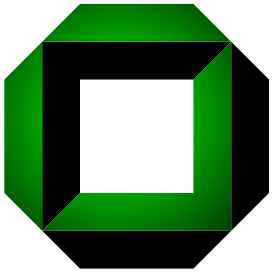


Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

Mit GrGen.NET zu den Sternen

Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken

Diplomarbeit von Edgar Jakumeit

Juli 2008

Betreuer:
Dr. Rubino Geiß

Verantwortlicher Betreuer:
Prof. em. Dr. Dr. h.c. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Inhaltsverzeichnis

Kapitel 1

Einleitung

Graphen sind ein anschaulicher, mathematisch präziser und ausdrucksstarker Formalismus zur Modellierung von in Beziehung stehenden Entitäten. Veränderungen von Graphen können mit Graphersetzungsregeln – bestehend aus einem zu suchenden Muster und einer durchzuführenden Ersetzung – deklarativ beschrieben werden. Das Graphersetzungswerkzeug GrGen übersetzt Graphersetzungsregeln in imperative Programm-Module, die es dem Nutzer zur Ausführung bereitstellt.

Im Rahmen dieser Arbeit wurde GrGen um wesentlich ausdrucksstärkere, zusammengesetzte Regeln erweitert. In die Regelspezifikationssprache wurden Teilmuster und alternative Muster aufgenommen, die zusammengenommen rekursive Muster ermöglichen. Diese Muster wurden zudem um eine abhängige Ersetzung erweitert, was zu Teilregeln, alternativen Regeln und schließlich rekursiven Graphersetzungsregeln führt.

Die Semantik der Musterzusammensetzung werde ich in dieser Arbeit aufbauend auf Sterngraphgrammatiken mit Musterzusammensetzungsgrammatiken fundieren, und die Semantik der Regelzusammensetzung aufbauend auf Paargraphgrammatiken mit Regelzusammensetzungsgrammatiken.

Zum Aufbau: In Kapitel 2 erfolgt eine Einführung in das Graphersetzungssystem GrGen, seine Sprachen und deren Semantik, in Kapitel 3 wird die Erweiterung der Muster um Teilmuster und alternative Muster beschrieben und in Kapitel 4 deren Erweiterung zu Teilregeln über eine abhängige Ersetzung; dabei folgt jeweils auf eine informelle Erläuterung eine formale Unterfütterung. In Kapitel 5 wird die Implementierung von GrGen vorgestellt, getrennt in den Codegenerator und den generierten Code. Auf dieser Basis aufbauend wird dann in den Kapiteln 6 und 7 die Implementierung der Erweiterungen erläutert, wobei zuerst die Änderungen am generierten Code und dann die Änderungen am Vorgang der Codegenerierung betrachtet werden. In Kapitel 8 schließlich vergleiche ich GrGen mit Viatra 2 im Hinblick auf die Ausführungsgeschwindigkeit der rekursiven Muster.

Kapitel 2

Graphersetzung mit GrGen

Graphen sind eine abstrakte und dennoch anschauliche Form der Datenorganisation, in der Beziehungen zwischen Entitäten durch Geflechte von durch Kanten verbundenen Knoten dargestellt werden. Mit ihnen können auch komplexe Aufgabenstellungen modelliert werden, die sich einer angemessenen Beschreibung durch weniger mächtige Datenstrukturen wie Listen und Bäume entziehen.

Berechnungen auf Daten in Graphform können auf intuitive und doch formale Weise durch Graphersetzungsregeln beschrieben werden, bestehend aus einem *Mustergraphen* L und einem *Ersetzungsgraphen* R. Bei der Anwendung einer Regel wird im *Arbeitsgraphen* G eine Instanz des Mustergraphen gesucht, und, wenn gefunden, durch eine Instanz des Ersetzungsgraphen ersetzt, mit einem veränderten Arbeitsgraphen, dem *Ergebnisgraphen* G' als Resultat. Das Vorgehen bei der Graphersetzung ist in Abbildung 2.1 abstrakt skizziert.

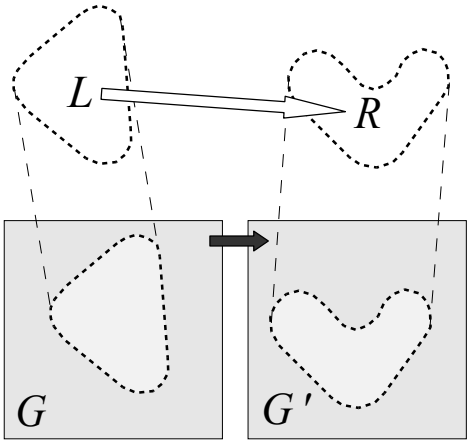


Abbildung 2.1: Graphersetzung abstrakt

Im Folgenden möchte ich die hinter GrGen.NET, dem Graph Rewrite GENerator in seiner .NET-Version stehenden Konzepte vorstellen, da sie die Basis meiner Erweiterungen darstellen; weitergehendere Ausführungen sind in [Gei08] und [BG08] zu finden.

2.1 Graphen

Graphen existieren in verschiedenen Formen, für uns interessant sind gerichtete, typisierte und attributierte Multigraphen, da diese das Metamodell bilden, welches dem GrGen-System zugrunde liegt. GrGen.NET wurde nebenläufig zu dieser Diplomarbeit um ungerichtete Kanten erweitert [Buc08], die zwei Knoten lediglich miteinander verbinden, ohne diese zu unterscheiden; mangels Relevanz für diese Arbeit werde ich sie im Folgenden aber nicht berücksichtigen.

Gerichtet heißt, dass eine Kante von einem ausgezeichneten Quell- zu einem ausgezeichneten Zielknoten führt.

Typisiert bedeutet, dass Knoten wie Kanten in Klassen eingeteilt werden und jedes Auftreten im Graphen mit einer solchen Klasse markiert ist.

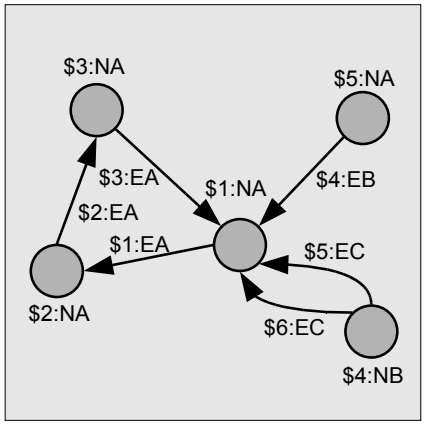
Attributiert sind Knoten und Kanten, die in Abhängigkeit von ihrem Typ mit Eigenschaften versehen sind.

Multigraph heißt ein Graph, bei dem zwei Knoten durch mehrere Kanten (desselben Typs) verbunden sein dürfen.

Über .gm-Dateien kann der Nutzer dem Metamodell entsprechende Graphmodelle definieren; dabei kann er zulässige Knoten- und Kantentypen sowie zugehörige Attribute festlegen und darüber hinaus Verbindungszusicherungen angeben. Die Typen sind in einer Untertypbeziehung angeordnet; sämtliche Knotentypen sind vom Basistyp Node, sämtliche Kantentypen vom Basistyp Edge abgeleitet.

Zur Veranschaulichung ist in der Abbildung 2.2 links ein Graph skizziert, der dem rechts oben stehenden Graphmodell entspricht und der durch die rechts unten stehende GrGen-Regel aus dem Nichts heraus materialisiert wurde (an die Knoten und Kanten ist ihre Identifikationsnummer und ihr Typ annotiert, nicht der Name des Knotens oder der Kante aus der erzeugenden Regel; so wurde zum Beispiel `$1`:NA durch `n1:NA` erzeugt).

Knoten und Kanten wollen wir zusammenfassend als Graphelemente bezeichnen. Im Folgenden werden wir die über Graphen getroffenen Aussagen formalisieren.




```
1  node class NA;
2  node class NB;
3  edge class EA;
4  edge class EB;
5  edge class EC;
```

```
1  rule materialize {
2    replace {
3      n1:NA -:EA-> n2:NA;
4      n2 -:EA-> n3:NA;
5      n3 -:EA-> n1;
6      :NA -:EB-> n1;
7      n4:NB -e1:EC-> n1;
8      n4 -e2:EC-> n1;
9    }
10 }
```

Abbildung 2.2: Graph mit erzeugender Regel

Graph

Unter einem Graphen wollen wir zunächst ein Tupel (N, E, src, tgt) verstehen, dabei gilt:

- N ist die Menge der Knoten (nodes),
  E die Menge der Kanten (edges).
- Die Abbildung $src : E \to N$ gibt zu einer Kante ihren Quellknoten, die Abbildung $tgt : E \to N$ zu einer Kante ihren Zielknoten an.

Eine Kante führt von einem Quell- zu einem Zielknoten, womit die Richtung der Kante festgelegt wird. Für einfache Graphen, bei denen zwischen je zwei Knoten in jeder Richtung nur *eine* Kante auftreten kann, wäre eine Knotenmenge N mit einer Kantenrelation $E = (N, N)$ ausreichend, für *Multi*graphen jedoch benötigen wir diese kompliziertere Modellierung. Schleifen, das sind Kanten mit $src(e) = tgt(e)$, sind zulässig.

Anliegend und Benachbart

Die Funktion $inc : N \to \mathfrak{P}(E)$ liefert die an einem Knoten *anliegenden*, oder auch *inzidenten* Kanten, sie ist definiert durch $inc(n) := \{e \in E | src(e) = n \vee tgt(e) = n\}$.
Die Funktion $inc : E \to \mathfrak{P}(N)$ liefert die an einer Kante anliegenden Knoten, sie ist definiert durch $inc(e) := \{src(e), tgt(e)\}$.
Die Funktion $adj : N \to \mathfrak{P}(N)$ bestimmt schließlich die zu einem Knoten n *benachbarten*, oder auch *adjazenten* Knoten; das sind alle Knoten, die an

einer Kante anliegen, die an n anliegt, bis auf n selbst. Die Funktion ist definiert durch $adj(n) := \{inc(e)|e \in inc(n)\} \setminus n$.

Typisierung

Wir wollen sowohl die Knoten als auch die Kanten mit Typen versehen. Die Typen sollen in einer Typhierarchie ähnlich der Klassenhierarchie in objektorientierten Programmiersprachen angeordnet sein. Da die Typisierung der Knoten und Kanten den gleichen Gesetzmäßigkeiten folgt, wollen wir den Begriff Typmodell zunächst generisch einführen. Ein Typmodell ist ein Tripel $T = (\Sigma, \leq, \bot)$, mit

- einer Menge von Typen Σ,
- und einer Halbordnung $\leq$ über Σ,
- die ein kleinstes Element $\bot \in \Sigma$ besitzt.

Eine solches Typmodell $T = (\Sigma, \leq, \bot)$ tritt im Graphbegriff von GrGen zweimal auf: in der Typhierarchie der Knoten $T_N = (\Sigma_N, \leq_N, \bot_N)$ und in der Typhierarchie der Kanten $T_E = (\Sigma_E, \leq_E, \bot_E)$; $\bot_N$ entspricht dem Basistyp `Node`, $\bot_E$ entspricht dem Basistyp `Edge`. Das *Typmodell* $\mathbb{T}$ eines Graphen definieren wir als das Paar (T_N, T_E) der genannten Typmodelle.

Unter einem Graphen über einem Typmodell $\mathbb{T} = (T_N, T_E)$ wollen wir ein Tupel $(N, E, src, tgt, type_N, type_E)$ verstehen, dabei gilt:

- N ist die Menge der Knoten (nodes),
  E die Menge der Kanten (edges).
- Die Abbildung $src : E \to N$ gibt zu einer Kante ihren Quellknoten, die Abbildung $tgt : E \to N$ zu einer Kante ihren Zielknoten an.
- Die Abbildung $type_N : N \to \Sigma_N$ ordnet jedem Knoten einen Knotentyp, die Abbildung $type_E : E \to \Sigma_E$ jeder Kante einen Kantentyp zu.

Graphen sind darüber hinaus in GrGen noch attributiert, da die Attributierung aber für diese Arbeit von nachrangiger Bedeutung ist, möchte ich den Leser auf [Gei08] verweisen. Für die Klasse aller Graphen über $\mathbb{T}$ wollen wir $\mathfrak{G}_{\mathbb{T}}$ schreiben.

2.2 Graphmustersuche

Vor der Anwendung einer Regel (Produktion) $p : L \to R$ muss im Arbeitsgraphen erst nach einer Instanz des Mustergraphen L gesucht werden. Eine gefundene Instanz heißt *Passung* (match) m von L in G, formal wird sie durch einen Graphhomomorphismus vom Mustergraphen in den Arbeitsgraphen beschrieben.

Graphhomomorphismus

Zuerst wollen wir den untypisierten Fall betrachten. Unter einem totalen Graphhomomorphismus H von einem Graphen G_1 in einen Graphen G_2 mit

$$G_1 = (N_1, E_1, src_1, tgt_1) \text{ und } G_2 = (N_2, E_2, src_2, tgt_2)$$

verstehen wir ein Paar von Abbildungen

$$(H_N : N_1 \to N_2, H_E : E_1 \to E_2),$$

das die Eigenschaft der strukturellen Gleichheit erfüllt, formal:

- $\forall e \in E_1 : (H_N \circ src_1)(e) = (src_2 \circ H_E)(e)$
- $\forall e \in E_1 : (H_N \circ tgt_1)(e) = (tgt_2 \circ H_E)(e)$

D.h. die Struktur des Quellgraphen findet sich im Zielgraphen wieder.

Wenn wir die Typisierung einbeziehen, ändern sich die Definitionen folgendermaßen: Unter einem totalen Graphhomomorphismus H von einem Graphen G_1 in einen Graphen G_2, beide über dem Typmodell $\mathbb{T} = (T_N, T_E)$, mit

$$G_1 = (N_1, E_1, src_1, tgt_1, type_{N_1}, type_{E_1})$$
$$G_2 = (N_2, E_2, src_2, tgt_2, type_{N_2}, type_{E_2})$$

verstehen wir ein Paar von Abbildungen

$$(H_N : N_1 \to N_2, H_E : E_1 \to E_2),$$

das folgende Eigenschaften erfüllt:

Strukturelle Gleichheit:

- $\forall e \in E_1 : (H_N \circ src_1)(e) = (src_2 \circ H_E)(e)$
- $\forall e \in E_1 : (H_N \circ tgt_1)(e) = (tgt_2 \circ H_E)(e)$

Typverträglichkeit:

- $\forall n \in N_1 : type_{N_1}(n) \leq_N type_{N_2}(H_N(n))$
- $\forall e \in E_1 : type_{E_1}(e) \leq_E type_{E_2}(H_E(e))$

Wir fordern also bei den Typen keine Gleichheit, sondern sehen eine Abbildung auch dann als Graphhomomorphismus an, wenn der Typ eines Zielelementes ein Untertyp eines Quellelementes ist.

