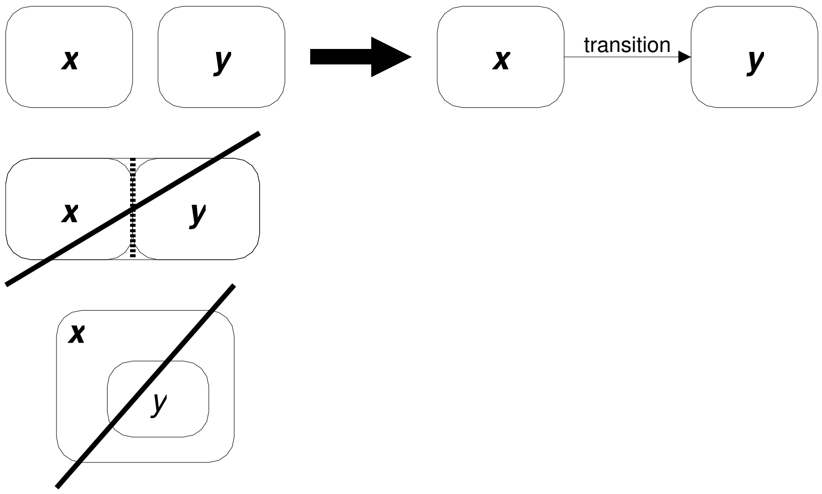



Abbildung 3.14: Zu Problem 3: Einfügen einer Transition

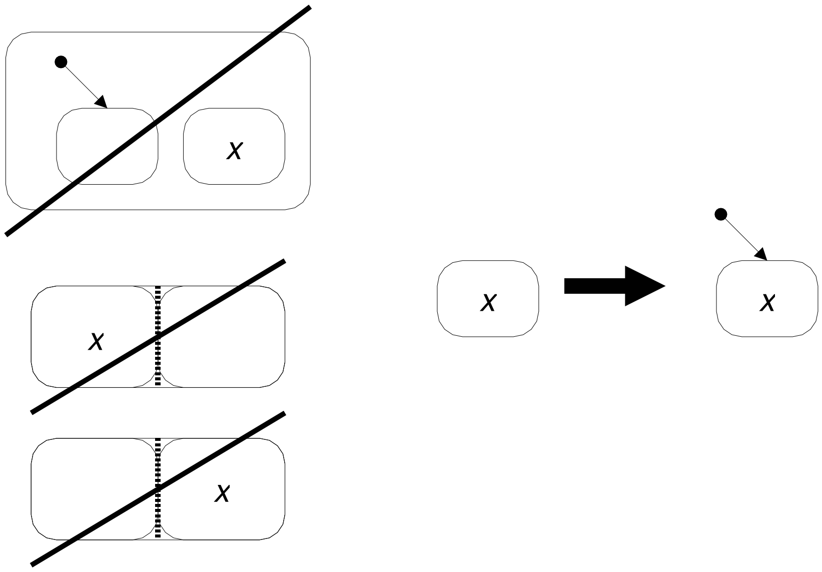



Abbildung 3.15: Zu Problem 4 und 5: Setzen eines Default-Zustands

Sind jetzt alle Probleme gelöst und lassen sich mit den geänderten Regeln korrekte Statecharts im Sinne der Beschreibung aufbauen? Mitnichten, denn es gilt beispielsweise, dass eine Transition **nicht** zwischen einem Zustand und **irgendeinem** seiner Oberzustände gezogen werden darf. Wir müssten also eine Regel bauen, in der ein Zustand

Verbindungen mit allen Oberzuständen bekommt, damit die negative Anwendungsbedingung für das Einfügen einer Regel in Abb. 3.14 überprüft werden kann. Da allerdings unsere Regeln nur lokal arbeiten, muss das zwangsläufig eine Hintereinanderanwendung von Regeln sein, die in etwa so aussieht:

> Solange der eingefügte Zustand noch einen Oberzustand ohne Verbindung besitzt, füge diese Verbindung ein.

Für die Verbindung können wir das Symbol `Hierarchy` problemlos weiterverwenden. Wir wollen hier nicht weiter auf die diversen Probleme eingehen, es genügt uns zu zeigen, dass die Komplexität der Definition einer visuellen Sprache auf keinen Fall zu unterschätzen ist, dass man allerdings mit Graphregeln – auf die unsere visuellen Regeln im nächsten Abschnitt zurückgeführt werden – sehr mächtige Werkzeuge hat, um Strukturen eines Diagramms auszudrücken.

#### 3.1.2.3 Löschregeln

Mit den im letzten Abschnitt genannten Änderungen sollte es uns möglich sein, Statecharts aufzubauen. Die Sprache ist komplett definiert! Da wir diese Sprache aber dazu verwenden wollen, die Regeln bzw. Befehle für einen Editor zu spezifizieren, sollten wir ebenso die Möglichkeit bieten, regelbasiert Teile eines Diagramms zu löschen.

Die einfachste Löschregel ist diejenige, die eine Transition zwischen zwei Zuständen löscht. Dazu drehen wir die Regel **InsertTransition** einfach um.

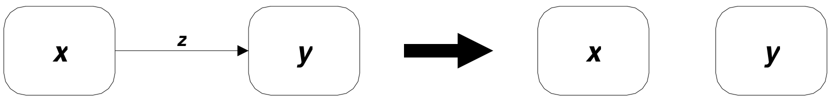



Abbildung 3.16: Löschen einer Transition

Ebenso können wir auch die Default-Markierung eines Zustands löschen.

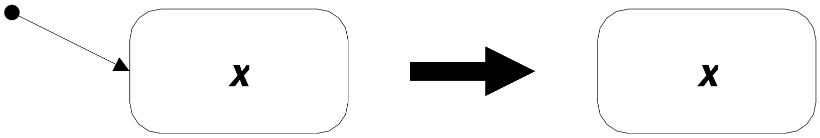



Abbildung 3.17: Löschen eines Default-Markierers

Schwieriger gestaltet sich das Löschen, wenn es um einen Zustand geht. Ein Zustand darf nur dann gelöscht werden, wenn alle Transitionen, die ihn als Start- oder Endzustand haben, gelöscht sind, kein Default-Connector auf diesen Zustand verweist und er außerdem keine Unterzustände besitzt. Das lässt sich jedoch elegant mit einer Reihe von negativen Anwendungsbedingungen lösen.

Das Aufheben von hierarchischer oder paralleler Komposition gestaltet sich im Grunde ebenso einfach. Es ist zwar so, dass durch das Löschen einer Parallelisierung beispielsweise die Unterzustände „in der Luft hängen“, das spielt aber für die Syntax

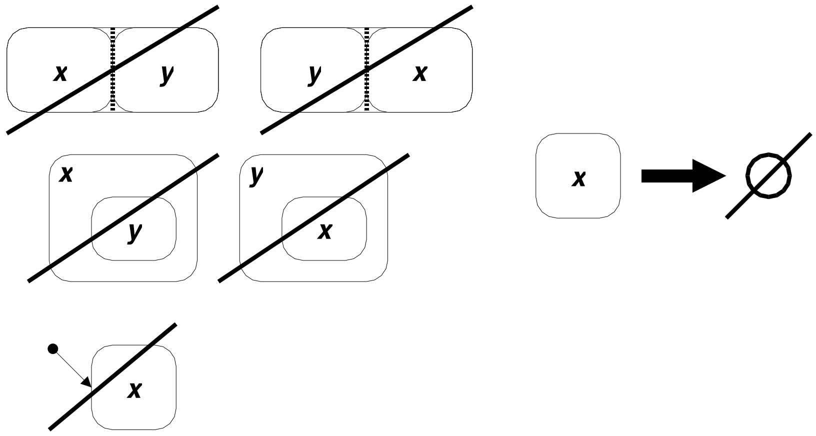



Abbildung 3.18: Löschen eines Zustands

der Statecharts, um die es uns hier ja geht, keine Rolle. Korrekterweise müssen wir aber anmerken, dass – falls wir das Regelkonstrukt „Verbinder zu allen Oberzuständen einfügen“ aus dem letzten Unterabschnitt verwendet haben – wir auch alle Verbinder zu Oberzuständen durch die Hintereinanderanwendung entsprechender Regeln löschen müssen.

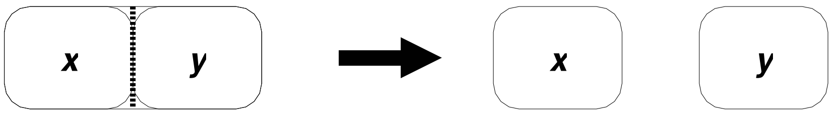



Abbildung 3.19: Löschen eines Parallelzustands

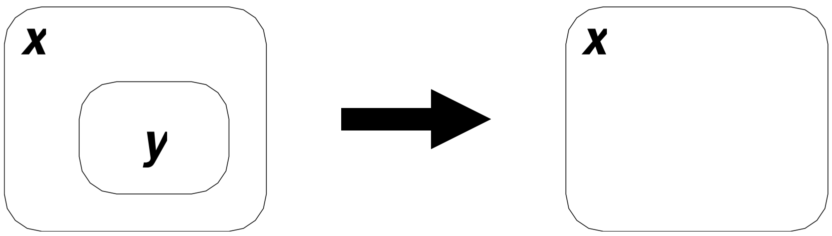



Abbildung 3.20: Löschen eines Unterzustands

Nun haben wir alle Regeln zusammen, um einen simplen Editor für Statecharts spezifizieren zu können. Es ist durchaus möglich, noch komplexere Strukturierungsmöglichkeiten durch geeignete Sprachregeln anzubieten. So kann man beispielsweise das Auflösen der parallelen Komposition als Regel oder Hintereinanderanwendung von Regeln gestalten, ebenso das Verbergen/Sichtbarmachen von Unterzuständen.

## 3.2 Sprachkonzepte

Wir wollen in diesem Abschnitt die Sprachkonzepte, die sich aus der Fallstudie ergeben haben, verallgemeinern und informell beschreiben. Eine ausführliche, formale Beschreibung findet sich in [Bar99].

### 3.2.1 Alphabet

Eine visuellen Sprache besteht aus Symbolen, diese bilden ein *Alphabet*. Dieses Alphabet besteht aus zwei Teilen, der *abstrakten* Syntax, die die Namen der Symbole und die der grafischen Links enthält und der *konkreten Syntax*, in der die grafische Beschreibung der Symbole und die Constraints sowohl der Symbole als auch der grafischen Links auftauchen. Bei dieser Definition werden wir unsere Fallstudie noch einmal aufarbeiten und ggf. Dinge dort ändern.

#### 3.2.1.1 Grafische Primitiva

Ein *grafisches Primitivum* ist definiert durch ein einfaches grafisches Objekt (Linie, Rechteck, Ellipse, Polylinie etc.) mit einer umgebenden Box (*Bounding-Box*). Eine *Bounding-Box* wiederum besteht aus einem Rechteck mit neun Anknüpfungspunkten `.nw`, `.n`, `.ne`, `.e`, `.se`, `.s`, `.sw`, `.w` und `.c`. Die wichtigsten Primitiva wurden in Unterabschnitt 3.1.1.2 vorgestellt, diese Menge ist aber beliebig erweiterbar.

Jedes grafische Primitivum hat einen Namen und die Punkte und Maße in einem Primitivum lassen sich durch eine Pfadangabe ansprechen. So hat der nordöstliche Anknüpfungspunkt einer Ellipse `ellipse` beispielsweise den Namen `ellipse.ne`, dessen X-Koordinate den Namen `ellipse.ne.x`.

#### 3.2.1.2 Constraints

Ein *grafischer Constraint* besteht aus einem Namen, $1 \ldots n$ Parametern und einer Menge von Ungleichungen über einer Variablenmenge. Die Parameter können grafische Primitiva oder ganze Zahlen sein, die Variablenmenge ergibt sich einerseits aus den Parameternamen für ganze Zahlen, andererseits aus den Maßen, die durch eine Pfadangabe von den als Parametern angegebenen Primitiva aus erreichbar sind. Ein Constraint z.B. wäre:

```
inside[obj1, obj2]:
  obj1.rp.x >= obj2.rp.x
  obj1.rp.y >= obj2.rp.y
  obj1.w <= obj2.w
  obj1.h <= obj2.h
```

Ein Constraint heißt *erfüllt* für eine Belegung seiner Parameter, wenn alle seine Ungleichungen erfüllt sind. Eine Constraintmenge ist *erfüllt*, wenn alle Constraints erfüllt sind. Weitere Beispiele findet man in der Fallstudie, Abschnitt 3.1.1.4.

#### 3.2.1.3 Grafische Symbole

Ein *grafisches Symbol* besteht aus einer Menge von grafischen Primitiva und einer Menge von Constraints über diesen Primitiva. Jedes Symbol hat einen eindeutigen Namen und eine umgebende Box, die die Vereinigung der umgebenden Boxen seiner Primitiva darstellt. Die umgebende Box eines Symbols `obj` wird mit dem Objektnamen angesprochen.

Jedes grafische Symbol besitzt ein initiales (Default-)Layout. Das Default-Layout ist eine Instanz des Symbols mit konkreten Werten, dh. sämtliche Primitiva des Symbols müssen mit konkreten Maßen belegt sein (Referenzpunkt, Breite, Höhe, Radius etc.). Die Constraints des Default-Symbols müssen erfüllt sein.

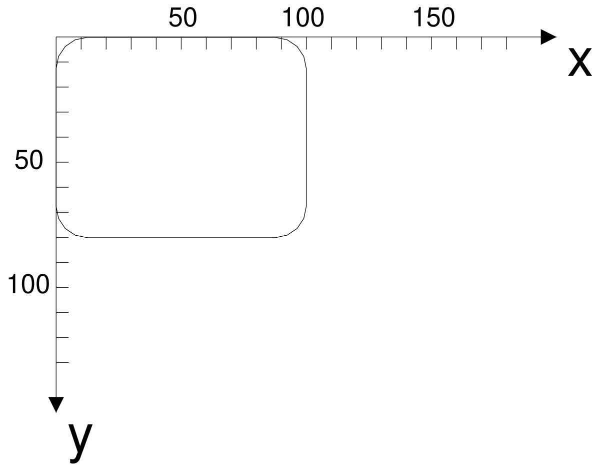



Abbildung 3.21: Beispiel: Default-Layout eines Zustands

Das Default-Layout des Zustands ist in diesem Fall ein Rechteck mit dem Referenzpunkt (0,0) und mit einer Breite von 100 und einer Höhe von 80 Einheiten. Der Constraint `minSize[rect, 50, 40]` ist damit erfüllt. Weitere Beispiele wurden in Tabelle 3.1 vorgestellt.

#### 3.2.1.4 Datentypen

Ein Datentyp ist ein grafisches Symbol, das aus einer umgebenden Box, einem Typen (Sorte) und einem Wert besteht. Alle Datentypen einer visuellen Sprache werden in einer Datentypsignatur zusammengefasst. Ein Wert eines Datentyps ist ein Element in einer Algebra zu diesem Datentypen. Die Datentypsignatur unseres Fallbeispiels ist in Abb. 3.22 dargestellt (erweitert um ein Tripel Ereignis-Bedingung-Aktion).

Wie in unserem Fallbeispiel werden die Datentypen als eigenständige grafische Symbole aufgefasst. Wir benötigen also weitere grafische Links, die die Symbole mit den dazugehörenden Datentypen verknüpfen. Daraus folgt außerdem, dass auch Datentypen ein Default-Layout besitzen müssen. Dieses Default-Layout ist ein festgelegtes Element (eine Konstante) aus der Trägermenge zur Sorte des Datentyps. Ein Datentyp wäre z.B. `TransitionName` oder kurz `TN:Triple` und das Default-Layout des Tripels für `TN`

$\textsc{DataSig}_{SC}$ =
**sorts**:
String Triple Event Condition Action
**opns**:
getEvent : Triple $\longrightarrow$ Event
getCondition : Triple $\longrightarrow$ Condition
getAction : Triple $\longrightarrow$ Action

Abbildung 3.22: Datentypsyntax für Statecharts

könnten die Werte `E(C)/A` mit einer bestimmten Fontart und -größe sein.

#### 3.2.1.5 Abstrakte Syntax

Die *abstrakte Syntax* einer visuellen Sprache besteht aus einer Signatur, deren Sorten den Namen der grafischen Symbole und deren Operationen den Namen der grafischen Links zwischen Symbolen entsprechen.

Für unsere Fallstudie ergibt sich nun die Signatur in Abb. 3.22.

$\textsc{LogSig}_{SC}$ =
**sorts**:
State Transition Default History SN TN Parallel Hierarchy
**opns**:
from : Transition $\longrightarrow$ State
to : Transition $\longrightarrow$ State
link_default_state : Default $\longrightarrow$ State
sName : SN $\longrightarrow$ State
tName : TN $\longrightarrow$ Transition
super : Hierarchy $\longrightarrow$ State
superName : Hierarchy $\longrightarrow$ SN
sub : Hierarchy $\longrightarrow$ State
left : Parallel $\longrightarrow$ State
right : Parallel $\longrightarrow$ State

Abbildung 3.23: Abstrakte Syntax für Statecharts

Eine solche abstrakte Syntax lässt sich selbst wieder grafisch (oder graphisch!) darstellen, indem man die Sortennamen als Knoten und die Operationen als Kanten auf-zeichnet. So erhält man einen Typgraphen wie in Abb. 3.11.

#### 3.2.1.6 Konkrete Syntax

Eine *konkrete Syntax* zu einer visuellen Sprache besteht nun

- aus den grafischen Symbolen für einen bestimmten Symbolnamen und
- aus den Constraints, die die grafischen Links konkretisieren.

Jedem Namen für ein grafisches Symbol entspricht ein Symbol selbst, jeder Operation der abstrakten Syntax wird ein grafischer Link zugeordnet. Für unser Fallbeispiel besteht die konkrete Syntax aus den Tabellen 3.1 und 3.2. Abstrakte und konkrete Syntax zusammengefasst bilden unser *Alphabet*. Den Begriff *Syntax* haben wir deshalb gewählt, weil das Alphabet tatsächlich eine einfache syntaktische Beschreibung der Sprache darstellt. Diagramme werden dann letztendlich aus den Instanzen der Alphabetsymbole gebildet.

### 3.2.2 Grammatik

Die Grammatik einer visuellen Sprache baut – wie bei textuellen Sprachen – auf einem Alphabet auf. Da bei textuellen Sprachen die Verbindung zwischen zwei Symbolen (Buchstaben) durch ihre Anordnung in der Liste oder dem Wort/Satz der Sprache vorgegeben ist, sind so etwas wie Links dort nicht nötig. Die Erweiterung unserer grafischen Links macht es möglich, grafische Symbole in einer bestimmten Art und Weise anzuordnen und in Beziehung zueinander zu setzen.

Analog zu einer textuellen Grammatik besteht eine *visuelle Grammatik* aus einem Startsatz und einer Menge von Regeln. Alle Sätze, die aus dem Startsatz durch Hintereinanderanwendung von Regeln erzeugt werden können, bilden unsere visuelle Sprache.

Mehr Material über die Verbindung von attributierter Graphtransformation und visuellen Sprachen findet sich in [BTMS, Roz99].

#### 3.2.2.1 Diagramme

Ein *Diagramm* oder *Satz* einer visuellen Sprache ist ein attributierter Graph. Die Struktur dieses Graphen hängt allerdings vom gewählten Alphabet ab, wobei jedem Knotentyp genau eine Sorte und jedem Kantentyp genau eine Operation aus der abstrakten Syntax des Alphabets entspricht. Durch die Layoutabbildungen wird dann jeder Knoten des Graphen auf ein grafisches Symbol und jede Kante auf einen grafischen Link abgebildet. Ein *Diagrammmorphismus* ist dann nichts weiter als ein Graphmorphismus auf attributierten Graphen.

Durch die Theorie, die hinter dieser Instatiierung steht, ergibt sich ein Problem: Da die Algebra, die ein Graph zu einem Diagramm darstellt, total sein muss, muss jeder grafische Link in einem Diagramm *total* sein. Das bedeutet, dass für jedes Objekt, das

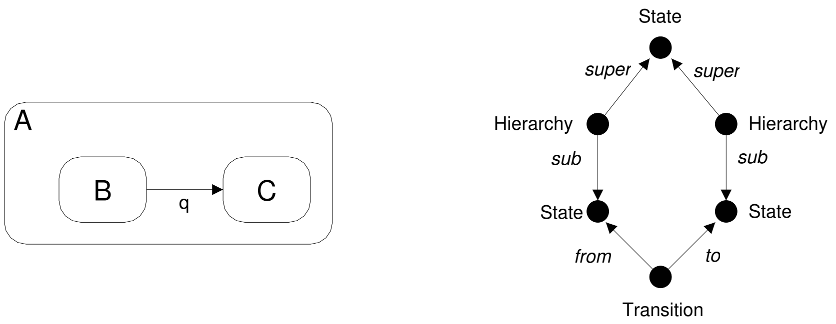



Abbildung 3.24: Statechart-Diagramm mit entsprechendem Graph

im Diagramm auftaucht, alle grafischen Links, die dieses Objekt als Quellobjekt haben, ebenfalls im Diagramm vorhanden sein müssen. Außerdem müssen für jeden grafischen Link die Zielobjekte im Diagramm sein. Genau aus diesem Grunde kann man Verbinder wie `Hierarchy` oder `Parallel` nicht als einfache grafische Links modellieren, da diese Verbinder optional sind, also auch wegfallen können.

#### 3.2.2.2 Regeln

Eine *Regel* einer visuellen Sprache besteht aus zwei Diagrammen $L$ und $R$ für die linke bzw. rechte Seite der Regel, einem Regelmorphismus $L \longrightarrow R$ sowie aus $0 \ldots n$ negativen Anwendungsbedingungen (NACs) mit den jeweiligen Morphismen $L \longrightarrow NAC_1, \ldots, L \longrightarrow NAC_n$.

Die *Anwendung* einer visuellen Regel sieht jetzt wie folgt aus:

**Ansatzsuche** : Zunächst wird die linke Seite über einen Morphismus in das Arbeitsdiagramm abgebildet, wobei alle negativen Anwendungsbedingungen erfüllt sein müssen. Dieser Schritt entspricht der Ansatzsuche für die Graphtransformation.

**Löschen** : Aus dem Arbeitsdiagramm werden nun alle Symbole und Links gelöscht, die in $L$, aber nicht in $R$ auftauchen. Auch die Constraints zu den Symbolen und Links werden aus der Menge der Constraints entfernt.

**Erhalten** : Die Symbole und Links, die vom Regelmorphismus abgebildet werden, werden in das neue Arbeitsdiagramm übernommen. Dabei bleiben auch ihre grafischen Maße wie Position und Größe erhalten.

**Hinzufügen** : Jetzt werden dem neuen Diagramm noch die Symbole und Links neu hinzugefügt, die in $R \setminus L$ sind. Neu hinzugefügte Symbole bekommen initial das im Alphabet festgelegte Default-Layout, außerdem werden die internen Constraints der Constraintmenge des Diagramms hinzugefügt. Neu dazugekommene Links addieren ebenfalls ihre Constraints zur Menge der Diagrammconstraints.

**Lösen der Constraints** : Der letzte Schritt besteht darin, durch Constraint-Solving alle Maße der Symbole im neuen Diagramm so anzupassen, dass das Diagramm sowohl von der Graphstruktur als auch von den Constraints her korrekt ist.

Da die Verwendung von Attributen stark davon abhängt, welche Implementierung wir letztendlich verwenden, soll darauf erst im nächsten Abschnitt und in der Designbeschreibung des Systems eingegangen werden.

## 3.3 Toolkonzepte

Bislang haben wir die Möglichkeit, visuelle Sprachregeln zu bauen und auf Diagramme anzuwenden. Für ein Tool, mit dem wir visuelle Grammatiken erzeugen und bearbeiten können, ist es nun günstig, dem Benutzer einen Satz einfacher Regeln an die Hand zu geben, mit der man grafische Symbole in Diagramme einfügen und aus ihnen löschen kann. Dabei kann, da das grafische Alphabet mit Links und Symbolen gegeben ist, sichergestellt werden, dass die Bedingungen für grafische Links, die sich aus der Theorie ergeben[2], immer erfüllt sind, die Diagramme also – aus Sicht des Alphabets betrachtet – syntaktisch korrekt sind. Mit Hilfe dieser einfachen Regeln lassen sich in einer Art Bootstrapping-Verfahren dann wiederum weitere Regeln aufbauen, bis man einen Satz von Regeln konstruiert hat, der die Sprache aus einem (meist leeren) Startdiagramm korrekt aufbaut.

### 3.3.1 Generierung einfacher Regeln

Aus einem gegebenen Alphabet lassen sich einfache Sprachregeln für das Einfügen und Löschen von Symbolen generieren, indem man den im Folgenden dargestellten Algorithmus anwendet:

**Algorithmus 3.1** (*Erzeugen einfacher Regeln aus einem Alphabet*)

Für jedes Symbol aus dem Alphabet:

**Einfügeregel**:

1. Erzeuge eine leere Regel.
2. Füge das Symbol[3] auf der rechten Regelseite ein.
3. Für jeden grafischen Link, der das Symbol als Quellsymbol hat:
   (a) Füge das Zielsymbol der Operation rechts und links ein.
   (b) Füge den graphischen Link selbst auf der rechten Seite ein, so dass er das Quell- und das Zielsymbol miteinander verbindet.
   (c) Erweitere den Regelmorphismus um eine Abbildung vom Zielsymbol auf der linken auf das Zielsymbol auf der rechten Seite.
   (d) Vervollständige die Regel für das eingefügte Zielsymbol (Das wird weiter unten erklärt!).

[2]Das bedeutet, dass die Abbildungen, die durch grafische Links gebildet werden, immer total sein müssen. Demnach muss für jedes Symbol, das als Quelle eines Links in einem Diagramm auftaucht, das Zielsymbol und der Link selbst auch definiert sein.

[3]Immer, wenn hier ein Symbol bzw. ein Link eingefügt wird, wird in den Diagrammgraphen auch der entsprechende Knoten bzw. die entsprechende Kante eingefügt.

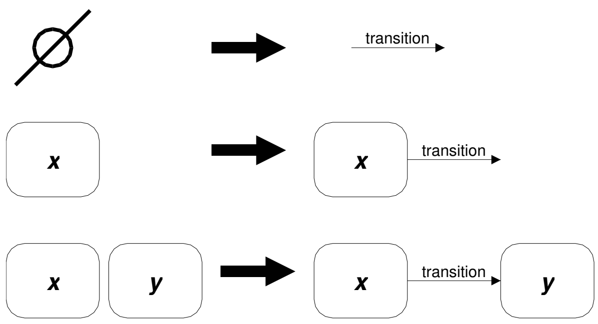



Abbildung 3.25: Generierung der Regel „Transition einfügen“

**Löschregel**:

1. Erzeuge eine leere Regel.
2. Füge das Symbol auf der linken Regelseite ein.
3. Für jeden grafischen Link, der das Symbol als Quellsymbol hat:
   (a) Füge das Zielsymbol der Operation rechts und links ein.
   (b) Füge den graphischen Link selbst auf der linken Seite ein, so dass er das Quell- und das Zielsymbol miteinander verbindet.
   (c) Erweitere den Regelmorphismus um eine Abbildung vom Zielsymbol auf der linken auf das Zielsymbol auf der rechten Seite.
   (d) Vervollständige die Regel für das eingefügte Zielsymbol (Das wird weiter unten erklärt!).
4. Für jeden grafischen Link, der das Symbol als Zielsymbol hat:
   (a) Erzeuge eine negative Anwendungsbedingung.
   (b) Füge die Operation und das Symbol in die NAC ein.
   (c) Erweitere den NAC-Morphismus um eine Abbildung vom Symbol in der linken Regelseite zum Symbol in der NAC.
   (d) Füge das Quellsymbol des Links in die NAC ein.
   (e) Vervollständige die NAC für die eingefügten Quellsymbole (Das wird weiter unten erklärt!).

Die Vervollständigung für ein Symbol ist rekursiv und bewirkt dasselbe wie die Einfügeregeln: Wenn es noch einen Link mit dem Symbol als Quellsymbol gibt, der nicht im Diagramm ist (bzw. in der rechten und linken Regelseite), wird dieser Link und das Zielsymbol eingefügt (und der Morphismus erweitert, falls eine Regel vervollständigt wird). Dann muss das Diagramm (bzw. die Regel) wieder für das Zielsymbol vervollständigt werden usw.

Diese Vervollständigung birgt Gefahren: Wenn es irgendwelche Zyklen im Typgraphen des Alphabets gibt, läuft der Algorithmus unendlich weiter, da immer wieder neue Operationen mit Zielsymbolen aneinandergehängt werden. Das ist einer der

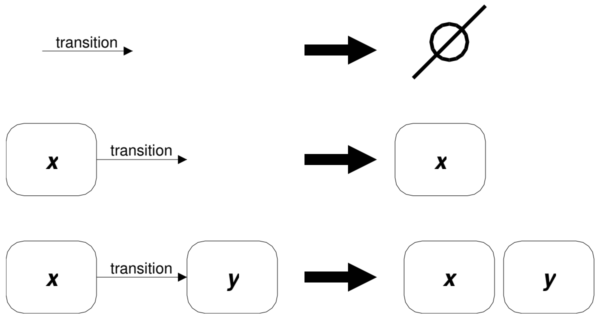



Abbildung 3.26: Generierung der Regel „Transition löschen“

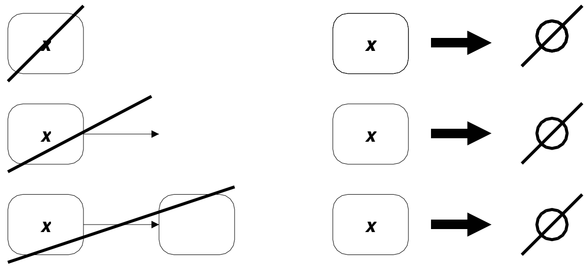



Abbildung 3.27: Generierung der Regel „Zustand löschen“ (mit einer NAC)

Gründe dafür, dass wir in der Fallstudie keinen Link `Hierarchy: State -> State` eingeführt haben. Solche Links müssen immer als Verbinder dargestellt werden, mit zwei Links, deren Quelle der Verbinder ist. Es ist möglich und angeraten, den Typgraphen auf einfache Weise auf Zyklen zu checken und damit das Alphabet zu validieren, bevor der Algorithmus die einfachen Regeln erzeugt. Dieser „Zyklen-Check“ kann auf unterschiedliche Weise implementiert werden, ein Beispiel befindet sich in der Implementierung (Kap. 6).

△

Durch diesen Algorithmus kann schon eine sehr schlichte und bestenfalls ungenaue Version der eigentlichen Sprache erstellt werden. Es lassen sich jedoch weitere Regeln aufbauen, indem man diese einfachen Regeln auf Regelseiten und NACs von Arbeitsregeln anwendet und die Morphismen der Arbeitsregeln entsprechend erweitert.

### 3.3.2 Korrektheit von Diagrammen

Unter *korrekten* Diagrammen verstehen wir solche, die im Sinne der Signatur korrekt sind. Das heisst, das für ein Symbol im Diagramm alle Links vorhanden sein müssen, die dieses Symbol als Quellsymbol haben können.

Durch den eben vorgestellten Algorithmus kann schon sichergestellt werden, dass mit den generierten einfachen Regeln nur korrekte Diagramme konstruiert werden können. Allerdings ist es möglich, mit Hilfe der einfachen Regeln weitere visuelle Regeln zu erzeugen, die **nicht** korrekte Diagramme erzeugen. So ist es möglich, mit unseren Statecharts-Regeln eine leere Regel derart zu bearbeiten, dass die rechte Seite leer bleibt, auf der linken Seite dagegen ein Zustand eingefügt wird (mit der Regel „Zustand einfügen“, siehe 3.28). Die so erhaltene Regel macht genau dasselbe wie die generierte Regel – sie enthält aber keine negativen Anwendungsbedingungen, die verhindern, dass beispielsweise ein Zustand gelöscht wird, an dem noch eine Transition hängt.

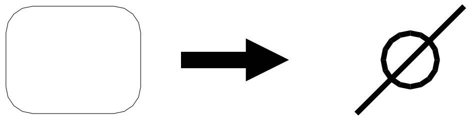


Abbildung 3.28: Regel „Zustand löschen ohne Schutz“

Da es uns nicht möglich ist, alle diese Fälle abzufangen, in denen der Benutzer eine Regel konstruieren kann, mit der ein nicht korrektes Diagramm erzeugt werden kann, müssen wir auf eine andere Methode zurückgreifen. Wenn durch irgendeine Regelanwendung ein nicht korrektes Diagramm entsteht, muss die Regelanwendung auf irgendeine Weise rückgängig gemacht werden und der Benutzer darüber informiert werden, dass die Regel nicht anwendbar war.

# Kapitel 4

# Design des Systems

In diesem Kapitel sollen zunächst die Anforderungen an unser System genauer spezifiziert werden, und zwar auf Basis der im letzten Kapitel vorgestellten Toolkonzepte. Dann wird mit der Unified Modeling Language eine Diagrammsprache erklärt, die wir im Weiteren für das Design und die Beschreibung der Implementierung verwenden. Anschließend wird ein Überblick über das Design des Systems, wie es sich aus den Anforderungen ergibt, gegeben. Dabei ist zu beachten, dass alles, was implementierungs*abhängig* ist, also konkrete Klassen und Algorithmen, erst in Kapitel 6 besprochen wird.