Zu beachten ist, dass bei dieser Definition mehrere Knoten/Kanten aus dem Quellgraphen auf einen Knoten/eine Kante im Zielgraphen abgebildet werden können; die Passung ist also, wie in Abbildung 2.3 dargestellt, nicht zwangsläufig injektiv.

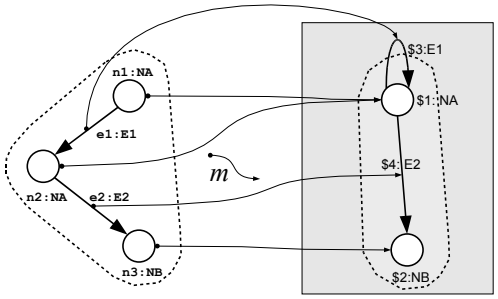


Abbildung 2.3: Nicht injektive Passung

2.3 Graphersetzung

Wir wissen nun, wie Graphen aufgebaut sind und was unter der Instanz eines Mustergraphen im Arbeitsgraphen verstanden wird. Noch offen ist die Beziehung vom Mustergraphen L zum Ersetzungsgraphen R in der Regel $p : L \to R$ und wie der Ergebnisgraph G' nach Anwendung der Regel auf den Arbeitsgraphen G aussieht.

In GrGen wird der Ansatz der Single Pushout-Graphersetzung, kurz *SPO* [CEH+97] verfolgt. Bei diesem wird die Beziehung zwischen dem Muster- und dem Ersetzungsgraphen durch einen partiellen und injektiven Graphmorphismus $r : L \to R$, auch *Erhaltungsmorphismus* genannt, hergestellt; eine Graphersetzungsregel besteht somit aus einem Tripel (L, R, r), auch $p : L \xrightarrow{r} R$ geschrieben. (Eine Abbildung $f : A \to B$ heißt partiell, wenn es Werte x in der Quellmenge A gibt, die kein Abbild in der Zielmenge B besitzen, $f(x)$ ist dann undefiniert, auch $f(x) = \perp$ geschrieben.) Der Erhaltungsmorphismus legt fest, welche Elemente von linker und rechter Seite sich entsprechen und beim Ersetzungsschritt erhalten bleiben.

Die Anwendung einer Regel wird bei gegebener Passung m als Konstruktion eines Pushouts (G', m^*, r^*) von (L, m, r) beschrieben. Ein Pushout ist ein Begriff der Kategorientheorie, für uns von Interesse ist die Kategorie Graph mit der Menge aller Graphen als Ansammlung von Objekten und der Menge aller Homomorphismen auf diesen Graphen als Ansammlung von Pfeilen; in dieser Kategorie existiert das Pushout immer und kann kanonisch konstruiert werden. Ein Pushout ist für m und r Pfeile und L einem Objekt ein Tripel (G', m^*, r^*), so dass das Diagramm 2.4 kommutiert und G' universell ist. Durch diese Eigenschaften wird sichergestellt, dass der Ergebnisgraph G' eine Abwandlung des Arbeitsgraphen G ist, in dem genau die für r erforderlichen Änderungen auf die Passung m angewandt wurden, nicht mehr und nicht weniger.

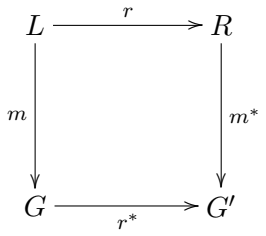


Abbildung 2.4: SPO

2.3.1 Eigenschaften der SPO-Graphersetzung

Diese Festlegungen führen grob gesprochen dazu, dass die von m nicht abgedeckten Elemente aus G unberührt, und die, durch die Abbildung r als sich entsprechend festgelegten Elemente aus der Passung m erhalten bleiben. Die nur in L vorhandenen Elemente werden aus G gelöscht, und die nur in R vorhandenen werden zu G hinzuinstanziiert. Diese Beschreibung ist nicht vollständig zutreffend, da es zu folgenden Konflikten kommen kann:

- Kanten können beim Löschen von Knoten Endpunkte abhanden kommen.
- Beim Einpassen können ein zu erhaltender Musterknoten und ein zu löschender Musterknoten auf ein- und denselben Arbeitsgraphknoten abgebildet werden.

Die Konflikte werden gemäß dem Formalismus der Single Pushout Graphersetzung gehandhabt:

- in der Luft hängende Kanten werden gelöscht
- Löschen hat Priorität vor dem Erhalten

Aufgrund der Festlegung für den ersten Konfliktfall können also auch nicht in m enthaltene Kanten aus G gelöscht werden, was zwar theoretisch unschön, aber praktisch nützlich ist. Durch die Festlegung für den zweiten Konfliktfall können auch Elemente gelöscht werden, die durch r als zu erhalten festgelegt wurden, dieser Fall kann aber nur bei nicht-injektiven Passungen auftreten. Dieses Verhalten wäre bei vielen Knoten, die potentiell zusammenfallen können, schwierig zu überblicken, der Regelschreiber könnte leicht durch nicht vorhergesehene Passungen überrascht werden. Deshalb ist in der GrGen-Syntax das injektive Matching als Standardfall gewählt worden. Da aber nicht-injektives Matching hin und wieder hilfreich ist, kann es in GrGen über eine „Dürfen-Zusammenfallen“-Deklaration `hom(n1,n2)` explizit angefordert werden.

2.3.2 Spezifikation einer Graphersetzungsregel

Eine Regel $p : L \xrightarrow{r} R$ wird in GrGen wie in Abbildung 2.5 spezifiziert.

```
"rule" name "{"
     BodyL
     "replace" "{"
          BodyR
     "}"
"}"
```

Abbildung 2.5: Regelsyntax

In dem nach dem Schlüsselwort `rule` geöffneten Namensraum des Musterrumpfes werden die Knoten des Mustergraphen L syntaktisch durch `Name:Typ;` und die Kanten durch `-Name:Typ->;` deklariert, ihre Verbindung untereinander wird durch Graphlets der Form `Name -Name-> Name;` festgelegt, wobei ein Stelle einer Namensnutzung auch eine Deklaration stehen kann. Auf die Musterelemente folgt das Schlüsselwort `replace`, im danach geöffneten Namensraum werden die Elemente des Ersetzungsgraphen R angegeben:

- Graphelemente, die hier deklariert werden, sind im Ersetzungsgraphen alleine enthalten, sie werden somit dem Arbeitsgraphen hinzuinstanziiert werden.
- Elemente l aus dem Muster, deren Namen hier genutzt werden, führen zum Hinzufügen eines Elementes r in den Ersetzungsgraphen und der Beziehung (l, r) in den Erhaltungsmorphismus. Sie werden erhalten bleiben.
- Im Muster deklarierte Elemente, die in der Ersetzung nicht mehr auftauchen, sind im Mustergraphen alleine enthalten und werden gelöscht werden.

Als alternative Regelnotation unterstützt GrGen vom Schlüsselwort `modify` umgebene Ersetzungen; im Muster deklarierte Elemente bleiben auch erhalten, wenn sie in dieser Ersetzung nicht genutzt werden, gelöscht werden können sie nur durch eine explizite `delete`-Anweisung. Werden unveränderte Elemente auch tatsächlich weggelassen, sind in der Ersetzung nur noch die Änderungen aufgeführt.

Die Syntax von GrGen wurde zu Beginn dieser Diplomarbeit auf die oben gezeigte Blockschachtelung des Musters in der Ersetzung umgestellt, um das auch vorher schon gegebene Enthaltensein des Namensraumes des Musters in der Ersetzung direkt zu repräsentieren. Das kommt der Intuition der blockschachtelungsgewohnten Programmierer entgegen und ermöglicht später, in verallgemeinerter Form, die Angabe der abhängigen Ersetzung

direkt bei den Alternativenzweigen. Auf diese Weise wird klar ersichtlich, welche Elemente die Ausgangsbasis für die Veränderung durch die Ersetzung darstellen.

Abbildung 2.6 konkretisiert die abstrakte Abbildung 2.1, die zugehörige GrGen-Regel wird in Abbildung 2.7 aufgelistet und in 2.8 schließlich wird die Passung der Musterelemente auf die Arbeitsgraphelemente tabellarisch angegeben (Musterelemente in `Schreibmaschinenschrift`, Arbeitsgraphelemente in serifenloser Schrift).

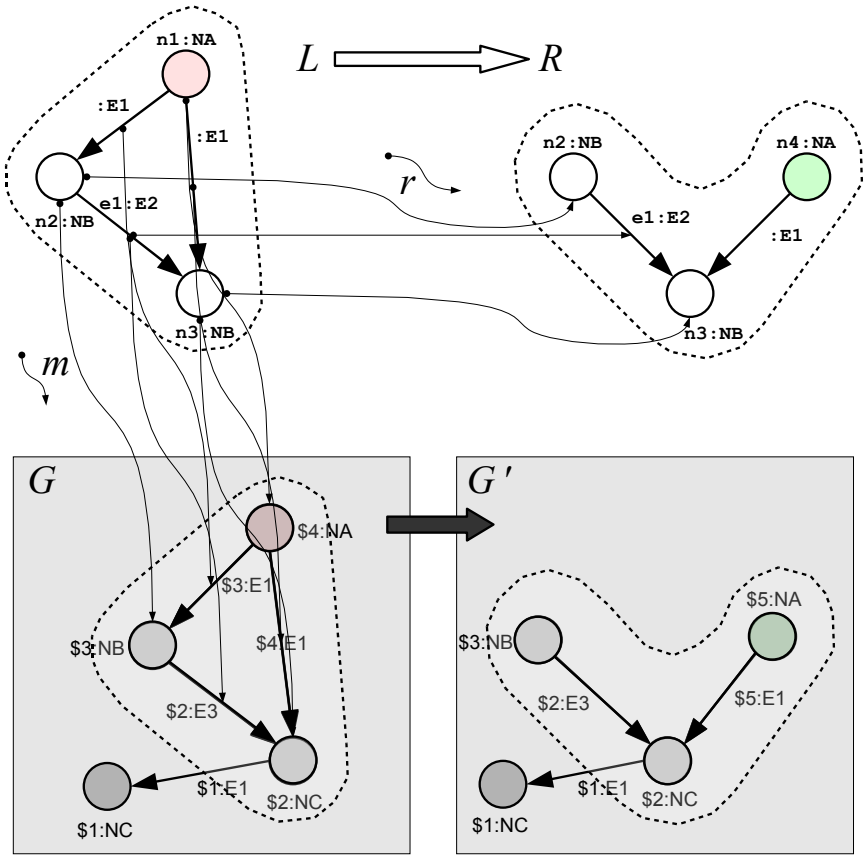


Abbildung 2.6: Graphersetzung konkret

2.4 Weitere Sprachelemente

Graphersetzungsregeln können mit Ein- und Ausgabeparametern versehen werden, durch Eingabeparameter können Teile des Passungsmorphismus m

```
1  rule foo {
2    n1:NA;
3    n1 -:E1-> n2:NB;
4    n1 -:E1-> n3:NB;
5    n2 -e1:E2-> n3;
6
7    replace {
8      n4:NA;
9      n4 -:E1-> n3;
10     n2 -e1:E2-> n3;
11   }
12 }
```

Abbildung 2.7: Graphersetzung konkret textuell in GrGen-Notation

Bindungen

L	G	R	G'
n1:NA	$4:NA	n2:NB	$3:NB
n2:NB	$3:NB	n3:NB	$2:NC
n3:NB	$2:NC	n4:NA	$5:NA
e1:E2	$2:E3	e1:E2	$2:E3

Hinweis zur Typhierarchie:
NB $\leq_N$ NC
E2 $\leq_E$ E3

Abbildung 2.8: Passung Musterelemente auf Arbeitsgraphelemente

```
1  rule foo(n1:NA, n2:NB) : (NB, NA) {
2    n1 -:E1-> n2;
3    n1 -:E1-> n3:NB;
4    n2 -e1:E2-> n3;
5
6    replace {
7      n4:NA;
8      n4 -:E1-> n3;
9      n2 -e1:E2-> n3;
10     return(n3,n4);
11   }
12 }
```

Listing 2.1: Regelparameter

vorbelegt werden, durch Ausgabeparameter können Teile des Morphismus m^* zurückgegeben werden. Das Beispiellisting 2.1 ist eine Abwandlung von Abbildung 2.7. Bei einer Anwendung der Regel über `(v1,v2) = foo(v1,v2)` mit initial `v1 = $4:NA` und `v2 = $3:NB` wird anhand der Parameterposition `n1` an `$4:NA` und `n2` an `$3:NB` gebunden, womit die Passungssuche den gleichen Match wie in 2.8 liefern muss. Nach der Anwendung der Regel gilt `v1 = $2:NC` und `v2 = $5:NA`, entsprechend der Bindungen der Musterelemente `n3` und `n4`, die in der `return`-Anweisung aufgeführt werden, an die Elemente aus dem veränderten Arbeitsgraphen.

Neben den Graphersetzungsregeln `rule` kennt GrGen noch Prüfungen `test`, die nur aus einem Mustergraphen bestehen. Bei ihrer Anwendung erfolgt lediglich eine Passungssuche, der Arbeitsgraph bleibt unverändert. Sie unterscheiden sich von Regeln mit einem leeren `modify`-Block durch die

```
1 test Foo {
2   beg:NA -e1:E-> end:NC;
3   negative {
4     beg -e2:E-> end; // Vorbelegung mit beg, end;
5   }
6 }
```

Listing 2.2: Vorbelegung negatives Muster

Möglichkeit, Ausgabeparameter bereits aus dem Muster zurückzugeben, was in einer Regel nur in der Ersetzung geschehen kann.

Die Graphersetzungsregeln von GrGen erlauben darüber hinaus noch weitere Mustergraphen als negative Anwendungsbedingungen anzugeben, die, wenn sie auf den Arbeitsgraphen passen, die Anwendung der Regel verhindern (siehe Abbildung 2.4). Die Passung eines negativen Musters wird unabhängig von der Passung des positiven Musters gesucht, in dem es syntaktisch geschachtelt aufgeführt wird. Eine Verbindung der Passung des negativen Musters mit der Passung des umgebenden Musters wird durch eine syntaktisch explizit anzufordernde Vorbelegung des Passungsmorphismus des negativen Musters mit Elementen aus dem Passungsmorphismus des positiven Musters hergestellt, was durch die Nutzung von Musterelementen aus dem positiven Muster im negativen Muster geschieht.

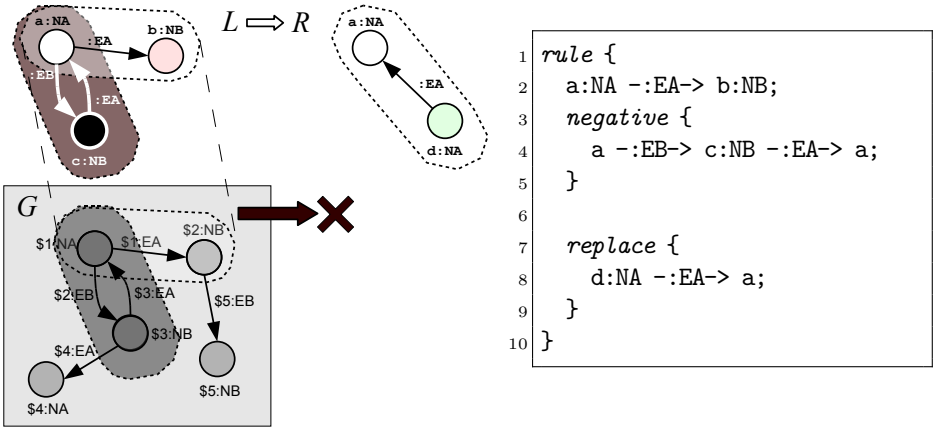


Abbildung 2.9: GrGen-Graph mit NAC grafisch und textuell

Im oben angeführten Beispiellisting 2.2 wird der Passungsmorphismus des negativen Musters mit den zu den Musterelementen `beg` und `end` gepassten Arbeitsgraphelementen aus dem Passungsmorphismus des umgebenden Musters vorbelegt, das zu `e1` gepasste Element hingegen ist im negativen

Passungsmorphismus unbekannt, somit wird `e2` unabhängig von ihm im Arbeitsgraph gesucht. Und da `e2` immer mit `e1` zusammenfallen kann, wird jede Passung des positiven Muster durch eine entsprechende Passung des negativen Musters verworfen, die obige Prüfung wird somit immer fehlschlagen.

Die zulässigen Muster können in einem `if`-Block innerhalb des Musters mit zusätzlichen Typ- und Attributbedingungen eingeschränkt werden; zudem können die Regeln durch den `dpo`-Modifizierer vor dem `rule`-Schlüsselwort auf DPO-Semantik [Ehr79] eingeschränkt werden, was bedeutet, dass zwei Pushouts konstruiert werden müssen, mit der Folgerung, dass die Passungen verworfen werden, die zum Auftreten von einem der beiden in Abschnitt 2.3.1 vorgestellten Konflikte führen würden. Des Weiteren können in einem `eval`-Block innerhalb der Ersetzung Attribute neu berechnet und zugewiesen werden, über `emit` kann eine Text-Ausgabe erzeugt werden und über `exec` kann die Ausführung einer Graphersetzungssequenz am Ende der Regelausführung angefordert werden.

2.5 Graphersetzungssequenzen

Die bei komplexeren Graphverarbeitungsaufgaben notwendige Kombination der Graphersetzungsregeln erfolgt durch eine speziellen Regelanwendungssteuerungssprache, die Graphersetzungssequenzen. Aus ihnen heraus werden die einzelnen Regeln aufgerufen, deren Anwendung wird in Abhängigkeit von Erfolg und Fehlschlag der Passungssuche vorhergehender Regeln und Prüfungen gesteuert, eine Anwendungsstelle kann über Ein/Ausgabeparameter weitergereicht werden. Die Sprache der Graphersetzungssequenzen ist induktiv definiert, dabei stehen unter anderem folgende Elemente zur Verfügung:

Regel R: Eine Passung des Regelmusters im Arbeitsgraphen wird gesucht, wurde eine (beliebige) gefunden, ist die Sequenz erfolgreich und die Ersetzung wird angewandt; wurde sie nicht gefunden, schlägt die Sequenz fehl. Ein Prüfung wird auf gleiche Weise verarbeitet, nur dass hier keine Ersetzung durchgeführt wird.

Logische Operationen: Die logischen Operationen Konjunktion S_1 && S_2 und Disjunktion S_1 || S_2 werden faul von links nach rechts ausgewertet, d.h. S_2 wird nicht ausgewertet, wenn der Erfolg oder Fehlschlag von S_1 bereits das Ergebnis der Operation bestimmt. Von beiden gibt es strikte Versionen S_1 & S_2 und S_1 | S_2, die immer beide Argumente auswerten; die logische Negation wird $!S$ geschrieben.

Iteration $S+$ oder $S*$: Die zu iterierende Sequenz wird so lange ausgeführt, bis sie das erste mal fehlschlägt; die Iteration $S+$ schlägt fehl, wenn die zu iterierende Sequenz keinmal erfolgreich ausgeführt wurde, die zu iterierende Sequenz $S*$ schlägt nie fehl, da auch 0-malige Anwendung als Erfolg gewertet wird.

Transaktionsklammern $< S >$**:** Die Effekte der zwischenzeitlich erfolgreichen Regelanwendungen innerhalb der von den Transaktionsklammern umschlossenen Sequenz auf den Arbeitsgraphen werden bei einem Fehlschlag der Sequenz als Ganzes rückgängig gemacht. Eine Operation zum Wiederaufsetzen einer fehlgeschlagenen Sequenz am letzten Entscheidungspunkt gibt es hingegen nicht, ein Durchprobieren aller möglichen Anwendungen einer Graphersetzungssequenz bis zum Erfolg ist somit nicht möglich.

Variablen v**:** dienen der Aufnahme von Knoten und Kanten, die Regeln oder Prüfungen nach ihrer Anwendung als Ausgabe zurückgeben oder vor ihrer Anwendung als Eingabe erhalten. Sie werden wie in `(v1,v2) = foo(v1,v2)` verwendet, die Deklaration erfolgt implizit bei der ersten Nutzung.

2.6 Aufbau des Systems

Der Aufbau des GrGen.NET-Systems ist in Abbildung 2.10 skizziert.

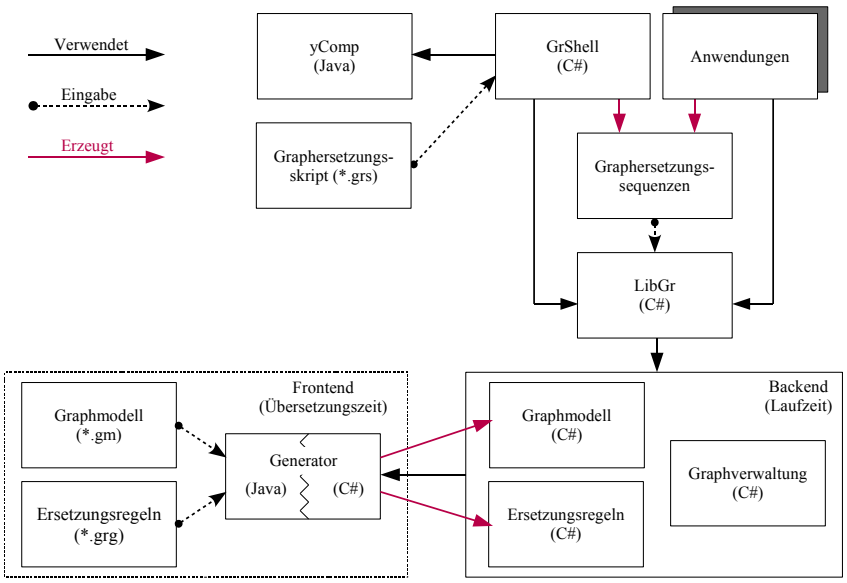


Abbildung 2.10: Aufbau GrGen.NET

Der GrGen.NET-Generator besteht aus einem in Java geschriebenen Frontend und einem in C# geschriebenen Backend. Das Frontend liest die Graphmodell- und Regelspezifikationsdateien ein und generiert daraus C# Code, der das Graphmodell implementiert, Beschreibungen der Muster ent-

hält, sowie den Code, der die Ersetzung durchführt. Das Backend generiert nach Reflektion über die Musterbeschreibungsobjekte den C#-Code, der nach der Passung des Musters sucht.

Die LibGr dient als Schnittstelle zum generierten Code, als Interpreter der Graphersetzungssequenzen bietet sie zudem einen einfachen Zugang zur programmierten Ablaufsteuerung.

Zum Entwickeln von Graphersetzungsanwendungen stellt das GrGen-System schließlich eine interaktive Kommandozeile, die GrShell, zur Verfügung. In ihr können Graphersetzungssequenzen schrittweise ausgeführt werden, unter Anzeige des Arbeitsgraphen und der gefundenen Passungen mit Hilfe des Graphvisualisierers yComp. Nach dem Rapid Prototyping mit diesen Werkzeugen können die entwickelten Graphersetzungsregeln und Sequenzen über die API der LibGr in beliebige Anwendungen integriert werden.

Aus den Graphersetzungsregeln wird ausführbarer Code generiert, der eine lokale Suche mit Rücksetzen im Arbeitsgraphen durchführt. Ein Suchplan legt dabei die Reihenfolge fest, mit der nach den einzelnen Musterelementen gesucht wird. Anhand von Graphanalysedaten und Heuristiken wird versucht, einen möglichst optimalen Suchplan aus der Menge der möglichen Suchpläne auszuwählen [Bat06, BKG08].

Die Semantik von GrGen ist über Graphmorphismen, SPO-Graphersetzung und Denotationelle Semantik definiert, ein umfassender Überblick ist in [Gei08] zu finden, für eine genaue Definition der Spezifikationssprachen von GrGen möchte ich auf [BG08] verweisen.

2.7 Verwandte Systeme

Mit GrGen konkurrieren diverse andere Graphersetzungssysteme um die Gunst der Nutzer, die wichtigsten möchte ich hier kurz vorstellen. Ihnen gemein ist die, manchmal nicht ganz vollständige, manchmal über sie hinausgehende, Verwendung von typisierten, attributierten, und gerichteten Graphen als Metamodell. Nicht ganz vollständig, weil nicht alle der Systeme Kanten mit Attributen versehen können und weil z.B. AGG im eigentlichen Kern untypisiert ist – es reicht die Typisierung über Prüfungen gegen einen Typgraphen nach. Über sie hinausgehend, weil z.B. Fujaba auch ungerichtete Kanten beherrscht, die es aber auf Paare von gerichteten Kanten abbildet. Die eigentlichen Unterschiede liegen jedoch in der Graphersetzung und in der Infrastruktur.

Progres (Programmed Graph Rewriting Systems)

war das erste allgemeine Graphersetzungswerkzeug und ist immer noch eines der ausdrucksstärksten. Es verfolgt den Ansatz der programmierten Graphersetzung, bei dem die Graphersetzungsregeln parametrisiert und durch

programmartige Konstrukte kombiniert werden können. Es verfügt über einen gemischt grafisch-textuellen Editor für die Graphersetzungsregeln, die im Format linke Seite getrennt von der rechten Seite angegeben werden, sowie für die Steuerung, es kann aber auch rein textuell gearbeitet werden. Graphmodelle werden durch Schemata vergleichbar mit den .gm-Dateien angegeben, die Musterbeschreibungssprache kennt negative Knoten und Kanten, aber keine allgemeinen negativen Graphen, zusätzlich verfügt sie über Pfadausdrücke, optionale Knoten und Mengenknoten. Die Passungssuche erfolgt injektiv, homomorphes Zusammenfallen kann aber für ausgewählte Elemente angefordert werden. PROGRES verfügt sowohl über einen Codegenerator wie auch über einen Interpreter für seine Spezifikationssprache, zur Passungsbestimmung verwendet es eine geplante lokale Suche. Die benötigte Umgebung ist schwierig herzustellen und die Komponenten weisen deutliche Alterungsspuren auf – es kann als nicht mehr zeitgemäß und FUJABA als sein Nachfolger angesehen werden.

Fujaba (From UML to Java and back again)

ist ein Werkzeug für die visuelle Softwareentwicklung, in dem der Nutzer durch das Zeichnen von modifizierten UML-Diagrammen Programme beschreibt, die dann durch einen Codegenerator in ausführbaren Java-Code umgesetzt werden; die Diagramme entsprechen programmierter Graphersetzung. Das Graphmodell wird durch UML-Klassendiagramme und die Steuerung durch UML-Aktivitätendiagramme beschrieben, die Graphersetzungsregeln werden durch Story-Patterns, die UML-Kollaborationsdiagrammen (Objektinteraktion) ähneln, angegeben. In einem Story-Pattern sind linke und rechte Seite zusammengefasst, zu erzeugende und zu löschende Elemente werden durch `create`/`delete`-Markierungen hervorgehoben. Die Musterbeschreibungssprache unterstützt negative Knoten und Kanten, negative Graphen hingegen müssen über die Steuerung realisiert werden; des Weiteren können optionale Knoten, Mengenknoten, und Pfade spezifiziert werden. Es wird nach einer injektiven Passung gesucht, für ausgewählte Elemente kann homomorphes Zusammenfallen angefordert werden. Zur Passungsbestimmung wird eine lokale Suche verwendet, bei der aber der Programmierer den Startpunkt der Suche festlegen muss. Eine zusätzliche Unterstützung für Triple Graph Grammars, die sich gut zur Modelltransformation eignen, ist als Plugin verfügbar.

AGG (Attributed Graph Grammar System)

ist eine Entwicklungsumgebung für allgemeine Graphersetzung mit einem grafischen Editor für Graphersetzungsregeln. Graphelemente können mit beliebigen Java-Objekten attributiert, das Graphmodell kann über einen Typgraphen eingeschränkt werden. Graphen werden im Format linke Seite ge-

trennt von der rechten Seite gezeichnet, zusätzlich können negative Mustergraphen angegeben werden; zwischen homomorpher und isomorpher Passung kann nur für ganze Graphen gewählt werden. Die Semantik der Graphersetzung ist im Gegensatz zu den beiden vorherigen Werkzeugen SPO-basiert und theoretisch wohlfundiert. AGG verwendet als einziges der hier vorgestellten Werkzeuge nicht den Ansatz der programmierten Graphersetzung, sondern den Ansatz der Graphgrammatiken, bei dem die anzuwendende Regel indeterministisch aus der Menge aller Regeln ausgewählt wird, so wie die Anwendungsstelle indeterministisch aus der Menge aller möglichen Anwendungsstellen (es ist keine Parameterübergabe möglich) ausgewählt wird. Als einziges Mittel der Steuerung ist eine Schichtung der Graphgrammatiken möglich, bei der zwischen den Schichten umgeschaltet wird, wenn keine Regel der Vorgängerschicht mehr angewendet werden kann. Die Passungssuche wird auf ein CSP-Problem abgebildet und erfolgt durch einen CSP-Löser, ein Generator steht nicht zur Verfügung. AGG bietet mehrere Hilfsmittel zur Analyse von Graphgrammatiken bezüglich Eigenschaften wie Konfluenz und Terminierung, wie die Analyse kritischer Paare zum Finden von Konflikten zwischen Regeln. Aufgrund des Verzichts auf eine programmartige Steuerung (abgesehen von einer API für Java), und insbesondere wegen seiner Langsamkeit ist AGG nicht für praxisrelevante Graphersetzungsaufgaben geeignet.