## 4.1 Anforderungen an einen Grammatikeditor

Aus den im letzten Kapitel genannten Konzepten und aus der Fallstudie ergeben sich nun etwas genauere Anforderungen an den zu implementierenden Grammatikeditor.

### 4.1.1 Datenstruktur

Da es sich bei dem zu implementierenden System um einen Grammatikeditor handelt, muss ein Alphabet schon vorhanden sein. Es muss uns also möglich sein, ein Alphabet in den Editor zu importieren. Dieses Alphabet muss folgendes bereitstellen:

- eine Menge von grafischen Symbolen, bestehend aus grafischen Primitiva und Constraints, wobei es möglich sein muss, diese Symbole zu instantiieren,
- eine Menge von grafischen Links mit den entsprechenden Link-Constraints, ebenso instantiierbar.

Der Grammatikeditor soll nun mit Hilfe des in Abschnitt 3.3 vorgestellten Algorithmus' in der Lage sein, einfache Einfüge- und Löschregeln zu generieren. Seine Datenstrukturen müssen also folgende Bestandteile haben:

- Diagramme, die aus einer Menge von Symbol- und Link-Instanzen bestehen,

- Morphismen, die Objekte und Links in Diagrammen aufeinander abbilden,
- Regeln, bestehend aus mindestens zwei Diagrammen für die linke und rechte Seite und einem Regelmorphismus (und zusätzlich $0 \ldots n$ NAC-Diagrammen und NAC-Morphismen),
- ein Startdiagramm und
- eine visuelle Sprache, bestehend aus einer Auswahl von Regeln und einem Startdiagramm.

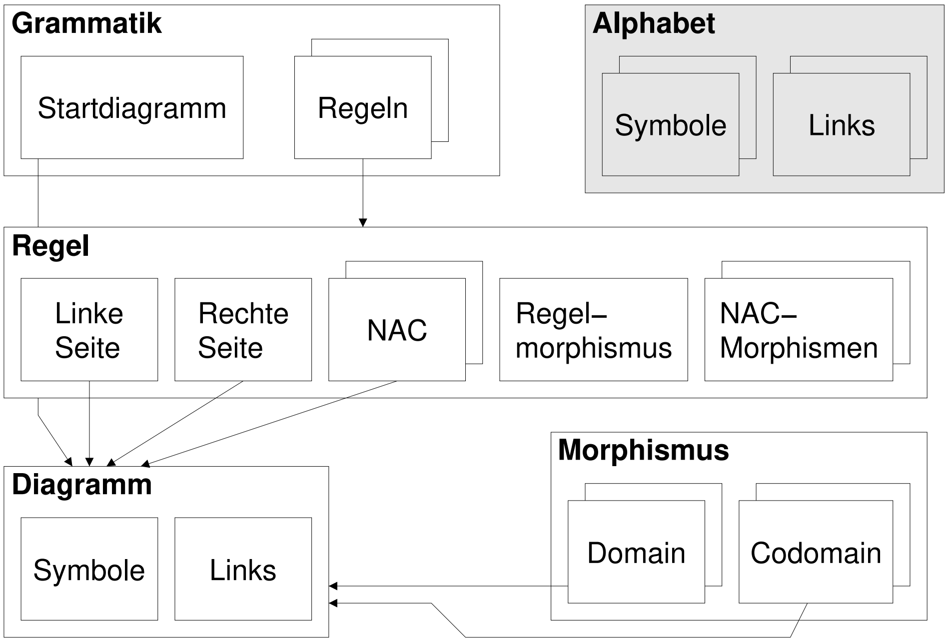



Abbildung 4.1: Strukturierung des Grammatikeditors

### 4.1.2 Funktionalität

Weiter spezifizieren wir auf Basis der o.g. Datenstrukturen die Funktionalität des Editors. Folgende Grundfunktionen sollen möglich sein:

- Importieren eines Alphabets,
- Generieren eines Satzes einfacher Lösch- und Einfügeregeln aus dem Alphabet,
- Bearbeiten von weiteren Regeln, indem schon vorhandene Regeln auf linke/rechte Regelseiten und NACs angewendet werden, im Zusammenhang damit
- Bearbeiten von Morphismen,
- Bearbeiten des Startdiagramms,

- Speichern von Regeln und Starrtdiagramm,
- Export von ausgewählten Regeln und dem Startdiagramm in eine Exportgrammatik.

Für eine grafische Benutzeroberfläche ergeben sich dann folgende Zusatzfunktionen:

- Anzeige von Diagrammen und Regeln mit dem Lösen von grafischen Constraints, dabei
- Anzeige von Diagrammen in einer Zeichenfläche mit den Funktionen Skalieren/Bewegen für grafische Symbole,
- Visualisierung von Morphismen,
- Übersicht über die Grammatik.

## 4.2 Unified Modeling Language

Wir werden für Design und Implementierung des Grammatikeditors in erster Linie eine Modellierungssprache verwenden, die *Unified Modeling Language* (UML). Hierbei handelt es sich um eine Spezifikations- und Beschreibungssprache für objektorientierte Softwaresysteme. UML ist Mitte der 90er Jahre von Booch, Rumbaugh und Jacobsen als Zusammenfassung der bisherigen Ansätze und Notationen für objektorientierte Softwareanalyse und -design entwickelt worden und hat sich seitdem zu einem Quasi-Standard in der OO-Gemeinde entwickelt [Rat97c, Rat97b, Rat97a, EFLR98].

UML ist aus unserer Sicht eine Sammlung von visuellen Sprachen, deren Elemente wie z.B. Klassen oder Objekte in mehreren Diagrammarten verwendet werden. Verschiedene Diagramme werden verwendet, um das zu entwickelnde System unter mehreren Aspekten zu betrachten. Wir wollen hier die UML-Sprachen vorstellen, die wir benutzen werden, um das System *Grammatikeditor* zu entwickeln. Dabei wollen wir uns aber auf eine informelle Beschreibung zurückziehen und nicht etwa wieder komplette Sprachbeschreibungen liefern. Das sollte dem Leser nach Lektüre dieser Arbeit als Übung überlassen bleiben.

Die von uns verwendeten Diagrammarten sind folgende:

**Use-Case-Diagramm** Diese Diagrammart beschreibt die Relationen zwischen Akteuren (außerhalb des Systems stehenden Komponenten) und dem System selbst. Dabei werden mögliche Aktionen durch Use-Cases (Transaktionen) beschrieben.

**Klassendiagramm** Diese Diagrammart gehört zur Gruppe der *Static-Structure-Diagramme* und bietet die Möglichkeit, wie schon der Name sagt, die statische Struktur des Systems und seiner Komponenten in Form von Klassen und deren Beziehungen (Vererbung, Aggregation etc.) zu beschreiben.

**Objektdiagramm** Objektdiagramme sind eine Form von Klassendiagrammen, die nur *Objekte*, also Instanzen von Klassen beschreiben. Diese Diagramme werden verwendet, um Datenstrukturen zu einer bestimmten Zeit genauer zu spezifizieren.

**Sequenzdiagramm** Diese Diagramme ermöglichen es, den dynamischen Anteil des Systems zu beschreiben.

Es ist möglich und üblich in UML, von einzelnen grafischen Symbolen (Objekt, Klasse, Assoziation u.ä.) *Stereotypen* abzuleiten, die es erlauben, quasi dynamisch eigene Untersymbole mit besonderer Bedeutung zu erzeugen. So gibt es beispielsweise das Symbol `Klasse` mit den Stereotypen `<<abstract>>` und `<<interface>>`. Stereotypen werden immer in doppelte spitze Klammern eingeschlossen.

### 4.2.1 Use-Case-Diagramme

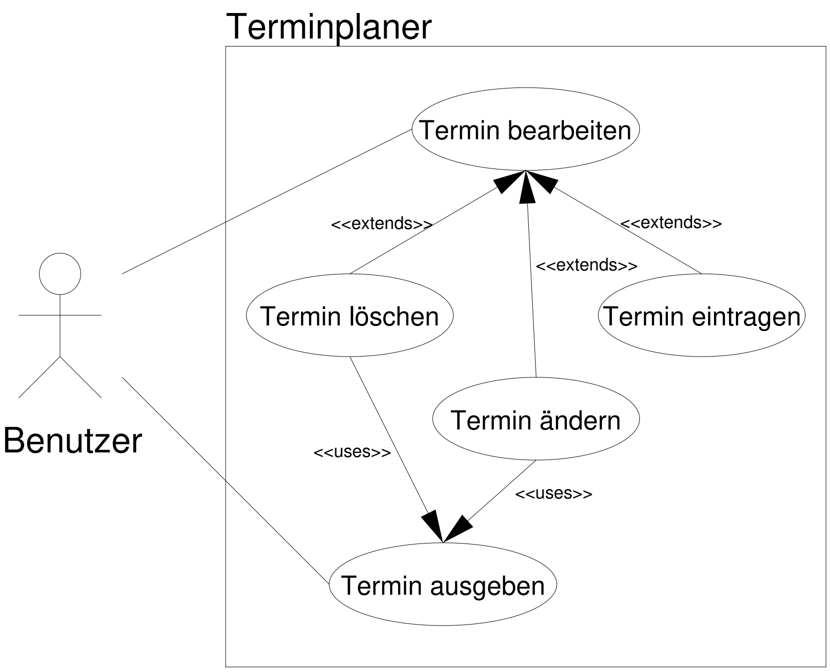



Abbildung 4.2: Use-Case-Diagramm: Beispiel für einen Terminplaner

Ein **Use-Case-Diagramm** besteht aus einer Menge von Akteuren sowie einer Menge von Use-Cases. Das System wird durch eine Grenze markiert, die die Use-Cases umfasst. Ein Akteur wird durch eine Strichmännchenzeichnung dargestellt, ein Use-Case durch eine Ellipse mit eingebundenem Text. Es gibt drei verschiedene Relationen zwischen den Objekten:

- Die *communicates*-Relation ist die einzige Relation zwischen Akteuren und Use-Cases. Sie wird durch eine durchgezogene Linie dargestellt.

- Die *uses*-Relation bezeichnet eine Aggregation von Use-Cases. Wenn ein Use-Case die Funktionalität eines zweiten benutzt, wird ein Pfeil auf den benutzten Use-Case gezogen.
- Die *extends*-Relation bezeichnet die Erweiterung der Funktionalität bzw. deren Spezialisierung. Der Pfeil führt vom spezialisierten Use-Case zum allgemeineren.

### 4.2.2 Klassendiagramme

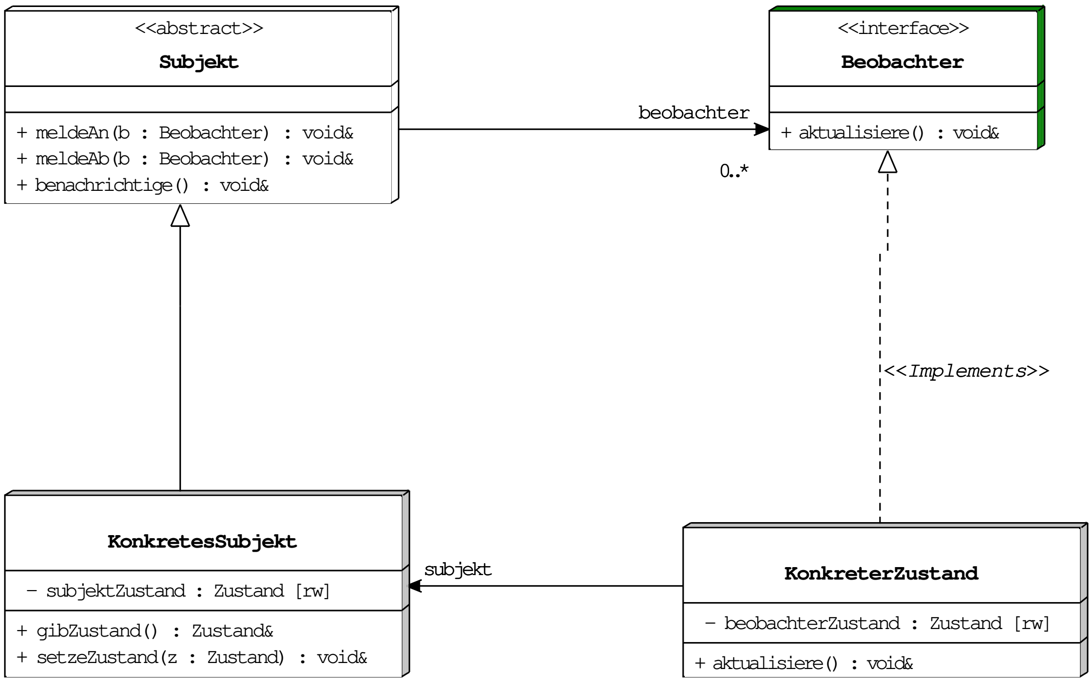



Abbildung 4.3: Klassendiagramm: Entwurfsmuster Beobachter

Die wichtigsten Elemente in einem **Klassendiagramm** sind *Klassen*, *Interfaces* sowie *Packages*. **Klassen** sind dreigeteilte Rechtecke, in deren Kopf fettgedruckt der Name der Klasse steht. Darunter befindet sich eine Liste von Attributen der Klasse und im untersten der drei Teile sind die Methoden der Klasse aufgelistet. Abstrakte Klassen (also Klassen, die als Templates für Unterklassen dienen) und Interfaces erhalten den zusätzlichen Bezeichner `<<abstract>>` bzw. `<<interface>>`. Attribute sowie Methoden können als öffentlich (*public*), geschützt (*protected*) und privat (*private*) deklariert werden, dazu werden ihnen die Zeichen +, # bzw. − vorangestellt. Typen von Methoden und Attributen werden mit `:type` nachgestellt, abstrakte Methoden kursiv gesetzt und Klassenattribute und -methoden (*static*) unterstrichen.

**Vererbungsbeziehungen** werden durch einen Pfeil mit einer Dreiecksspitze beschrieben, die Spitze zeigt zur Oberklasse. Eine Oberklasse kann mehrere Unterklassen besitzen, die alle an derselben Vererbungsbeziehung teilhaben. Ebenso wird auch die Implementierung eines Interfaces als Vererbungsbeziehung dargestellt, allerdings mit einer gestrichelten Linie.

Es können noch weitere **Assoziationen** zwischen den Klassen beschrieben werden. In der allgemeinsten Form wird eine solche Assoziation als einfacher Pfeil beschrieben, an dessen Anfang und Ende Rollenbezeichnungen und Kardinalitäten angegeben werden können. Die Rolle gibt an – wie schon der Name sagt – welche Rolle das so bezeichnete Element in der Assoziation spielt, die Kardinalität bezeichnet die Anzahl der Elemente, die an der Assoziation teilnehmen (können). Wir werden Assoziationen hauptsächlich dafür verwenden, Attribute von Klassen, die selbst wieder Objekte sind, außerhalb der Klasse zu schreiben. So hält in unserem Beispiel ein Subjekt eine Referenz auf alle seine Beobachter, was mit der Rolle `beobachter` und der Kardinalität `0..*` beschrieben ist.

Weitere spezielle Assoziationen sind die **Aggregation** und die **Komposition**. Hierbei handelt es sich um *besteht-aus*-Relationen zwischen Klassen. Die Komposition wird durch eine ausgefüllte Raute an der Kompositklasse dargestellt, wobei Komposition eine Verschärfung der Aggregation dahingehend darstellt, dass die Lebensdauer der Elemente vom der des Kompositums abhängt: Stirbt das Gesamtobjekt, sterben auch die Elemente. Aggregation wird durch eine nicht ausgefüllte Raute symbolisiert. Die Kardinalität ist auf der Seite des Kompositums in beiden Fällen 1.

Ein **Package** ist eine Zusammenfassung von Interfaces und Klassen zu einem Ganzen. Packages bestimmen den Namensraum von Klassen, wie es z.B. in Java üblich ist. So hat die Klasse `Beobachter` im Package `util` eigentlich den Namen `util.Beobachter`. Packages können ihrerseits wieder Subpackages enthalten. In unseren Diagrammen werden importierte Klassendiagramme immer bestimmten Packages zugeordnet. Importierte Klassen, also solche, die nicht aus dem aktuellen Klassendiagramm stammen, werden grau hinterlegt. Die *benutzt*-Beziehung zwischen zwei Packages kann durch einen gestrichelten Pfeil dargestellt werden.

### 4.2.3 Objektdiagramme

**Objektdiagramme** sind nichts weiter als eine starke Einschränkung von Klassendiagrammen. An die Stelle der Klassen und Interfaces treten Objekte, die als Instanzen von einer bestimmten Klassen den Namen `Objekt:Klasse` bekommen. Vererbungen und Packages fallen weg, was bleibt, sind Assoziationen, Aggregation und Komposition. Wir verwenden in der Designphase zur Darstellung der Datenstrukturen nur Objekt- und keine Klassendiagramme, da es so möglich ist, das Design des Systems implementierungsunabhängig zu halten. So geben wir auch noch keine konkreten Klassen für Objekte an, sondern halten die Struktur offen und eher informell.

### 4.2.4 Sequenzdiagramme

In den bisher vorgestellten Diagrammsprachen sind bislang nur die statischen Anteile des zu beschreibenden Systems zur Sprache gekommen. Mit den **Sequenzdiagram-**

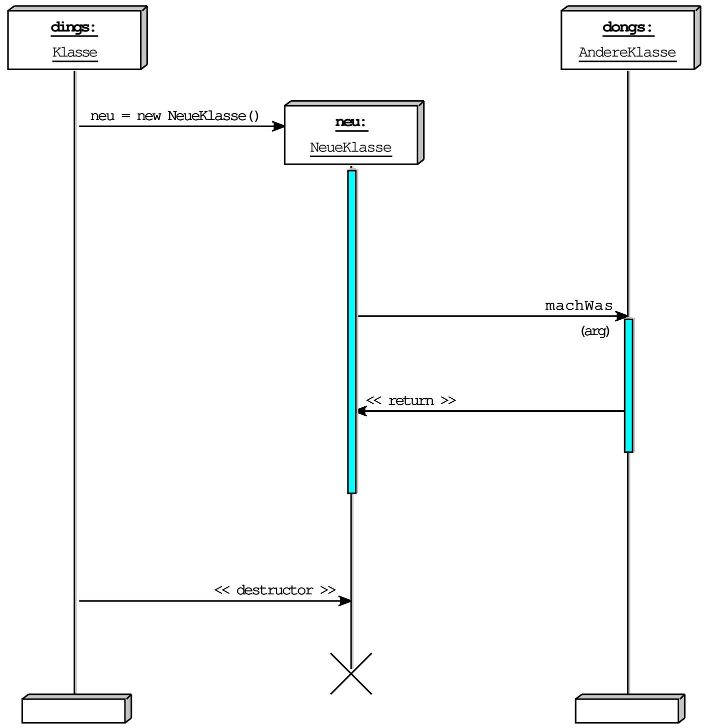



Abbildung 4.4: Sequenzdiagramm: Beispiel

**men** haben wir jetzt die Möglichkeit, dynamische Aspekte wie die Interaktion, das Erzeugen und Löschen von Objekten über der Lebensdauer des Systems darzustellen. Sequenzdiagramme sind meistens Szenarien, sprich Instanzen von Use-Cases zugeordnet.

Objekte werden – wie ihre Klassen – in einem Rechteck dargestellt, im Rechteck taucht der Objektname (optional) und, durch einen Doppelpunkt getrennt, der Klassenname auf. Beide werden unterstrichen. Entlang der senkrechten Achse des Diagramms wird die Zeit abgetragen und jedes Objekt erhält eine eigene vertikale „Lebenslinie“. Das Ende der Lebensdauer wird mit einem „X“ markiert. Objekte rufen gegenseitig oder auf sich selbst Methoden auf, die durch Pfeile zum jeweiligen Zielobjekt dargestellt werden. Es gibt besondere Stereotypen für Konstruktoren, Destruktoren und die Rückgabe von Methoden. Solange eine aufgerufene Methode abgearbeitet wird, gibt es auf der Lebenslinie des entsprechenden Objekts einen Methodenfokus, der durch eine schmale Box symbolisiert wird.

### 4.2.5 Verwendete Tools

Um das Design und die Erstellung von UML-Diagrammen für diese Arbeit zu vereinfachen, verwenden wir – wo immer möglich[1] – das Tool *SoftModeler*™ der Firma Softera Ltd. Es handelt sich dabei um ein in Java implementiertes Entwicklungswerkzeug, das Analyse, Design und Implementierung von objektorientierten Systemen unterstützt. Das geschieht durch eine Reihe von speziell auf die Diagrammarten zugeschnittenen Editoren. Es ist zudem möglich, mit dem SoftModeler einerseits implementierte Klassen zu analysieren und daraus Klassendiagramme zu erzeugen, andererseits aber auch Java-Quellcode aus Klassendiagrammen zu exportieren. Das Tool ist im Betastadium frei verfügbar und für Bildungseinrichtungen auch in der Endversion kostenlos. Weitere Informationen erhält man unter der URL `http://www.softera.com`.

Es ist natürlich denkbar, mit der fertigen Version des hier beschriebenen Grammatikeditors denselben zu beschreiben, dann muss man sich allerdings darüber im Klaren sein, dass alle UML-Beschreibungssprachen, die hier vorgestellt wurden, zunächst designt und implementiert werden müssen.

## 4.3 Die logische Ebene

Wir werden unser Tool in zwei Ebenen betrachten, die sich aus den Sprachkonzepten ergeben: Die logische sowie die grafische Ebene. Es ist – wie wir später sehen werden – angemessen, alles im Editor, was sich mit Grafiken, Constraints und dergleichen beschäftigt, von der reinen Struktur der visuellen Sprache zu trennen.

Die *logische Ebene* ist für die Graphen zuständig, die den Diagrammen zugrundeliegen. Auf dieser Ebene betrachten wir also die reine Struktur der Diagramme und Regeln, unabhängig von der tatsächlichen grafischen Darstellung und etwaiger vorhandener Constraints.

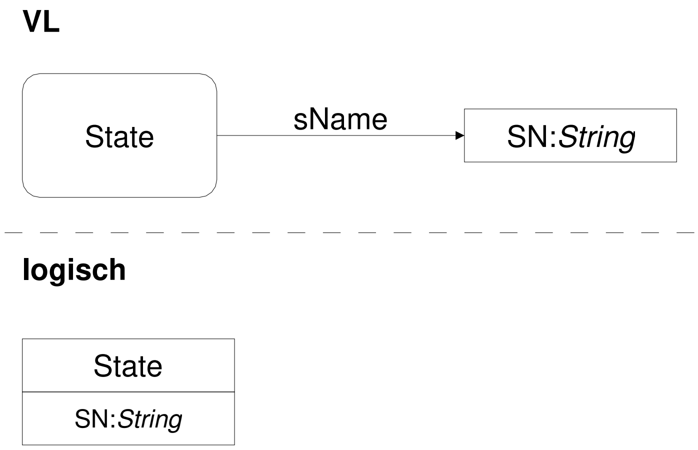



Abbildung 4.5: Datentypen auf der logischen Ebene

[1]Wie sich herausstellte, ist ein spezieller Grafikeditor auch nicht besser als ein allgemeiner, so mussten wir dann doch sehr oft auf die Hilfe von StarOffice™ zurückgreifen!

Wir nehmen aus praktischen Gründen (wie sich bei der Implementierung ergeben wird) die Datentypen mit in die logische Ebene hinein. Eine der Änderungen, die wir im Hinblick auf das System machen, besteht darin, Datentypen nicht als gesonderte grafische Symbole, sondern auf dieser Ebene als zum Attribut eines Knotens gehörig zu betrachten (siehe Abb. 4.5). Die Grafik eines Datentypen dagegen wird hier nicht betrachtet, sondern nur der Wert. Im Weiteren werden wir die Datentypen außerdem direkt in die entsprechenden Symbole schreiben, um die Übersichtlichkeit zu wahren.

Da jeder Knoten nur ein Attribut besitzen kann, ein grafisches Symbol jedoch unter Umständen mehrere Datentypen besitzen kann, müssen wir die Attributtypen so gestalten, dass sie mehrere Attributtupel aufnehmen können. So besteht in einem Klassendiagramm beispielsweise das Attribut eines Knotens, der einer Klasse entspricht, aus den drei Datentypen `ClassName:String`, `MethodList:List` und `AttributeList:List`. In unserer Fallstudie enthält jedes Attribut maximal einen Datentypen, ein Zustand beispielsweise hat genau einen Datentypen `StateName:String`.

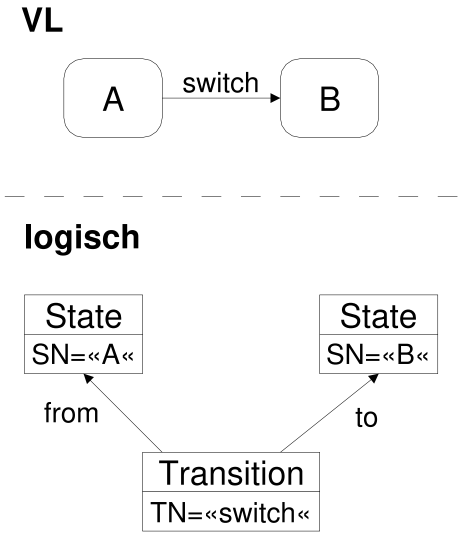



Abbildung 4.6: Diagramme auf der logischen Ebene

Symbole werden auf dieser Ebene also durch Knoten mit Attributen dargestellt, grafische Links werden auf Kanten abgebildet. Auf dieser Ebene findet die gesamte Regelanwendung statt: Die linke Regelseite wird auf ein Arbeitsdiagramm gematcht, die Datentypvariablen ggf. belegt und die Transformation wie in Abschnitt 2.3 durchgeführt. Da die Transformation nach dem Single-Pushout-Verfahren stattfindet, werden hängende Kanten (also Links) nach der Transformation gelöscht.

Die Datenstruktur auf der logischen Ebene sieht also wie in Abb. 4.8 und 4.9 dargestellt aus: Eine Grammatik besteht aus Regeln, diese wiederum aus Morphismen und Diagrammen, die ihrerseits aus Knoten und Kanten bestehen. Als nächstes müssen wir diese Graphstrukturen geeignet visualisieren, das geschieht in der grafischen Ebene.

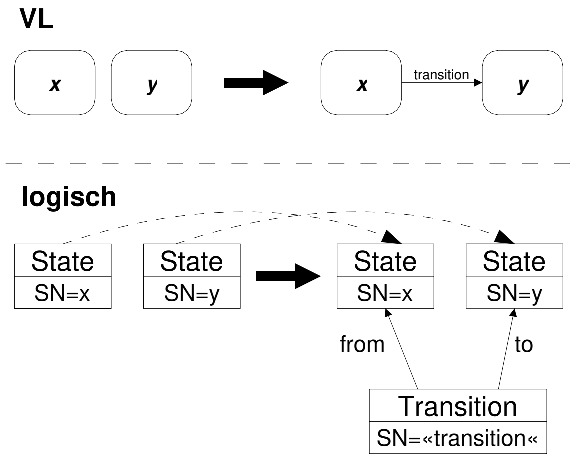



Abbildung 4.7: Regeln auf der logischen Ebene

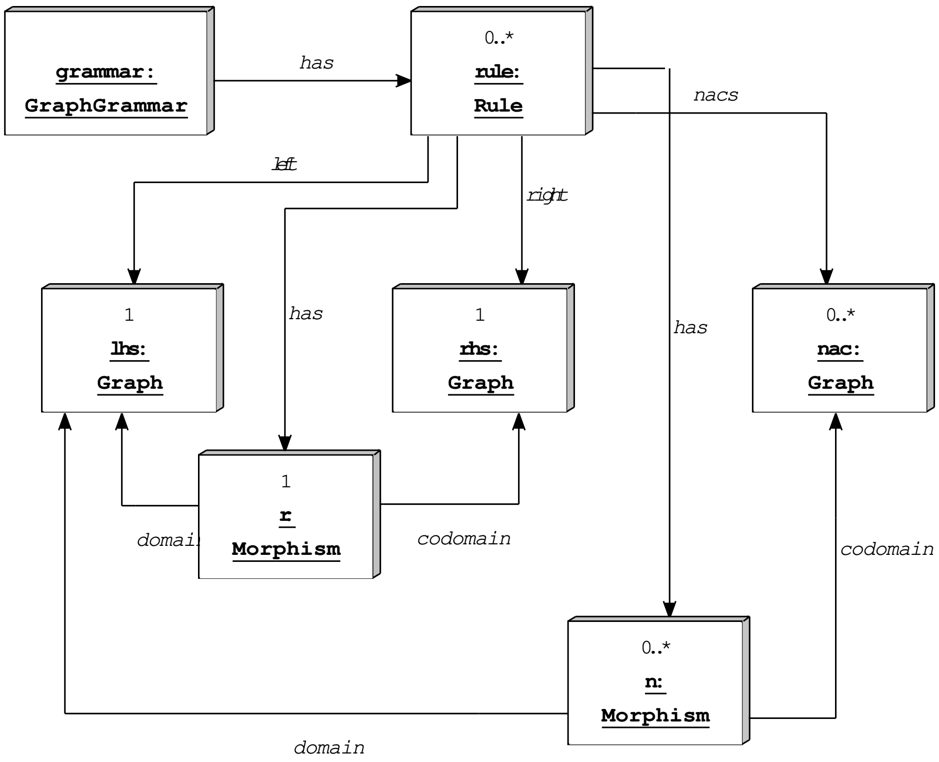



Abbildung 4.8: Objektdiagramm: Datenstrukturen auf der logischen Ebene

## 4.4 Die grafische Ebene

Auf der grafischen Ebene befinden sich nun alle Informationen über die visuelle Sprache, die auf der logischen Ebene vernachlässigt wurden. Dazu gehören die grafischen Primitiva, die Constraints und die Visualisierung der Datentypen.

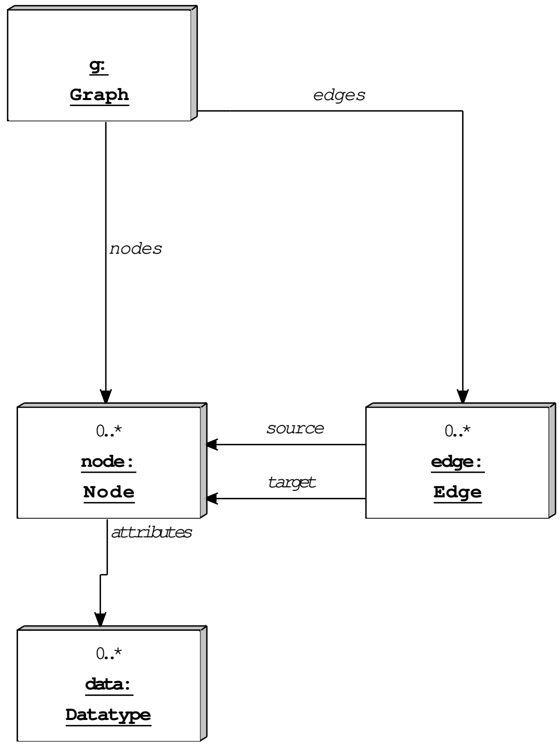



Abbildung 4.9: Objektdiagramm: Ein Graph auf der logischen Ebene

Da die Aufgabe, einen Editor für visuelle Alphabete zu implementieren, an uns vorübergegangen ist, können wir davon ausgehen, dass wir eine Struktur importieren können, die vordefinierte grafische Symbole mit Default-Werten sowie grafische Links und Constraints enthält. Ein solches Alphabetsymbol bezeichnen wir auch als *Vorlage* (engl. Template). Es gibt Vorlagen sowohl für die grafischen Symbole, die dann die Default-Symbole und die Default-Datentypen darstellen ebenso wie für die grafischen Links. Außer den Default-Werten, enthalten diese Vorlagen alle Constraints.

Was passiert, wenn wir – im Laufe einer Regelanwendung – ein Symbol in ein Diagramm einfügen? Wir bilden eine *Instanz* von der Vorlage, was bedeutet, dass ein neues Objekt erzeugt wird und die Constraints aus der Vorlage kopiert und den Diagrammconstraints hinzugefügt werden.

Was uns nun zufällt, ist die Aufgabe, anhand der Vorgabe des grafischen Alphabets eine logische Struktur aufzubauen, sprich eine initiale Graphgrammatik, und diese so mit den Instanzen von grafischen Symbolen und Links zu koppeln, dass eine eindeutige Zuordnung von Knoten zu Symbolen und Kanten zu Links erfolgen kann. Außerdem muss dafür gesorgt werden, dass zu jedem Zeitpunkt die grafischen Constraints für ein Diagramm erfüllt sind.