GReAT (Graphical Rewrite and Translation)

ist zusammen mit dem GME (Generic Modeling Environment) ein Werkzeug zur Entwicklung von Modelltransformationen, genauer der Abbildung des Graphen eines Graphmodells in einen Graphen eines anderen Graphmodells. Graphmodelle werden über UML-Klassendiagramme gezeichnet, Graphersetzungsregeln werden grafisch in einer UML-ähnlichen Notation angegeben, linke und rechte Seite sind dabei zusammengefasst, zu erzeugende und zu löschende Elemente werden durch `create`/`delete`-Markierungen hervorgehoben. Eine Besonderheit von GReAT ist die Möglichkeit, zu Knoten und Kanten Kardinalitäten notieren zu können, über die festgelegt wird, in welcher Anzahl das entsprechende Element im Graph auftreten muss. Negative Graphen werden nicht unterstützt, aber über die Kardinalität 0 negative Graphelemente. Die Steuerung wird ebenfalls grafisch spezifiziert, durch Regelsequenzen, die in einer Art Datenflußdiagramm über ihre Ein/-Ausgabeparameter verbunden werden, wobei nicht nur Abfolgen, sondern auch Entscheidungen oder Wiederholungen möglich sind. Es stehen sowohl ein Interpreter wie auch ein Codegenerator zur Verfügung.

Viatra 2 (Visual Automated Model Transformations)

ist ein Eclipse-basiertes Modelltransformationswerkzeug, das auf Graphersetzung aufbaut und Graphen unterschiedlicher Modelle ineinander überführen kann. Es implementiert eine textuelle Sprache zur Beschreibung von Modellen und Metamodellen, wie auch für die Graphersetzungsregeln und ihre Steuerung, wobei die Regelbeschreibung sowohl im Format linke Seite nach rechter Seite, als auch zusammengefasst in einem Muster mit `create`/`delete`-Annotationen möglich ist. Im Format linke Seite nach rechter Seite erfolgt die Entscheidung Erhalten/Löschen nur für Elemente, die explizit als Parameter vom Muster zur Ersetzung übergeben werden, nicht wie in GrGen anhand von syntaktisch geschachtelten Elementen. Viatra 2 verfügt über eine mächtige Mustersprache, die es erlaubt, Teilmuster zu verwenden und alternative Muster anzugeben, durch deren Kombination rekursive Muster spezifiziert werden können. Außerdem sind geschachtelte negative Muster unbeschränkter Tiefe möglich, allerdings sind nur injektive Passungen vorgesehen. Die Steuerung der Graphersetzungsregeln erfolgt durch ASM-Programme (Abstract State Machine), zudem können die Regeln selbst um ein ASM-Programm erweitert werden. Die Passungssuche erfolgt über einen Interpreter, als Besonderheit bietet das System Unterstützung für generische und Metatransformationen, das sind Typ-Parameter und die Möglichkeit der Veränderung von Transformationen so wie gewöhnliche Modelle.

Kapitel 3

Erweiterungen der Muster

Mit den bisher vorgestellten Sprachmitteln müssen komplexe, flexible oder wiederholte Muster über operationale Graphersetzungssequenzen unter Zuhilfenahme von Schiffchen und Zurückrollen realisiert werden, was bei größeren Aufgaben zu einer umfangreichen und komplexen Steuerung führt. In meiner Studienarbeit [Jak07] habe ich als Abhilfe eine Erweiterung um rekursive Muster und Regeln entwickelt, die eine elegante, effiziente und deklarative Spezifikation erlaubt. Die Semantik habe ich in ihr nur informal eingeführt, ich möchte sie hier nun mit kontextfreien Graphgrammatiken fundieren.

3.1 Graphgrammatiken

Eine Graphgrammatik entspricht im Wesentlichen einem Graphersetzungssystem, d.h. einer Menge von Graphersetzungsregeln – der Hauptunterschied liegt in der Zielsetzung. Während mit einem Graphersetzungssystem eine Berechnung auf einem Arbeitsgraphen spezifiziert wird, dient eine Graphgrammatik der Beschreibung des Aufbaus einer Graphsprache. Eine Graphsprache besteht aus der Menge aller *Wortgraphen*, die aus einem *Startgraphen* durch fortgesetzte Regelanwendung ableitbar sind.

Besteht die Zielsetzung im Spezifizieren einer Graphsprache, werden für gewöhnlich in der Menge der Knoten- und/oder Kantentypen nichtterminale von terminalen Typen unterschieden. Die Sprache eines Startgraphen besteht dann aus allen, aus diesem durch fortgesetzte Regelanwendung ableitbaren terminalen Wortgraphen.

So wie bei den Zeichenkettengrammatiken sind auch bei den Graphgrammatiken die Grammatiken mit kontextfreien Regeln am bedeutendsten, da sich bei ihnen zu einem gegebenen Graphen dessen Ableitungen noch in praktikabler Zeit bestimmen lassen; außerdem können Menschen diese Klasse von Regeln noch gut kognitiv verarbeiten.

Innerhalb der kontextfreien Graphgrammatiken wird zwischen der Kno-

tenersetzung und der (Hyper-)Kantenersetzung unterschieden. Bei der Knotenersetzung bestehen die linken Seiten der Regeln aus jeweils einem nichtterminalen Knoten, der durch den Graphen auf der rechten Seite ersetzt wird. Dabei werden die Knoten der rechten Seite anhand von Verbindungsanweisungen mit den, vor der Regelanwendung in Nachbarschaft zum Nichtterminal stehenden Knoten verbunden. Bei der Kantenersetzung besteht die linke Seite einer Regel aus einer nichtterminalen Kante mit ihren anliegenden Knoten; die Kante wird durch den Graphen auf der rechten Seite ersetzt, dieser wird über die anliegenden Knoten mit dem Rest des Wortgraphen verklebt. Wirklich interessant ist diese Ersetzungsart erst nach einer Erweiterung auf Hyperkanten [Hab92], die über ihre Tentakel eine beliebige, aber zum Spezifikationszeitpunkt feststehende Anzahl von Knoten miteinander verbinden können.

Ich möchte eine neuere Form von kontextfreien Graphgrammatiken verwenden, die Sterngraphgrammatiken, die als Zwischenschritt hin zu den adaptiven Sterngraphgrammatiken in [DHJ$^+$06] eingeführt wurden; sie entsprechen den Hyperkantengraphgrammatiken, ohne aber die Begrifflichkeit der normalen Graphen mit ihren Knoten und Kanten zu verlassen.

3.1.1 Sterngraphgrammatiken

Ein *Stern* ist ein Graph G von einer speziellen Form:
Er besteht aus

- einer Mitte $center(G)$, einem Nichtterminalknoten
- seinen Strahlen $ray(G)$, das sind die anliegenden nichtterminalen Kanten,
- sowie den andersseitig an den Kanten anliegenden terminalen Knoten $ifc(G)$, den Anknüpfungspunkten bzw. Schnittstellenknoten des Sterns.

In Abbildung 3.1 ist ein Stern mit 5 Kanten grafisch dargestellt.

Um mit Sternen arbeiten zu können, müssen wir das Typmodell $\mathbb{T} = (T_N, T_E)$ erweitern: Wir unterscheiden in den Mengen Σ_N / Σ_E die bereits eingeführten, terminalen Typen von den nichtterminalen Typen $\mathring{\Sigma}_N$ / $\mathring{\Sigma}_E$, wobei jeder nichtterminale Typ $\mathring{\sigma}$ nur zu sich selbst in der Beziehung $\mathring{\sigma} \leq \mathring{\sigma}$ steht.

Eine *Sternersetzungsregel* $\mathring{p} : L \xrightarrow{r} R$ ist eine SPO - Graphersetzungsregel mit einem Stern auf der linken Seite, dessen Strahlen zu paarweise disjunkten Zielknoten führen, und einem Graphen auf der rechten Seite, über einem gemeinsamen Typmodell $\mathbb{T}$, für die gilt:

- der Nichtterminalknoten wird gelöscht $r(center(L)) = \perp$
- die Nichtterminalkanten werden gelöscht $\forall x \in ray(L) : r(x) = \perp$

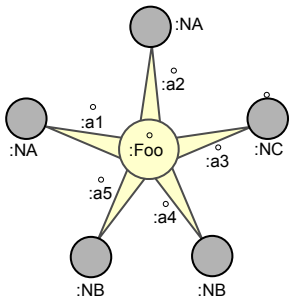


Abbildung 3.1: Stern

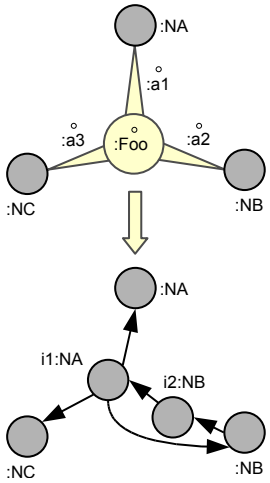


Abbildung 3.2: Sternersetzung

- die Anknüpfungspunkte bleiben erhalten $\forall x \in ifc(L) : r(x) \neq \bot$

An die übrigen Elemente der rechten Seite werden keine Einschränkungen gestellt, sie werden dem Wortgraphen hinzugefügt. Insbesondere dürfen auf der rechten Seite weitere Sterne auftreten. Eine Sternersetzungsregel wird in Abbildung 3.2 visualisiert.

Eine Menge P von Sternersetzungsregeln über einem Typmodell $\mathbb{T}$ nennen wir eine *Sterngrammatik*. Ein Ableitungsschritt $G \overset{P}{\Rightarrow} G'$ einer Sterngrammatik P besteht aus der (SPO)-Anwendung einer Sternersetzungsregel $\mathring{p} \in P$ auf den Graphen G, die zu dem Ergebnisgraphen G' führt. Wir sprechen von einer Sternableitung $G \overset{P}{\Rightarrow}_+ G'$, wenn eine Folge von Wortgraphen $X_1, \ldots, X_n$ existiert, so dass $G \equiv X_1$ und $G' \equiv X_n$ ist, und $X_i \overset{P}{\Rightarrow} X_{i+1}$ gilt, für $i = 1, \ldots, n-1$. Die Sprache $\mathbb{L}(Z)$ eines Startgraphen Z sind alle aus diesem ableitbaren terminalen Wortgraphen $\mathbb{L}(Z) = \left\{ G \in \mathfrak{G}_{\mathbb{T}\setminus\mathring{\mathbb{T}}} | Z \overset{P}{\Rightarrow}_+ G \right\}$.

3.1.2 Musterzusammensetzungsgrammatiken

Wir wollen die eingeführten Sterngraphgrammatikregeln nutzen, um die Beschreibungsmächtigkeit der Graphersetzungsregeln von GrGen zu erhöhen. Das ist ein interessantes Ziel angesichts der Tatsache, dass wie sie als *eingeschränkte* SPO-Graphersetzungsregeln eingeführt haben. Wir lösen diesen Widerspruch auf, indem wir sie nicht auf der Objektebene nutzen, sondern mit ihrer Hilfe die Mustergraphen zu Mustergraphschemata erweitern, die auf der Metaebene zu Mustergraphen aussubstituiert werden, bevor sie auf der Objektebene angewendet werden.

Wir geben dem Nutzer somit eine Sterngraphgrammatik an die Hand, mit der er die Zusammensetzung der zu suchenden Muster aus Musterteilen auf der Metaebene beschreibt, anstelle der Muster an sich. Hierzu erweitern wir die Typenmenge des Graphmodells von GrGen um Nichtterminaltypen für Knoten und um Nichtterminaltypen für Kanten. Zusätzlich zur Menge der Graphersetzungsregeln nehmen wir noch Musterzusammensetzungsregeln in eine Musterzusammensetzungsgrammatik auf. Die Nichtterminaltypen dienen der Musterzusammensetzung und treten nur in den Muster- und Ersetzungsgraphen der beiden Regelmengen auf, sowie in den Wortgraphen der Ableitung; sie werden niemals im Arbeitsgraphen selbst erscheinen. Außerdem treten sie in den Graphen nicht an beliebiger Stelle auf, sondern nur innerhalb von Sternen. Eine weitere Forderung besteht darin, dass in den Graphen nur Sterne vorkommen, für die auch exakt passende Muster in den Regeln der Musterzusammensetzungsgrammatik existieren. Mit diesem Wissen können wir nun die Erweiterungen der Muster der GrGen-Regeln durchgehen und ihre Semantik präzisieren.

3.2 Eingebettete Teilmuster

GrGen wurde im Rahmen dieser Diplomarbeit um eingebettete Teilmuster erweitert. Sie wurden mit dem pragmatischen Ziel entwickelt, Nutzern den Überblick über komplexe Graphen zu ermöglichen, sowie die Wiederverwendung von Teilgraphen zu erlauben, ohne deren Spezifikation von Hand kopieren zu müssen.

Dazu wurden die Sprachmittel *Teilmusterspezifikation* und *Teilmusterdeklaration* eingeführt. Eine Teilmusterspezifikation, in EBNF (siehe A.1)
`SubPattern ::= "pattern" Name "(" Connections ")" "{" Body "}"`
`Connections ::= (NodeDeclaration|EdgeDeclaration)||","`
enthält in ihrem Rumpf `Body` eine Musterspezifikation wie sie auch innerhalb einer GrGen-Regel auftreten kann. Im Gegensatz zu einer GrGen-Regel kann sie aber keine Graphelemente zurückgeben.

Zwischen den Klammern befinden sich die *Anknüpfungen*, die als eine Liste von Knoten- und Kantendeklarationen notiert werden und über die das Teilmuster mit seinem umgebenden Muster verbunden wird. Die deklarierten Graphelemente sind Verweise auf Graphelemente aus dem Musterkontext, sie werden in einer Teilmusterdeklaration an Elemente aus dem umgebenden Muster gebunden.

Durch eine Teilmusterspezifikation wird ein Teilmustertyp definiert, er kann dann in einer Teilmusterdeklaration zum Einführen eines Teilmusters dieses Typs in ein Regelmuster verwendet werden, in EBNF:
`Declaration ::= EntityName ":" TypeName "(" Element||"," ")"`
Eine Teilmusterdeklaration spezifiziert im Muster, welches sie enthält, das Auftreten einer Teilmusterentität mit dem angegebenen Namen und dem an-

```
1 pattern Foo(beg:NA, end:NC) {
2   beg --> i1:NB --> end;
3   beg --> i2:NB --> end;
4 }
```

Listing 3.1: Beispiel Teilmusterspezifikation

```
1 pattern Bar {
2   a:NA --> e:NC;
3   foo:Foo(a,e);
4 }
```

Listing 3.2: Beispiel Teilmusterdeklaration

gegebenen Teilmustertypen. Somit können Teilmuster innerhalb einer Musterspezifikation als Bestandteile neben Knoten und Kanten auftreten, eingeführt durch Teilmusterdeklarationen neben den Knoten- und Kantendeklarationen.

Innerhalb der Klammern der Teilmusterdeklaration werden die Bindungen der Anknüpfungen aus der Teilmusterspezifikation an Elemente aus dem, die Teilmusterdeklaration enthaltenden, Muster aufgeführt. Die Anzahl der Elemente muss gleich sein, das zugewiesene Graphelement muss vom gleichen Typ wie das Anknüpfungselement sein, oder einen von diesem abgeleiteten Untertyp aufweisen.

Ein Beispiel für eine Teilmusterspezifikation ist Listing 3.1, grafisch veranschaulicht in der Abbildung 3.3. Ein Beispiel für ein Muster, das eine Teilmusterdeklaration enthält ist in Listing 3.2 gegeben und in der Abbildung 3.4 visualisiert.

Semantik

Zu jeder Teilmusterspezifikation
`"pattern"` *name* `"("` $(cn_i$ `":"` $t_i)$ $||$`","` `")"`
mit dem Namen *name* und den Anknüpfungsnamen cn_i und Anknüpfungstypen t_i wird für das Teilmuster ein Nichtterminalknotentyp $\mathring{name}$ der Typmenge $\mathring{\Sigma}_N$ hinzugefügt, und werden für die Anknüpfungen Nichtterminalkantentypen $\mathring{cn}_i$ der Typmenge $\mathring{\Sigma}_E$ hinzugefügt. Wir wollen bei der Formalisierung nur Knoten als Anknüpfungen betrachten, da wir ansonsten mit Hilfsknoten oder Superkanten [Den07], also Kanten mit Kanten als Zielen arbeiten müssten. Der dadurch erlangte Erkenntnisgewinn stünde in keinem Verhältnis zur Verkomplizierung der Modellierung.

Den Musterzusammensetzungsregeln wird für jede Teilmusterspezifikation eine Sternersetzungsregel hinzugefügt. Der Stern auf der linken Seite hat einen Knoten des Nichtterminalknotentyps $\mathring{name}$ als Mitte und Kanten der

Nichtterminalkantentypen $\mathring{cn}_i$ als Strahlen, deren andersseitige Enden aus Knoten mit den Knotentypen t_i bestehen. Die rechte Seite besteht aus dem Rumpf der Teilmusterspezifikation. Mitte und Strahlen des Sterns werden somit entfernt, nur die Schnittstellenknoten verbleiben, für gewöhnlich mit dem Rumpfgraphen verknüpft.

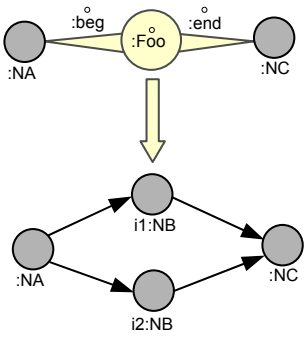


Abbildung 3.3: Teilmusterspezifikation

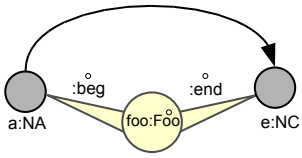


Abbildung 3.4: Teilmusterdeklaration

Für eine Teilmusterdeklaration
$name_{entity}$ `":"` $name_{type}$ `"("` $name_{element,i}||$`","` `")"`
innerhalb eines Regelmusters oder innerhalb einer Teilmusterspezifikation wird ein ihr zugehöriger Stern in den Graphen des Regelmusters oder der Teilmusterspezifikation eingefügt, die Strahlen des Sterns zeigen auf die Graphelemente $name_{element,i}$, entsprechend ihrer Position in der Bindungsliste.

Mit diesen Festlegungen können wir nun definieren, wie zu einer Graphersetzungsregel, deren Muster Teilmusterdeklarationen, also Sterne, enthält eine Passung zu suchen ist: Das Regelmuster wird als Ausgangsgraph genommen, es werden alle aus ihm mittels der Musterzusammensetzungsgrammatik ableitbaren terminalen Wortgraphen bestimmt, und erst mit diesen Worten der Graphsprache wird der eigentliche Passungsvorgang durchgeführt. Die Ein-Schritt-Ableitung des Musters aus Abbildung 3.4 mit der Regel aus Abbildung 3.3 ist in Abbildung 3.5 dargestellt.

Mit dem bisher definierten Sprachmittel der eingebetteten Teilmuster besteht die berechnete Graphsprache aus genau einem Graphen. Direkt oder indirekt rekursive Muster sind derzeit noch verboten (was sich im Abschnitt 3.4 ändern wird), da mit den aktuellen Sprachmitteln die Zusammensetzung nicht terminieren würde (nur ein unendlich großes Muster angegeben werden könnte).

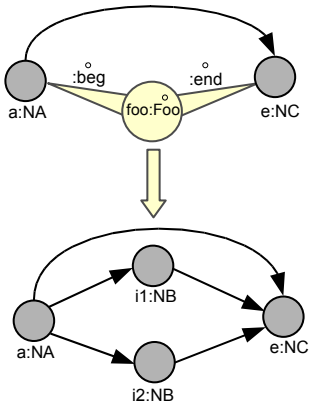


Abbildung 3.5: Teilmusterableitung

Passungsobjekt

Da nicht nur das Enthaltensein eines Elementes in der Passung eines zusammengesetzten Musters von Interesse ist, sondern auch zu welchem Teilmuster es gehört, erhält der Nutzer auf API-Ebene eine gefundene Passung in Form eines Ableitungsbaums bestehend aus Teilpassungsobjekten (siehe A.2), die aus der Teilpassung des terminalen Teils des Musters und den Teilpassungen aller enthaltenen Teilmuster bestehen. Der terminale Teil wird über eine Reihung für die lokalen Knoten und eine Reihung für die lokalen Kanten gegeben, mit fest vorgegebenen Indizes für jedes Graphelement, der nichtterminale Teil über eine Reihung von Teilpassungsobjekten für die lokal enthaltenen Teilmuster, ebenfalls mit fest vorgegebenen Indizes für jedes Teilmuster. Der Ersetzer für die Teilregeln (siehe Kapitel 4) verändert den Arbeitsgraphen ebenfalls anhand dieser Struktur.

3.3 Alternative Muster

Da sich die in Kapitel 2 beschriebenen starren Muster in der Anwendung als zu unflexibel erwiesen haben, wurde GrGen in dieser Diplomarbeit um Muster mit variablen Bestandteilen erweitert.

Um Muster mit Varianten beschreiben zu können, habe ich das Sprachmittel *Alternative* in die Regelbeschreibungssprache eingeführt, die Syntax in EBNF ist gegeben durch

```
Alternative ::= "alternative" "{" Case* "}"
Case ::= Name "{" PatternBody "}"
```

Der Name des Alternativenzweiges dient i) dem Herausfinden bei einer Inspektion des Passungsobjektes welcher Zweig gepasst wurde, ii) der Auswahl

des zu instanziierenden Zweiges bei einer späteren Erweiterung um das Hinzufügen von Teilmustern mit Alternativen und iii) der Selbstdokumentation. Die Alternative kann an einer beliebigen Stelle in einem Teilmuster oder im Muster einer Regel vor der Ersetzung eingeführt werden. Sie spezifiziert eine Auswahl zwischen den Teilmustern in ihren Zweigen; ein Muster mit Alternativen kann mit jedem der Zweige, so er denn im Graph vorhanden ist, zur Passung gebracht werden – aber innerhalb einer Passung immer nur mit einem zur gleichen Zeit.

```
1  pattern Foo {
2    alternative {
3      AwithB {
4        :NA --> :NB;
5      }
6      Awith2C {
7        a:NA --> :NC;
8        a --> :NC;
9      }
10   }
11 }
```

Im obigen Beispiel treten die Varianten nur als unterschiedliche Rümpfe einer Musterspezifikation auf, sie sind aber ein eigenständiges Sprachmittel, das eingebettet innerhalb eines Musters erscheinen kann. Ein Teilmuster `PatternBody` eines Alternativenzweiges ist, analog dem Teilmuster einer NAC, innerhalb des umgebenden Musters geschachtelt, die Graphelemente des umgebenden Musters sind im Teilmuster sichtbar. Damit ist es möglich, ein Muster bestehend aus einem festen Kern mit unterschiedlichen Anhängseln innerhalb nur einer Musterbeschreibung zu spezifizieren, wie es im unteren Beispiel gezeigt wird.

```
1  pattern Foo {
2    beg:NA --> i1:NB --> end:NC;
3    beg --> i2:NB --> end;
4    alternative {
5      Attachment {
6        end --> e1:NA;
7        end --> e2:NA;
8      }
9      Empty {
10     }
11   }
12 }
```

Semantik

Für jede Alternative wird ein Nichtterminalknotentyp $\mathring{alt}_i$ der Typmenge $\mathring{\Sigma}_N$ hinzugefügt, mit i fortlaufend aus $\mathbb{N}$; für alle in den Alternativenrümp-

fen verwendeten Graphelemente $name_{element}$ des umschließenden Musters werden Nichtterminalkantentypen $\mathring{name}_{element}$ der Typmenge $\mathring{\Sigma}_E$ hinzugefügt.

Zu jedem Alternativenzweig mit Namen *case* wird eine Sternersetzungsregel der Musterzusammensetzungsgrammatik hinzugefügt. Die linke Seite besteht aus dem oben eingeführten Nichtterminalknoten als Mitte, den Nichtterminalkanten als Strahlen und den Graphelementen aus dem umschließenden Muster als Schnittstellenknoten. Die rechte Seite besteht aus den Schnittstellenknoten sowie dem Rumpf des Alternativenzweiges.

Für jede Alternative wird der Stern der Alternative in das Muster eingefügt, welches sie enthält. Die Strahlen des Sterns zeigen auf die Graphelementen aus dem Muster, an die die Alternative, d.h. mindestens einer ihrer Zweige, anknüpft.

Zu einer Graphersetzungsregel, deren Muster Alternativen enthält, wird nun so wie auch schon bei den Teilmustern eine Passung gesucht: Das Regelmuster wird als Ausgangsgraph genommen, es werden alle aus ihm mittels der Musterzusammensetzungsgrammatik ableitbaren terminalen Wortgraphen bestimmt, und mit diesen Worten der Graphsprache wird der eigentliche Passungsvorgang durchgeführt. Im Gegensatz zum vorher eingeführten Konzept der Teilgraphen sind hier für jeden Alternativenstern mehrere Ersetzungen möglich, entsprechend der Anzahl der Alternativenzweige. Die Graphsprache besteht hier also aus einer (endlichen) Menge von terminalen Graphen, ein jeder dieser zusammengesetzten Graphen wird zu passen versucht, bis die gewünschte Anzahl an Passungen erreicht wurde oder alle möglichen Gesamtmuster ausprobiert worden sind. Die Reihenfolge, in der die Alternativenzweige zum Zusammensetzen des Gesamtmusters bei der Passungsbestimmung ausgewählt werden, ist undefiniert (und somit der Implementierung überlassen).

Passungsobjekt

Die Reihung der Teilmuster im Teilpassungsobjekt (siehe A.2) wird um Felder für die enthaltenen Alternativen erweitert. Jeder Alternative wird ein Index zugewiesen, an dieser Position befindet sich nach erfolgter Passungssuche die Teilpassung des gefundenen Alternativenzweiges. Um es dem Nutzer auf einfache Weise zu ermöglichen, festzustellen welcher Alternativenzweig gewählt wurde, enthalten die Teilpassungsobjekte einen Zeiger auf die Beschreibung des zugrundeliegende Musters aus der Metaschnittstelle mit den Regelrepräsentationen. Der Aufbau der Muster von allen Regelmustern, Teilmustern und Alternativenzweigen ist in dieser aufgeführt.

3.4 Rekursive Muster

Mit der Erweiterung von GrGen um rekursive Muster im Rahmen dieser Diplomarbeit können nun auch Muster mit einer statisch nicht bekannten Anzahl k von Wiederholungen eines Teilmusters verarbeitet werden, dem Hauptentwicklungsziel meiner Studienarbeit. Das Schöne an dieser Erweiterung ist, dass keine neuen Konstrukte in die Regelspezifikationssprache aufgenommen, sondern nur die Bestehenden konsequent implementiert werden müssen. Der Nutzer hält mit den eingebetteten und den alternativen Mustern bereits alle Mittel in der Hand: die Interaktion der beiden Sprachmerkmale ermöglicht rekursive Muster und damit wiederholte Teilmuster. Ein rekursiv spezifiziertes Muster besteht aus einem alternativen Muster, dessen eine Alternative, der Basisfall, das Muster aus einem elementaren Mustergraphen zusammensetzt, und dessen andere Alternative, der Rekursivfall, das Muster aus einem Graphen, in den das Muster selbst wieder eingebettet ist, zusammensetzt. Als Beispiel folgt die Spezifikation einer Kette (auch als iterierter Pfad bekannt), mit dem Basisfall in den Zeilen 3-5, und dem Rekursivfall in den Zeilen 6-9. Die zugehörigen Musterzusammensetzungsregeln sind in Abbildung 3.6 dargestellt, eine Beispielableitung ist in Abbildung 3.7 zu sehen.

```
1  pattern Chain(from:Node, to:Node) {
2    alternative {
3      Base {
4        from --> to;
5      }
6      Recursive {
7        from --> intermediate:Node;
8        rest:Chain(intermediate, to);
9      }
10   }
11 }
```

Semantik

Rekursive Muster führen zu rekursiven Regeln in der Musterzusammensetzungsgrammatik, in diesen wird der Stern des Teilmusters durch den Rumpfgraphen des Teilmusters ersetzt, der den Stern der Alternative enthält. Der Stern der Alternative wird mit ihnen zu einem terminalen Muster oder einem Muster, das wiederum den Stern des Teilmusters enthält, ersetzt. Das Musterschemata einer Graphersetzungsregel führt damit zu einer unendlichen Sprache von zu suchenden Regelmustern, die durch rekursives Aufzählen bestimmt werden können. Da der Arbeitsgraph endlich ist, ist die Aufgabe dennoch *effektiv* lösbar – das rekursive Aufzählen kann abgebrochen werden, sobald das Muster größer als der Arbeitsgraph wird (modulo nichtinjektivem Zusammenfallen). Und da beim Aufzählen viele Muster einen

gemeinsamen Anfang haben, ist das dynamische Zusammensetzen in Abhängigkeit vom Arbeitsgraphen während der Passungssuche sogar in vielen Fällen *effizient*.