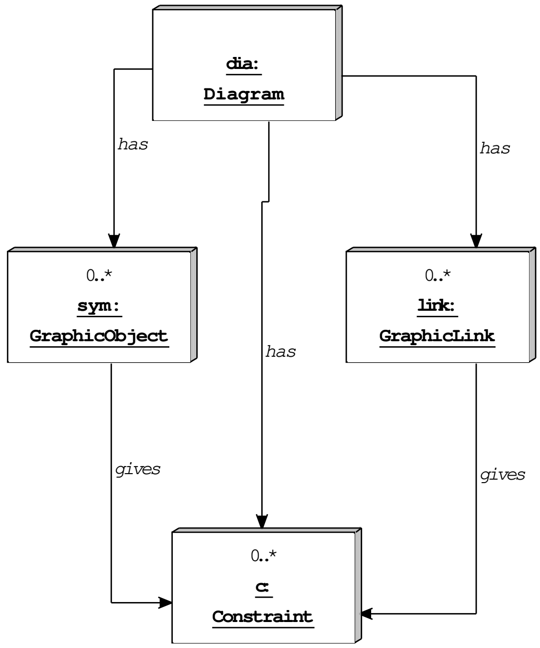



Abbildung 4.10: Objektdiagramm: Ein Diagramm auf der grafischen Ebene

Wir bauen also zunächst die benötigte Datenstruktur auf[2]; diese besteht aus einer visuellen Sprache, die ihrerseits visuelle Regeln enthält, die aus Morphismen und Diagrammen bestehen, ganz analog zur logischen Ebene. Der Unterschied besteht nun darin, dass Diagramme – abgesehen von der eindeutigen Zuordnung Symbol $\mapsto$ Knoten und Link $\mapsto$ Kante – eine Constraintmenge speichern müssen, die in sich konsistent sein muss und bei Änderung der grafischen Werte eines Symbols alle Werte, die in den Constraints festgehalten sind, derart angepasst werden müssen, dass alle Constraints erfüllt sind.

### 4.4.1 Transformation

Eine Regelanwendung muss also sicherstellen, dass sowohl Symbole als auch Links, deren zugehörige Knoten respektive Kanten gelöscht werden, mitgelöscht werden **und** (und das ist das Wichtigste) dass die Constraints aus der Constraintmenge des Arbeitsdiagramms verschwinden. Dann müssen für neu entstandene Knoten und Kanten die entsprechenden Symbole und Links neu erzeugt werden, deren Constraints der Constraintmenge hinzugefügt werden und zu guter Letzt alle Constraints gelöst werden.

Ein weiterer Aspekt ist die Korrektheit des transformierten Diagramms. Wie in Abschnitt 3.3.2 dargestellt, muss nach der Regelanwendung überprüft werden, ob das Dia-

[2]Wir verwenden hier ein Objektdiagramm, dessen Objekte allerdings keine konkreten Klassen besitzen, da das Design von der Implementierung unabhängig sein soll.

gramm korrekt im Sinne der Signatur ist und ggf. die Transformation rückgängig gemacht werden. Eine Transformation auf der grafischen Ebene sieht also wie folgt aus:

1. Sichere das Arbeitsdiagramm zur späteren Wiederherstellung.
2. Führe die Graphtransformation auf der logischen Ebene durch.
3. Überprüfe das transformierte Diagramm auf Korrektheit (wird später erklärt). Falls es nicht korrekt ist, stelle das ursprüngliche Arbeitsdiagramm wieder her.
4. Lösche alle Symbole aus dem Diagramm, deren Knoten nicht mehr im Graphen sind, zusammen mit den internen Constraints für diese Symbole.
5. Lösche alle Link-Constraints aus dem Diagramm, deren zugeordnete Kanten nicht mehr im Graphen sind.
6. Füge für alle Knoten im Arbeitsgraphen, die noch keine grafische Repräsentation haben, die entsprechenden Symbole und Constraints zum Diagramm hinzu.
7. Füge für alle Kanten im Arbeitsgraphen, die noch keinen zugehörigen Link haben, die entsprechenden Links und Constraints zum Diagramm hinzu.
8. Passe die Werte aller grafischen Objekte im Diagramm derart an, dass alle Constraints gelöst sind (Lösen der Constraints).

**Korrektheits-Überprüfung**

Um das transformierte Diagramm darauf zu überprüfen, ob es im Sinne der Signatur korrekt ist, müssen wir für jeden Knoten im transformierten Graph (auf der logischen Ebene) folgendes machen:

- Wir lassen uns den Knotentyp, also den Namen des Symbols zu diesem Knoten geben.
- Nun müssen wir für jeden Link, der dieses Symbol als Quelle hat, überprüfen, ob auch eine ausgehende Kante für diesen Link existiert.
- Wenn für irgendeinen Knoten eine wie eben dargestellte ausgehende Kante nicht existiert, ist das transformierte Diagramm nicht korrekt.

**Bau von Regeln mit einfachen Regeln**

Zu den Toolkonzepten aus Abschnitt 3.3 gehört auch, dass in unserem System neue Regeln nur entweder mit dem Algorithmus zu Regelerzeugung oder durch die Anwendung von Regeln auf Regelseiten oder NACs erzeugt bzw. – im zweiten Fall – gebaut werden. Das lässt sich so modellieren, weil ja sowohl die linke/rechte Regelseite als auch das Bild (Codomain) einer negativen Anwendungsbedingung Diagramme in unserer visuellen Sprache sind. Wir können beim Bau einer neuen Diagrammregel also wie folgt vorgehen:

- Wir erzeugen zunächst eine leere Regel ohne NACs. Leer bedeutet, dass sowohl die linke als auch die rechte Seite leer sind und der Regelmorphismus keine Abbildungen enthält.
- Nun können wir die vorhandenen Regeln (beispielsweise Einfügeregeln) auf beide Seiten anwenden, um dort Diagramme zu bauen.
- Ebenso können wir auch eine leere NAC erzeugen, mit einem leeren NAC-Morphismus von der linken Regelseite in das NAC-Diagramm, und dieses NAC-Diagramm mit Hilfe der vorhandenen Regeln aufbauen.
- Um die Definition abzuschließen, müssen wir den Regelmorphismus und die NAC-Morphismen (soweit vorhanden) per Hand angeben. Das bedeutet, dass wir Abbildungen von Symbolen auf Symbole und von Links auf Links angeben müssen.

Die so erzeugte Regel können wir in den großen Pool der verfügbaren visuellen Regeln aufnehmen, um mit ihr wiederum weitere Regeln zu bauen. Es sollte zudem die Möglichkeit geschaffen werden, die linke Seite eines Morphismus auf die rechte Seite zu kopieren und gleichzeitig den Morphismus durch die Abbildung der kopierten Elemente zu vervollständigen.

Alle Morphismen in unseren visuellen Regeln und auch der Ansatzmorphismus bei Anwendung einer Regel können nicht-injektiv sein. Es ist also durchaus erlaubt, mehrere Symbole auf ein zweites abzubilden.

### 4.4.2 Regelparameter und Datentypen

Wir wollen jetzt noch etwas genauer klären, wie mit den Datentypen bei Anwendung einer Regel verfahren wird und zusätzlich noch die Möglichkeit bieten, Datentypen in einem Diagramm durch Eingabe von außen zu ändern.

#### Datentypen

Wenn durch eine Regelanwendung ein neues Symbol erzeugt und dem Diagramm hinzugefügt wird, werden für dieses Symbol die Default-Werte des Alphabets genommen. Ebenso werden für die Datentypen des Symbols die Default-Werte aus der Vorlage kopiert. Auf der logischen Ebene werden Datentypen zwar im Attribut des entsprechenden Knotens gehalten, auf der grafischen Ebene aber werden die Symbole, die den Datentyp kapseln, durch einen grafischen Link mit dem Symbol verbunden. Jetzt stehen aber noch zwei weitere Probleme zur Lösung an:

1. Wie können Datentypen verändert werden?

2. Wie kann ein Regelansatz gefunden werden, wenn ein Datentyp **nicht** den Default-Wert enthält?

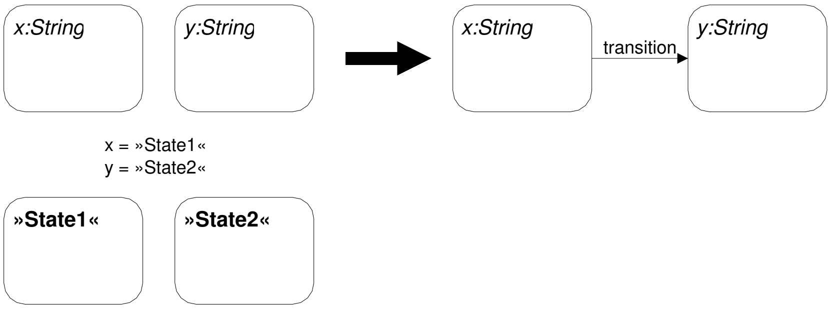



Abbildung 4.11: Ein Ansatz mit Datentypen

Das erste Problem wird dadurch gelöst, dass wir einerseits Variablen für Datentypen verwenden, wie in Abschnitt 3.1.1.7 angedeutet, andererseits Operationen auf diesen Datentypen verwenden, um deren Werte zu ändern. Die Verwendung von Variablen löst auch das zweite Problem. Wann immer in einer linken Regelseite eine Variable für einen Datentypen steht, wird diese Variable beim Ansatz mit dem Wert belegt, den der Datentyp des entsprechenden Symbols im Arbeitsdiagramm hat (siehe Abb. 4.11). Ergeben sich Konflikte bei der Belegung der Variablen, schlägt der Ansatz fehl, so z.B. dann, wenn dieselbe Variable mit verschiedenen Werten belegt werden müsste. Die Variablen werden nun entweder für sich oder in komplexeren Operationen auf der rechten Seite verwendet, um die Datentypen für die Symbole auf der rechten Seite aufzubauen. Eine Erweiterung: Ist der Wert für den Datentypen leer, entspricht dies einer Art „universeller Variable“, der Ansatz mit einem solchen Datentypen gelingt also immer. Konstanten werden auf Konstanten gematcht.

Zusammenfassend können wir also folgendes sagen:

- Auf der linken Seite der Regeln dürfen Datentypen entweder leer sein oder in Form von Variablen oder Konstanten auftreten.
- Auf der rechten Seite dürfen Datentypen in Form von Variablen, Konstanten oder komplexen Ausdrücken auftreten. Dabei ist zu beachten, dass alle Variablen, die auf der rechten Seite verwendet werden, auch links vorkommen.

Die Regelanwendung ändert sich wie folgt:

- Zuerst wird ein Ansatz der linken Regelseite auf der logischen Ebene gesucht, dabei werden Variablen und konstante Datentypwerte gematcht.
- Die Transformation auf der logischen Ebene findet wie gehabt statt. Danach werden die Datentypen im geänderten Graphen neu berechnet: Entweder werden Variablen durch ihre Belegung ersetzt und komplexere Ausdrücke ausgewertet oder Konstanten eingesetzt.

- Auf der grafischen Ebene findet die Transformation ebenfalls wie oben beschrieben statt. Vor dem Lösen der Constraints allerdings werden die Datentypsymbole mit den neuen Werten belegt. Dadurch ändert sich ggf. deren Größe, so dass erst, wenn alle Datentypen die neuen (berechneten oder konstanten) Werte haben, die Constraints gelöst werden können.

### Regelparameter

Was wir als nächstes lösen wollen, ist das Problem, dass wir Datentypwerte nicht nur automatisch erzeugen oder ändern, sondern deren Werte auch von außen festlegen wollen. So soll beispielsweise vom Benutzer ein Zustandsname abgefragt und dann in den Zustand eingesetzt werden. Das lösen wir, indem wir unsere Regeln durch *Regelparameter* erweitern.

Ein *Regelparameter* ist ein Paar, bestehend aus einem *Parameternamen* und dem *Parametertypen*. Der Parametertyp entspricht dabei dem Typen eines Datentypen (`String`, `List` etc.). Bei einer Regelanwendung wird dann nicht nur die Regel und der Ansatz, sondern zusätzlich auch ein Wert für den Parameter angegeben. Die Parametervariable kann dann – ganz analog zu den Datentypvariablen – auf der rechten Seite verwendet werden, da sie ja vor Regelanwendung belegt wurde. Wir können die Bedingungen für die Verwendung von Datentypen also wie folgt erweitern:

- Auf der linken Seite der Regeln dürfen Datentypen entweder leer sein oder in Form von Variablen oder Konstanten auftreten.
- Auf der rechten Seite dürfen Datentypen in Form von Variablen, Konstanten oder komplexen Ausdrücken auftreten. Dabei ist zu beachten, dass alle Variablen, die auf der rechten Seite verwendet werden, entweder links vorkommen oder als Regelparameter angegeben werden.

Nun lassen sich auch zusätzlich zu den automatisch generierten einfachen Einfüge- und Löschregeln für jeden Datentypen in jedem Symbol Änderungsregeln schreiben. Das geschieht mit folgendem Algorithmus:

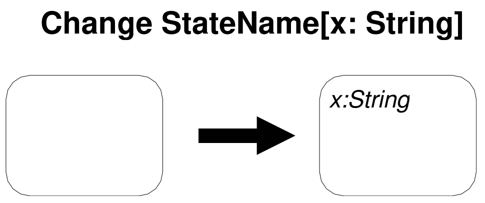



Abbildung 4.12: Die Regel „Zustandsnamen ändern“

Für jedes Symbol und jeden Datentyp in diesem Symbol:

**Änderungsregel**:

1. Erzeuge eine leere Regel.

2. Füge das Symbol auf der linken und rechten Regelseite ein.
3. Erweitere die Regel um einen Parameter x mit dem Typen des Datentypen.
4. Füge die Variable x auf der rechten Seite als Datentypvariable ein.
5. Lösche den Datentypwert auf der linken Seite.
6. Vervollständige die Regel für das eingefügte Symbol (wie oben erklärt).

# Kapitel 5

# Verwendete Tools

In diesem Kapitel werden die Hilfsmittel vorgestellt, die wir einsetzen, um einerseits die im letzten Kapitel dargestellte logische Ebene, also den Bereich der Graphtransformation einer visuellen Sprache aufzubauen und zu bearbeiten und um andererseits das Alphabet und den grafischen Anteil der Sprache darzustellen und zu bearbeiten. Dabei legen wir besonderen Wert auf die Beschreibung der Schnittstellen der jeweiligen Programme, da es denkbar ist, einzelne Komponenten wie z.B. den Constraintsolver durch andere zu ersetzen.

Aus dem Design und den Anforderungen für den Grammatikeditor ergibt sich, dass wir in großem Maß darauf angewiesen sind, externe Komponenten zu benutzen. Dazu gehört zum einen ein *Alphabeteditor*, im Prinzip ein Editor für Vektorgrafiken, der um die Eingabe und Verarbeitung von grafischen Constraints und Links erweitert wird. Dieser Editor liefert die gesamte grafische Basis für die Grammatiken, die wir bearbeiten und erstellen werden. Desweiteren benötigen wir ein Tool, das es uns ermöglicht, mit attributierten Graphen umzugehen, also Graphregeln und Graphen zu erzeugen und Transformationen mit gegebenen Regeln durchzuführen. Außerdem benötigen wir eine Constraint-Maschine, die wir mit unseren grafischen Constraints füttern können und die in der Lage ist, diese Constraints effizient zu lösen. Der Gesamtaufbau unseres Systems mit den Schnittstellen zu externen Komponenten stellt sich dann wie in Abb. 5.1 dar.

Wenn wir die externen Akteure, die mit unserem System arbeiten, betrachten, müssen wir immer zwischen zwei Arten von Benutzern unterscheiden. Der *Designer* einer visuellen Sprache ist derjenige, der sowohl den Alphabeteditor als auch den Grammatikeditor benutzt, um die Sprache aufzubauen. Bei dieser Person können wir einiges an Vorwissen über Programmierung, Constraints und Grammatiken voraussetzen. Der *Benutzer* einer visuellen Sprache hingegen bekommt nur das Endprodukt, also den syntaxgesteuerten Diagrammeditor zu einer bestimmten Sprache zu sehen. Dieser soll so benutzbar sein, dass man keine weiteren Vorkenntnisse benötigt, um Diagramme in der entsprechenden Sprache zu zeichnen.

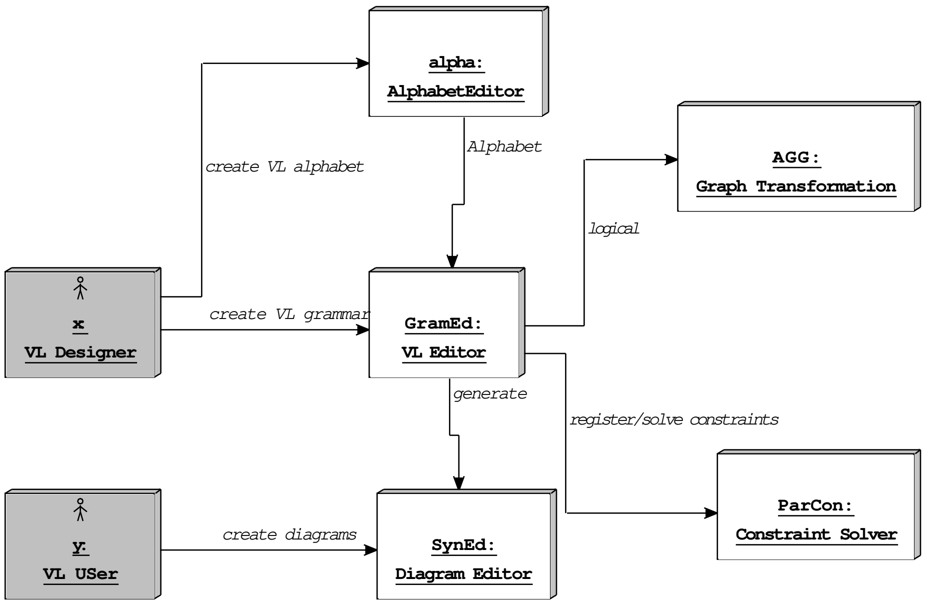



Abbildung 5.1: Objektdiagramm: Aufbau des Systems GRAMED

## 5.1 Das Graphtransformationssystem AGG

Für die Ebene der attributierten Graphtransformation greifen wir auf das AGG-System (AGG = Algebraische Graph-Grammatiken) zurück [TER]. Es wurde ursprünglich 1991 an der Technischen Universität Berlin in der Sprache Eiffel als Grapheditor entwickelt [Bey91, LB93] und letztendlich 1997 in Java reimplementiert [Rud97]. Das System besteht im Großen und Ganzen aus vier Komponenten:

**Graphmodell** Das System erlaubt den Aufbau einer Graphgrammatik mit Regeln und einem Arbeitsgraphen. Als Grundlage dienen ALR-Graphen [Löw93], mit denen herkömmliche attributierte Graphen modelliert werden.

**Attributierung** Knoten und Kanten in den Graphen können in Regeln mit Variablen und sogar ganzen Java-Programmen attributiert werden, wobei die Graphtransformation die Belegung von Variablen und die Ausführung der Programme steuert [Mel99].

**Transformation** Der Arbeitsgraph kann durch Regelanwendungen transformiert werden, die Transformation geschieht auf der Basis einer kategoriellen Bibliothek [Wol97] und ist damit konsistent mit der formalen Beschreibung.

**Ansatzsuche** Die Suche des Ansatzes einer linken Regelseite im Arbeitsgraphen wird als Constraint Satisfaction Problem formuliert, so dass zur Lösung dieses Ansatzes vorhandene Lösungsalgorithmen verwendet werden können.

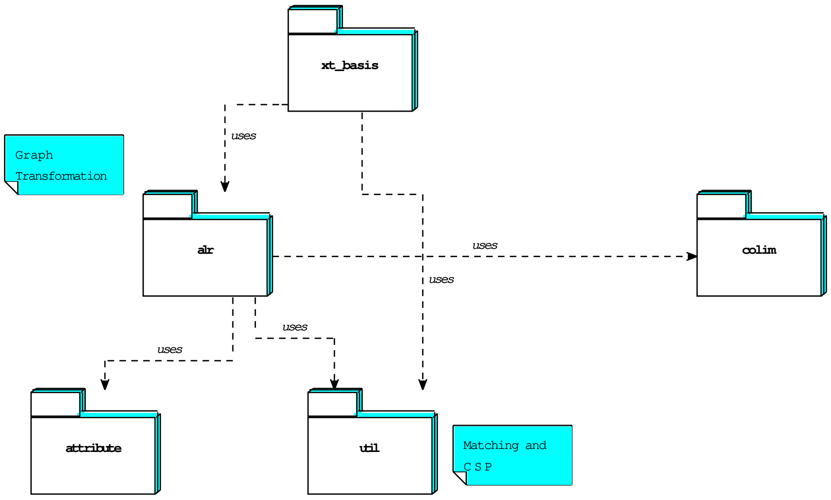



Abbildung 5.2: Klassendiagramm: Das AGG-System

### 5.1.1 Graphen und Regeln

Wir beschreiben nun, wie eine Graphgrammatik aufgebaut und verwendet werden kann. Die Graphgrammatik selbst lässt sich mit dem Konstruktor `GraGra()` erzeugen, wir erhalten eine Grammatik ohne Regeln mit einem leeren Arbeits- oder Startgraphen, den wir über die Methode `Graph getGraph()` bekommen. Weitere Regeln lassen sich in der Grammatik mit der Methode `Rule createRule()` erzeugen. Einen Regelmorphismus brauchen wir nicht gesondert zu kreieren, da die Regel selbst ein Morphismus ist (siehe Abb. 5.3). Wir bekommen die linke bzw. rechte Seite der Regeln als Graphen mit den Methoden `Graph getLeft()` sowie `Graph getRight()`. Außerdem hält die Graphgrammatik alle Typen, die Graphobjekte annehmen können, wir können diese Typen über `Type createType()` erzeugen.

Wollen wir nun Knoten und Kanten in einem Graphen erzeugen, so geschieht das über die Methoden `Node createNode(Type typ)` und `Arc createArc(Type typ, Node src, Node tar)` – es muss also immer der Typ für ein zu erzeugendes Graphobjekt angegeben werden. Da die Typen auch Informationen über die Attribute enthalten, betrachten wir sie im nächsten Unterabschnitt noch genauer. Die ein- bzw. ausgehenden Kanten eines Knotens erhalten wir über `Enumeration getIncomingArcs()` bzw. `Enumeration getOutgoingArcs()`, den Quell- bzw. Zielknoten einer Kante mit `Node getSource()` bzw. `Node getTarget()`.

Was jetzt noch fehlt, um Regeln angeben zu können, ist der Morphismus selbst, den die Regel darstellt. Wir fügen dem Morphismus eine Abbildung hinzu, indem wir die Methode `void addMapping(GraphObject o, GraphObject i)` anwenden. Ausgangs-

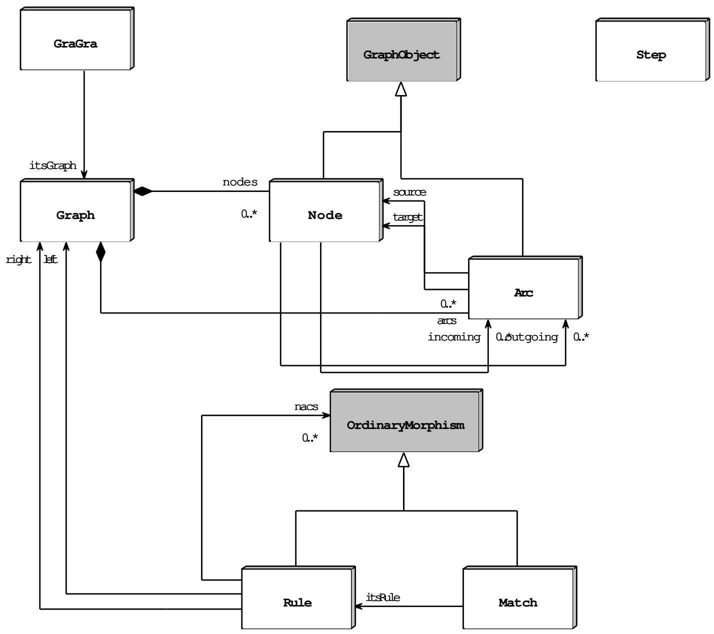



Abbildung 5.3: Klassendiagramm: Die Graphgrammatik in AGG

und Zielgraphen eines Morphismus erhalten wir über `Graph getOriginal()` und `Graph getImage()`, Definitions- und Wertebereich selbst erhalten wir dagegen über `Enumeration getDomain()` und `Enumeration getCodomain()`.

Desweiteren können wir für eine Regel negative Anwendungsbedingungen mit `OrdinaryMorphism createNAC()` erstellen, NACs sind also auch nur schlichte Morphismen.

### 5.1.2 Typen und Attribute

Die Theorie sieht vor, dass jedem Knoten und jeder Kante genau ein Attribut zugeordnet wird. AGG implementiert jetzt diese Attribute so, dass jedes Graphobjekt – wie oben dargestellt – genau einen *Graphobjekttyp* besitzt (Klasse `Type`). Jedem dieser Graphobjekttypen ist wiederum genau ein *Attributtyp* zugeordnet. Ein konkretes Graphobjekt besitzt jetzt genau eine *Attributinstanz* oder kurz ein Attribut zum jeweiligen Attributtypen. Da wir aber aus praktischen Gründen gerne mehrere konkrete Datentypen an einem Graphobjekt haben wollen, so z.B. einen Knoten, der einen Wert vom Typ `String` und einen weiteren vom Typ `boolean` hält, können einem Attributtypen jetzt null oder mehrere *Einzeldeklarationen* zugeordnet werden. Diese bestehen aus einem *Deklarationsnamen* und einem *Deklarationstyp*.

Der Deklarationstyp hängt nun von dem *Attributhandler* ab, den wir verwenden, um die Daten zu verwalten, auf Konsistenz zu prüfen und – im Verlauf der Graphtransformation – Berechnungen durchzuführen. Wir verwenden an dieser Stelle einen Handler für Java-Ausdrücke, das impliziert, dass jetzt als Deklarationstypen Java-Klassen verwendet werden und wir uns bei der Beschreibung der Attributierung auf Java-Ausdrücke beschränken.

Bei der Erzeugung eines Knotens oder einer Kante werden aus den Einzeldeklarationen des Attributtyps jetzt Instanzen erzeugt. Die Instanz einer Deklaration besteht jetzt aus Datentyp (Java-Klasse), Name und entweder einem Java-Ausdruck oder einem konstanten Wert (also einem Java-Objekt). Der Java-Ausdruck muss zum entsprechenden Datentyp auswerten, das wird vom Attributhandler schon bei der Eingabe des Ausdrucks geprüft. Ein solcher Ausdruck wäre z.B. `s.toUpperCase()`. Dieser Ausdruck konvertiert den String in der Variable `s` in Großbuchstaben und gibt ihn zurück. Variablen, die man verwendet, müssen bei der Auswertung belegt sein. Das wird bei der Graphtransformation sichergestellt, wir benötigen für unsere Graphen einige Einschränkungen in den Attributen:

- In der linken Seite einer Regel dürfen nur Variablen oder Konstanten vorkommen.
- Auf der rechten Seite dürfen beliebige Java-Ausdrücke oder Konstanten auftreten, allerdings müssen rechts verwendete Variablen links auch vorkommen.
- Im Arbeitsgraphen selbst dürfen nur Konstanten stehen.

### 5.1.3 Transformation

Um eine Regel auf einen Arbeitsgraphen anzuwenden, muss zunächst ein Ansatz (Match) der linken Regelseite im Arbeitsgraphen erstellt werden. Das geschieht innerhalb der Graphgrammatik mit der Methode `Match createMatch(Rule r)`. Dieser Match ist leer und damit nicht korrekt im Sinne der Theorie, da nicht total. Da ein Match wieder ein Morphismus ist, können ihm mit `addMapping(..)` wieder Abbildungen hinzugefügt werden. Wir können den Match jetzt entweder per Hand vervollständigen, bis er total ist, oder wir verwenden die Methode `boolean nextCompletion()`, um den nächsten Ansatz zu bekommen.

Ein Transformationsschritt selbst wird durch den Aufruf der Methode `Morphism execute(Match m)` in der Klasse `Step` angestoßen. Diese Methode liefert den Morphismus aus der rechten Regelseite in den – jetzt transformierten – Arbeitsgraphen zurück. Da ein *in-place-editing* stattfindet, bleibt der Arbeitsgraph bestehen.

Die Transformation der Attribute mittels einer Regel geht nun folgendermaßen vor sich: Zunächst wird ein Ansatz der linken Seite gesucht, wobei auf Seite der Attribute entweder Konstanten gematcht werden oder Variablen mit den Werten aus dem Arbeitsgraphen belegt werden. Dann wird der Graph transformiert und die Java-Ausdrücke auf der rechten Seite der Regel (falls vorhanden) ausgewertet, wobei die Variablenbelegungen aus dem Match verwendet werden.

## 5.2 Der Constraintsolver Parcon

Der Constraintsolver, den wir als Komponente einsetzen, ist *Parcon*. Dieses System wurde 1995 von Peer Griebel an der Universität-GH Paderborn entwickelt [Gri96]. Parcon ist darauf spezialisiert, zweidimensionale – also grafische – Constraints in Echtzeit zu lösen, was unseren Anforderungen an einen Constraintsolver sehr entgegenkommt. Constraints können interaktiv hinzugefügt und gelöscht werden, es können sowohl Gleichungen als auch Ungleichungen eingegeben werden und konjunktiv und disjunktiv verknüpft werden. Der Vorteil von Parcon ist der, dass durch die Konzentration auf geometrische Constraints und die Implementierung in einer sehr schnellen Programmiersprache es gerade für unseren Editor möglich ist, die Constraints in Echtzeit zu lösen. Wenn also beispielsweise Objekte, deren Werte am Solver angemeldet sind, verschoben werden, ist es möglich, während des Verschiebens die Beziehungen zu anderen Grafikobjekten zu korrigieren und darzustellen.

Parcon ist in C++ implementiert und bietet uns, die wir mit Java arbeiten, nichts weiter als eine ASCII-Schnittstelle, in die wir Constraints schreiben und aus der wir Werte auslesen können. Diese Java-Parcon-Schnittstelle ist Bestandteil des Alphabeteditors und wird deshalb hier nicht näher betrachtet, sondern wird dort [Sch99] beschrieben. Zum besseren Verständnis des gesamten Bereichs Constraint-Solving stellen wir die Funktionalität des Tools überblicksweise vor.

### 5.2.1 Constrainteingabe in Parcon

Wie wird nun ein Constraint am Constraintsolver angemeldet? Parcon besitzt keine höhere Constraintsprache, wie sie in Abschnitt 3.1.1 vorgestellt wurde, stattdessen muss jede Constraint(un)gleichung einzeln eingegeben werden. Die Syntax einer Constraintformel lautet

**add** $\{$ ***Wert Variable*** $\}^+$***relop Wert***

Jede Formel ist eine lineare (Un)gleichung, wobei $1 \ldots n$ Koeffizienten-Variablen-Paare auf der linken Seite stehen können. Es folgt eine der Relationen **¡=, =, ¿=** und ein konstanter Wert auf der rechten Seite. Beispiel: **add 1 a.x 2 b.x -1 c.x = 5**. Variablennamen sind eine Kombination aus Ziffern und Buchstaben und jeder Variablenname muss einen Punkt enthalten. Geometrische Punkte lassen sich in Kurzform schreiben, indem man den Teil hinter dem trennenden Punkt weglässt, so ist **P** eine Kurzform für **P.x P.y**.

Produkte lassen sich wie folgt schreiben: **add *Variable Variable* = *Variable***. Eine weitere nichtlineare Formel ist die Berechnung des euklidischen Abstands zwischen zwei Punkten (damit wird festgelegt, dass zwei Punkte immer einen bestimmten Abstand haben sollen): **add P1 R= r.v P2** .

Es ist möglich, Formeln Prioritäten zuzuweisen, die beim Lösen der Constraints berücksichtigt werden. Dazu verwendet man **add**$n$ ***Formel***, wobei $n$ eine Zahl von 0 (geringste Priorität) bis 9 (höchste Priorität) sein kann. Formeln ohne Angabe haben automatisch die niedrigste Priorität. Außerdem kann man Formeln disjunktiv (mit „oder“) verknüpfen. Dazu gibt man eine Formel mit **add—*Formel*** ein, diese bekommt dieselbe

Priorität wie die vorher eingegebene und wird mit dieser disjunktiv verknüpft. Disjunktionen haben höhere Priorität als Konjunktionen.

Variablen werden mit **set *Variable*** auf bestimmte Werte gesetzt, ansonsten erhalten sie vom System automatisch den Wert 0.

Formeln erhalten eine interne Nummerierung, sie können mit **del *Formelnummer*** wieder aus dem Constraintsystem gelöscht werden.

### 5.2.2 Constraint-Solving

Parcon hat folgende Eigenschaften beim Lösen seines Constraintsystems:

- *Least Astonishment*: Benutzereingaben werden – wenn überhaupt – als letze Eingaben verändert. Wenn also ein Wert geändert wird und dadurch nicht mehr alle Constraints erfüllt sind, wird zuallererst versucht, ältere Werte anzupassen, ohne den neu geänderten Wert anzufassen.
- Dazu gehört, dass während der Lösung der Constraints die Variable mit einem neuen Wert belegt wird, deren Wert die längste Zeit unverändert war.
- Von zwei gleichwertigen Variablen in einer Formel wird die Variable, die am weitesten links steht, als erstes geändert.

Diese Eigenschaft muss beachtet werden, will man – wie es im Alphabeteditor getan wurde – eine eigene High-Level-Constraintsprache basierend auf Parcon-Formeln definieren. Ein Lauf des Constraintsolvers besteht jetzt aus zwei Teilen (angenommen, es wurde ein Constraintsystem definiert). Zunächst werden Parcon mit **new** $\{$ ***Variablenname*** $\}^+$ die Namen der Variablen mitgeteilt, deren Wert sich ändern wird. Die Werte schließlich werden mit **val** $\{$ ***Wert*** $\}^+$ an den Solver übergeben.