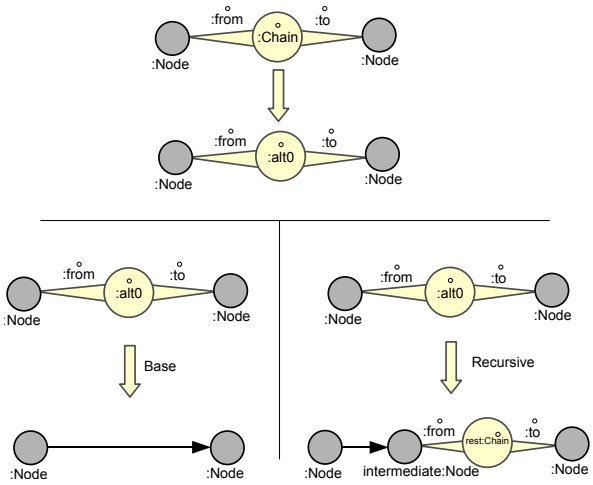


Abbildung 3.6: Teilmusterspezifikation Kette

Bemerkungen

Sind für eine Kette Anfang und Ende über Anknüpfungen gegeben, erhalten wir als Passung eine vollständige Kette zwischen diesen Punkten oder überhaupt keine Passung (je nachdem ob eine durchgängige Kette zwischen den Punkten existiert oder nicht). Ist hingegen nur der Anfang gegeben, muss beachtet werden, dass bereits der Basisfall eine zulässige Passung darstellt, obwohl die Kette im Arbeitsgraphen noch mehrere Schritte weitergegangen werden könnte. Ist diese Verhalten nicht gewünscht, müssen die zulässigen Muster eingeschränkt werden. Dazu ist der Basisfall mit einem negativen Muster zu versehen, welches sicherstellt, dass kein weiterer Rekursivschritt mehr möglich ist. Als Beispiel ist in Abb. 3.8 eine Kette ohne vorzeitigen Abbruch gegeben.

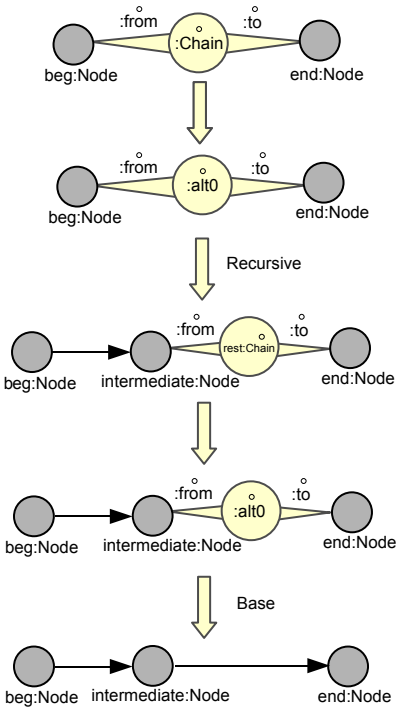


Abbildung 3.7: Teilmusterableitung Kette

```
1  pattern CompleteChain(from:NA) {
2    alternative {
3      Base {
4        negative {
5          from --> to:NA;
6        }
7      }
8      Recursive {
9        from --> to:NA;
10       rest:CompleteChain(to);
11     }
12   }
13 }
```

Abbildung 3.8: Vollständige Kette

Globale Isomorphie

Wir wollen noch einen Blick auf das Verhältnis von Homomorphie und Isomorphie bei Teilmustern werfen.

Frage: In welcher Beziehung stehen Elemente aus unterschiedlichen Teilmustern?
Antwort: Sie werden analog dem Standard innerhalb eines Musters gepasst: isomorph.

Und da ein Zusammenfallen über mehrere Teilmuster hinweg kaum noch zu durchschauen wäre, ist auch keine `hom`-Deklaration über mehrere Muster hinweg vorgesehen. Innerhalb eines Teilmusters hingegen können Elemente ebenso wie in einem Regelmuster über die `hom`-Deklaration als potentiell nicht-injektiv zueinander spezifiziert werden.

In die Begrifflichkeit des Musterzusammensetzens über die Sternregeln übertragen bedeutet dies, dass die durch eine Sternersetzung dem Regelmuster hinzugefügten Elemente in der Passung injektiv zu den bereits vor dieser Sternersetzung vorhandenen Elementen abgebildet werden müssen.

Eine Ausnahme gibt es bei Sternregeln für Alternativen, da die Alternativenzweige in der Spezifikationssprache lokal innerhalb des umschließenden Musters geschachtelt sind. Hier kann über eine `hom`-Deklaration eine potentiell nicht-injektive Passung für Elemente aus dem umschließenden Muster und Elemente aus dem Alternativenzweig erlaubt werden.

3.5 Negative Muster

Wie in Abschnitt 2.4 erläutert, wird die Passung eines negativen Musters bis auf die explizite Vorbelegung unabhängig von der Passung des positiven Musters gesucht; Elemente aus dem positiven und dem negativen Muster können also beliebig zusammenfallen, d.h. auf das gleiche Arbeitsgraphelement abgebildet werden, so wie bei einer `hom`-Deklaration für Elemente aus dem gleichen Muster.

Da wir aber im vorherigen Abschnitt die Passungen von Teilmustern als zueinander isomorph festgelegt haben, stellt sich nun die interessante Frage, wie sich negative Muster in einem Teilmuster zu den Mustern verhalten, die dieses Teilmuster enthalten. Man könnte sowohl der Isomorphie der Teilmuster untereinander wie auch der Unabhängigkeit der negativen Muster Vorrang einräumen.

Wegen der Möglichkeit, Mengenknoten realisieren zu können habe ich mich für den Vorrang der globalen Isomorphie der Teilmuster vor einer globalen Homomorphie der negativen Muster entschieden. Da der zweite Fall jedoch auch gewisse Vorzüge hat, ist in der Musterbeschreibungssprache für ihn bereits der `independent negative`-Block reserviert (er weist aber derzeit noch die gleiche Semantik wie der `negative`-Block auf).

Wenn wir weiterhin der Sichtweise folgen, dass der Passungsmorphismus eines negativen Musters bis auf die Vorbelegung unabhängig vom Passungsmorphismus des positiven Musters ist, ist die Semantik wie folgt festgelegt: Enthält die rechte Seite einer Grammatikregel einer Musterzusammensetzungsgrammatik ein negatives Muster, werden beim Musterzusammensetzen in dieses alle bis dahin abgeleiteten Elemente als Vorbelegung aufgenommen.

Durch diese Festlegung ist es möglich, eine maximal umfangreiche Passung sicherzustellen, indem ein negatives Muster erst dann am Feuern gehindert wird, wenn alle relevanten Elemente in seine Vorbelegung aufgenommen wurden. Dies wird im folgenden Beispiel ausgenutzt, in dem von einem festgehaltenen Anfangsknoten `head` aus schrittweise alle benachbarten Knoten aufgesammelt werden. Das entspricht einem Muster, das einen Knoten über eine Kante mit einem Mengenknoten verbindet; da keine weiteren Verbindungen zum Mengenknoten existieren, kann das Ergebnis – anschaulich gesprochen – als Kopf einer Pusteblume angesehen werden.

```
1  pattern Blowball(head:Node) {
2    alternative {
3      Base {
4        negative {
5          head --> :Node;
6        }
7      }
8      Recursive {
9        head --> :Node;
10       :Blowball(head);
11     }
12   }
13 }
```

Geschachtelte negative Muster

Im Rahmen dieser Diplomarbeit wurde GrGen um beliebig tief geschachtelte negative Muster erweitert, sie stellen eine konsequente Verallgemeinerung der bereits bekannten einfach geschachtelten negativen Muster dar. Eine erfolgreiche Passung des negativen Musters verhindert eine erfolgreiche Passung des umgebenden Musters, nur dass dieses jetzt ebenfalls ein negatives Muster anstelle eines positiven Musters sein kann. Die Passungssuche des negativen Musters erfolgt prinzipiell (lokal) unabhängig von der Passungssuche aller umgebenden Muster, nicht nur des direkt umgebenden Musters; alle Elemente aus den umgebenden Mustern, nicht nur die des direkt umgebenden Musters können als Vorbelegung verwendet werden.

Durch die links in der Beispielabbildung 3.9 notierte Prüfung wird eine Verbindung von einem Knoten a zu einem Knoten b gesucht, aber nur wenn a nicht mit einem weiteren Knoten c verbunden ist, mit der Ausnahme, dass

auch b mit diesem Knoten c verbunden ist. Ihr Muster passt auf den rechts dargestellten Arbeitsgraphen, da das innere negative Muster die Passung des äußeren negativen Musters verwirft.

```
1 test foo {
2   a:NA --> b:NB;
3   negative {
4     a --> c:NC;
5     negative {
6       b --> c;
7     }
8   }
9 }
```

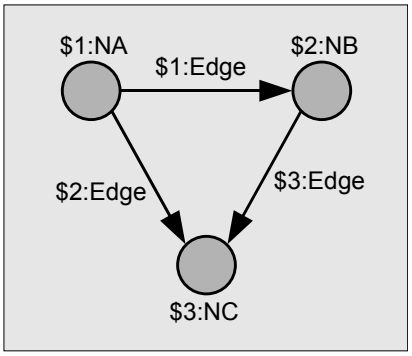


Abbildung 3.9: Geschachtelte negative Muster

3.6 Verwandte Arbeiten

Das einzige mir bekannte Graphersetzungssystem, das eine, der hier entwickelten Spezifikation von eingebetteten und alternativen Mustern mit deklarativen Regeln ähnlich mächtige und angenehme Beschreibung unterstützt, ist VIATRA 2 [Via08]. Mit iterierten Pfaden und Mengenknoten bieten PROGRES und FUJABA einen kleinen (aber wichtigen) Teil der durch rekursive Muster ermöglichten Funktionalität. Shaped Generic Graph Transformation [DHJ+08] ist eine Arbeit mit ähnlicher Richtung, in ihr werden die mächtigeren adaptiven Sterne verwendet, sie ist aber dafür rein theoretischer Natur.

Kapitel 4

Erweiterungen der Regeln

Im vorangegangenen Kapitel habe ich mich auf die Beschreibung der Erweiterungen des Mustergraphen konzentriert, was uns zu rekursiven Mustern und Musterzusammensetzungsgrammatiken geführt hat. Aber wir wollen die Muster nicht nur im Arbeitsgraphen finden, sondern diesen auch verändern. Wir benötigen also auch eine zusammengesetzte Ersetzung, und zwar eine, die in Abhängigkeit der Zusammensetzung des Musters aufgebaut wird, oder anders ausgedrückt: wir benötigen rekursive Regeln. Zur Modellierung von deren Semantik wollen wir auf Paargraphgrammatiken zurückgreifen.

4.1 Paargraphgrammatiken

Paargraphgrammatiken wurden in [Pra71] zur Spezifikation der Übersetzung zwischen Zeichenketten und Graphen eingeführt, sie ermöglichen aber auch die Übersetzung zwischen unterschiedlichen Graphen. Im genannten Artikel wurden sie, der Zielsetzung der Übersetzung von Zeichenketten in Graphsprachen folgend, für eine eingeschränkte Knotenersetzungsgrammatik mit je einem ausgezeichneten Eingangs- und Ausgangsknoten auf der rechten Seite einer Regel definiert, wobei nur diese Knoten mit dem umgebenden Graphen verbunden werden konnten. Ich werde sie im Folgenden für Sterngraphgrammatiken modifizieren; eine Erweiterung dieser Paarsterngraphgrammatiken dient dann schließlich der Spezifikation des Regelzusammensetzens.

Paarsterngraphgrammatiken

Eine Paarsterngraphgrammatik ist ein Paar von Sterngraphgrammatiken über einem gemeinsamen Typmodell, zusammen mit einer Entsprechung zwischen den Regeln der Grammatiken und den Nichtterminalknoten in den Regeln. Sich entsprechende Nichtterminale weisen den gleichen Typ auf.

Seien

- G und H Graphen über dem Typmodell $\mathbb{T}$,
- $\mathring{N}_G = \left\{ n \in N_G | type(n) \in \mathring{\Sigma}_N \right\}$ die nichtterminalen Knoten in G, d.h. die Sternmitten in G,
- $\mathring{N}_H = \left\{ n \in N_H | type(n) \in \mathring{\Sigma}_N \right\}$ die nichtterminalen Knoten in H, d.h. die Sternmitten in H

dann ist eine *Sternpaarung* $\overset{\circ\circ}{h}$ zwischen G und H

- eine injektive (und partielle) Abbildung $\overset{\circ\circ}{h}\colon \mathring{N}_G \to \mathring{N}_H$,
- so dass für alle Paare (x, y) mit $\overset{\circ\circ}{h}(x) = y$ gilt: $type(x) = type(y)$.

Eine *Paarsternregel* p ist ein Tripel $(\mathring{l}, \mathring{r}, \overset{\circ\circ}{h})$ mit

- $\mathring{l} : L_l \to R_l$ ist eine Sterngrammatikregel,
- $\mathring{r} : L_r \to R_r$ ist eine Sterngrammatikregel,
- für sie gilt: $type(center(L_l)) = type(center(L_r))$,
- und $\overset{\circ\circ}{h}$ ist eine Sternpaarung zwischen R_l und R_r.

Eine *Paarsterngraphgrammmatik* über einem Typmodell $\mathbb{T}$ ist eine Menge P von Paarsternregeln, jede über $\mathbb{T}$.

Für eine Paarsternregel $p = (\mathring{l}, \mathring{r}, \overset{\circ\circ}{h})$ in P heißt $\mathring{l}$ die linke Regel von P und $\mathring{r}$ die rechte Regel von P. Die linke/rechte Sterngraphgrammatik $P_{\mathring{l}}/P_{\mathring{r}}$ (über $\mathbb{T}$) der Paarsterngraphgrammatik P über $\mathbb{T}$ besteht aus der Menge der linken/rechten Regeln von P. Die linke/rechte Sprache der Paarsterngraphgrammatik ist die Sprache der linken/rechten Sterngraphgrammatik.

Die Sprache einer Paargraphgrammatik P besteht aus Paaren von Wortgraphen aus der linken und rechten Sprache von P. Die Paargraphgrammatik bestimmt, wie diese Paare von Graphen aus einem initialen Graphenpaar im Gleichschritt abgeleitet werden. Zu jedem Zeitpunkt der Ableitung haben wir ein Paar von Graphen mit einer Paarung zwischen den Sternen in den Graphen. Bei jedem Ableitungsschritt wird ein, durch eine Sternpaarung als zueinander gehörig definiertes Paar von Sternen (ein Stern in jedem Graph), gemäß einer Regel der Paargraphgrammatik ersetzt und eine neue Paarung zwischen den entstehenden Graphen erzeugt.

Ein *Graphenpaar* ist eine Tripel $(G, H, \overset{\circ\circ}{h})$ mit G und H Graphen über $\mathbb{T}$ sowie $\overset{\circ\circ}{h}$ einer Sternpaarung zwischen G und H. Ein „terminales“ Graphenpaar ist ein Graphenpaar mit leerer Sternpaarung.

Seien

- P eine Paarsterngraphgrammatik über $\mathbb{T}$,
- $X = (G_X, H_X, \overset{\circ\circ}{h_X})$ ein Graphenpaar über $\mathbb{T}$,
- $Y = (G_Y, H_Y, \overset{\circ\circ}{h_Y})$ ein Graphenpaar über $\mathbb{T}$,

dann schreiben wir $X \overset{P}{\Rightarrow} Y$ bzw. $X \overset{p}{\Rightarrow} Y$ für einen Ableitungsschritt, auch direkte Ableitung genannt, wenn

- es eine Paarsternregel $p = (\mathring{l}, \mathring{r}, \overset{\circ\circ}{h})$ in P gibt
- und einen Stern n in G_X gibt,
- und einen Stern m in H_X gibt,
- so dass die beiden Sterne durch $\overset{\circ\circ}{h}$ gepaart sind: $\overset{\circ\circ}{h}\ (center(n)) = center(m)$,

und

- $G_X \overset{\mathring{l}}{\Rightarrow} G_Y$ gilt, durch Anwendung der Sterngrammatikregel $\mathring{l}$ auf n,
- $H_X \overset{\mathring{r}}{\Rightarrow} H_Y$ gilt, durch Anwendung der Sterngrammatikregel $\mathring{r}$ auf m,
- $\overset{\circ\circ}{h_Y} = \overset{\circ\circ}{h} \cup \overset{\circ\circ}{h_X} \setminus (n, m)$ gilt, d.h. in die Paarung wird die Paarung aus der Regel aufgenommen und dafür das ersetzte Sternenpaar entfernt.

Ein Ableitungsschritt ist in Abbildung 4.1 skizziert.

$X \overset{P}{\Rightarrow}_* Y$ heißt Ableitung, wenn es eine Folge von Ableitungsschritten gibt, die mit X beginnt und mit Y endet.

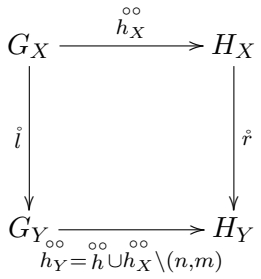


Abbildung 4.1: Ableitungsschritt

4.2 Regelzusammensetzungsgrammatiken

Wir möchten die Paarsterngraphgrammatiken zur Spezifikation des Regelzusammensetzens verwenden; wenn wir uns aber an Abschnitt 2.3 erinnern, besteht eine SPO-Graphersetzungsregel nicht nur aus zwei Graphen, dem Muster L und der Ersetzung R, sondern auch aus dem Erhaltungsmorphismus r zwischen ihnen. Zum einen müssen wir den Erhaltungsmorphismus

also mit L und R zusammen aufbauen, um festzulegen, welche Knoten und Kanten sich entsprechen. Zum anderen können wir ihn aber auch so erweitern, dass er zusätzlich die Sternpaarung $\overset{\circ\circ}{h}$ aufnimmt (mit r werden dann, über sich entsprechende Knoten und Kanten hinaus, noch die nichtterminalen Sternmitten von zueinander gehörigen Sternen in Beziehung gesetzt).

Wir betrachten somit anstelle von Paarsternregeln $p = (\mathring{l}, \mathring{r}, \overset{\circ\circ}{h})$ aus einer Paarsterngraphgrammatik *Regelzusammensetzungsregeln* $p = (\mathring{l}, \mathring{r}, r)$ mit $\mathring{l} : L_l \to R_l$ und $\mathring{r} : L_r \to R_r$ aus einer *Regelzusammensetzungsgrammatik*. Durch r werden die Graphen R_l und R_r zueinander in Beziehung gesetzt, genauer $R_l \setminus \mathit{ifc}(L_l)$ und $R_r \setminus \mathit{ifc}(L_r)$; die angeknüpften Elemente der Strahlen beider Seiten sind bereits weiter vorne in der Ableitung eingeführt und zueinander in Beziehung gesetzt worden.

4.3 Löschphase und Schattenknoten

Um den Nutzer einer Teilregel vor Überraschungen zu schützen wird festgelegt, dass Teilregeln angeknüpfte Elemente aus dem umgebenden Muster nicht verändern oder gar löschen dürfen. Von den angeknüpften Knoten des Sterns der linken Seite einer Regelzusammensetzungsregel wird somit verlangt, dass sie auch im Stern auf der rechten Seite vorhanden sind, mit einer entsprechenden Zuordnung im Erhaltungsmorphismus. Es wird festgelegt, dass der Stern der rechten Seite immer auch implizit alle Anknüpfungen der linken Seite enthält. Aufgrund der syntaktischen Schachtelung der Ersetzung im Muster wird dieses Verhalten auch vom Nutzer erwartet; der Zugriff auf die Anknüpfungen der linken Seite erlaubt es zudem sehr einfach, weitere Kanten an die angeknüpften Knoten anzufügen, ohne sie explizit über Ersetzungsanknüpfungen hineinreichen zu müssen.

Um eine Regel aber nicht unnötig zu beschränken, ist es in dieser dennoch durchaus erlaubt, einen, in ihr eingeführten, von einer Teilregel angeknüpften Knoten zu löschen. Damit würde aber dem Stern der rechten Seite eine Anknüpfung abhanden kommen, womit die Sternersetzungsregel der rechten Seite nicht mehr passen würde bzw. erst gar kein gültiger Wortgraph mehr vorläge. Um dieses Verhalten dennoch zu ermöglichen, erweitern wir die Knotenmenge des Graphen der rechten Seite N um eine weitere Menge N_{del} und führen nach der Phase des Zusammensetzen eine Löschphase aus. Zuerst erfolgt das Zusammensetzen der Regel als Ableitung mit den Regelzusammensetzungsregeln, dabei wird zu einem Knoten der linken Seite ohne Entsprechung auf der rechten Seite ein Schattenknoten in N_{del} und damit auf der rechten Seite eingefügt. Er fungiert als Anknüpfungsknoten für die implizite Anknüpfungen der gepaarten Sterne auf der rechten Seite. Nach erfolgtem Zusammensetzen wird einfach die Menge N_{del} vergessen, womit der Graph der rechten Seite nur noch die Knoten in N enthält, dabei werden alle an Knoten aus N_{del} anliegenden Kanten mit entfernt. Mit diesem

Rüstzeug wollen wir nun die Spezifikation der abhängig zusammengesetzten Ersetzung angehen.

4.4 Abhängig zusammengesetzte Ersetzung

Teilmuster werden durch die Angabe einer `replace-` oder `modify-`Ersetzung zu Teilregeln erweitert, mit ihnen kann das Teilmuster durch Ersetzungsoperationen auf seinen Elementen feinkörnig verändert werden.

Die Regel, welche das Teilmuster enthält, verwendet die im Teilmuster spezifizierte Ersetzung über eine *Teilmusterersetzungsnutzung*, bei der sie in ihrer Ersetzung den Namen der deklarierten Teilmusterentität, gefolgt von einer geöffneten und einer geschlossenen Klammer anführt (Intuition: Aufruf der abhängigen Ersetzung auf dem gepasstem Musterobjekt).

Wir folgen damit der Semantik der zusammengesetzten Muster und setzen den Ersetzungsgraph – in Abhängigkeit von der Struktur des Mustergraphen – zusammen. Die gepasste Teilmusterstruktur, d.h. die Schachtelung der Teilmuster ineinander, kann in der Ersetzung nicht geändert werden, wohl aber der Inhalt eines jeden Teilmusters (seine terminalen Knoten und Kanten). Es ist also z.B. nicht möglich, eine gepasste Baumstruktur in der Ersetzung in eine Listenstruktur zu plätten; es können aber die einzelnen Teilmuster der Baumstruktur beliebig geändert werden.

Analog den Musteranknüpfungen in den Teilmustern können auch die Ersetzungen der erweiterten Teilmuster Anknüpfungen erhalten, über die die einzelnen Ersetzungsteile miteinander verbunden werden. Die Anknüpfungen des Musters sind in der Ersetzung sichtbar, können aber nur zum Auslesen der Attribute oder dem Hinzufügen von Kanten zu einem angeknüpften Knoten verwendet werden. Der schreibende Zugriff auf die Attribute hingegen ist verboten, so wie das Löschen eines angeknüpften Elementes.

In Abb. 4.2 werden die eingeführten Konzepte mit den Teilregeln `Foo` und `Bar`, die in einer Regel `R` verwendet werden, in GrGen-Notation gezeigt.

Formalisierung

Wir wollen die bisherige informale Beschreibung im Folgenden formal fundieren. Dazu betrachten wir eine Teilmusterspezifikation mit abhängiger Ersetzung, wie sie in Abbildung 4.3 gegeben ist, mit dem Namen *name*, den Anknüpfungsnamen cn_i und Anknüpfungstypen t_i sowie den Ersetzungsanknüpfungsnamen rcn_j und Ersetzungsanknüpfungstypen rt_j. Für sie wird, so wie bei der Definition der Semantik der Teilmusterspezifikation, für das Teilmuster selber ein Nichtterminalknotentyp $\mathring{name}$ der Typmenge $\mathring{\Sigma}_N$ hinzugefügt, und es werden für die Anknüpfungen Nichtterminalkantentypen $\mathring{cn}_i$ der Typmenge $\mathring{\Sigma}_E$ hinzugefügt.

Außerdem wird eine Regelzusammensetzungsregel $p = (\mathring{l}, \mathring{r}, r)$ (mit $\mathring{l} : L_l \rightarrow R_l$ und $\mathring{r} : L_r \rightarrow R_r$) der Regelzusammensetzungsgrammatik hin-

```
1  pattern Foo(beg:NA, end:NC)
2  {
3    beg --> i1:NB --> end;
4    beg --> i2:NB --> end;
5    replace {
6      beg <-- end;
7    }
8  }
9
10 pattern Bar(li:NA)
11 {
12   a --> i3:NB;
13   modify(re:NC) {
14     a --> i4:NB <-- re;
15   }
16 }
17
18 rule R
19 {
20   a:NA --> e:NC;
21   foo:Foo(a,e);
22   bar:Bar(a);
23   replace {
24     n:NC --> e;
25     foo();
26     bar(n);
27   }
28 }
```

Abbildung 4.2: Beispiel Teilregeln

"pattern" *name* "(" (cn_i ":" t_i) ||"," ")" "{"
Body
"replace" "(" (rcn_j ":" rt_j) ||"," ")" "{"
ReplBody
"}"
"}"

Abbildung 4.3: Teilmusterspezifikation mit abhängiger Ersetzung

zugefügt, wobei $\mathring{l}$ eine Sternersetzungsregel ist, wie sie schon bei der Definition der Semantik der Teilmusterspezifikation eingeführt wurde, d.h. der Stern auf der linken Seite hat einen Knoten des Nichtterminalknotentyps $\mathring{name}$ als Mitte und Kanten der Nichtterminalkantentypen cn_i als Strahlen, deren andersseitige Enden aus Knoten mit den Knotentypen t_i bestehen. Die rechte Seite von $\mathring{l}$ wird durch den Rumpf *Body* der Teilmusterspezifikation bestimmt, die Schnittstellenknoten sind verblieben, die Mitten und die Strahlen des Sterns wurden durch den Rumpfgraphen ersetzt. Die Sternersetzungsregel $\mathring{r}$ besteht aus einem Stern auf der linken Seite mit einem Knoten des Nichtterminalknotentyps $\mathring{name}$ als Mitte und Kanten der Nichtterminalkantentypen cn_i, dann rcn_j als Strahlen, deren andersseitige Enden aus Knoten mit den Knotentypen t_i, dann rt_j bestehen. Die rechte Seite von $\mathring{r}$ besteht aus dem Graphen der Teilmusterspezifikation *ReplBody*. Mitte und Strahlen des Sterns wurden somit entfernt, nur die Schnittstellenknoten sind verblieben, für gewöhnlich mit dem Rumpfgraphen verknüpft. Es gilt $type(center(L_l)) = type(center(L_r))$ wie bei Paarsternregeln, zusätzlich gilt $\forall x \in ray(L_l)\ \exists y \in ray(L_r)$ so dass $type(x) = type(y)$.

Der die terminalen Knoten und Kanten in Beziehung setzende Teil des Erhaltungsmorphismus r zwischen R_l und R_r wird durch das Auftreten der Namen im Muster und in der Ersetzung wie in 2.3 definiert, nur hier für Teilregeln anstelle von gewöhnlichen Regeln. Der die Sterne im Sinne einer Sternpaarung in Beziehung setzende Teil von r wird durch die Ersetzungsnutzungen in *ReplBody* zu den Teilmusterdeklarationen in *Body* bestimmt, weshalb wir zuerst diese betrachten müssen:

Für eine Teilmusterdeklaration
$name_{entity}$ `":"` $name_{type}$ `"("` $name_{element,i}||$`","` `")"`
innerhalb des Rumpfes *Body* der Teilmusterspezifikation wird ein ihr zugehöriger Stern in den Mustergraphen der Teilmusterspezifikation L_r eingefügt, die Strahlen des Sterns zeigen auf die Graphelemente $name_{element,i}$, entsprechend ihrer Position in der Bindungsliste.

Die zugehörige Teilmusterersetzungsnutzung (der Zusammenhang wird durch den gleichen Namen $name_{entity}$ hergestellt)
$name_{entity}$ `"("` $name_{element,j}||$`","` `")"`
innerhalb des Ersetzungsrumpfes *ReplBody* führt zur Aufnahme eines, dem Stern der linken Seite entsprechenden Sterns in den Ersetzungsgraphen R_r, mit einer Mitte vom Typ $\mathring{name}$ und den Strahlen der impliziten Anknüpfungen sowie den Strahlen der Ersetzungsanknüpfungen aus der zugehörigen Teilmusterspezifikation. Die Strahlen der Ersetzungsanknüpfungen zeigen auf die Knoten $name_{element,j}$ aus dem Graph R_r, entsprechend ihrer Position in der Bindungsliste. Die Strahlen der impliziten Anknüpfungen zeigen auf die, den in L_r angeknüpften Knoten gemäß r entsprechenden Knoten aus R_r, wobei für angeknüpfte Knoten aus L_r ohne Entsprechung in R_r der Schattenknoten aus N_{del,R_r} verwendet wird. Der Sternpaarungsteil von r

besteht somit aus den Abbildungen der Mitten der Sterne aus L_r auf die Mitten der Sterne ihrer Entsprechungen in L_r.