Dann beginnt der Solver, das Constraintsystem zu lösen, indem er die übergebenen Variablen möglichst nicht anfasst. Ist der Lauf fertig, gibt Parcon alle Variablen mit ihren Werten aus, wobei Variablen, deren Werte sich nicht geändert haben, mit einer vorangestellten Tilde beschrieben werden. Wie die Einbindung von Parcon in unser System funktioniert und wie die Constraintsolver-Ein- und Ausgabe vonstatten geht, wird im Abschnitt 5.3 genauer beschrieben.

## 5.3 Der Alphabeteditor

Der **Alphabeteditor** ist die Komponente, mit der unser Grammatikeditor die engste Verbindung eingeht. Er wird im Rahmen einer weiteren Diplomarbeit an der Technischen Universität Berlin von Manuel Schwarze implementiert, und zwar – genau wie unser Grammatikeditor – in Java. Im Prinzip besteht dieser Editor aus drei Subkomponenten: einem Symboleditor, mit dem sich die grafischen Symbole bauen lassen, einem Verbindungseditor, in dem man die grafischen Links für die Symbole festlegt und einem Constrainteditor, der als Hilfsmittel in beiden Untereditoren auftaucht, und mit dem man Constraints für grafische Links und Symbole angeben kann. Wir wollen hier nur überblicksweise die Schnittstelle zeigen, eine ausführliche Beschreibung findet sich in [Sch99].

Sämtliche Klassen, die wir benötigen, um auf ein Alphabet zuzugreifen, befinden sich im Package `genged.language`.

### 5.3.1 Symbole und Links

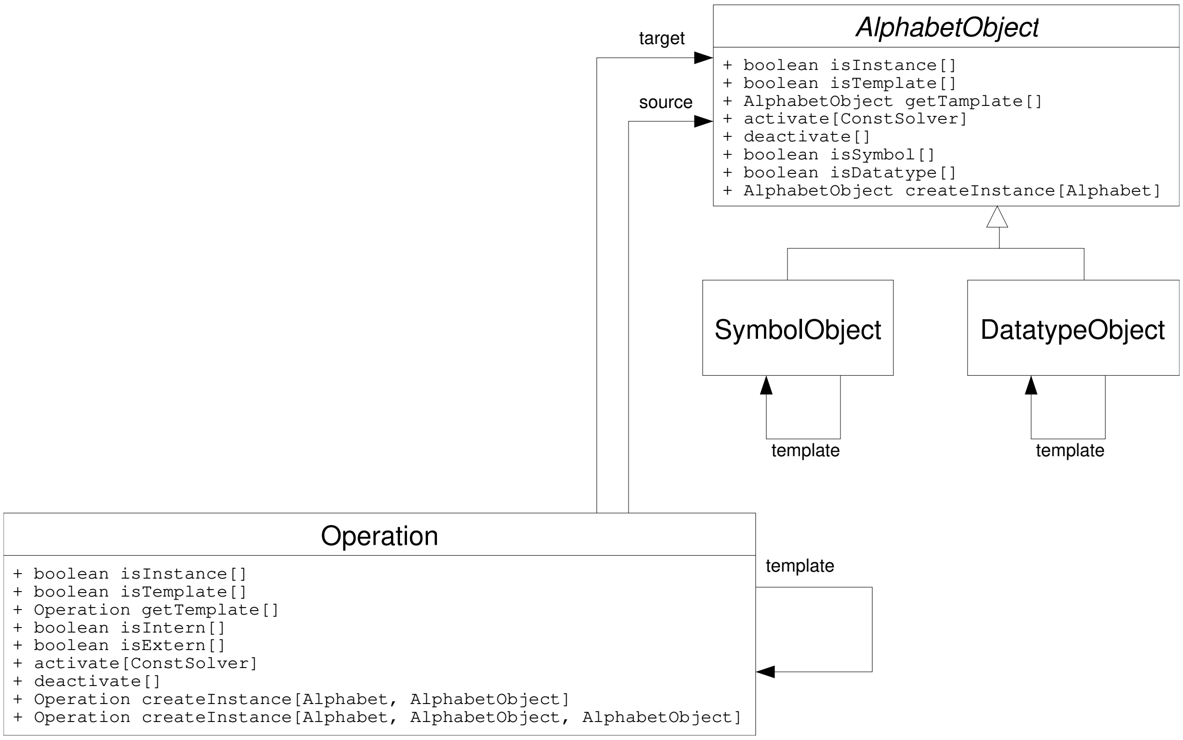



Abbildung 5.4: Klassendiagramm: Symbole und Operationen

Das Grundobjekt eines grafischen Alphabets, das wir vom Alphabeteditor erhalten, wird durch das Interface `AlphabetObject`. repräsentiert. Ein `AlphabetObject` kann ein `SymbolObject` oder ein `DatatypeObject` sein. Alle solche grafischen Objekte können entweder ein *Template* (Vorlage) oder eine Instanz sein. Im grafischen Alphabet, das der Alphabeteditor erzeugt, sind alle Symbole und Datentypobjekte Templates.

Die Klasse `SymbolObject` beschreibt alle grafischen Symbole des Alphabets, dieses Objekt kann sich aus mehreren Primitiva zusammensetzen. Die Klasse `DatatypeObject` beschreibt ein grafisches Objekt, das einen Datentypen kapselt.

Die Constraints, die unsere Symbole intern zusammenhalten, werden nicht innerhalb des Symbol- oder Datentypobjekts gehalten. Stattdessen erhält jedes Alphabetsymbol eine Art Pseudooperation, die als *intern* gekennzeichnet ist und die alle internen Constraints des Symbols hält. Damit kommen wir zu den Operationen:

Eine externe Operation entspricht dem grafischen Link in unserer Sprachbeschreibung, verbindet also zwei Symbolobjekte oder ein Symbolobjekt und ein Datentypobjekt miteinander.

### 5.3.2 Alphabet und Diagrammalgebra

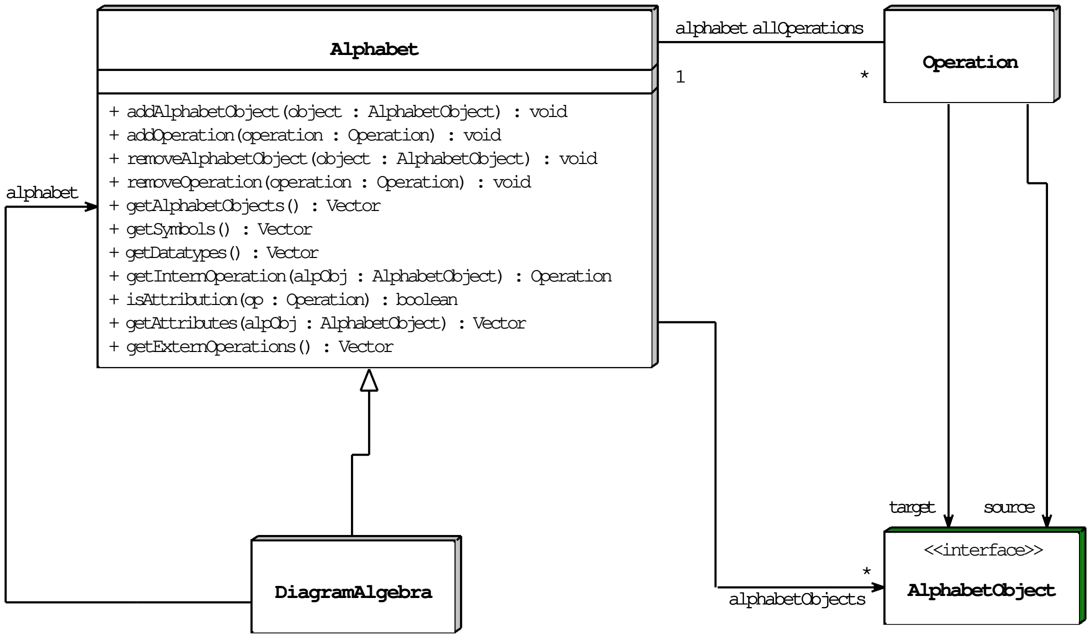



Abbildung 5.5: Klassendiagramm: Alphabet

Alle Templates (sowohl die für Symbole als auch die für Operationen) werden in einem Objekt der Klasse `Alphabet` gehalten. Ein Alphabet stellt mannigfaltige Möglichkeiten zur Verfügung, sich Listen von Symbolen, Datentypen, internen oder externen Operationen, Operationen im Zusammenhang mit einem bestimmten Symbol etc. ausgeben zu lassen. Wir haben im Klassendiagramm 5.5 nicht alle Methoden aufgeführt, diese werden in der Beschreibung des Alphabeteditors genauer definiert. Da das Alphabet ja – wie oben angedeutet – nur die Vorlagen enthält, benötigen wir außerdem eine Datenstruktur, die die Instanzen hält. Das liefert uns die `DiagramAlgebra` („Algebra“ deshalb, da es sich um eine Instantiierung des Alphabets handelt). Wenn wir nun ein Symbol oder eine Operation instantiieren wollen, müssen wir zunächst eine Diagrammalgebra aus einem Alphabet erzeugen. Dann benutzen wir die Methode `createInstance(...)`, der wir die Diagrammalgebra und – wenn wir eine Operation instantiieren wollen – zwei Symbolinstanzen übergeben. Die erzeugte Instanz müssen wir noch mir der jeweiligen `add...`-Methode zur Diagrammalgebra hinzufügen.

### 5.3.3 Constraints

Die Behandlung und das Lösen der Constraints ist denkbar einfach. Wir müssen die Verbindung zu einem Constraintsolver, also einem Parcon-Prozess erzeugen, dafür gibt es die Klasse `genged.constsolver.ConstSolver`, und die Symbole bzw. Operationen mit der Methode `activate(ConstSolver)` an diesem Constraint-Solver anmelden oder sie mit `deactivate()` abmelden. Eine genaue Beschreinbung der Schnittstelle befindet sich wiederum in [Sch99].

# Kapitel 6

# Implementierung

Dieses Kapitel beschreibt die Implementierung des Grammatikeditors sowie unsere Schnittstellen zu den in Kapitel 5 vorgestellten externen Komponenten. Unter Implementierung verstehen wir nicht die Beschreibungen des Sourcecodes, sondern – im Gegensatz zum Design, das von der verwendeten Programmiersprache und dem System unabhängig ist – die Darstellung der konkret verwendeten Java-Packages, -Klassen und -Methoden und der objektorientierten Techniken, die wir benutzt haben. Als grafische Diagrammsprachen verwenden wir dabei hauptsächlich die in Abschnitt 4.2 eingeführten Klassen- und Sequenzdiagramme.

## 6.1 Java

Warum haben wir Java als Implementierungssprache gewählt? Es gibt natürlich einen ganz simplen Grund: Sachzwänge! Da das AGG-System, auf dem wir aufbauen und dessen Schnittstellen wir intensiv nutzen, komplett in Java implementiert worden ist, blieb uns fast keine andere Wahl, als die beiden Editoren (Alphabet- und Grammatikeditor) auch in Java zu schreiben. Die Programmiersprache bietet allerdings auch ohne diese Zwänge genügend Vorteile, die hier kurz dargestellt werden sollen:

**Objektorientierung** Java ist eine objektorientierte Sprache. Das OO-Paradigma hat den Vorteil, dass sich Sachverhalte, die wir in der Theorie vorfinden, sehr schnell in Form von Objekten, Interfaces und Methoden modellieren und dann auch implementieren lassen. Nach einer gewissen Einarbeitungszeit ist die Modellierung dann schön intuitiv, so erscheint es natürlich, dass z.B. ein Objekt *Diagramm* weitere Objekte *Symbol* enthält, und dass Symbole beispielsweise Methoden zum Zeichnen oder Bewegen auf einer Zeichenfläche besitzen. Auch für die Modellierung und Implementierung von grafischen Benutzungsoberflächen sind Objekte am besten geeignet, da vordefinierte Klassen für Fenster, Buttons etc. leicht abgeleitet und erweitert werden können.

**Portabilität** Java ist plattformunabhängig. Obwohl diese Marketingaussage nicht völlig richtig ist, ist Java doch größtenteils portabel. Das ermöglicht den Einsatz unseres Systems auf verschiedensten Plattformen. Entwickelt wird dieser

Grammatikeditor beispielsweise auf einem Linux-System, der Einsatz findet jedoch größtenteils unter Solaris statt. Gerade für Applikationen, die an einer Universität entwickelt werden und die im Laufe ihres Lebens von vielen Programmierern betreut werden müssen, ist diese Unabhängigkeit ein klarer Vorteil.

**API** Java beinhaltet von Anfang an sehr umfangreiche Bibliotheken. Da wir neben dem von *Java Development Kit* von Sun Microsystems die ebenfalls von Sun entwickelten *Java Foundation Classes* (auch Swing-Klassen) verwenden, lassen sich Benutzeroberflächen mit wenig Aufwand verwirklichen.

**Robustheit** Da Java im Vergleich zu C++ keine Pointer und keine Möglichkeit, zwischen *call-by-reference* und *call-by-value* zu wählen, bietet, sinkt die Wahrscheinlichkeit von Programmierfehlern deutlich. Auch die Tatsache, dass der Programmierer sich dank der eingebauten Garbage Collection keine Gedanken mehr über die Freigabe von Speicher machen muss, macht zumindest die Implementierung leichter.

**Verbreitung** Java ist sehr weit verbreitet. Gerade seit dem Aufstieg des Internets und Web-Based Programming hat Java inzwischen einen hohen Bekanntheitsgrad erreicht, was uns hoffen lässt, dass es auch in nächster Zukunft noch Unterstützung und Weiterentwicklungen für diese Sprache geben wird.

Wir wollen allerdings die Nachteile nicht verschweigen:

**Geschwindigkeit** Java ist, da es sich um eine interpretierte Sprache handelt, immer noch ca. zehnmal langsamer als z.B. C++. Auch der Speicherbedarf ist sehr viel größer. Es steht allerdings zu hoffen, dass beide Probleme in nächster Zeit – vor allem durch den Einsatz von Just-In-Time-Compilern – zumindest verkleinert werden.

**Portabilität** Java ist nicht völlig plattformunabhängig. So ist besonders die Funktionalität der grafischen Benutzeroberflächen von System zu System stark unterschiedlich. Aus dem Marketingspruch „write once, run anywhere“ ist deshalb allzuoft ein „write once, debug everywhere“ geworden.

Zum letzten Punkt ist noch anzumerken, dass dadurch, dass der verwendete Constraintsolver Parcon in C++ implementiert wurde und deshalb nur unter Linux und Solaris läuft, unser gesamtes System leider ohnehin auf diese beiden Plattformen angewiesen ist.

## 6.2 Struktur des Systems

In Abb. 6.1 ist die gesamte Strukturierung des Grammatikeditors aufgezeigt. Alle diese Unterpackages sind in einem Systempackage namens `gramed` enthalten. Dieses Systempackage enthält nur eine Klasse, nämlich `GramEd`, die dazu dient, das System zu starten. Die Packages haben folgende Aufgaben (in alphabetischer Reihenfolge):

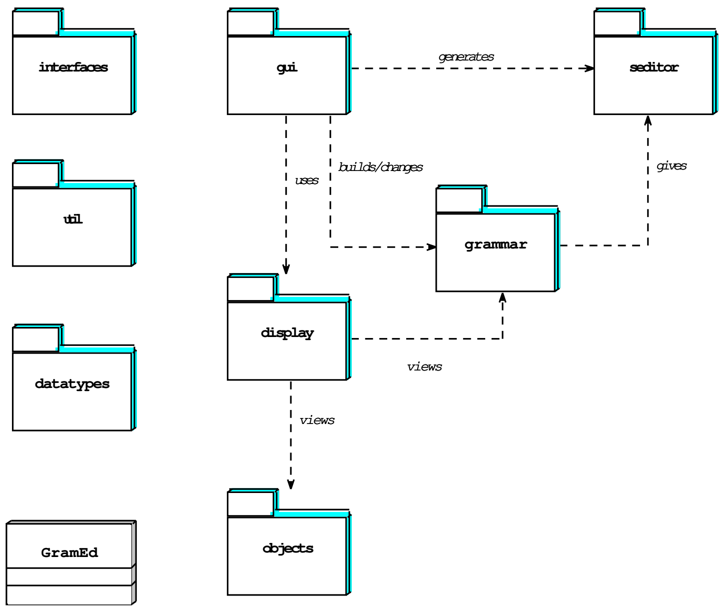



Abbildung 6.1: Klassendiagramm: Packages für den Grammatikeditor

**`display`** Dieses Package enthält die Klassen, die *Sichten* auf verschiedene Bestandteile der visuellen Sprache enthalten: Grammatik, Regeln, Diagramme etc. *Sicht* ist aber nicht – wie im üblichen Sinne – passiv zu verstehen, es handelt sich eher um Editoren für die Sprachbestandteile.

**`grammar`** Hier ist die logische Ebene des Grammatikeditors zu Hause, also die Grammatik selbst mit den Verbindungen zum AGG-System.

**`gui`** Dieses Package enthält die Klassen für die grafische Benutzeroberfläche, abgesehen von den einzelnen Displays.

**`interfaces`** Hier stehen alle Interfaces, die package-übergreifend benutzt werden sowie die Interfaces zum Alphabeteditor.

**`objects`** In diesem Package finden wir die grafische Ebene des Editors, dh. Objekte für Regeln, Diagramme etc.

**`seditor`** Dieses Package enthält den zu generierenden Syntax- oder Diagrammeditor. „Zu generieren“ ist nicht wörtlich zu nehmen, da wir letztendlich nur die Grammatik für den Editor erzeugen und ihn damit füttern.

**`util`** Hier hinein kommen Hilfsklassen, die von anderen Packages benutzt werden.

## 6.3 Entwurfsmuster „Beobachter“

*Entwurfsmuster* [GHJV95] sind Lösungsentwürfe für Probleme, mit denen man bei der objektorientierten Programmierung – unabhängig von der verwendeten Sprache – immer wieder konfrontiert wird, besonders wenn es um die Auflösung von Redundanzen und um die Wiederverwendung von Datenstrukturen geht. Ein Entwurfsmuster, das wir in der Implementierung sehr häufig verwenden, ist das des *Beobachters*.

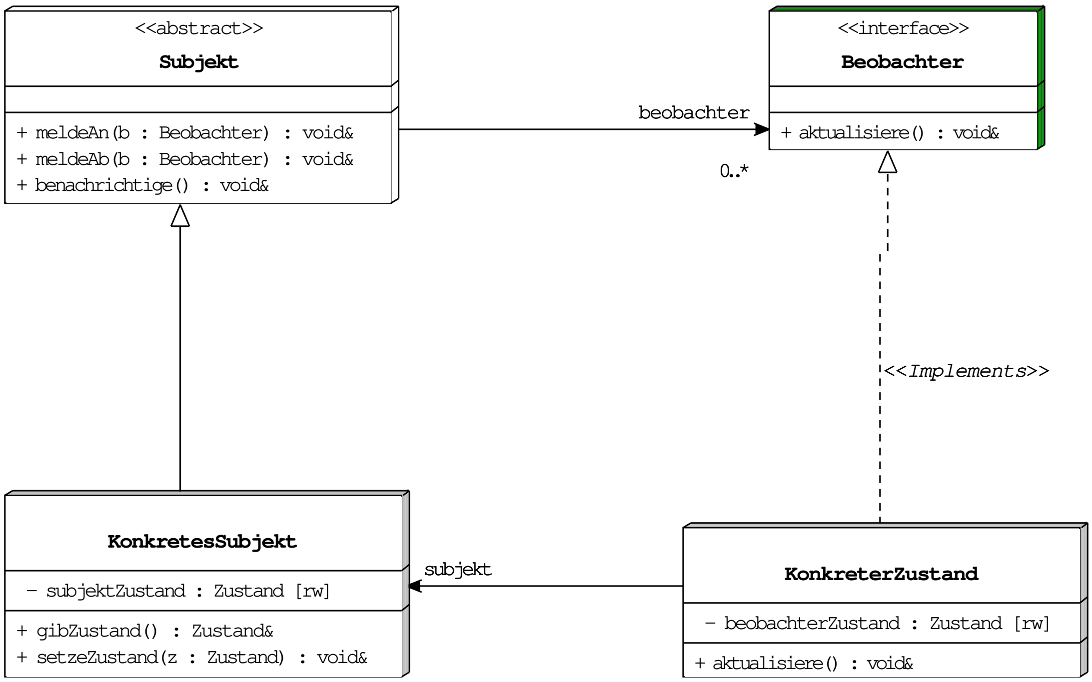



Abbildung 6.2: Klassendiagramm: Entwurfsmuster Beobachter

Das Muster *Beobachter* modelliert die Abhängigkeit zwischen einem Basisobjekt und $1 \ldots n$ Beobachtern, wobei Zustandsänderungen im Basisobjekt dazu führen, dass alle Beobachter automatisch benachrichtigt und aktualisiert werden. So lassen sich dynamisch weitere Beobachter oder Sichten auf ein Objekt definieren und auch wieder abmelden, ohne dass die Verbindung zu eng wäre. Die Struktur des Musters finden wir in Abb. 6.2.

Das *Subjekt* hält eine Liste von seinen Beobachtern. Wird ein neuer Beobachter an- bzw. abgemeldet, muss er in diese Liste aufgenommen bzw. daraus entfernt werden. Außerdem muss das Subjekt bei Änderung seines Zustands alle Beobachter über die Änderung informieren. Das *konkrete Subjekt* muss diese Funktionalität implementieren.

Ein *Beobachter* definiert eine Schnittstelle für die Benachrichtigung von Objekten.

Ein *konkreter Beobachter* muss eine Referenz auf ein Subjekt halten und die Methode `aktualisiere` bereitstellen, die vom Subjekt aufgerufen wird, wenn sich dessen Zustand ändert und die die Konsistenz zwischen Sicht und Beobachter wiederherstellen soll.

In unserem System entspricht das *Subjekt* der abstrakten Klasse `gramed.objects.GEObject` mit den Methoden `void register(GEObserver o)`, `void unRegister(GEObserver o)` und `void notifyObservers(UpdateInfo changes)`. Abgeleitet davon werden konkrete Subjekte wie Diagramme, Regeln und die Grammatik der visuellen Sprache. Die Schnittstelle *Beobachter* finden wir in `gramed.interfaces.GEObserver` wieder, sie wird von den Displays oder Editoren für die Diagramme, Regeln und die Grammatik implementiert. Aus Effizienzgründen liefert die Methode `void notifyObservers(UpdateInfo changes)` in dem Objekt `changes` genaue Informationen darüber, **was** sich im Subjekt geändert hat. Das verhindert, dass Beobachter jedesmal das Subjekt komplett nach den Änderungen durchsuchen müssen und erlaubt uns, lokale Änderungen im Subjekt auch durch nur lokale Änderungen im Beobachter widerzuspiegeln.

## 6.4 Datenstrukturen

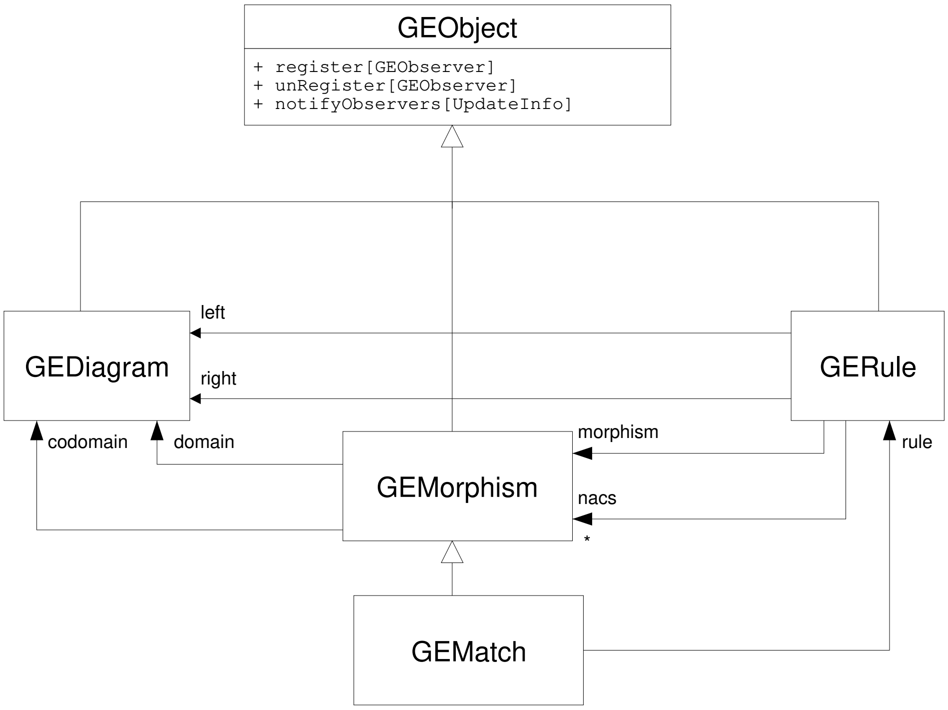



Abbildung 6.3: Klassendiagramm: Das Package `objects`

Bis auf die visuelle Grammatik, die in einem eigenen Packages residiert, befinden sich alle Datenstrukturen unseres Grammatikeditors im Package `gramed.objects`. Die wichtigsten Klassen sind `GERule`, `GEDiagram` und `GEMorphism`. Sie werden im Folgenden einzeln vorgestellt.

Alle diese Klassen basieren auf GEObject. Wie schon im Abschnitt 6.3 beschrieben, implementiert diese Klasse ein Subjekt, das in der Lage ist, mit mehreren Beobachtern zu kommunizieren. Jedes GEObject hat einen (änderbaren) Namen und ein Alphabet mit folgender Funktionalität: Ein neues GEObject wird auf Basis eines initialen grafischen Alphabets erzeugt. Damit basiert das Objekt auf genau einer Signatur (nämlich der, die vom initialen Alphabet geliefert wird), das Alphabet zu dieser Signatur kann sich jedoch ändern. Das macht es uns möglich, Grafiken und Links für ein Objekt dynamisch zu ändern, solange diese dieselben Namen und – im Falle der Links – dieselben Quell- und Zielobjekte haben wie die ursprüngliche Signatur.

### 6.4.1 Klasse **GEDiagram**

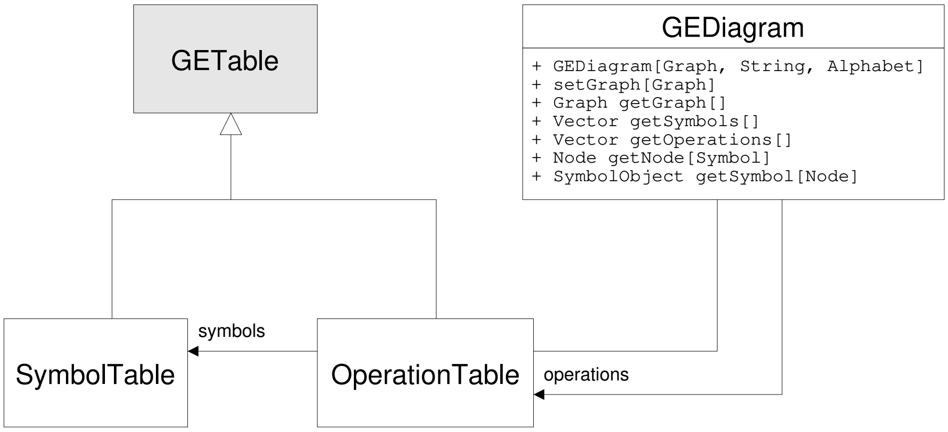



Abbildung 6.4: Klassendiagramm: GEDiagram

Diese Klasse liefert ein Diagramm, basierend auf einem Graphen. Da AGG bei jeder Transformation neue Knoten erzeugt, ist es nicht ratsam, die Instanzen von grafischen Symbolen und Links mit Referenzen an die AGG-Objekte zu koppeln. Stattdessen halten wir zwei Tabellen, in denen die Abbildungen Knoten $\longleftrightarrow$ Symbole und Kanten $\longleftrightarrow$ Operationen gehalten werden. Wann immer der zugrundeliegende Graph für das Diagramm sich ändert oder neu gesetzt wird, müssen wir alle Knoten und Kanten des Graphen durchgehen und in der jeweiligen Tabelle nachsehen, ob eine grafische Repräsentation für das Graphobjekt existiert. Wenn ja, dann wird diese Repräsentation stehen gelassen, wenn nein, wird ein neues Grafiksymbol bzw. ein Link vom jeweiligen Template erzeugt und in die Tabelle eingetragen. Außerdem werden die Beobachter dieses Diagramms über Änderungen benachrichtigt, so muss beispielsweise ein Display für dieses Diagramm Constraints löschen bzw. neu anmelden und ein Constraint-Solving durchführen, um das Diagramm korrekt darzustellen.

Die Tabellen SymbolTable und OperationTable sind vom Hilfsobjekt GETable abgeleitet und implementieren eine Art doppelte Hashtable. So lassen sich effizient 1:1-Abbildungen speichern und in beide Richtungen auswerten.

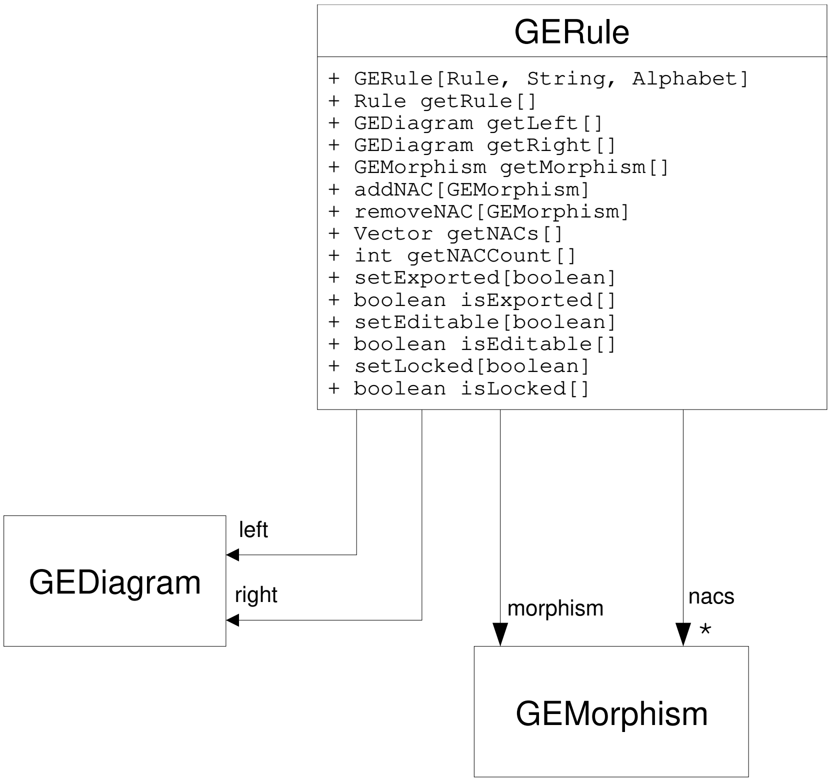



Abbildung 6.5: Klassendiagramm: `GERule`

### 6.4.2 Klasse **GERule**

Diese Klasse stellt eine visuelle Regel dar. Eine visuelle Regel basiert auf einer Graphregel und besteht aus zwei Diagrammen für die linke und rechte Seite sowie einer Liste von NACs, die ihrerseits wieder Diagramme sind. Analog zur Graphebene hat eine visuelle Regel einen Regelmorphismus und $0 \dots n$ NAC-Morphismen. Eine Regel kann *geschützt* sein, was bedeutet, dass sie weder bearbeitet noch gelöscht werden kann, sie kann *gesichert* („locked“) sein, was bedeutet, dass sie nicht auf andere Diagramme angewendet werden kann und sie kann für den Export markiert werden. Darüberhinaus kann eine Regel *Regelparameter* besitzen, diese werden in Abschnitt 6.5 genauer beleuchtet.

### 6.4.3 Klassen **GEMorphism** und **GEMatch**

Diese Klasse baut auf einem Graphmorphismus auf und erweitert diesen um Domain- und Codomaindiagramm. Außerdem speichert ein Morphismus die Abbildungen von grafischen Symbolen aufeinander, das ermöglicht es uns, einen Morphismus schnell zu visualisieren. Die Visualisierung wird im Abschnitt 6.7 genauer erläutert.

Ein Match (Klasse `GEMatch`) ist ein besonderer Morphismus, der zusätzlich zu den Domain- bzw. Codomain-Diagrammen noch die Regel speichert, in die der Ansatz gemacht wird. Außerdem weiss ein Match, ob er auf eine Regel angewendet wird oder auf

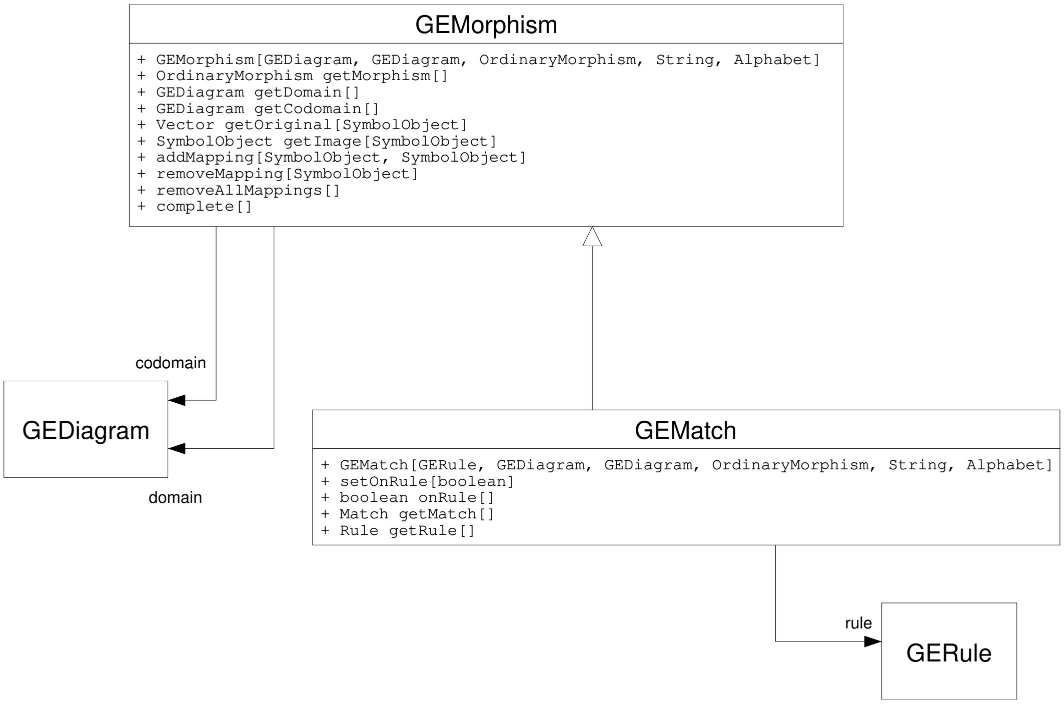



Abbildung 6.6: Klassendiagramm: `GEMorphism`

ein Arbeitsdiagramm – dieses Verhalten wird in Abschnitt 6.6 genauer erklärt.

## 6.5 Datentypen

*Datentypen*, wie schon in [Sch99] beschrieben, sind Klassen, die das Interface `genged.language.Datatype` implementieren. Alle Datentypklassen müssen sich im Package `genged.language.datatypes` befinden, da sie sonst vom Attributhandler des AGG-Systems nicht gefunden werden. Will man eigene Datentypen definieren, sollte man eine Unterklasse von `genged.language.AbstractDatatype` bilden, im Alphabeteditor-Handbuch befindet sich diesbezüglich eine ausführliche Anleitung. Ein Datentyp, der in einem unserer Diagramme vorkommt, wird in der Klasse `GEAttributeTuple` gespeichert, und zwar mit folgenden Attributen:

- dem Namen des Datentyps,
- der Java-Klasse des Datentyps (und zwar der vollständigen Klasse),
- dem Java-Ausdruck, der diesen Datentypen beschreibt,
- dem Datentypobjekt (Klasse: `DatatypeObject`), das diesen Datentypen kapselt und darstellt,
- dem Wert des Datentypen, also dem Datentypen selbst und
- dem Attribut im AGG-System, das diesen Datentypen bearbeitet.

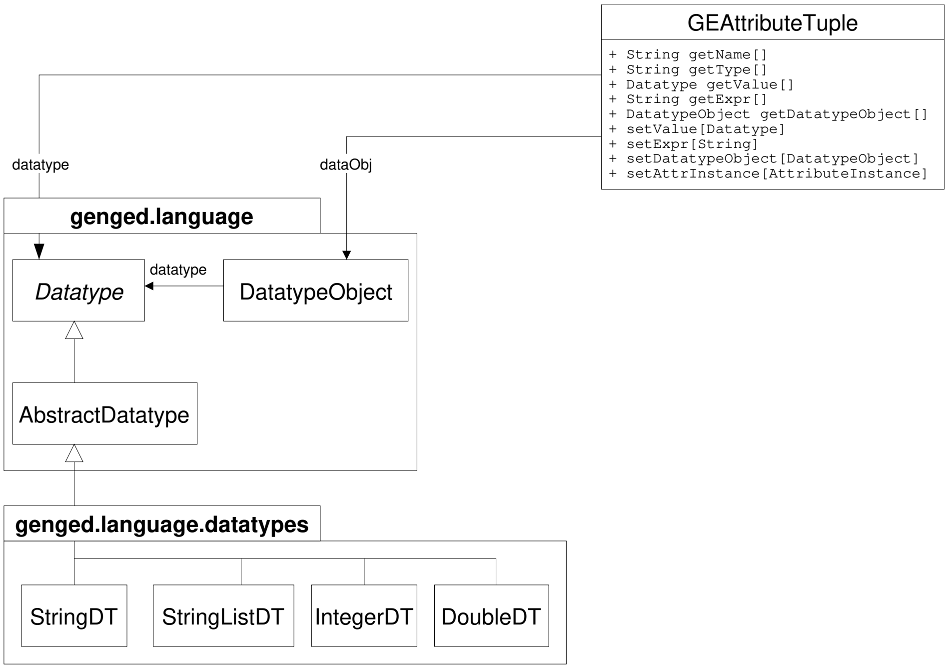



Abbildung 6.7: Klassendiagramm: Datentypen

Dieser Aufwand ist nötig, da wir Regeln auf andere Regelseiten anwenden wollen, um mit Hilfe der einfachen, erzeugten Regeln komplexere aufzubauen. Da die Attributkomponente des AGG-Systems es allerdings nicht erlaubt, dass im Arbeitsgraphen (der ja in unserem Fall durchaus auch eine Regelseite sein kann) andere Werte für Datentypen als Konstanten (also Objekte) vorkommen, könnten wir weder Variablen in der linken Regelseite auf Variablen noch auf komplexe Java-Ausdrücke im Arbeitsgraphen matchen. Wir umgehen dieses Problem, indem wir, wenn wir einen Ansatz einer Regelseite in einen Arbeitsgraphen (`GEMatch`) erzeugen, explizit sagen, ob wir eine Regel mit einer anderen oder einen herkömmlichen Arbeitsgraphen bearbeiten.

Wenn wir eine Regel bearbeiten, müssen wir dafür Sorge tragen, dass die Attribute auf der linken Regelseite *nicht* gematcht werden. Das lösen wir wie folgt: Wird eine Regel auf eine andere angewendet, so werden zunächst alle Attribute in den AGG-Graphen der Regel und dem Arbeitsgraphen entfernt (`unbindAttributes()`)[1]. Dann wird die Regel, wie in Abschnitt 6.6 erläutert, angewendet und nach Anwendung der Regel, ob erfolgreich oder nicht, die Attribute wieder an die Graphen gebunden (`bindAttributes()`).

Natürlich werden bei dieser Technik nur vollkommen leere Datentypen erzeugt. Diese müssen dann – wie in Kapitel 7 beschrieben – von Hand bearbeitet werden. Genauso werden auch Parameter nicht interaktiv abgefragt.

Ein weiteres Problem besteht darin, dass – auf Grund einiger Mängel der Attributkomponente des AGG-Systems – auch Konstanten auf der linken bzw. rechten Seite der Regel *nicht* zugelassen sind. Wir lösen das Problem zumindest für rechte Regelseiten,

[1]Die Attribute verschwinden nicht, sondern werden in `GEAttribute` gespeichert.

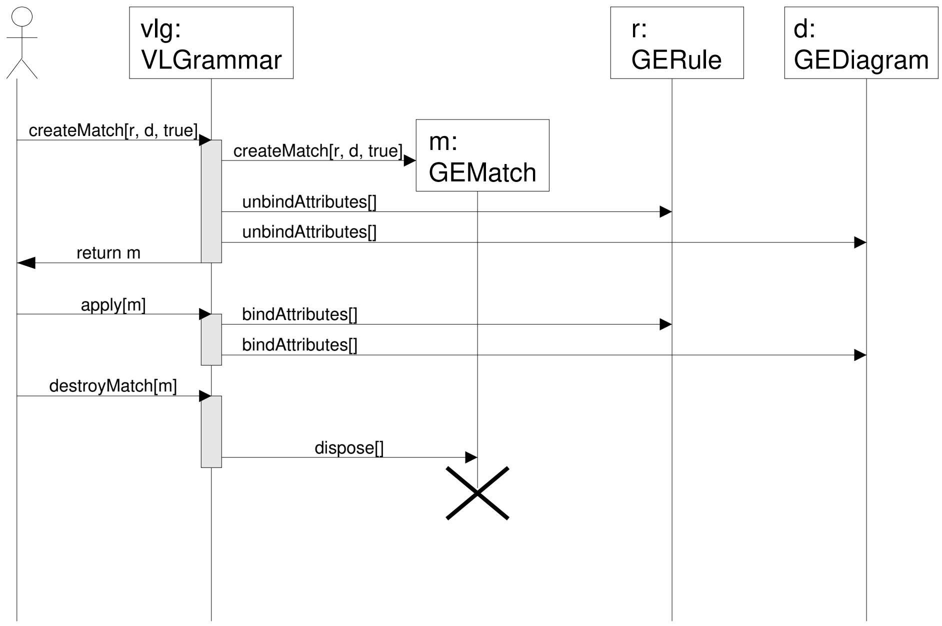



Abbildung 6.8: Sequenzdiagramm: Anwenden einer Regel auf eine Regel

indem wir einen Standard-Java-Ausdruck vorgeben, der neu erzeugte Datentypen bei der Auswertung gleich mit einem Wert versieht. Deshalb muss die dafür verwendete statische Methode `createDatatype(String)` auch von jeder Datentypklasse implementiert werden.

### 6.5.1 Regelparameter

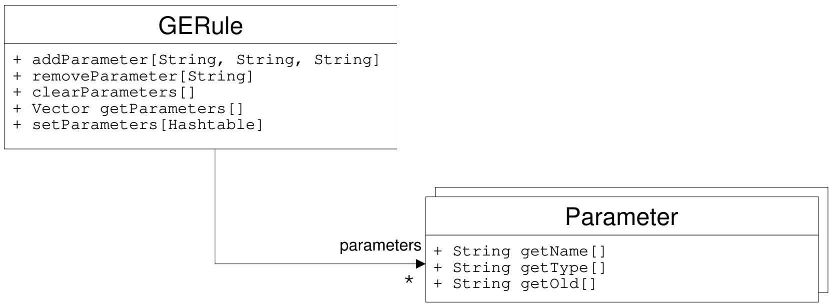



Abbildung 6.9: Klassendiagramm: Parameter

Jede Regel erhält jetzt eine Liste von Parametern. Ein Parameter besteht aus einem Namen, einem Typen und als Erweiterung optional aus dem Namen einer Variablen, die den alten Wert des Parameters enthält und damit auf der linken Regelseite vorhanden sein muss. Bei Anwendung der Regel wird der Wert des Parameters durch einen (datentypspezifischen) Dialog abgefragt, wobei der Wert der alten Variablen, falls vorhanden, als Vorgabewert verwendet wird.

## 6.6 Visuelle Grammatik

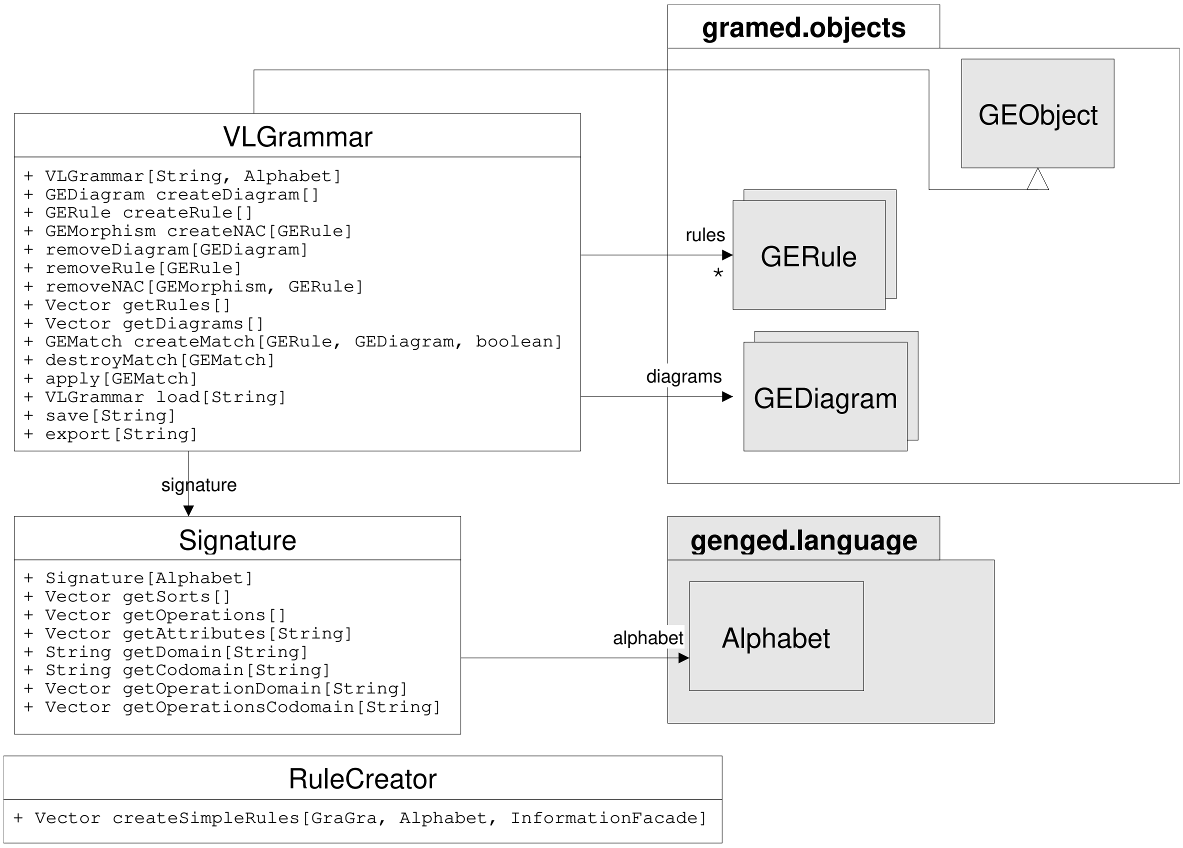



Abbildung 6.10: Klassendiagramm: Das Package `grammar`

Diese Ebene wird in erster Linie durch das Package `grammar` bereitgestellt, eine Übersicht findet sich in Abb. 6.10. Die Hauptbestandteile sind die Klasse `VLGrammar`, die uns eine komplette visuelle Grammatik zur Verfügung stellt sowie die Klasse `RuleCreator`, die dazu dient, den Algorithmus 3.1 zu implementieren, der die einfachen Regeln erzeugt und die Klasse `Signature`. Letztere stellt einen Adapter[2] für ein Alphabet des Alphabeteditors zur Verfügung. Dadurch können wir auf rein textuelle, stringbasierte Art auf das Alphabet und dessen Informationen zugreifen.

### 6.6.1 Klasse `VLGrammar`

Eine neue Grammatik lässt sich nur auf der Basis eines gegebenen Alphabets erzeugen. Die Struktur eines Alphabets wurde in Abschnitt 5.3 beschrieben. Wir setzen auf ein Alphabet einen Adapter `Signature`. Diese Signatur liefert einerseits als Sorten die Namen der grafischen Symbole, andererseits als Operationen die Namen der Links, au-

[2]Ein Adapter bietet eine Schnittstelle für ansonsten inkompatible Objekte. In unserem Fall ist das Alphabet nicht inkompatibel, sondern einfach unbequem zu benutzen. Der *Adapter* ist, wie der *Beobachter*, ein Entwicklungsmuster [GHJV95].

ßerdem Methoden, um Domain und Codomain für eine Operation und die Attribute für ein Symbol auszugeben.

Die Grammatik wird dann mit dem Konstruktor `VLGrammar(String n, Alphabet a)` erzeugt. Dabei geschieht folgendes: Zuerst wird eine neue Graphgrammatik im AGG-System erzeugt. Dieser Graphgrammatik werden nun Knoten- und Kantentypen hinzugefügt, wobei für jede Sorte in der Signatur ein Knotentyp und für jede Operation ein Kantentyp erzeugt wird. Desweiteren werden für jede Sorte alle ihre Datentypen als Deklarationen zum entsprechenden AGG-Knotentyp hinzugefügt. Damit besitzt unsere Graphgrammatik alle Typen, um Graphen für die Regeln erzeugen zu können.

Da wir nicht zulassen, dass von Hand Knoten bzw. Kanten in die Regeln eingefügt bzw. aus ihnen gelöscht werden, benötigen wir einen Satz einfacher Regeln, die aus der vorgegebenen Signatur erzeugt werden. Dafür rufen wir die Klasse `RuleCreator` mit unserer Graphgrammatik und dem Alphabet auf und erhalten einen Satz einfacher Regeln, mit deren Hilfe wir nun kompliziertere aufbauen können. Der gesamte Vorgang wird in Abb. 6.11 dargestellt.

Spätestens an dieser Stelle müssen wir aus praktischen Gründen die logische Ebene verlassen und auf die grafische Ebene wechseln. Die erzeugten Regeln sind nämlich nicht mehr reine Graphregeln, es werden stattdessen Objekte der Klasse `gramed.objects.GERule` erzeugt. Die Grammatik hält jetzt also eine Liste von Grammatikregeln der Klasse `GERule` und eine Liste von Diagrammen der Klasse `GEDiagram`.

Um Regeln bzw. Diagramme zu erzeugen oder zu löschen, können wir die folgenden Methoden verwenden:

**Regel erstellen/löschen** : Mit der Methode `GERule createRule()` wird eine neue visuelle Regel erstellt, die Methode `removeRule(GERule)` löscht die angegebene Regel aus der Menge der Grammatikregeln. Gelöscht werden dürfen nur Regeln, die als *editierbar* gekennzeichnet und nicht gesperrt sind.

**Diagramm erstellen/löschen** : Die Methode `GEDiagram createDiagram()` erzeugt ein neues Diagramm, die Methode `removeDiagram(GEDiagram)` löscht ein Diagramm aus der Grammatik.

**NAC erstellen/löschen** : Mit `GEMorphism createNAC(GERule)` wird eine negative Anwendungsbedingung für eine Regel erzeugt, mit `removeNAC(GERule, GEMorphism)` wieder gelöscht. Auch hier gilt, dass die Regel editierbar sein muss.

Im Fehlerfalle wird eine `GrammarException` geworfen, die von einem Editor aufgefangen werden muss. Da die Grammatik ebenso wie alle Regeln, Diagramme und Morphismen von `GEObject` ist, können wir Beobachter für diese Klasse bestimmen, die dann über neue oder gelöschte Regeln, Diagramme und NACs informiert werden.

Bei der Erzeugung der Grammatik wird außerdem ein leeres *Startdiagramm* erzeugt.

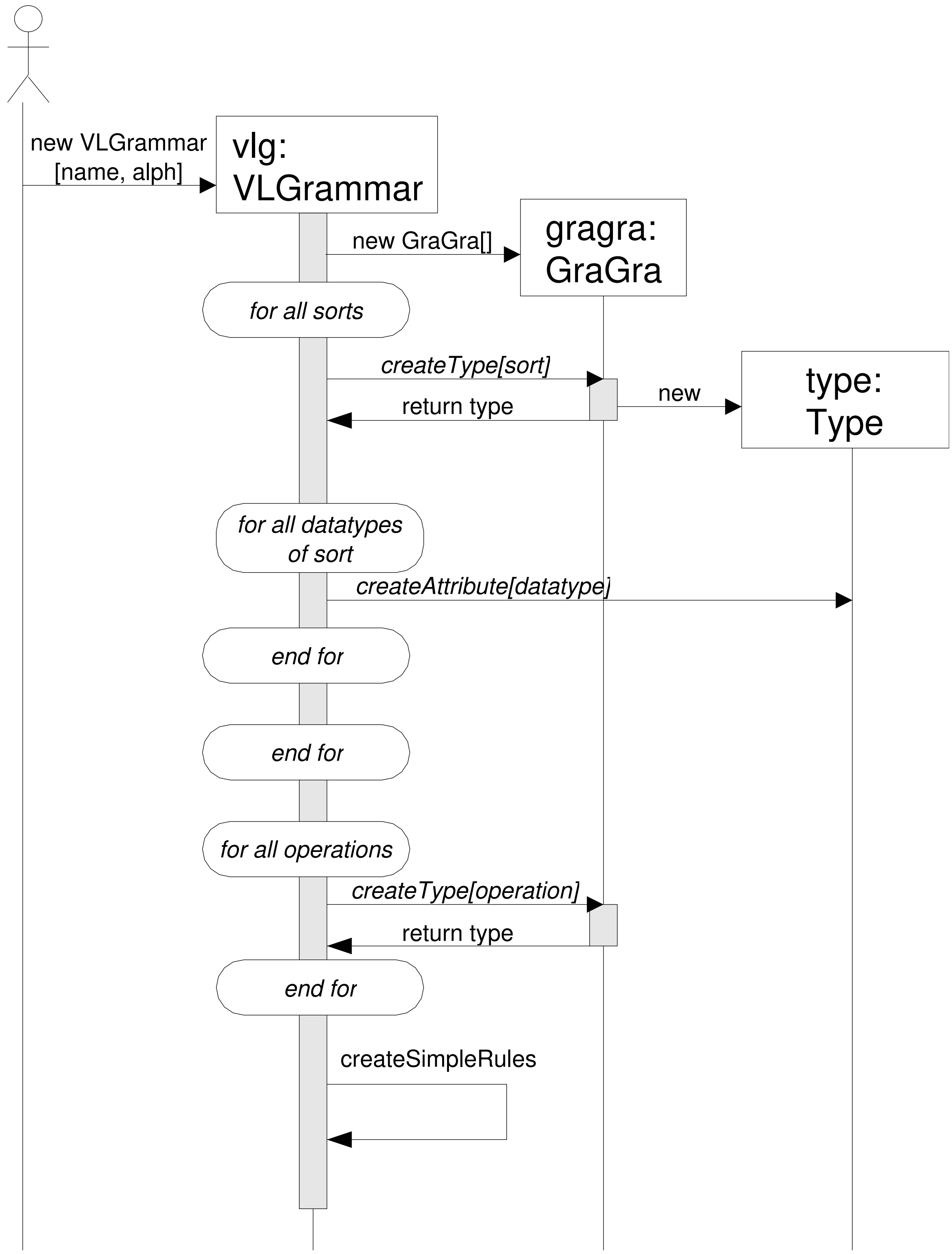



Abbildung 6.11: Sequenzdiagramm: Neue visuelle Grammatik

### 6.6.2 Ansatz und Transformation

Einen (leeren) Ansatz erzeugen wir durch die Methode `GEMatch createMatch(GERule, GEDiagram, boolean)`, der wir die Regel, deren Ansatz gegeben werden soll, und ein Arbeitsdiagramm übergeben. Das Flag in der Methode gibt, wie oben beschrieben, an, ob wir die Regel auf eine weitere Regel anwenden oder auf ein normales Arbeitsdiagramm. Im ersteren Falle werden keine Attribute betrachtet. Es muss darauf geachtet werden, dass jeder Ansatz auch wieder mit `destroyMatch(GEMatch)` gelöscht wird, da das AGG-System ansonsten nicht mehr mit den Morphismen zurechtkommt. Außerdem werden spätestens beim Zerstören eines Matches die Attribute wieder an die AGG-Graphen gebunden.

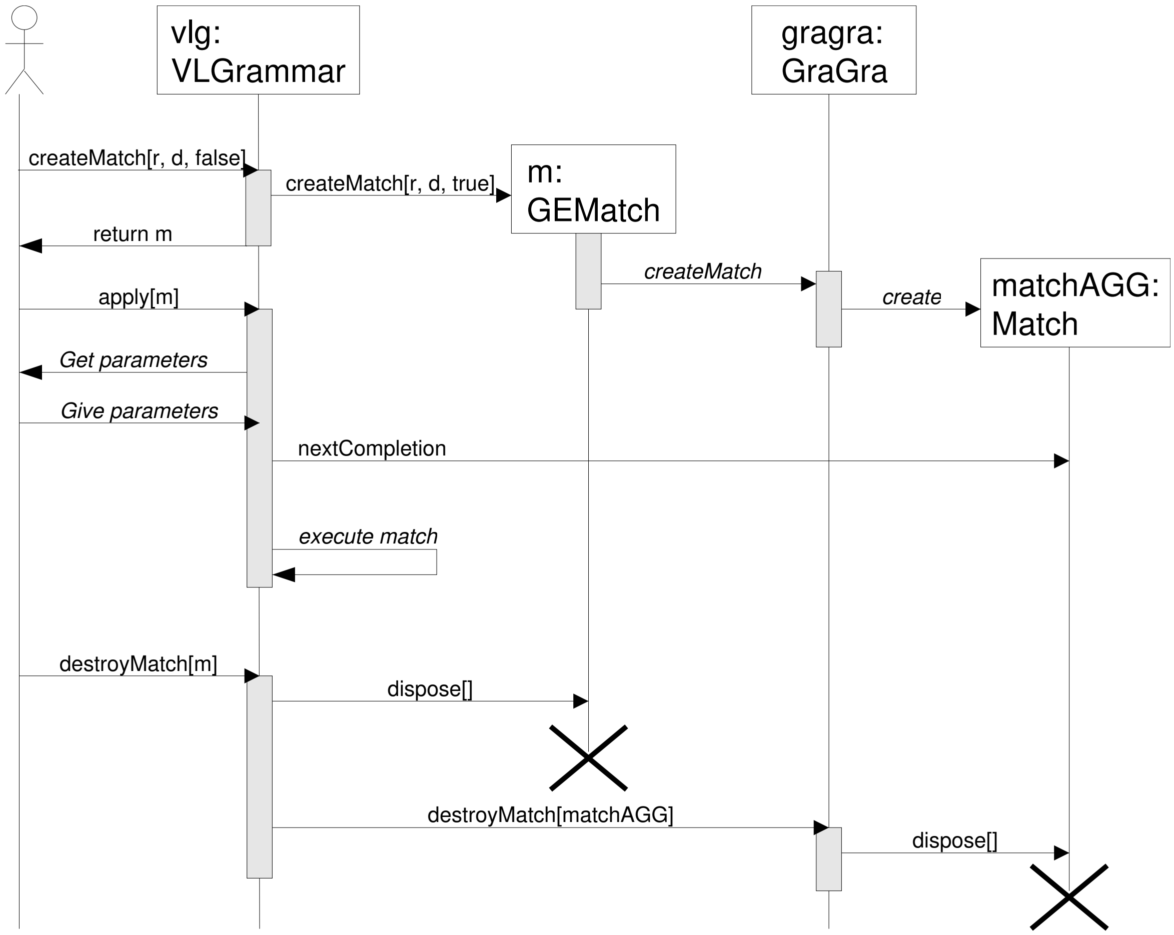



Abbildung 6.12: Sequenzdiagramm: Normale Regelanwendung

In Abb. 6.12 wird noch einmal aufgezeigt, wie eine Transformation eines normalen Arbeitsdiagramms mit Hilfe des AGG-Systems stattfindet:

- Es wird ein Match erzeugt (diesmal mit dem Flag `false`, da wir auf einem normalen Graphen arbeiten).
- Dieser Match kann jetzt mit Hilfe der Methoden der Klasse `GEMorphism` ver-

vollständigt werden.

- Durch `apply(GEMatch)` wird dieser Match angewendet. Sofern er noch nicht vollständig, also total war, wird er durch `nextCompletion` vervollständigt.
- Jetzt wird das Arbeitsdiagramm kopiert und die Graphtransformation im AGG-System durchgeführt.
- Zuerst wird jetzt, wie in Abschnitt 4.4.1, der AGG-Graph auf Korrektheit geprüft.
- War die Regelanwendung erfolgreich, wird der Graph für das Arbeitsdiagramm neu gesetzt, was zur Folge hat, dass automatisch alte Symbole und Links gelöscht und die neuen hinzugefügt werden.
- Ist die Transformation fehlgeschlagen, wird eine Exception geworfen und das alte Diagramm wiederhergestellt.

### 6.6.3 Erzeugung einfacher VL-Regeln

Die Klasse `RuleCreator` stellt nun eine direkte Implementierung des Algorithmus' 3.1 dar. Bevor der Algorithmus allerdings damit beginnt, die einfachen Einfüge-, Lösch- und Änderungsregeln zu erzeugen, wird der Typgraph bzw. die Signatur auf Zyklen geprüft und ggf. eine *Exception* erzeugt. Die statische Methode `Vector createSimpleRules(Alphabet a, GraGra g)` liefert einerseits alle einfachen Regeln als Objekte der Klasse `gramed.objects.GERule`, verändert andererseits aber auch die Graphgrammatik dergestalt, dass die erzeugten Graphregeln dort mit aufgenommen sind. Alle so generierten Regeln sind *geschützt*.

## 6.7 Grafische Ebene

Jetzt, wo wir die Basis für die Transformation von Diagrammen mit visuellen Regeln geschaffen haben, werden wir uns auf die Darstellung der Grammatikelemente konzentrieren. Jede Sicht auf eine Regel oder ein Diagramm wird durch ein *Display* im Package `gramed.display` bereitgestellt. Jedes Display implementiert das Interface `gramed.interfaces.GEObserver` mit der Methode `void update(GEObject o, UpdateInfo changes)`. Die Klasse `gramed.util.UpdateInfo` implementiert ein einfaches String/Objekttupel, in dem wir eine Änderungsbeschreibung und geänderte Objekte selbst speichern können. Wir stellen im Folgenden die Displays mit ihrer jeweiligen Funktionalität vor.

### 6.7.1 `GraphDisplay`, `DiagramDisplay` und `DebugDisplay`

Es gibt zwei Displays für Diagramme, die beide von der abstrakten Klasse `GraphDisplay` abgeleitet sind: `DiagramDisplay` bietet eine VL-Sicht auf ein Diagramm, während `DebugDisplay` es ermöglicht, den Graphanteil, dh. die logische Ebene eines Diagramms zu betrachten.

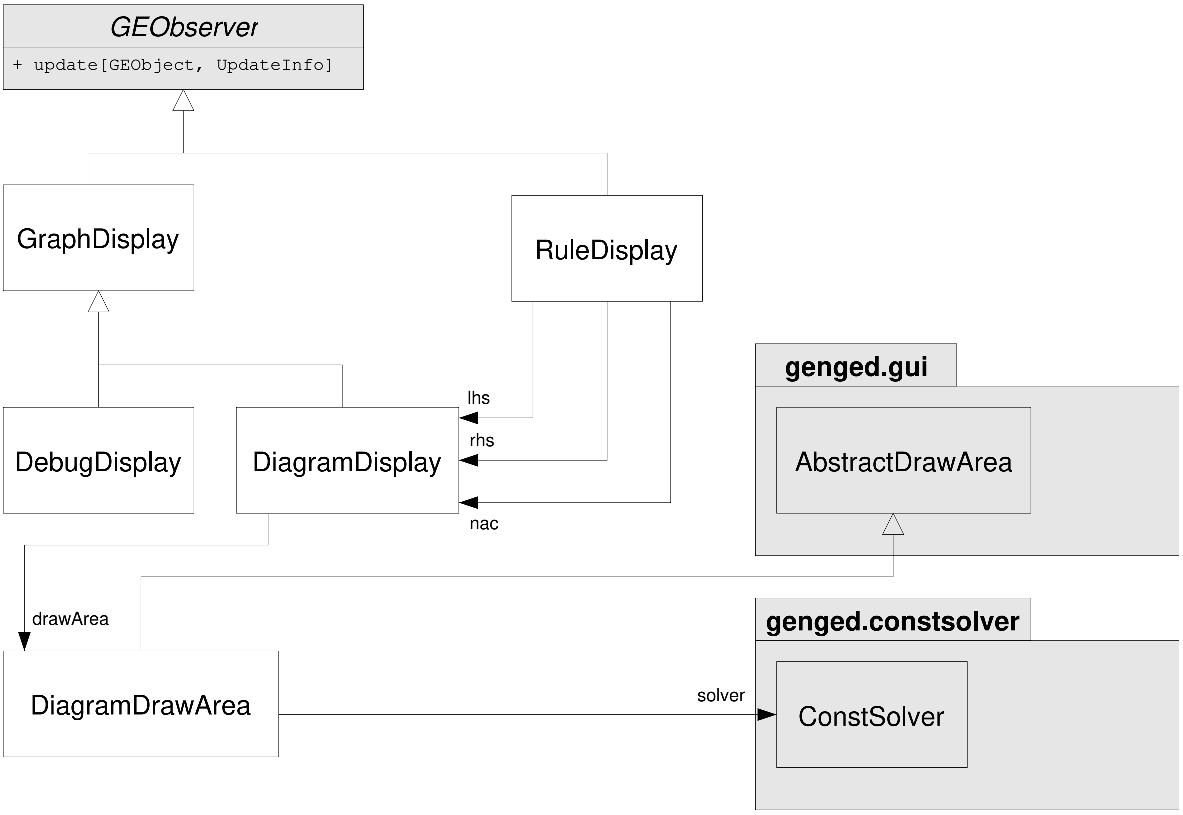



Abbildung 6.13: Klassendiagramm: Das Package `display`

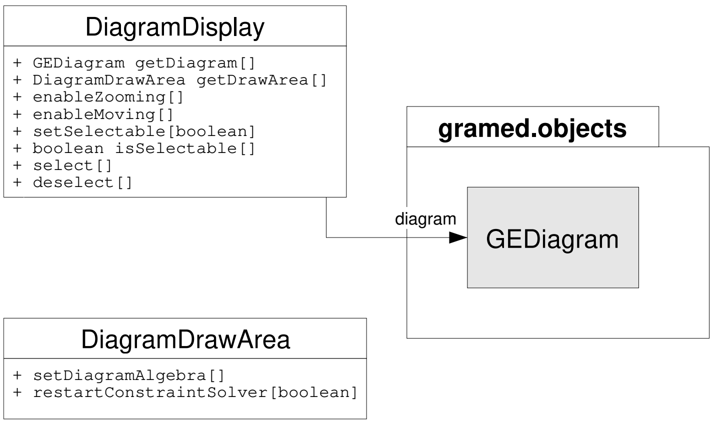



Abbildung 6.14: Klassendiagramm: Die Klasse `DiagramDisplay`

Das `DiagramDisplay` liefert einen Beobachter für ein Objekt der Klasse `gramed.objects.GEDiagram`. Die eigentliche Zeichenfläche ist die Klasse `Diagram-DrawArea`, abgeleitet von der Klasse `AbstractDrawArea` im Alphabeteditor[3]. Um die Darstellung des Diagramms gerade bezüglich der Constraints in dem Diagramm konsi-

[3]Für die Darstellung wollten wir das Rad nicht neu erfinden und deshalb auf die GUI-Klassen des Alphabeteditors zurückgreifen.

stent zu halten, erhält die Zeichenfläche eine Referenz auf einen Constraint-Solver, eine Java-Klasse, die eine einfache Schnittstelle zum Parcon-System herstellt und außerdem Referenzen zu allen grafischen Primitiva und Symbolen hält, die durch Constraints gebunden sind. Jede Zeichenfläche hat ihren eigenen Constraint-Solver. Die Information, dass sich das zugrundeliegende Diagramm geändert hat oder gar ausgetauscht wurde, wird so aufgefangen, dass alle Constraints, die nicht mehr aktuell sind, beim Constraintsolver abgemeldet werden und alle, die neu hinzugekommen sind, dort angemeldet werden.

Von der Funktionalität her bietet diese Zeichenfläche außerdem die Möglichkeit, Symbole zu verschieben und ihre Größe zu ändern sowie den Zoomfaktor der Zeichenfläche zu ändern. Wir erweitern das Ganze noch durch ein Pop-Up-Menü, mit Hilfe dessen man Objekte verlagern kann. Alle diese Funktionalitäten werden im Handbuch, Kapitel 7 erklärt.

Ebenso finden wir auch in `DebugDisplay` eine Sicht auf ein Diagramm vor – der Unterschied besteht darin, dass hier darauf verzichtet wird, mit konkreten grafischen Symbolen und Constraints zu arbeiten und stattdessen Knoten und Kanten des zugrundeliegenden AGG-Graphen dargestellt werden. Es findet kein Layout statt, der Benutzer muss selbst die Knoten dergestalt verschieben, dass sich eine für ihn befriedigende Sicht auf das Diagramm ergibt.

### 6.7.2 Klasse **`RuleDisplay`**

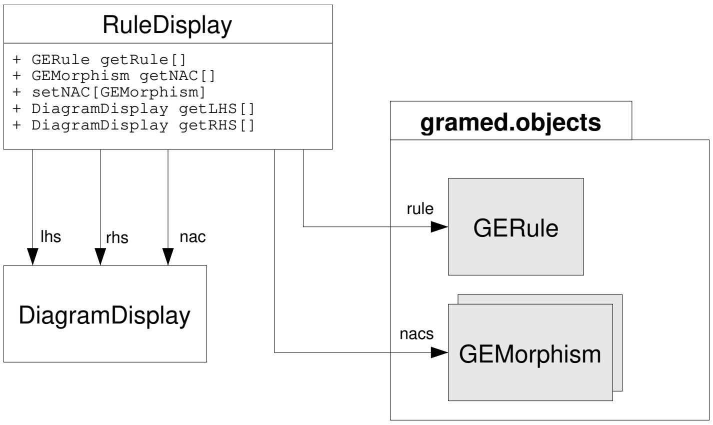



Abbildung 6.15: Klassendiagramm: Die Klasse `RuleDisplay`

Ein Objekt der Klasse `RuleDisplay` ist jetzt nichts weiter als eine Aggregation von drei `DiagramDisplay`-Objekten. Von diesen stellen zwei die linke und rechte Seite einer Regel dar, das dritte Display dient zur Darstellung der aktuell ausgewählten NAC der Regel. Die Methoden `GEMorphism getNAC()` und `setNAC(GEMorphism)` dienen dazu, die Anzeige im dritten (NAC)Display zu steuern und abzufragen.

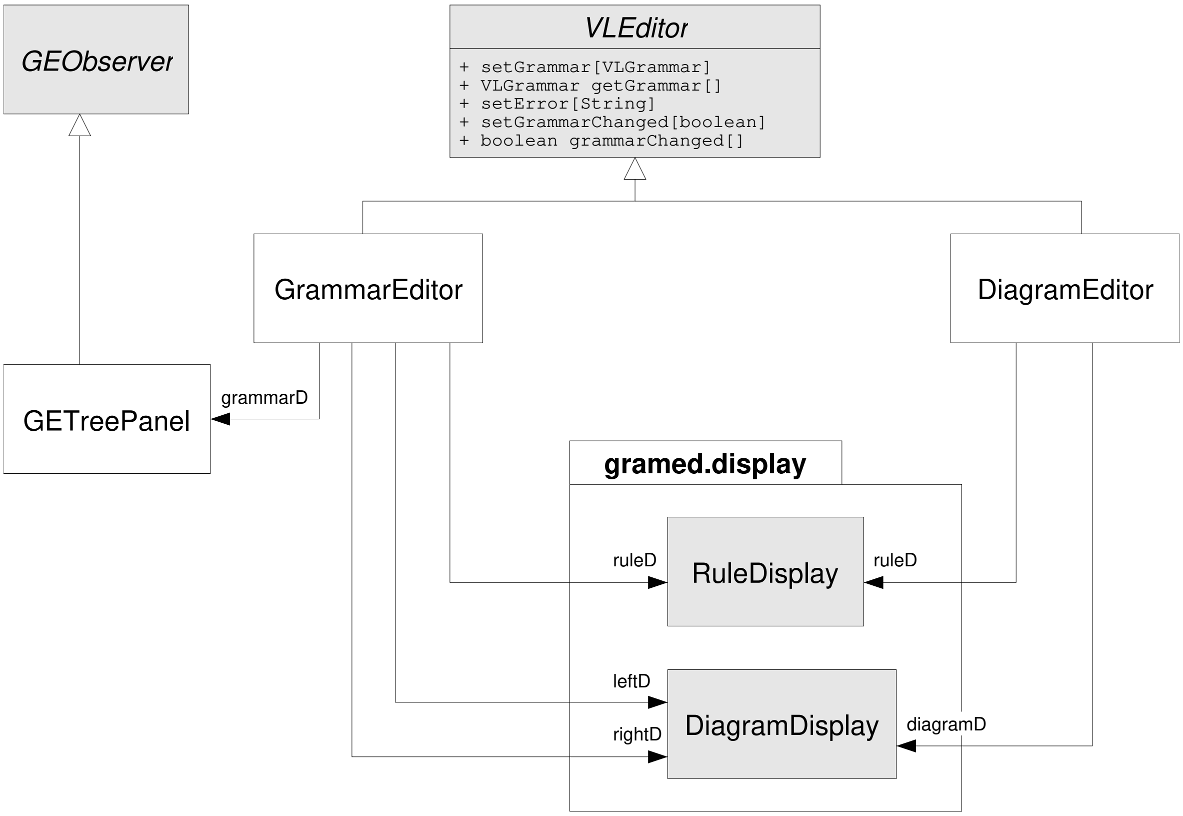



Abbildung 6.16: Klassendiagramm: Das Package `gui`

## 6.8 Benutzungsoberfläche

Die Funktionalität der Benutzungsoberfläche (GUI) soll hier nicht beschrieben werden, das wird im Handbuch, Kapitel 7, ausführlich getan. Hier soll nur ein Überblick über die verwendeten Klassen gegeben werden.

Das Interface `VLEditor` haben wir eingeführt, weil sich so die meisten der Editoraktionen im Package `gramed.gui.actions` sowohl für den Grammatikeditor `GrammarEditor` als auch den Diagrammeditor `DiagramEditor` verwenden lassen. Der Grammatikeditor dient dazu, Regeln mit Hilfe von Regeln zu bauen. Er besteht aus einer Werkzeugleiste, einem Baumdisplay, einem Regeldisplay und zwei Diagrammdisplays, in denen sich entweder linke und rechte Seite oder linke Seite und NAC einer zu bearbeitenden Regel befinden. Wird das Startdiagramm bearbeitet, so ist das rechte Diagrammdisplay leer und das linke nimmt das Startdiagramm auf.

Der Diagrammeditor sieht ganz genauso aus, nur das die Funktionalität eingeschränkt ist und statt der Arbeitsfläche mit zwei Diagrammdisplays nur eines vorhanden ist.

Das oben genannte Baumdisplay `GETreePanel` bietet einen Blick auf eine gesamte Grammatik und dazu die Möglichkeiten, Namen von Regeln, NACs und Diagrammen zu ändern und Objekte zu selektieren, die dann im Editor weiter verwendet und bearbeitet werden. Das Baumdisplay passt sich automatisch dem Editor an, in dem es benutzt wird, so ist es im Diagrammeditor nicht möglich, Namen von Regeln oder NACs zu ändern.

### 6.8.1 Visualisierung der Morphismen

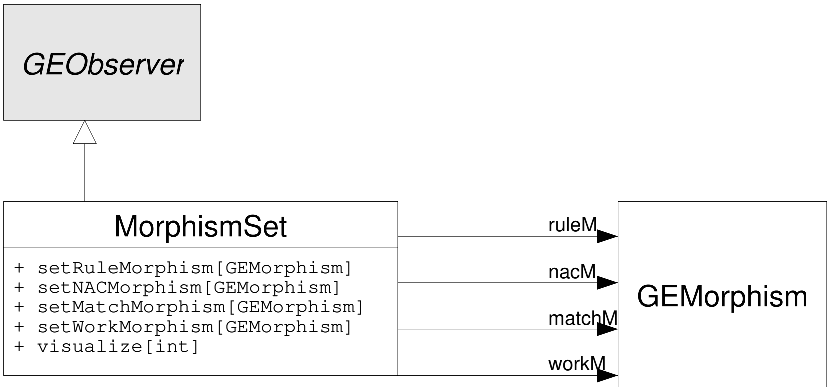



Abbildung 6.17: Klassendiagramm: Die Klasse `MorphismSet`

Da die Visualisierung der Morphismen ein Hauptproblem der Benutzungsoberfläche war, wollen wir noch einen Blick auf die Implementierung werfen.

Alle Morphismen, die zu visualisieren sind, werden in der Klasse `MorphismSet` gehalten. Dazu gehört der Regelmorphismus der Regel, die ausgewählt ist und auf ein Diagramm angewendet wrden soll, der NAC-Morphismus ebendieser Regel (falls eine NAC ausgewählt oder überhaupt vorhanden ist), der Morphismus, der den Match der Regel in das Arbeitsdiagramm beschreibt sowie der Morphismus, an dem im Arbeitsbereich gerade gearbeitet wird. Mit den entsprechenden `set`-Methoden können diese vom Editor geändert werden, die Methode `visualize(int)`, aufgerufen mit einem der Modi „Alle“, „Regelmorphismus“, „NAC-Morphismus“, „Match“, „Arbeitsmorphismus“ oder „Keiner“, sorgt dann dafür, dass genau die Morphismen angezeigt werden, die spezifiziert wurden.

Die Anzeige der Morphismen wird durch Zahlen deutlich gemacht, wobei gleiche Zahlen Abbildungen bedeuten. Werden zwei Symbole auf dasselbe abgebildet, erhält das Zielsymbol beide Zahlen, durch Kommas getrennt. Um größere Schwierigkeiten bei der Anzeige zu vermeiden (Hintereinanderschaltung von Morphismen), wird der Arbeitsmorphismus anders dargestellt als die anderen Morphismen.

`MorphismSet` implementiert `GEObserver`, und zwar beobachtet es seine Morphismen und ändert die Darstellung, wenn sich die Morphismen ändern.

## 6.9 Stand der Implementierung

Fast alle Elemente der vorgestellten Theorie sind im Augenblick implementiert. Was noch wünschenswert wäre, sind folgende Features:

- Die Benutzungsoberfläche sollte an den Alphabeteditor angepasst werden, auf Dauer ist auch eine engere Kopplung denkbar, so dass zwischen Alphabet- und Grammatikdefinition der visuellen Sprache fließend gewechselt werden kann.
- Letzteres würde auch eine Implementierung intelligenter Undo-Mechanismen implizieren.
- Die Visualisierung ist noch nicht elegant gelöst.
- Was unbedingt noch gemacht werden muss, ist der Umbau des Diagrammeditors zu einem Grafikeditor, in dem die Regeln zu Editieroperationen (`insert`, `delete`) werden. Dazu sollten die Regeln dahingehend erweitert werden, dass der Match auf irgendeine geschickte Art angelegt wird.

# Kapitel 7

# Benutzerhandbuch

In diesem Kapitel wird die Installation, der Aufruf und die Benutzung des Grammatikeditors und des dazugehörigen Diagrammeditors beschrieben. Da beide Editoren auf grafischen Alphabeten basieren, die mit dem Alphabeteditor von Manuel Schwarze [Sch99] erstellt werden, ist es angeraten, zunächst das sehr ausführliche Handbuch dieses Editors durchzuarbeiten, da das vorliegende Handbuch darauf aufbaut.

Im oben genannten Handbuch wird der Aufbau eines grafischen Alphabets mit entsprechenden Symbolen, Datentypen, Links und Constraints sehr detailliert dargestellt, so dass wir uns in der Beschreibung des Grammatikeditors ganz auf den Aufbau der tatsächlichen visuellen Sprache konzentrieren können. Als laufendes Beispiel verwenden wir allerdings - um die Kontinuität zur Fallstudie zu gewährleisten - die vorher vorgestellte Statecharts-Notation.

## 7.1 Installation und Start

### 7.1.1 Hardware-Voraussetzungen

Der Grammatikeditor wurde auf einem Intel-Pentium-System (133 MHz) unter S.u.S.E. Linux 6.0 entwickelt, was allerdings nicht bedeutet, dass dieses System – besonders die Hardware– für den dauerhaften Betrieb des Editors ausreicht. Wir empfehlen daher folgende Mindestvoraussetzungen für die Hardware:

**Linux:** Intel-Pentium 233 (oder vergleichbare Prozessoren) mit 64 Megabyte Hauptspeicher

**Solaris:** Sun 10 Workstation mit 64 Megabyte Hauptspeicher

**Bildschirm:** 17 Zoll-Bildschirm mit mindestens 1024x768 Pixeln Auflösung, wie immer bei visuellen Tools gilt: je größer, desto besser

Die Einschränkung auf die zwei genannten Plattformen ergibt sich dadurch, dass der Constraint-Solver Parcon nur unter diesen Systemen läuft.

### 7.1.2 Software-Voraussetzungen

Als Betriebssystem kommt Solaris oder Linux in Frage, wobei das System mit dem Linux-Kernel 2.0 und Solaris 2.0 getestet wurde. Außerdem wird folgendes benötigt:

- eine funktionsfähige Installation des Alphabeteditors, wie in dessen Handbuch beschrieben,
- eine funktionsfähige Installation des AGG-Systems,
- das Java Development Kit von Sun Microsystems (mindestens in der Version 1.1.6, unter Linux 1.1.7),
- die Java Foundation Classes (auch „Swing“ Classes genannt) in der Version 1.1, alternativ dazu auch Java 2, dort sind die JFC integriert.

Die Umgebung muss für den Betrieb des Alphabeteditors eingerichtet worden sein (Umgebungsvariable `$CLASSPATH` entsprechend gesetzt).

### 7.1.3 Installation

Das System wird in einem gepackten Tar-File zusammen mit einem Installations-Shell-Skript (`install.sh`) ausgeliefert (gramed.XX.tgz). Beide Dateien sollten in ein temporäres Verzeichnis kopiert und dann das Installationsskript aufgerufen werden. Als Argument erhält das Skript das Verzeichnis, in das der Grammatikeditor installiert werden soll. Es empfiehlt sich, ein Verzeichnis zu wählen, das schon in der Umgebungsvariablen `$CLASSPATH` vorhanden ist.

### 7.1.4 Aufruf

Zum Start des Systems wird im Verzeichnis `gramed` ein Installationsskript (`GramEditor`) bereitgestellt. Aufgerufen wird das System entweder mit

`GramEditor` (Startet den Grammatikeditor)

oder mit

`GramEditor -test` (Startet den Diagrammeditor).

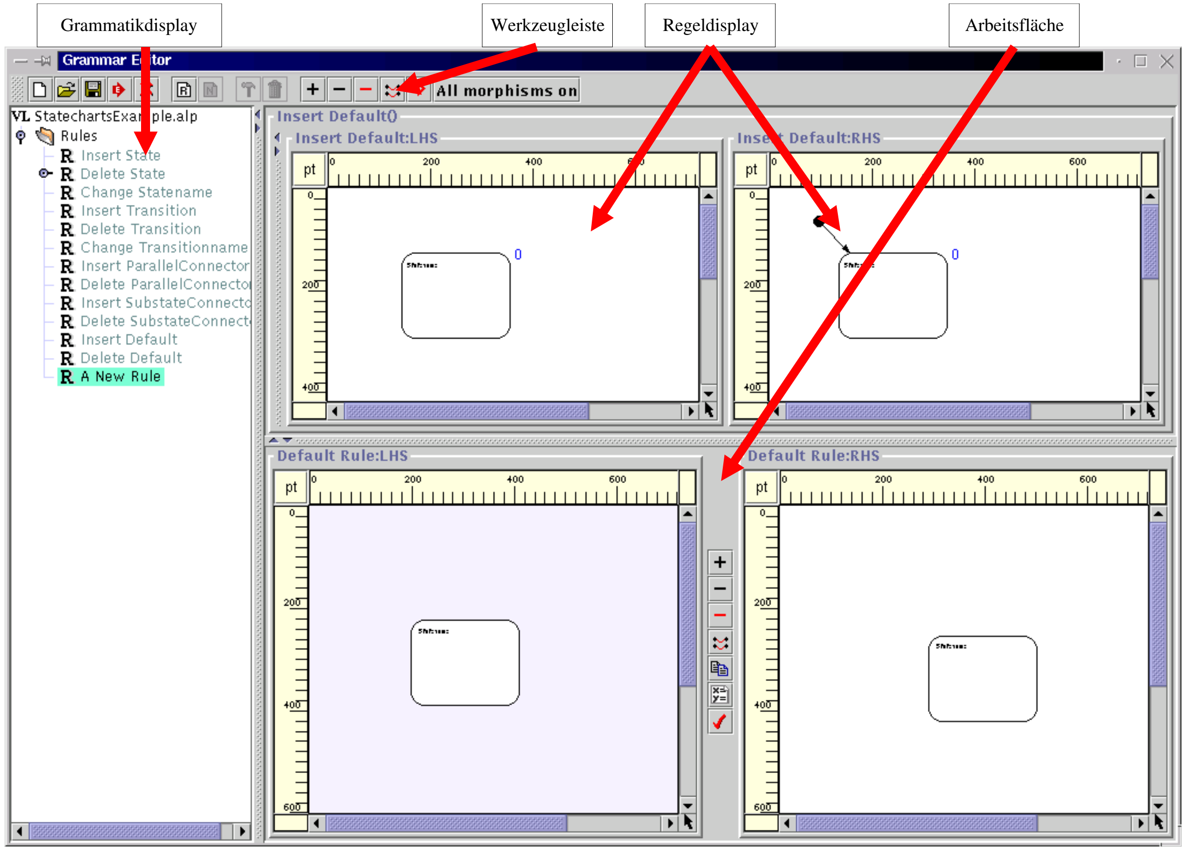



Abbildung 7.1: Der Grammatikeditor

## 7.2 Überblick

Das Hauptfenster des Grammatikeditors stellt sich wie in Abbildung 7.1 dar.

Am oberen Rand befindet sich eine *Werkzeugleiste*, die fast alle Funktionen des Editors integriert. Auf der linken Seite ist ein baumartiges *Grammatikdisplay*, in dem eine Grammatik mit ihren Regeln im Überblick angezeigt werden. Die rechte Seite nimmt der eigentliche Editor ein: In der oberen Hälfte finden wir ein *Regeldisplay*, das die Regel, die wir anwenden wollen, aufnimmt; in der unteren Hälfte haben wir zwei Diagrammdisplays, mit Hilfe derer eine neue Regel bearbeitet wird. Zwischen den *Arbeitsdisplays* befindet sich eine weitere *Werkzeugleiste*, die benutzt wird, um die Regel zu definieren.

Zu beachten sind die senkrechten und waagerechten Trennbalken, mit denen man durch Anklicken mit der linken Maustaste und Ziehen die einzelnen Fenster von der Größe her anpassen kann. Es befindet sich ein Trennbalken zwischen dem Grammatikdisplay und dem übrigen Editor, ein Trennbalken zwischen dem Regeldisplay und den zwei Arbeitsdisplays sowie ein Trennbalken zwischen dem Display für die negative Anwendungsbedingung und der linken/rechten Regelseite.

### 7.2.1 Werkzeugleiste

Hier sind fast alle Funktionen des Editors untergebracht.

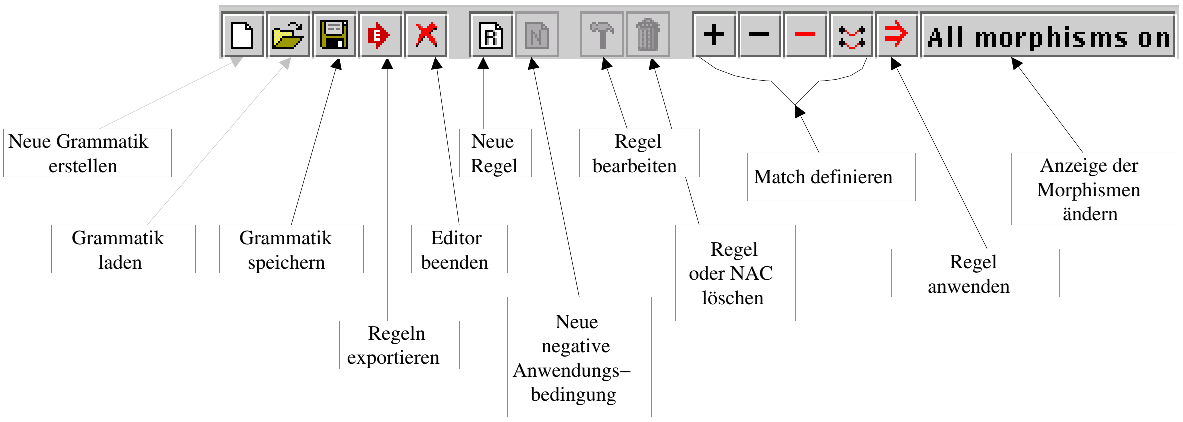



Abbildung 7.2: Die Werkzeugleiste

#### Funktionen für die Grammatik

 Erstellt eine neue Visuelle Grammatik auf der Grundlage eines grafischen Alphabets. Der Benutzer wird aufgefordert, ein Alphabet anzugeben. Dann werden die einfachen Regeln mit Hilfe des Algorithmus in 3.1 erzeugt.
**Fehlermeldung**: Es gibt Zyklen im Typgraphen oder die Datei ist nicht lesbar.

 Lädt eine Visuelle Grammatik aus einer Datei.
**Fehlermeldung**: Die Datei ist nicht lesbar.

Speichert eine Visuelle Grammatik in eine Datei.
**Fehlermeldung**: Die Datei ist nicht schreibbar.

Speichert eine Visuelle Grammatik, die nur die Regeln enthält, die als *exportiert* gekennzeichnet sind ().
**Fehlermeldung**: Die Datei ist nicht schreibbar.

Beendet den Editor. Hier wird zunächst abgefragt, ob die Visuelle Grammatik, die bearbeitet wurde, gespeichert werden soll.
**Fehlermeldung**: Die Datei ist nicht schreibbar.

### Funktionen für die Regelbearbeitung

Erzeugt eine neue Regel mit leerer linker und rechter Regelseite und ohne negative Anwendungsbedingungen (NAC). Die Regel wird in die Regelliste im Grammatikdisplay eingefügt.

Erzeugt eine neue negative Anwendungsbedingung für die im Grammatikdisplay ausgewählte Regel. Ist dort keine Regel ausgewählt, wird die Regel genommen, die gerade in Arbeit ist, falls eine solche existiert.
**Fehlermeldung**: Die ausgewählte Regel ist nicht editierbar.

Übernimmt die ausgewählte Regel zur Bearbeitung. Die Regel wird im Grammatikdisplay farbig unterlegt und die linke und rechte Regelseite werden im den Arbeitsdisplays angezeigt.
**Fehlermeldung**: Die ausgewählte Regel ist nicht editierbar.

Löscht eine Regel oder eine negative Anwendungsbedingung.
**Fehlermeldung**: Das ausgewählte Objekt ist nicht editierbar oder die Regel ist in Bearbeitung.

### Funktionen für das Matchen und Anwenden einer Regel

+ Fügt dem Match zwischen der linken Regelseite und dem ausgewählten Diagramm eine Abbildung hinzu. Dafür muss sowohl in der linken Regelseite als auch im ausgewählten Arbeitsdisplay genau ein Symbol markiert sein. Wird ein Symbol auf der linken Regelseite verwendet, für das schon eine Abbildung existiert, so wird die alte Abbildung durch die neue ersetzt.
**Fehlermeldung**: Die Auswahl ist falsch oder die Abbildung ist nicht möglich.

− Löscht eine Abbildung aus dem Match zwischen der linken Regelseite und dem ausgewählten Diagramm. Dafür muss zumindest im ausgewählten Arbeitsdisplay genau ein Symbol markiert sein, da die zu löschende Abbildung sonst nicht eindeutig ist.
**Fehlermeldung**: Die Auswahl ist falsch oder die Abbildung existiert nicht.

− Löscht alle Abbildungen aus dem Match zwischen der linken Regelseite und dem ausgewählten Diagramm.

Vervollständigt die Abbbildungen im Match zwischen der linken Regelseite und dem ausgewählten Diagramm. Da für die Anwendung der Regel (siehe Abschnitt 2.3) der Match total sein muss, bietet die Funktion die Möglichkeit, die Vervollständigung zu automatisieren und den enstandenen Match anzuzeigen und zu bearbeiten.
**Fehlermeldung**: Der Match kann nicht vervollständigt werden.

Verwendet den definierten Match, um die Regel auf das ausgewählte Diagramm anzuwenden. Ist der Match noch nicht total gewesen, so wird er spätestens jetzt vervollständigt.
**Achtung**: Da wir in diesem Editor Regeln auf Regeln anwenden, werden die Datentypen im Arbeitsdiagramm **nicht** verändert - wenn die Regel Java-Ausdrücke in den Datentypen enthält, werden diese nicht ausgewertet und wenn neue Symbole erzeugt werden, sind deren Datentypen leer und die Java-Ausdrücke in den neu erzeugten Datentypen müssen von Hand editiert werden. Wie das geht, steht im Abschnitt über die Bearbeitung von Regeln!
**Fehlermeldung**: Der Match kann nicht vervollständigt werden.

### Visualisierung von Morphismen und Abbildungen

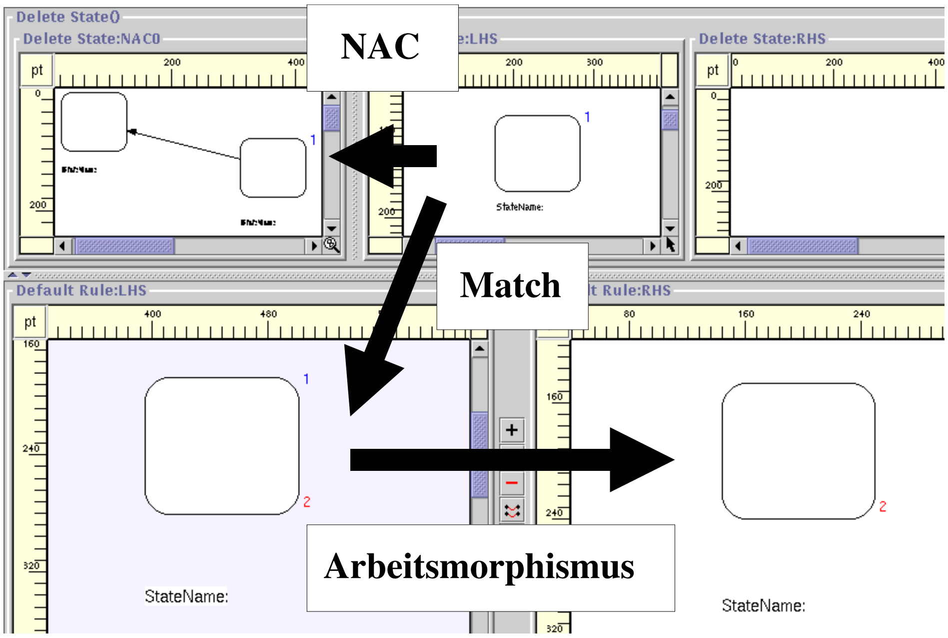



Abbildung 7.3: Morphismen

Die letzte Schaltfläche in der Werkzeugleiste dient dazu, zwischen den einzelnen Visualisierungen für die Morphismen im Editor umzuschalten. Ein Morphismus besteht immer aus einer Menge von Einzelabbildungen zwischen Symbolen. Diese Abbildungen werden durch gleiche Zahlen auf beiden Seiten dargestellt. Werden zwei Symbole auf dasselbe abgebildet, werden die Zahlen am Zielsymbol durch Kommas getrennt dargestellt. Der Morphismus, der in der zu bearbeitenden Regel angezeigt ist, wird durch rote, alle anderen Morphismen durch blaue Zahlen dargestellt. Es gibt fünf Modi:

**All morphisms on** : Alle Abbildungen werden dargestellt.

**Show rule morphism** : Nur die Abbildung der ausgewählten Regel von der linken in die rechte Regelseite wird dargestellt.

**Show NAC morphism** : Nur die Abbildung der ausgewählten Regel von der linken Regelseite in die negative Anwendungsbedingung (falls vorhanden) wird dargestellt.

**Show match morphism** : Nur der Match von der linken Regelseite in das Arbeitsdiagramm wird (falls vorhanden) dargestellt.

**Show work morphism** : Nur die Abbildung vom linken in das rechte Arbeitsdisplay wird dargestellt.

**All morphisms off** : Die Anzeige der Abbildungen wird ausgeschaltet.

### 7.2.2 Grammatikdisplay

In diesem Display können Regeln ausgewählt werden, einerseits um diese Regeln auf die →Arbeitsdiagramme anzuwenden, andererseits um Regeln weiter zu bearbeiten oder Regeln bzw. negative Anwendungsbedingungen zu löschen. Außerdem wird dieses Display dafür verwendet, die Namen von Regeln und negativen Anwendungsbedingungen zu ändern sowie Regeln für den Export zu markieren. Und mit Hilfe dieses Displays kann das Startdiagramm zur Bearbeitung ausgewählt werden.

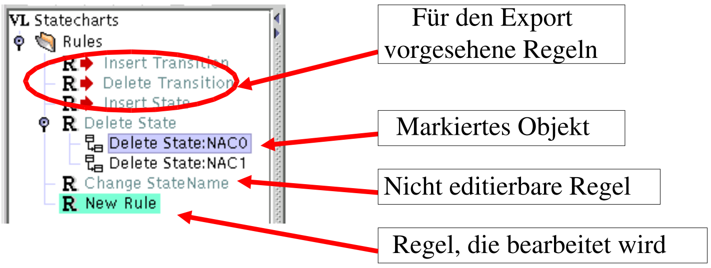



Abbildung 7.4: Das Grammatikdisplay

Das Display verwendet folgende Darstellungen:

- VL Die Visuelle Grammatik.
- R Eine Regel der Visuellen Grammatik.
- Eine negative Anwendungsbedingung.
- R Eine für den Export vorgesehene Regel.

Regeln, die automatisch erzeugt wurden und nicht editierbar sind, werden in blaugrüner Schrift dargestellt, alle editierbaren Regeln in schwarzer Schrift. Die Regel, an der gerade gearbeitet wird, ist türkis hinterlegt. Markierte Objekte bekommen einen blaugrauen Hintergrund.

Dieses Display hält folgende Funktionen bereit:

**Anklicken einer Regel mit der linken Maustaste:** Die Regel wird selektiert und im →Regeldisplay angezeigt, falls sie sich nicht gerade in Bearbeitung befindet.

**Anklicken einer Regel mit der rechten Maustaste:** Die Regel wird, falls sie nicht gerade bearbeitet wird, für den Export ausgewählt und das Icon vor dem Regelnamen ändert sich. War die Regel bereits zum Export vorgesehen, wird die Exportmarkierung entfernt.

**Doppelklick auf einen Regel- oder NAC-Namen:** Falls die Regel oder die NAC editierbar ist, kann der Name geändert werden und mit `Enter` gesetzt werden.

Außerdem können alle Regeln sowie die Grammatik ein- bzw. ausgeblendet werden, diese Funktionalität unterscheidet sich in nichts von der aus Dateimanagern u. ä. bekannten.

### 7.2.3 Diagrammdisplays

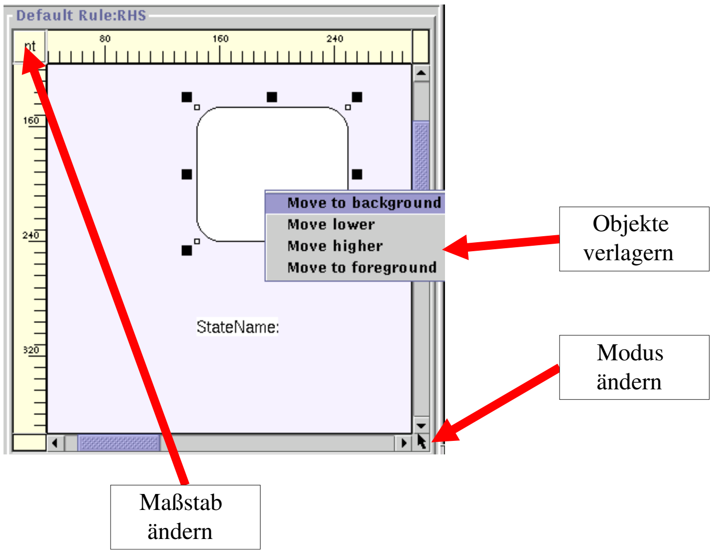



Abbildung 7.5: Das Diagrammdisplay

Diese Anzeige taucht sowohl im Regeldisplay als auch in den Anzeigen für die zu bearbeitende Regel auf. Hierbei handelt es sich um eine Anzeige mit fast derselben Funktionalität wie das Diagrammdisplay im Testsystem des Alphabeteditors (siehe dort).

Im Kopf eines Diagramms wird der Name des jeweiligen Diagramms dargestellt. Der Maßstab eines Diagramms wird auf dem oberen und rechten Lineal angezeigt, mit Klicken in die linke obere Ecke kann der Maßstab geändert werden. Alle Objekte im Zeichenbereich können durch Klicken oder Aufziehen eines Rahmens markiert werden, markierte Objekte können dann wiederum mit Drag&Drop verschoben werden. Außerdem kann die Größe von Objekten geändert werden, indem man die dicken schwarzen „Griffe“ verwendet, die am Rand eines markierten Objekts erscheinen.

Nachdem die Maus losgelassen wurde, findet ein automatisches Lösen aller Constraints im entsprechenden Diagramm statt, ebenso nach dem Anwenden einer Regel - wenn Objekte zum Diagramm hinzugekommen sind, Objekte gelöscht wurden oder Datentypwerte sich geändert haben.

Jedes Diagrammdisplay kann zwei Zustände annehmen, zwischen denen mit der Schaltfläche in der rechten unteren Ecke umgeschaltet werden kann:

Dies ist der normale Modus, in dem Objekte selektiert, skaliert und verschoben werden können.

Dies ist der Zoom-Modus. Zieht man mit der linken Maustaste einen Rahmen auf, so wird das Display auf die Anzeige des Rahmeninhalts vergrößert. Mit einem Rechtsklick wird die Anzeige verkleinert und mit Klick auf die mittlere Maustaste wird die Anzeige auf die voreingestellte Vergrößerungsstufe gebracht.

Wie man schnell feststellen wird, gibt es keine Möglichkeit, in Regeln festzulegen, in welcher Reihenfolge Objekte in einem Diagramm gezeichnet werden, so dass oft einige Objekte andere überdecken. Um die Anordnung der Objekte zu ändern, klickt man im normalen Modus mit der rechten Maustaste auf ein oder mehrere markierte Objekte. Jetzt geht an der Stelle des Klicks ein Pop-Up-Menü auf, in dem die Objekte nach vorne, nach hinten und ganz in den Vorder- bzw. Hintergrund bringen kann.

Was bei einem Doppelklick auf ein Objekt passiert, hängt vom Kontext ab, in dem das jeweilige Diagrammdisplay verwendet wird und wird an geeigneter Stelle erklärt.

### 7.2.4 Regeldisplay

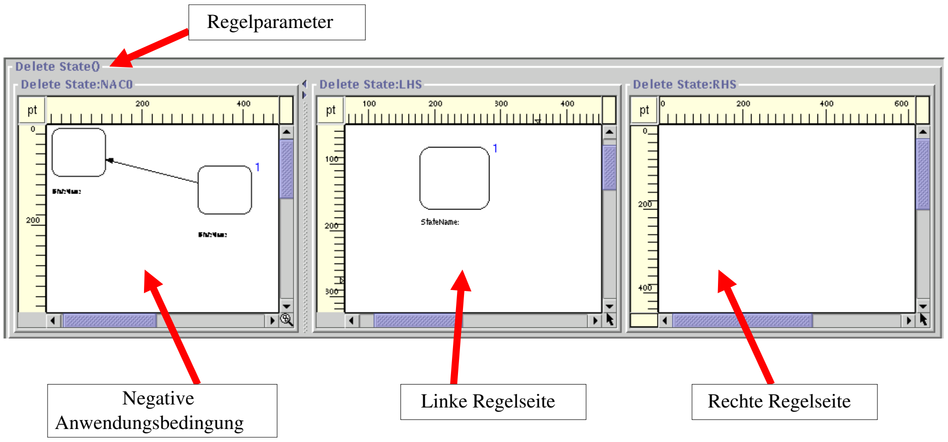



Abbildung 7.6: Das Regeldisplay

In der oberen Hälfte des Regeleditors finden wir das Regeldisplay. Es dient dazu, Regeln mit ihren negativen Anwendungsbedingungen anzuzeigen. Ganz links ist ein Display für die jeweils sichtbare negative Anwendungsbedingung. Diese wird dann angezeigt, wenn sie im →Grammatikdisplay ausgewählt wurde. Zu Anfang ist diese Anwendungsbedingung nicht sichtbar, da sich der Trennbalken des Regeldisplays ganz links befindet.

Zwei weitere Displays stellen die linke und rechte Seite der Regel dar. Im Kopf des Regeldisplays wird der Name der Regel und daneben - in Klammern - die Parameter der Regel mit Typ und Namen dargestellt.

### 7.2.5 Arbeitsdiagramme mit Werkzeugleiste

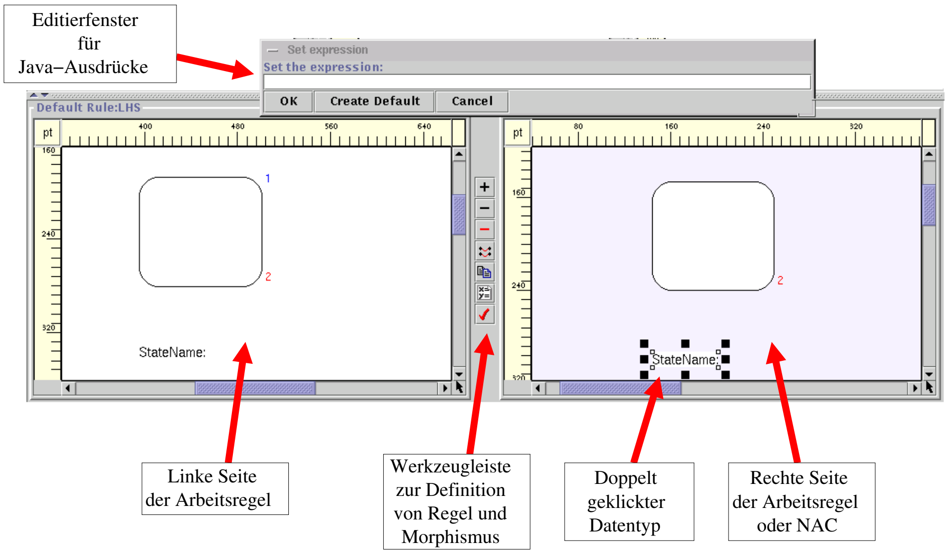



Abbildung 7.7: Arbeitsfläche

Im unteren Bereich des Editors befinden sich die zwei Arbeitsdisplays zur Regelbearbeitung und eine weitere Werkzeugleiste, die diverse Funktionen bietet, mit denen die Arbeitsregel aufgebaut werden kann.

Im linken Arbeitsdisplay wird **immer** die linke Seite der Arbeitsregel dargestellt, während im rechten Display nach Bedarf entweder die rechte Seite der Arbeitsregel oder eine negative Anwendungsbedingung angezeigt wird. Der Morphismus, der mit Hilfe der Regelwerkzeugleiste bearbeitet werden kann, ist dementsprechend entweder der Regelmorphismus von der linken in die rechte Seite der Arbeitsregel oder der NAC-Morphismus von der linken Seite der Arbeitsregel in das Diagramm der negativen Anwendungsbedingung. Was jeweils ausgewählt ist, hängt von der Selektion im →Grammatikdisplay ab. Wird dort die Arbeitsregel angeklickt, wird die linke und rechte Seite in den Arbeitsdisplays dargestellt, wird dort eine NAC angeklickt, wird linke Seite und NAC ausgewählt, ebenso geschieht die Auswahl der entsprechenden Morphismen automatisch.

Einen weiteren Modus in der Arbeitsfläche gibt es, wenn das Startdiagramm zur Bearbeitung ausgewählt wurde. In diesem Fall ist das rechte Display leer, während das linke das Startdiagramm aufnimmt. Auch die Werkzeugleiste ist dann nicht benutzbar.

Durch einen Klick in ein Arbeitsdisplay wird dieses selektiert (mintgrüner Hintergrund). Der *Match* von der linken Seite der selektierten Regel (im oberen Regeldisplay dargestellt) bezieht sich immer auf das selektierte Arbeitsdisplay und auch der Match schaltet bei Selektion des jeweils anderen Displays um. Das sichert, dass die ausgewählte Regel immer auf das selektierte Display angewendet wird.

Die ersten vier Funktionen der Regelwerkzeugleiste sind analog zu den Matchfunktionen in der Hauptwerkzeugleiste dazu da, den ausgewählten Morphismus zu definieren und zu vervollständigen. Im Gegensatz zum Match muss dieser Morphismus, wenn wir uns die Theorie anschauen, jedoch nicht total sein.

Desweiteren gibt es folgende Funktionen:

Kopiert das Diagramm in der linken Arbeitsfläche auf die rechte Seite. Diese Hilfsfunktion erstellt eine exakte Kopie des Diagramms auf der linken Seite in der rechten Seite. Außerdem wird der Morphismus soweit vervollständigt, dass alle Objekte links auf die jeweiligen Kopien rechts abgebildet werden.

Öffnet ein Dialogfenster, in dem die Regelparameter editiert werden können. Diese Funktion wird im Abschnitt über Datentypen genauer erklärt.

Beendet die Definition einer Regel. Die Regel wird aus dem Arbeitsbereich entfernt und steht für den Export und die Löschfunktionen wieder zur Verfügung.

## 7.3 Datentypen und Parameter

Wie schon in Abschnitt 3.1.1.7 beschrieben, werden alle Attribute in den Regeln durch Java-Ausdrücke erzeugt oder verändert. Selbst dem voreingestellten Wert für einen Datentypen entspricht ein Java-Ausdruck, der genau diesen Datentypen erzeugt.

Im gesamten Grammatikeditor werden die Datentypen nur in Form ihrer Java-Ausdrücke dargestellt, wobei den Ausdrücken der Name des Datentyps (z.B. `Statename`) vorangestellt wird[1]. So lassen sich auch leere Java-Ausdrücke bearbeiten. Java-Ausdrücke, die eine Länge von 10 Zeichen überschreiten, werden im Display mit „. . .“ abgekürzt.

Java-Ausdrücke für einzelne Datentypen lassen sich bearbeiten, indem man auf den entsprechenden Datentyps doppelklickt. Es erscheint ein einfacher Texteingabedialog mit dem alten Wert des Datentypen. Zu den üblichen Schaltflächen für das Bestätigen bzw. Abbrechen der Eingabe kommt eine Schaltfläche, die – für die bessere Handhabung – einen Java-Ausdruck erstellt, der bei Auswertung einen Default-Wert für diesen Datentyp erzeugt. Dieser Default-Wert ist derjenige, der im Alphabeteditor angegeben wurde (z.B. „`Default State\` für einen Zustandsnamen). Das spielt vor allem dann eine Rolle, wenn man selbst Einfügeregeln definieren will.

Als Java-Ausdrücke für einen Datentypen (z.B. mit der Klasse `StringDT`) sind alle Ausdrücke erlaubt, die ein Objekt der Datentypklasse erzeugen, außer folgenden Einschränkungen:

- Operatoren mit Seiteneffekten (=, += etc.) dürfen nicht verwendet werden,

[1] Nicht zu verwechseln mit dem *Typ* des Datentypen, der die Java-Klasse `StringDT` ist

- Alle Ganzzahlen werden mit der Präzision von `int` behandelt, alle Fließkommazahlen mit der Präzision von `float`.
- Werden Klassen angegeben (z.B. für statische Methoden wie `createDatatype()`), so müssen diese in `$` eingeschlossen werden.

Konform zur Theorie und zum Aufbau der Attributkomponente dürfen auf der linken Regelseite nur Variablen auftauchen. Eine weitere Änderung gegenüber den im AGG-System üblichen Möglichkeiten besteht darin, dass auf der linken Regelseite auch keine konstanten Ausdrücke verwendet werden können. Es ist also leider nicht möglich, bestimmte konstante Datentypwerte zu matchen. Allerdings ist es erlaubt, an Stelle von Variablen den Java-Ausdruck einfach leer zu lassen. Ein leerer Ausdruck passt auf jeden beliebigen Datentypen und ist also so etwas wie eine universelle Variable.

Wir wollen nun noch ein Beispiel für die komplexere Anwendung von Datentypen geben:

Nehmen wir an, wir hätten ein Klassendiagramm. Das Symbol für eine Klasse enthält drei Datentypen: `ClassName:StringDT`, `AttributeList:StringListDT` und `MethodList:StringDT`. Außerdem gibt es eine Assoziation mit einem Namen `AssocName:StringDT`. Wir wollen nun eine Regel bauen, in der eine Assoziation zwischen zwei Klassen in eine Methode innerhalb der Zielklasse umgewandelt wird (siehe Abb. 7.8).

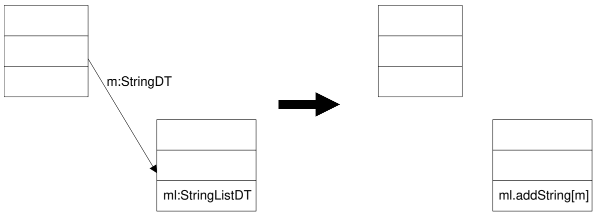



Abbildung 7.8: Beispielregel: Assoziation in Methode umwandeln

Wir geben nur Variablen für die Methode und die Methodenliste an, die letztendlich geändert werden. Alle anderen Datentypen bleiben leer, denn – wie in Abschnitt 4.4.2 beschrieben – leere Datentypen matchen auf alle Datentypen im Arbeitsdiagramm. In der Datentypklasse `StringListDT` gibt es nun eine Methode `addString(StringDT)`, mit der man dieser Liste einen String hinzufügen kann. Diese gibt die geänderte Liste zurück, so dass diese nach Auswertung des Java-Ausdrucks in den Datentyp zurückgeschrieben werden kann. Genau diese Methode wird auf der rechten Regelseite verwendet, um die Methodenliste zu ändern.

Auf der rechten Regelseite können nun komplexe Java-Ausdrücke auftauchen. Für die Variablen, die in der rechten Regelseite vorkommen, gilt folgende Erweiterung:

**Variablen auf der rechten Regelseite müssen entweder links oder als Regelparameter vorkommen!**

Ein Regelparameter besteht aus einem Datentypen, einem Namen und einer Variablen, die den alten Wert enthält. Das ist eine Erweiterung der Theorie, die zur Be-

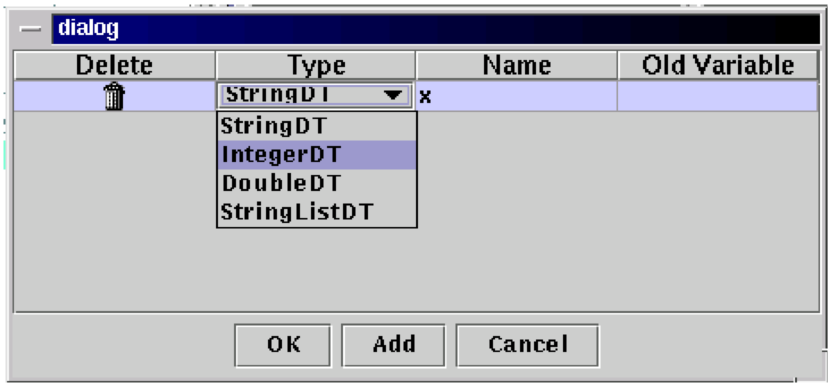



Abbildung 7.9: Parameter definieren

quemlichkeit dient: Man kann jetzt festlegen, welcher alte Wert eines Parameters im Eingabefenster für den Parameterwert erscheinen und geändert werden soll. Ist der Name der alten Variablen leer, wird dem Benutzer kein alter Wert für diesen Datentypen vorgeschlagen. Der Regelparameter kann als Variable auf der linken Regelseite vorkommen, sein Wert wird allerdings dann durch den interaktiv eingegebenen Wert ersetzt. Als Beispiel für eine solche „interaktive" Änderung eines Datentypwerts kann man sich jede beliebige generierte Änderungsregel ansehen.

## 7.4 Beispielanwendung: Fallstudie

Wir wollen nun in einem beispielhaften Arbeitsablauf darstellen, wie eine Grammatik für Statecharts (siehe Kapitel 3) aus einem gegebenen Alphabet erstellt und verfeinert werden kann.

### 7.4.1 Start

Starten wir den Grammatikeditor mit dem Aufruf `GramEditor`. Zunächst benötigen wir die Alphabetdatei `StatechartsExample.alp`, die sich im Unterverzeichnis `genged/alphabets` des Alphabeteditors befinden sollte. Diese Alphabetdatei enthält alle Definitionen für die grafischen Links, Symbole und Datentypen, die in Abschnitt 3.1 eingeführt wurden.

Eine neue Visuelle Grammatik erzeugen wir, indem wir die Schaltfläche „Neue Grammatik" (🗋) anklicken. Im anschließenden Dateimenü ist das Verzeichnis `genged/alphabets` schon vorausgewählt, so dass wir nur die Datei `StatechartsExample.alp` auswählen müssen. Jetzt passiert eine Weile lang scheinbar gar nichts, da der Grammatikeditor den oben vorgestellten Algorithmus verwendet, um für die Symbole `State`, `Transition`, `ParallelConnector`, `SubstateConnector` und `Default` Einfüge- und Löschregeln und außerdem Änderungsregeln für die Datentypen `Statename` und `Transitionname` generiert. Stößt er bei der Vervollständigung der Regeln auf Zyklen, so wird eine Fehlermeldung ausgegeben und die Grammatik nicht erzeugt.

Ist der Editor mit der Generierung fertig, tauchen im Grammatikdisplay die erzeugten Regeln auf. Alle diese Regeln sind in grüngrauer Schrift dargestellt, wir können diese Regeln also nicht ändern.

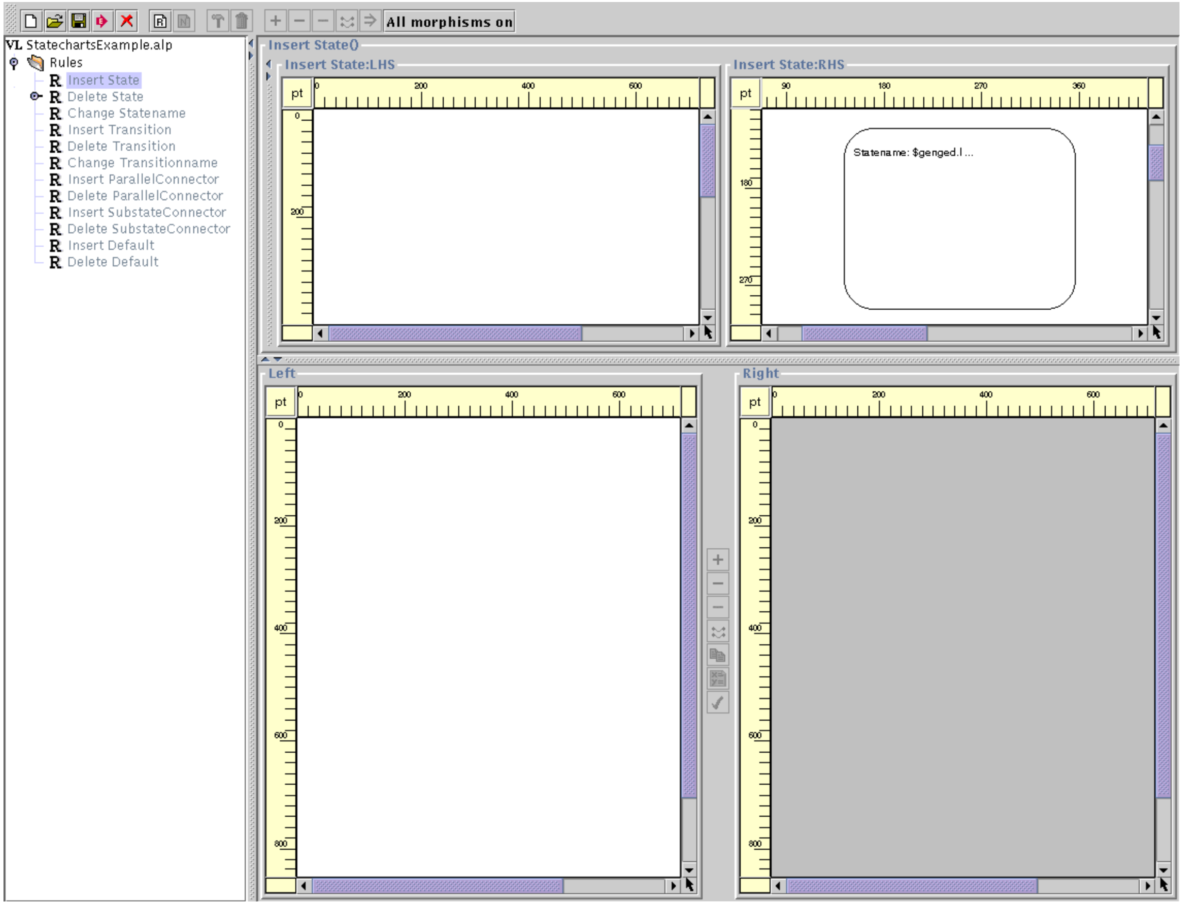



Abbildung 7.10: Nach dem Laden des Alphabets

### 7.4.2 Konstruktion einer visuellen Regel

Wir können natürlich sämtliche erzeugte Regeln exportieren, allerdings entspricht unsere Grammatik dann noch nicht der im Abschnitt 3.1 vorgestellten. So könnte man mit dieser einfachen Grammatik Transitionen zwischen zwei parallelen Zuständen oder zwischen einem Unter- und seinem Oberzustand einfügen. Das können wir ändern, indem wir die gegebenen Regeln dazu verwenden, eine eigene Einfügeregel für Transitionen zu schreiben, die mit Hilfe von negativen Anwendungsbedingungen die zwei eben genannten Fehlerfälle ausschließt.

Erzeugen wir also eine neue, leere Regel, indem wir die Schaltfläche "Regel erzeugen" ( ) anklicken. Dann geben wir dieser Regel einen Namen, z.B. *Insert Valid Transition* , indem wir den Regelnamen doppelklicken und den neuen Namen eingeben.

Als nächstes wollen wir die Regel bearbeiten, dazu selektieren wir sie im Grammatikdisplay und klicken die Schaltfläche “Regel bearbeiten“ ( ) an. Die neue Regel erscheint nun mit der linken und rechten Regelseite im Arbeitsbereich, die linke Regelseite ist für die Bearbeitung ausgewählt (erkennbar am mintgrünen Hintergrund).

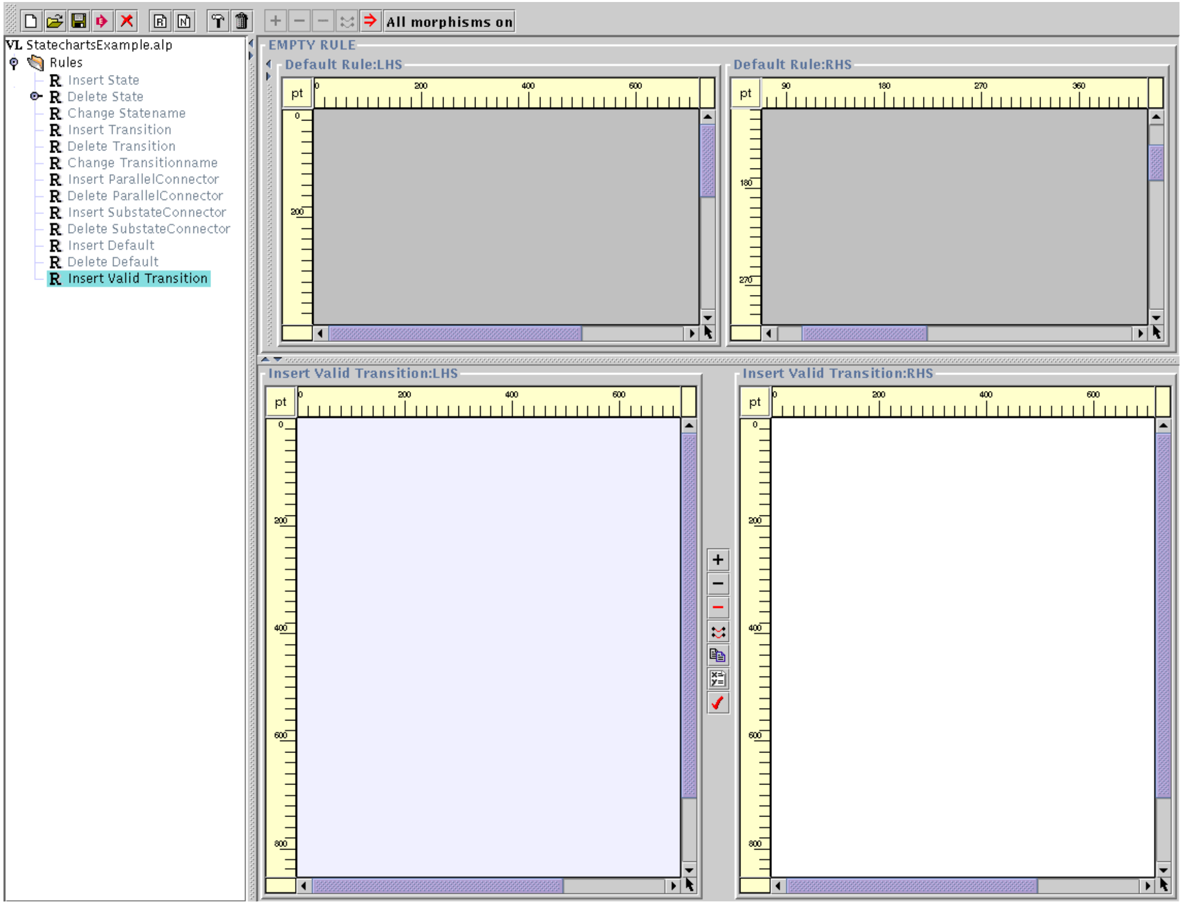



Abbildung 7.11: Bearbeiten der Regel

Wir wollen nun auf der linken Seite zwei Zustände einfügen, also wählen wir im Grammatikdisplay die Regel *Insert State* aus, die im Regeldisplay dargestellt wird. Da die linke Regelseite der Zustands-Einfügeregel leer ist, brauchen wir keinen Match zu definieren, ein Klick auf die Schaltfläche “Regel anwenden“ ( ) genügt, und der Zustand wird in die linke Seite der Arbeitsregel eingefügt. Das Ganze wiederholen wir noch einmal – und wir sehen nur einen Zustand auf der Arbeitsfläche! Das liegt daran, dass die Symbole, die eingefügt werden, immer ihre Defaultwerte bekommen. Danach werden eventuelle Constraints zwischen den Symbolen gelöst, da aber in diesem Fall keine existieren, kommen die zwei Zustandssymbole übereinander zu liegen und wir müssen sie per Hand auseinanderziehen. Zur Übung selektieren wir die rechte Arbeitsfläche und wiederholen das Ganze. Nun sollten wir rechts und links je zwei Zustände haben.

Jetzt können wir zwischen die zwei rechten Zustände eine Transition einfügen. Dazu wählen wir im Grammatikdisplay die Regel *Insert Transition* aus. Wie wir sehen, hat diese Regel keine leere linke Seite, sondern enthält dort den Ausgangs- und Endzustand der Transition. Diese Zustände werden auf die rechte Regelseite abgebildet, wobei die

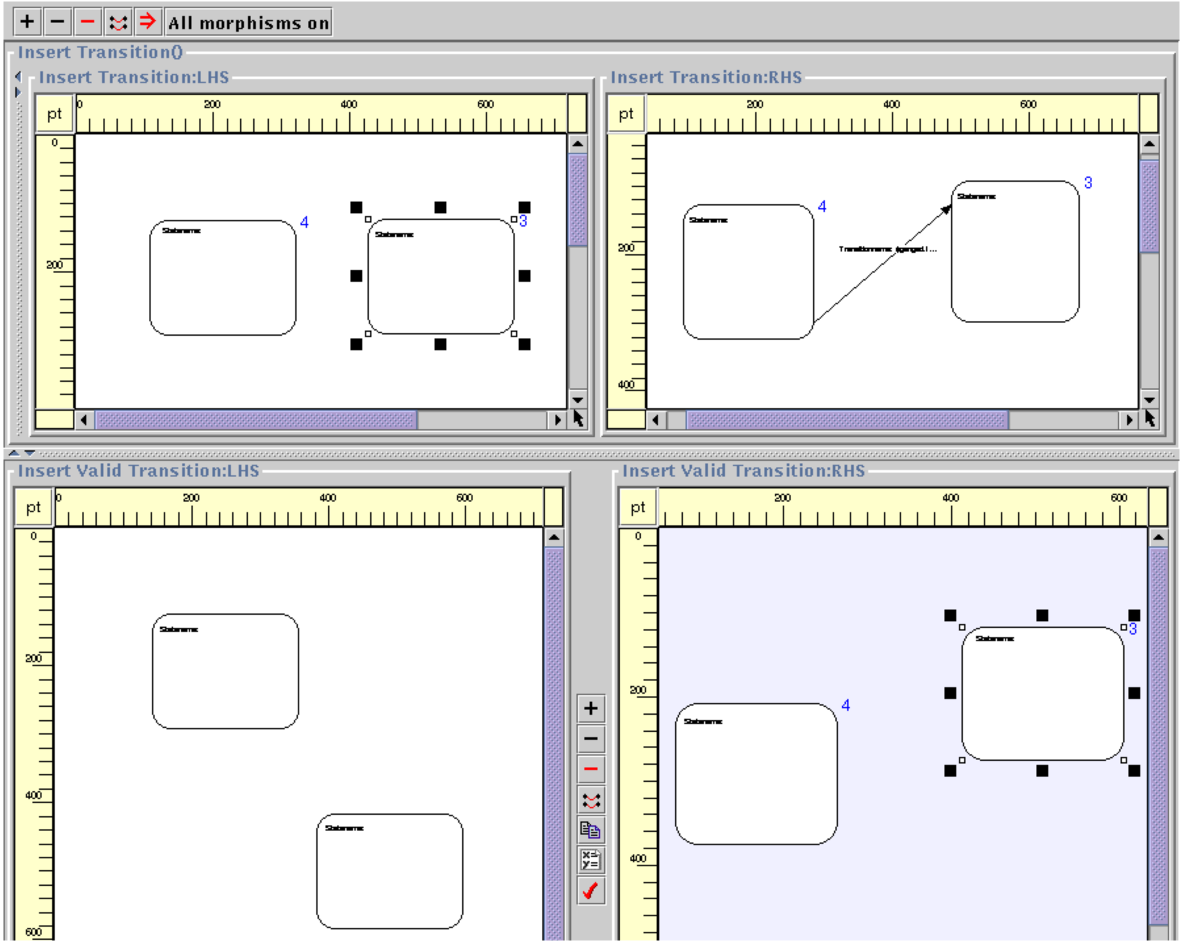



Abbildung 7.12: Matchen der Regel *Insert Transition*

Abbildung durch blaue Zahlen rechts oben an den Symbolen visualisiert ist. Würden wir jetzt einfach die Regel auf das Diagramm in der rechten Arbeitsfläche anwenden, wäre nicht eindeutig, wo die Transition beginnt und wo sie endet. Also werden wir einen expliziten Match für die beiden Zustände angeben. Dazu markieren wir zunächst den Ausgangszustand (denjenigen, der auf den Ausgangszustand der Transition auf der rechten Seite abgebildet wird) und dann den entsprechenden Zustand in der rechten Arbeitsfläche und klicken auf die Schaltfläche „Abbildung hinzufügen“ (**+**). Die Zahl am oberen rechten Zustand taucht jetzt auch in der Arbeitsfläche auf und signalisiert, dass eine Abbildung im Match explizit definiert worden ist. Dasselbe machen wir für den Zielzustand.

Jetzt können wir wieder die Regel anwenden, und zwar auf die rechte Seite der Arbeitsregel, und erhalten die Transition. Es hätte in diesem Fall sogar genügt, nur einen Zustand explizit abzubilden, da der andere automatisch bei der Vervollständigung des Matches – bevor die Regel angewendet wird – abgebildet werden würde[2]. In großen Dia-

[2]Die Vervollständigung ist immer injektiv, so werden – sofern man es nicht selbst macht – nie zwei Symbole auf dasselbe abgebildet.

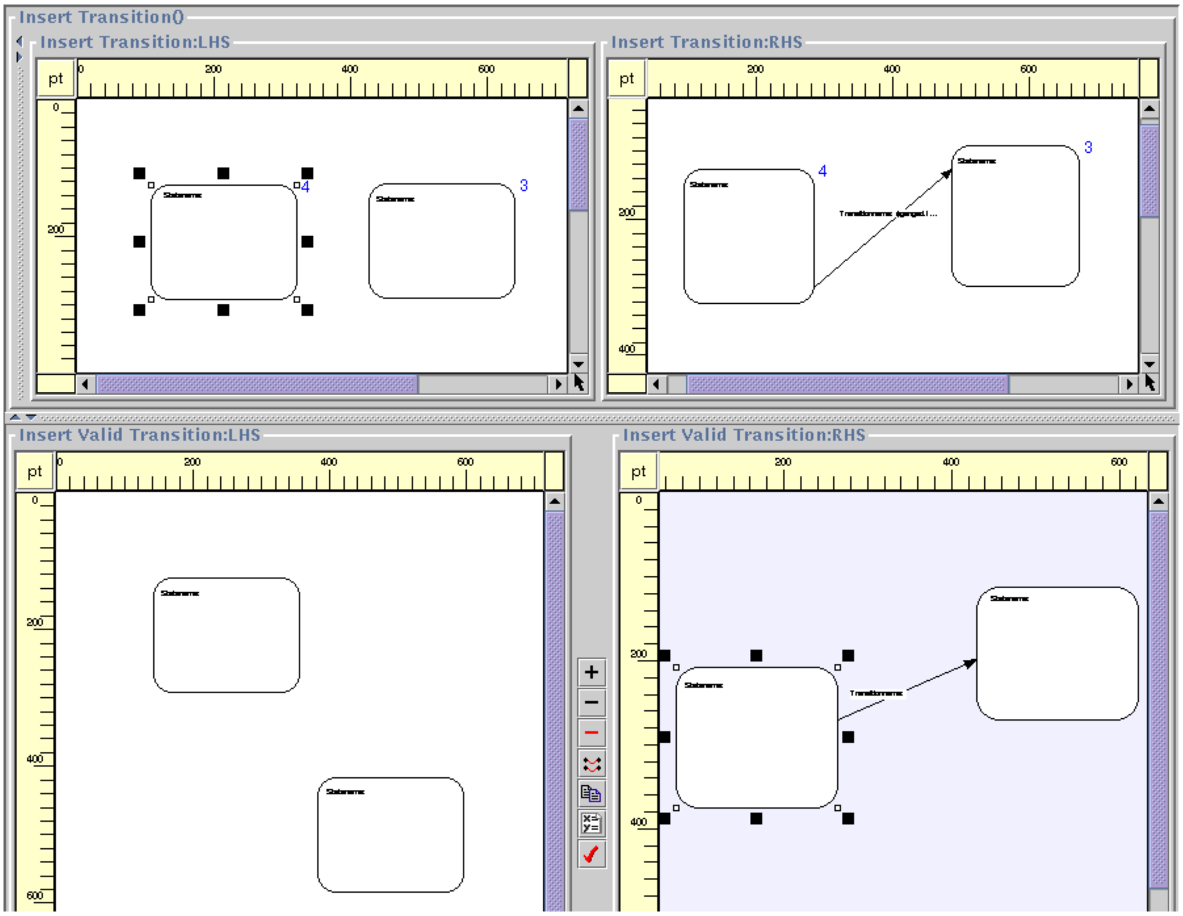



Abbildung 7.13: Regel *Insert Transition* angewendet

grammen gibt es aber meistens mehrere Möglichkeiten, einen Match zu definieren[3], so dass wir uns daran gewöhnen sollten, einen Match größtenteils per Hand zu bauen.

Als nächstes müssen wir einen Regelmorphismus, also eine Abbildung der linken auf die rechte Seite der Arbeitsregel bauen. Das geschieht ebenso wie das Matchen, bloß dass wir dafür die Schaltflächen zwischen dem linken und dem rechten Arbeitsdiagramm verwenden. Der Morphismus wird durch eine rote Zahl rechts unten an den abzubildenden Symbolen gekennzeichnet.

Noch ist diese Regel sogar ungünstiger als die alte Regel „Insert Transition". Warum? Nun, da der Java-Ausdruck für den Transitionsnamen auf der rechten Seite der Regel leer ist, wird auch in einem Diagramm eine namenlose Transition erzeugt. Wir würden aber gerne den im Alphabeteditor kreierten Default-Transitionsnamen dort sehen. Also müssen wir den Java-Ausdruck ändern, der diesen Default-Namen erzeugt. Dazu doppelklicken wir auf den Datentypen, der im Augenblick nur aus dem Namen des Datentyps besteht (`Transitionname`) und erhalten folgendes Fenster:

Wir könnten natürlich mit Hilfe der statischen Java-Methode `createDatatype()`,

[3]So kann eine Regel mit einem Zustand auf der linken Seite auf alle Zustände in einem umfangreichen Arbeitsdiagramm gematcht werden.

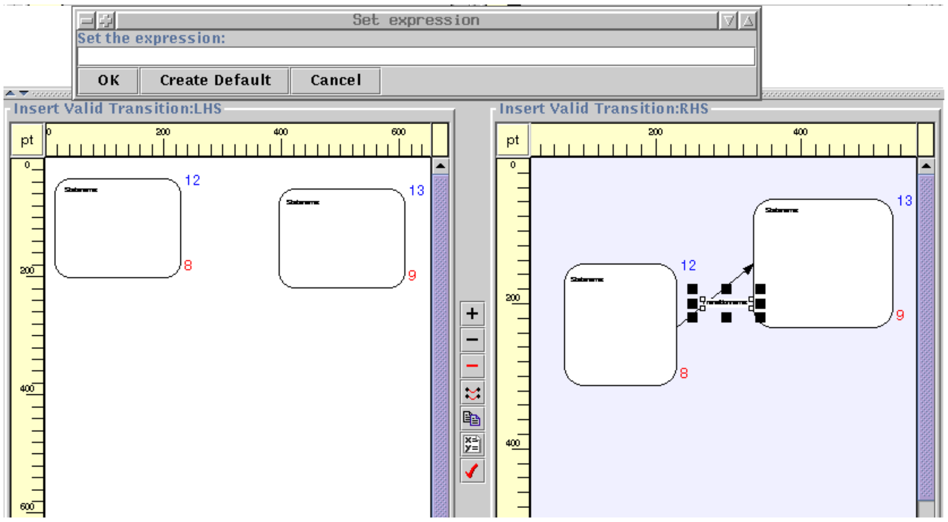



Abbildung 7.14: Erzeugen eines Default-Datentypen

die in jeder Datentypklasse existiert, einen Datentypen erzeugen, einfacher geht es jedoch, indem wir die Schaltfläche „Create Default“ auswählen, die uns den Java-Ausdruck erzeugt, der bei Auswertung den Default-Wert („`Default State\`) für diesen Datentypen (in diesem Fall den Datentypen `Transitionname:StringDT`) erzeugt.

Nun sind wir an dem Punkt angelangt, an dem die neue Regel genau dasselbe macht wie die alte. Jetzt machen wir die Regel „korrekt im Sinne der Spezifikation“, indem wir die negativen Anwendungsbedingungen hinzufügen. Dazu klicken wir die Regel im Grammatikdisplay einmal an und wählen die Schaltfläche „NAC erzeugen“ (+) aus. Die erzeugte negative Anwendungsbedingung (NAC) können wir nun selektieren, um sie zusammen mit der linken Seite der Arbeitsregel im Arbeitsbereich darzustellen. Jetzt wollen wir eine negative Anwendungsbedingung bauen, die verbietet, dass eine Transition zwischen Unter- und Oberzustand eingefügt wird. Also müssen wir in die rechte Regelseite zwei Zustände und zusätzlich einen Verbinder zwischen Unter- und Oberzustand einfügen. Das könnten wir nun wieder von Hand machen, günstiger ist es jedoch, das Diagramm in der linken Arbeitsfläche jetzt einfach auf die rechte (NAC-)Arbeitsfläche zu kopieren. Dazu verwenden wir die Schaltfläche „Identische Kopie“ (+), die genau das erledigt und uns zusätzlich noch den Morphismus von der linken in die rechte Seite baut. Alles, was jetzt noch fehlt, ist der Verbinder zwischen Ober- und Unterzustand auf der rechten Seite, den wir – auf die übliche Weise – mit Hilfe der Regel *Insert Substate-Connector* einfügen. Jetzt sagt die NAC aus, dass ein Zustand nicht Unterzustand des zweiten sein darf, wenn eine Transition zwischen den beiden eingefügt werden soll.

Analog zu der eben definierten NAC benötigen wir eine weitere NAC, die genau die umgekehrte Aussage macht: Ein Zustand darf nicht Oberzustand des anderen sein. Die Definition geschieht ganz analog, außer dass wir dieses Mal dir beiden Zustände vertauschen. Außerdem benötigen wir zwei NACs für den Parallelzustand-Verbinder, die wir auch auf die eben genannte Weise erzeugen.

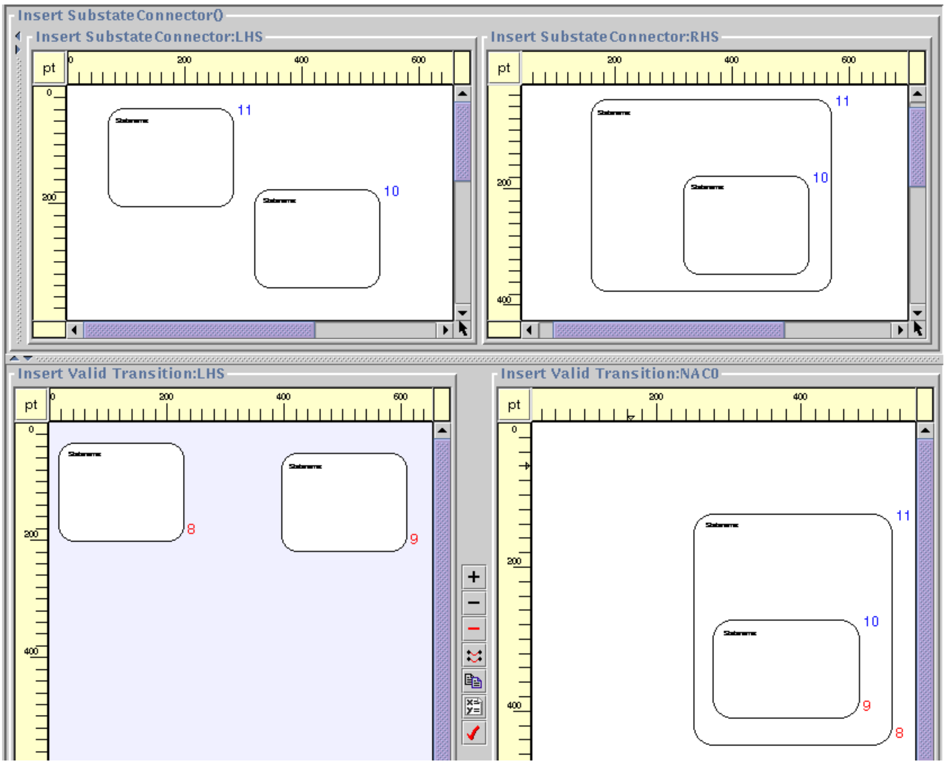



Abbildung 7.15: Bau einer negativen Anwendungsbedingung (NAC)

### 7.4.3 Konstruktion einer parametrisierten visuellen Regel

Nun wollen wir noch ein einfaches Beispiel für eine parametrisierte Regel angeben. Vielleicht ist schon aufgefallen, dass für das Einfügen eines Zustands mit einem bestimmten, vom Benutzer gewünschten Namen, zwei der erzeugten Regeln hintereinander angewendet werden müssen: Als erstes muss ein Zustand mit dem Default-Zustandsnamen eingefügt werden (Regel: *Insert State*), dann kann man den Namen des Zustands ändern (Regel: *Change Statename*). Das wollen wir ändern, indem wir eine Regel definieren, die einen Zustand einfügt und dessen Namen aus einem Regelparameter generiert. Dieser Regelparameter wird dann beim Ausführen der Regel vom Benutzer abgefragt.

Wir erzeugen zunächst auf gewohnte Weise eine Regel, deren linke Seite leer ist und auf deren rechter Seite ein Zustand ohne Namen steht. Diese benennen wir mit *Insert Named State*. Jetzt klicken wir auf die Schaltfläche „Parameter setzen“ (). Es erscheint ein Dialog, in dem wir Parameter hinzufügen oder löschen können sowie deren Typ, Namen und den Namen der alten Variablen, auf deren Basis der Parameter abgefragt wird, eingeben können. Wir fügen mit „Add“ einen Parameter hinzu, belassen den Namen bei x und wählen den Datentypen `StringDT` aus der Liste der verfügbaren Datentypen aus. Das Feld für die alte Variable bleibt leer – woher sollten wir auch den alten Wert nehmen? Die linke Regelseite ist ja leer.

Jetzt müssen wir natürlich noch angeben, wo auf der rechten Regelseite der Parameter Einsatz findet. Die Auswahl fällt leicht, gibt es doch nur einen möglichen Kandida-

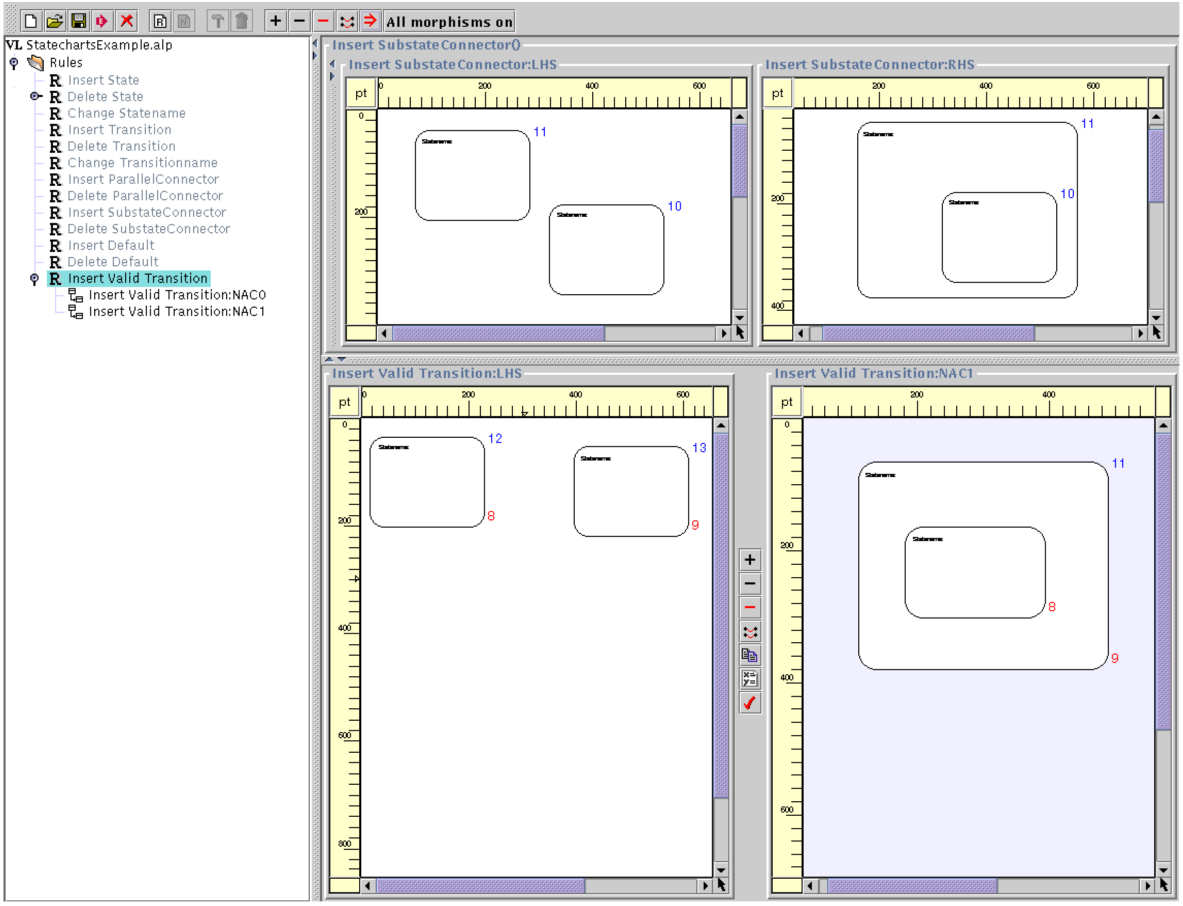



Abbildung 7.16: Die analoge NAC

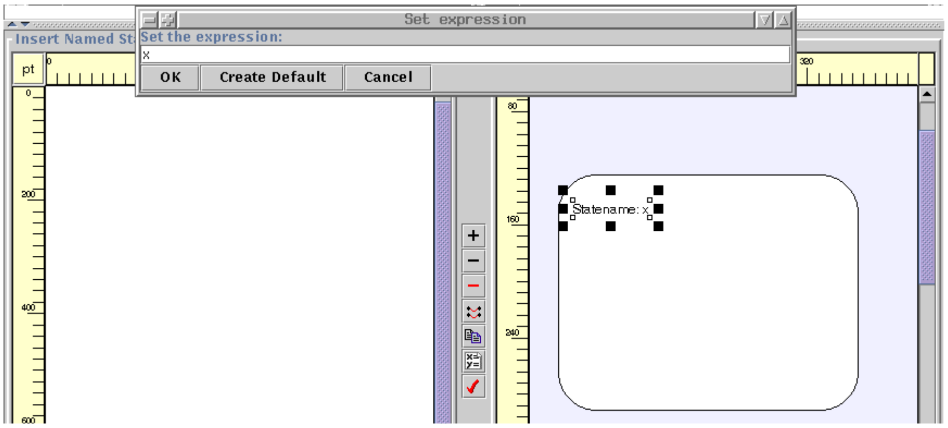



Abbildung 7.17: Bau einer Regel „Zustand einfügen“

ten, nämlich den noch leeren Zustandsnamen. Diesen öffnen wir durch Doppelklick und geben als Java-Ausdruck die Variable x, die wir als Parameter definiert haben, ein.

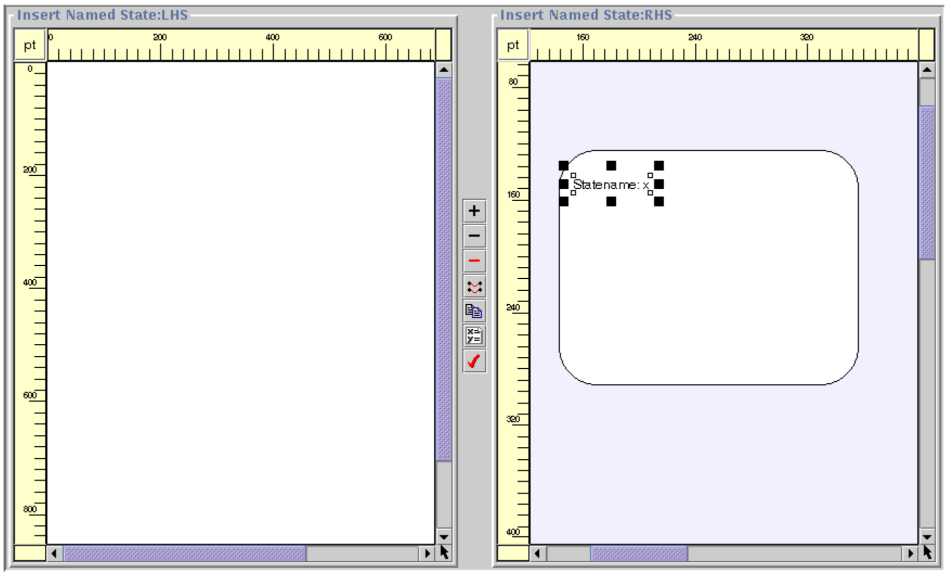



Abbildung 7.18: Regel „Zustand einfügen“ ist fertig

### 7.4.4 Ende

Nun wollen wir – als erste Näherung – alle Regeln der Grammatik exportieren, mit Hilfe derer wir Statecharts erzeugen können. Die alte, automatisch generierte Regel *Insert Transition* gehört nicht dazu, ebensowenig etwas unbequeme Regel *Insert State*. Wir markieren also alle Regeln, die unsere neue Grammatik enthalten soll, durch einen Rechtsklick auf den Regelnamen im Grammatikdisplay. Die zu exportierenden Regeln sollten jetzt mit dem Symbol R➔ markiert sein. Durch Klick auf die Schaltfläche „Exportieren“ (R➔) können wir nun eine Grammatik speichern, die nur die ausgewählten Regeln enthält. Jetzt können wir den Grammatikeditor beenden und den Diagrammeditor aufrufen, um unsere Regeln an einem Testdiagramm auszuprobieren.

## 7.5 Diagrammeditor

Der Diagrammeditor ist eigentlich eine stark vereinfachte Version des Grammatikeditors, die im Arbeitsbereich nur ein Diagrammdisplay besitzt. Aufgerufen wird er mit `GramEditor -test`. Er bedient sich genau wie der Regeleditor, mit folgenden Unterschieden:

- Alle Regeldefinitionsfunktionen entfallen.
- Die Arbeitsfläche enthält nur ein Diagrammdisplay und keine Werkzeugleiste für die Definition von Morphismen.
- Alle Morphismen werden immer angezeigt, falls vorhanden.
- Doppelklick auf einen Datentypen im Diagrammdisplay hat keine Wirkung, da

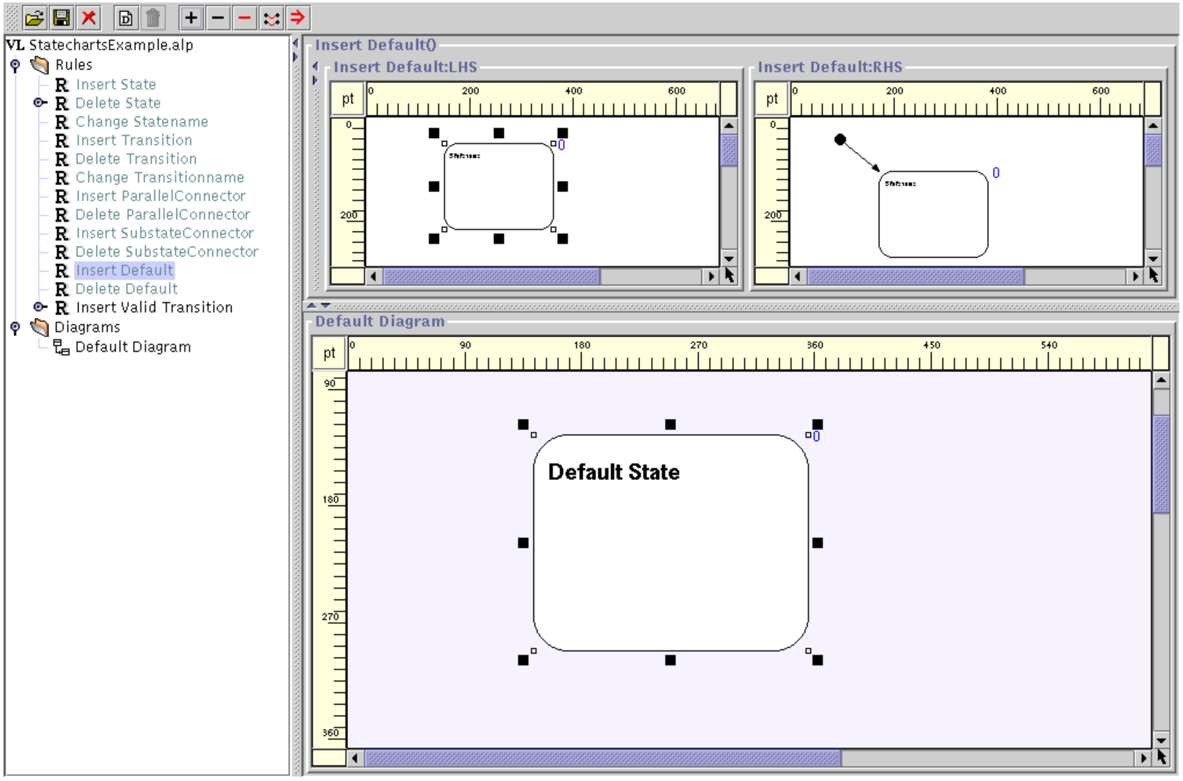



Abbildung 7.19: Der Diagrammeditor

die Werte von Datentypen nur über entsprechende Regeln gesetzt und verändert werden können.

- Es gibt eine Schaltfläche „Diagramm erzeugen“ (, außerdem enthält das Grammatikdisplay einen weiteren Ordner für Diagramme. Die eben genannte Schaltfläche erzeugt eine Kopie des Startdiagramms der Grammatik.

- Es gibt (noch) keine Startdiagramme, da man im Grammatikeditor z. Z. nur Regeln bearbeiten kann. Jedes neu erzeugte, leere Diagramm ist also ein Startdiagramm für unsere visuelle Sprache.

# Kapitel 8

# Zusammenfassung und Ausblick

Wir wollen nun noch einmal zusammenfassen, was wir in dieser Arbeit beschrieben und implementiert haben.

Nach einem kurzen Überblick über textuelle und visuelle Sprachen haben wir die zwei grundlegenden Formalismen eines Systems, mit dem man visuelle Sprachen entwickeln kann, vorgestellt: Auf der grafischen Seite haben wir Constraints und auf der logischen, strukturellen Seite attributierte Graphtransformation beschrieben.

Anhand einer Fallstudie wurden dann die Konzepte einer visuellen Sprache (Alphabet, Regeln, Datentypen) vorgestellt und dann genauer beschrieben. Aus den Anforderungen für einen Grammatikeditor ergaben sich weitere Toolkonzepte wie ein Algorithmus zur Erzeugung einfacher Sprachregeln, mit deren Hilfe sich dann komplexere aufbauen lassen und Maßnahmen, um die Korrektheit der mit Regeln erzeugten Diagramme zu sichern.

Da sich der Grammatikeditor zu großen Teilen aus Schnittstellen zu schon bestehenden Systemen zusammensetzt, haben wir diese Systeme in einem weiteren Kapitel genauer beleuchtet, insbesondere deren Schnittstellen. Design-, Implementierungsbeschreibung und ein Benutzerhandbuch ergaben dann die umfassende Beschreibung unseres Systems. Das fertige System erlaubt es, auf der Basis eines Alphabets einfache Regeln zu generieren und mit Hilfe dieser Regeln weitere komplexe Regeln und ein Startdiagramm zu konstruieren. Mit diesem Regelsatz und dem Startdiagramm lassen sich dann Diagramme der visuellen Sprache editieren.

Auf der Basis des vorliegenden Grammatikeditors sind mehrere Erweiterungen und Verbesserungen denkbar, von denen wir nun noch die wichtigsten vorstellen wollen:

## 8.1 Transformation und Regeln

- Was im AGG-System noch nicht optimal gelöst ist, ist die Tatsache, dass die Attributkomponente sehr stark auf Strings fixiert ist. Das Matchen von beliebigen konstanten Objekten mit der Methode `equals(Object)` sollte möglich sein. Ebenso sollte es eine Möglichkeit geben, Regeln **unter Einbeziehen der Attribute** auf Regeln anzuwenden und Konstanten sowohl in Regeln als auch im Startdiagramm zu verwenden.

- Weiter denkbar ist es, Editoroperationen durch Hintereinanderanwendung von Regeln und – noch besser – Kontrollstrukturen auf Regeln zu beschreiben, um Probleme wie die Hierarchien bei Statecharts (siehe Abschnitt 3.1) in den Griff zu bekommen.

- Ebenso sollte es möglich sein, Bedingungen für einen Ansatz auf der Datentypebene anzugeben.

- Das Startdiagramm wird noch etwas stiefmütterlich behandelt, da es weder Datentypwerte noch -ausdrücke enthalten darf. Es wäre denkbar, Variablen im Startdiagramm zuzulassen, die dann bei Initialisierung des Diagramms interaktiv mit Werten belegt werden können.

## 8.2 Sichten und Filter

- Vorgesehen, aber nicht vollständig implementiert ist die Möglichkeit, dass man – auf der Basis derselben textuellen Signatur – das Alphabet, also Symbole und Linkconstraints ändern kann. So lassen sich Diagramme mit verschiedenem Aussehen, aber der gleichen Struktur darstellen.

- Eine andere Erweiterung ist die Verbindung mehrerer Diagrammarten zu „Sichten“ auf ein bestimmtes Modell, wie es oft in UML gemacht wird. So können einzelne Symbole wie z.B. Klassen in mehreren Diagrammen verwendet werden und eine Änderung in einem Diagramm schlägt sich dann auch in allen anderen Diagrammen nieder.

- Außerdem sollte es möglich sein, die Anzeige von Datentypen zu filtern, sich also beispielsweise nur Klassennamen in einem Klassendiagramm anzeigen zu lassen.

## 8.3 Sonstiges

Die Fixierung auf „korrekte“ Diagramme ist – wie man schnell merken wird, wenn man eine visuelle Sprache definiert – ungünstig. Besser wäre es, im Verlauf der Konstruktion eines Diagramms auch nicht korrekte Diagramme zuzulassen. So könnte man auch – analog zu textuellen Sprachen – Nonterminalsymbole einführen. Ein „fertiges“ Diagramm wäre dann eines, das nur Terminalsymbole enthält.

Was auch denkbar wäre, wäre es, dem Benutzer des Editors mehr Freiheit zu geben, indem man ihn Diagramme nur auf Basis sehr einfacher Einfüge- und Löschoperationen zeichnen lässt und zusätzlich einen Parser konstruiert [Ear70]. Dieser Parser könnte auf den komplexen Regeln basieren und Diagramme auf ihre syntaktische Korrektheit hin überprüfen.

# Literaturverzeichnis


[Bar99] R. Bardohl. *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Dissertation, Technische Universität Berlin, Berlin, 1999. In preparation.

[Bey91] M. Beyer. GAG: Ein graphischer Editor für algebraische Graphgrammatiksysteme. Diplomarbeit, 1991.

[BTMS] R. Bardohl, G. Taentzer, M. Minas und A. Schürr. Application of Graph Transformation to Visual Languages. In: *[Roz99]*.

[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.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 13(2):94–102, 1970.

[EFLR98] Andy Evans, Robert France, Kevin Lano und Bernhard Rumpe. Developing the uml as a formal modelling notation. In: Pierre-Alain Muller und Jean Bezivin (Hrsg.), *UML'98 Beyond the notation. International Workshop Mulhouse France*. Ecole Superieure Mulhouse, Universite de Haute-Alsace, 1998.

[EMC$^+$99] H. Ehrig, B. Mahr, F. Cornelius, M. Große-Rhode und P. Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series, 1995.

[Gol91a] E. Golin. *A method for the specification and parsing of visual languages*. Dissertation, Brown University, 1991.

[Gol91b] E. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, 1991.

[Gri96] P. Griebel. *Parcon - Paralleles Lösen von grafischen Constraints*. Dissertation, Universität-Gesamthochschule Paderborn, Paderborn, 1996.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har90] David Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.

[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, 1993.

[Lef94] G. Lefrancois. *Psychologie des Lernens*. Springer-Verlag, Berlin Heidelberg New York, 3. Auflage, 1994.

[LKW93] M. Löwe, M. Korff und A. Wagner. An algebraic framework for the transformation of attributed graphs. In: M.R. Sleep, M.J. Plasmeijer und M.C. 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.

[Mar94] K. Marriott. Constraint multiset grammars. In: *[VL94]*, Seiten 118–125, 1994.

[Mel99] B. Melamed. Grundkonzeption und -implementierung einer Attributkomponente für ein Graphtransformationssystem. Diplomarbeit, Technische Universität Berlin, 1999. In preparation.

[MM97] K. Marriott und B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):375–402, 1997.

[PSG95] M. Pöpping, G. Szwillus und P. Griebel. Objection - eine Entwicklungsumgebung zum Entwurf constraint-basierter Benutzungsoberflächen. In: *Proc. GI Softwaretechnik'95-Tagung Braunschweig*, Seiten 136–143, 1995.

[Rat97a] Rational. UML Notation – version 1.1. Technischer Bericht, Rational Software Corporation, September 1997.

[Rat97b] Rational. UML Semantics – version 1.1. Technischer Bericht, Rational Software Corporation, September 1997.

[Rat97c] Rational software cooperation. Unified modeling language. available via http://www.rational.com, 1997.

[Roz99] G. Rozenberg (Hrsg.). *Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 2: Applications*. World Scientific Publishing, 1999.

[Rud97] M. Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Diplomarbeit, Technische Universität Berlin, 1997. `http://www.cs.tu-berlin.de/~mich/Dipl.ps.gz`.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[Sch99] M. Schwarze. Konzeption und Implementierung eines generischen Alphabeteditors für visuelle Sprachen. Diplomarbeit, Technische Universität Berlin, 1999. In preparation.

[TER] G. Taentzer, C. Ermel und M. Rudolf. The AGG Approach: Language and Tool Environment. In: *[Roz99]*.

[VL94] *Proc. IEEE Symposium on Visual Languages (VL'94)*, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[VL95] *Proc. IEEE Symposium on Visual Languages (VL'95)*, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[VL96] *Proc. IEEE Symposium on Visual Languages (VL'96)*, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[VL97] *Proc. IEEE Symposium on Visual Languages (VL'97)*, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[Wol97] D. Wolz. Colimit Computations for Graph Structures and Algebraic Specification Languages. PhD Thesis, TU Berlin, FB Informatik, in preparation, 1997.