Mit diesen Festlegungen können wir nun definieren, wie zu einer Graphersetzungsregel, deren Muster Teilmusterdeklarationen und deren Ersetzung zugehörige Teilmusterersetzungsnutzungen enthält, die Passung zu suchen und die Ersetzung durchzuführen ist: Das Tripel (L, R, r) der SPO-Graphersetzungsregel mit dem, um die Sternpaarungen erweiterten Erhaltungsmorphismus r ist der Ausgangspunkt. Aus ihm werden alle, mittels der Regelzusammensetzungsgrammatik ableitbaren terminalen Regeln bestimmt, eine jede davon wird zu passen versucht, bis eine erste Passung gefunden wurde, deren zugehörige Ersetzung wird schließlich ausgeführt.

In den folgenden Abbildungen 4.6, 4.4, 4.5 wird die beschriebene Semantik für das Beispiel aus Abb. 4.2 visualisiert. In der Abbildung 4.6 wird dabei das Zusammensetzen der Regel `R` gezeigt, zuerst mittels der in Abbildung 4.4 gezeigten Teilregel `Foo`, dann mittels der in Abbildung 4.5 gezeigten Teilregel `Bar`.

Alternativen

Alternativenzweige können einen lokal geschachtelten Ersetzungsteil erhalten, mit dem der Ersetzungsgraph zusammengebaut wird, wenn für das Gesamtmuster dieser Zweig der Alternative gewählt wurde.

```
1  pattern Foo {
2    beg:NA --> i1:NB --> end:NC;
3    beg --> i2:NB --> end;
4
5    alternative {
6      Attachment {
7        end --> e1:NA;
8        end --> e2:NA;
9        replace {
10         end --> e1;
11         end --> e2;
12         e1 --> e2; e2 --> e1;
13       }
14     }
15     Empty {
16       replace {
17       }
18     }
19   }
20
21   replace {
22     beg --> end;
23   }
24 }
```

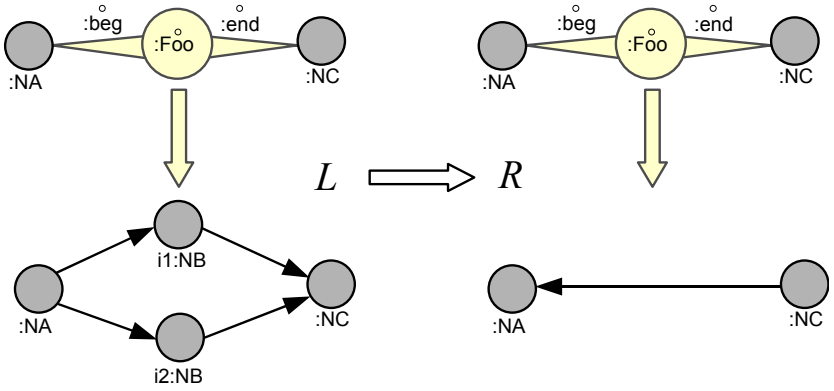


Abbildung 4.4: Teilregelspezifikation Foo zu Abb. 4.2

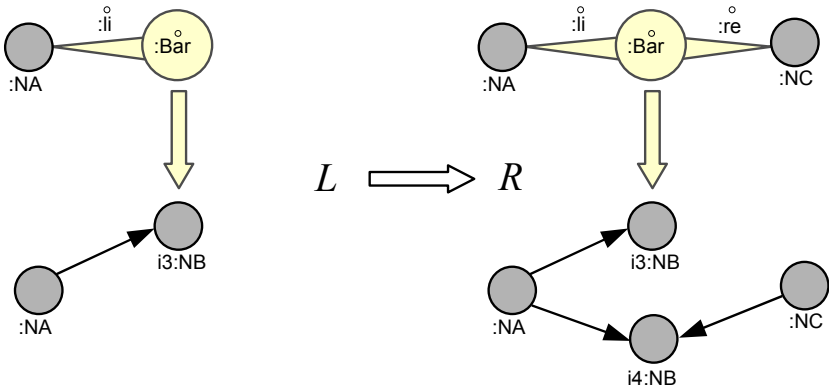


Abbildung 4.5: Teilregelspezifikation Bar zu Abb. 4.2

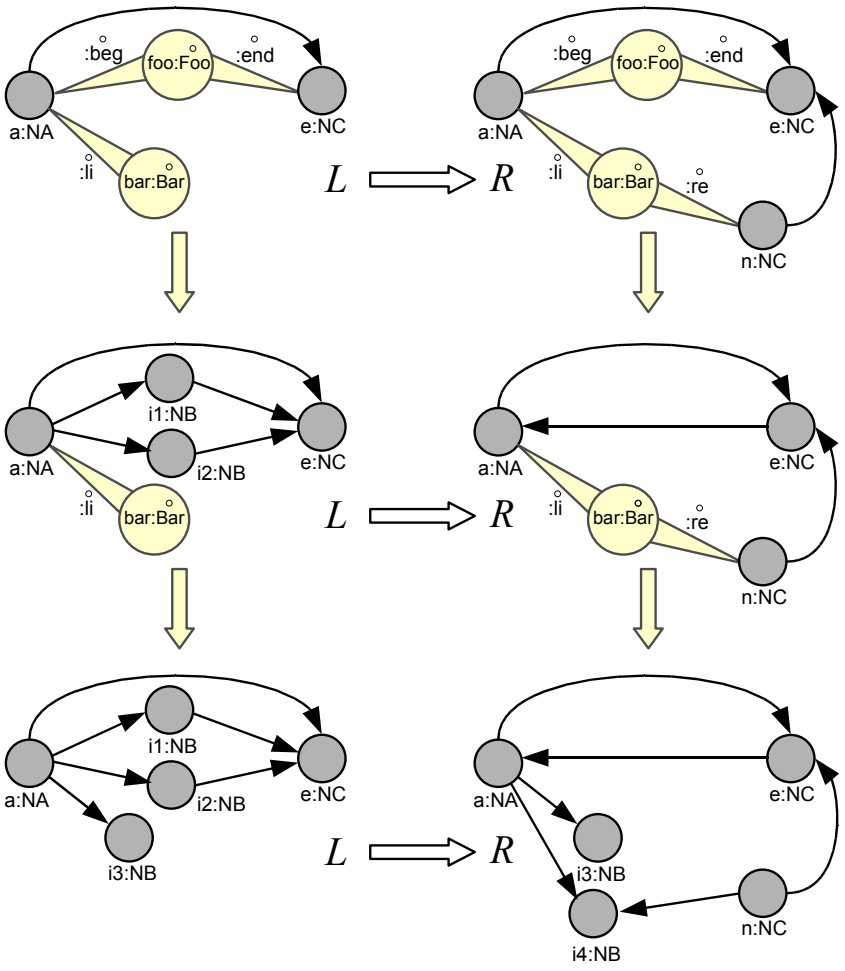


Abbildung 4.6: Teilregelableitung zu Abb. 4.2

Die Teilmusterzusammensetzungsregeln der einzelnen Alternativenzweige werden bei vorhandener abhängiger Ersetzung so wie die Teilmusterzusammensetzungsregeln der Teilmusterspezifikationen mit abhängiger Ersetzung auf das Niveau der Teilregeln und Regelzusammensetzungsregeln angehoben. Zu jeder Alternative entstehen damit so viele Regelzusammensetzungsregeln wie sie Zweige hat, die Auswahl eines Zweiges im Verlauf einer Ableitung legt somit zusammen mit dem Muster auch die Ersetzung fest.

4.5 Erhalten, Löschen und Einfügen

Die bisher beschriebene abhängige Ersetzung erlaubt eine feinkörnige Manipulation der gefundenen Teilmuster, häufig ist aber ein grobschlächtigeres Verhalten ausreichend und angemessen, bei dem gefundene Sterne *als Ganzes* zu *Erhalten*, zu *Löschen* oder *Einzufügen* sind. Dies kann als eine Verallgemeinerung der genannten Graphersetzungsoperationen von Knoten und Kanten auf Teilmuster angesehen werden, die Syntax folgt der bereits bekannten, d.h. in einer `replace`-Ersetzung wird ein nicht aufgeführtes Element aus dem Muster im Arbeitsgraph gelöscht, ein wieder angeführtes Element bleibt erhalten, ein neu eingeführtes Element wird dem Arbeitsgraph hinzugefügt. In einer `modify`-Ersetzung bleibt ein nicht aufgeführtes Element erhalten, wird ein innerhalb von `delete()` angeführtes Element aus dem Arbeitsgraph gelöscht und ein neu eingeführtes Element dem Arbeitsgraph hinzugefügt. Nicht aufgeführt bedeutet, dass der Name aus dem Muster in der Ersetzung nicht verwendet wird, angeführt heißt, dass der Name in der Ersetzung verwendet wird, und eingeführt wird ein Element durch eine Deklaration analog einer Deklaration in einem Muster.

```
1  rule R {
2    foo:Foo();
3    bar:Bar();
4    replace {
5      foo; // bleibt erhalten
6      // bar nicht aufgeführt, wird gelöscht
7      blub:Blub(); // wird neu eingeführt
8      // foo(); -- würde abhängige Ersetzung aufrufen
9    }
10 }
```

Zu beachten ist der Unterschied zwischen `foo` und `foo()`. Ersteres bewirkt ein Erhalten der gepassten Teilmusterentität `foo` vom Teilmustertyp `Foo`, unabhängig davon, ob in `Foo` eine abhängige Ersetzung definiert wurde oder nicht; letzteres hingegen wendet die in `Foo` definierte abhängige Ersetzung (die dann dort auch existieren muss) auf die gefundene Instanz `foo` an.

Beim Einfügen von Teilmustern mit Alternativen werden die Sterne der Alternativen derzeit weggelassen, da nicht klar ist, mit welchem Zweig weiter ausgefaltet werden soll. Dieses Verfahren ist unvollständig, eine Erweiterung, bei der die Auswahl durch den Nutzer vorgenommen wird, habe ich in meiner Studienarbeit [Jak07] vorgestellt, doch war ihre Implementierung im Verlauf dieser Diplomarbeit aus Zeitmangel nicht mehr möglich.

Das Erhalten, Löschen oder Einfügen von Teilmustern wollen wir formal durch nicht von h gepaarte Sterne und eine dritten Phase der Graphersetzung modellieren, die nach dem vollständigen Aussubstituieren der Paarsterne und vor dem Entfernen von N_{del} ausgeführt wird. In ihr werden die verbleibenden, da ungepaarten, einzelnen Sterne auf der linken wie auf der rechten Seite aussubstituiert. Ein einzelner Stern auf der linken Seite ohne Entsprechung auf der rechten Seite, genauer dessen ausgefalteter Graph, wird dadurch schließlich aus dem Arbeitsgraphen gelöscht. Ein einzelner Stern auf der rechten Seite ohne Entsprechung auf der linken Seite, genauer dessen ausgefalteter Graph, wird dadurch schließlich in den Arbeitsgraphen eingefügt. Zum Ausfalten wird bei Teilmustern die Musterzusammensetzungsgrammatik oder bei Teilregeln, d.h. Mustern mit abhängiger Ersetzung, die linke Grammatik der Regelzusammensetzungsgrammatik verwendet.

Beim Erhalten hingegen muss ein Stern der rechten Seite zu einem Graph ausgefaltet werden, der isomorph zum ausgefalteten Graphen des zugehörigen Sterns der linken Seite ist, wobei die isomorphen Elemente zudem im Erhaltungsmorphismus in Beziehung gesetzt sein müssen – wir benötigen hierfür also eine Regelzusammensetzungsgrammatik. Wir nehmen für jedes Teilmuster eine *Erhaltungsregel* in die Regelzusammensetzungsgrammatik auf, die eine abhängige Ersetzung mit leerem `modify` simuliert. Bei vorhandener abhängiger Ersetzung geschieht das zusätzlich zu deren Regelzusammensetzungsregel, dabei gilt für die Erhaltungsregel:

- die rechte Sternersetzungsregel ist eine Kopie der linken Sternersetzungsregel,
- ein terminales Graphelement aus dem Original wird über den Graphelementeteil des Erhaltungsmorphismus r zu seiner Kopie in Beziehung gesetzt,
- alle Sterne aus dem Original werden über den Sternpaarungsteil des Erhaltungsmorphismus r zu ihrer Kopie in Beziehung gesetzt.

Wird nun im Muster einer Regel ein Teilmuster deklariert und in der abhängigen Ersetzung dessen Erhaltung spezifiziert, wird eine entsprechende Sternpaarung in den Erhaltungsmorphismus der Regel aufgenommen und in der Ableitung durch die Erhaltungsregel ausgefaltet.

```
1  pattern ChainReverse(from:Node, to:Node) {
2     alternative {
3        base {
4           from --> to;
5
6           replace {
7              from <-- to;
8           }
9        }
10       rec {
11          from --> intermediate:Node;
12          cr:ChainReverse(intermediate, to);
13
14          replace {
15             from <-- intermediate;
16             cr();
17          }
18       }
19    }
20
21    replace {
22       from; to;
23    }
24 }
```

Abbildung 4.7: Kette umdrehen mit einer rekursiven Regel

4.6 Vergleich mit Graphersetzungssequenzen

Die bereits in früheren GrGen-Versionen eingeführten Graphersetzungssequenzen und die in dieser Arbeit vorgestellten rekursiven Regeln haben unterschiedliche Ziele: Die Graphersetzungssequenzen dienen der Steuerung der Graphersetzung, dem Zusammensetzen der Lösung einer Graphtransformationsaufgabe durch die gesteuerte Abfolge von Lösungen einzelner Teilaufgaben. Die rekursiven Regeln hingegen dienen dem Finden und Umschreiben von rekursiven Strukturen (von Graphen, die durch kontextfreie Hyperkantenersetzungsgrammatiken erzeugt werden können).

Manche der rekursiven Strukturen lassen sich auch mit den Graphersetzungssequenzen finden und umschreiben, wie z.B. eine Kette, deren Kanten es umzudrehen gilt; eine Lösung mit rekursiven Regeln ist in Abb. 4.7 zu sehen, eine entsprechende Lösung mit den GES in Abb. 4.8. Schon bei diesem einfachen Beispiel treten die Vorteile der rekursiven Regeln zu Tage:

```
( Base(from,to) || (from)=Rec(from) )*
```

```
rule Base(from:Node, to:Node)
{
   from --> to;

   replace {
      from <-- to;
   }
}
rule Rec(from:Node) : (Node)
{
   from --> intermediate:Node;

   replace {
      from <-- intermediate;
      return(intermediate);
   }
}
```

Abbildung 4.8: Kette umdrehen mit Graphersetzungssequenzen

Erst umschreiben, wenn die ganze Struktur gefunden wurde

Die GES werden Schritt für Schritt, Kettenglied für Kettenglied ausgeführt. Nach jeder Ausführung wird ein veränderter Arbeitsgraph hinterlassen, in dem die Kette ein Glied weiter umgeschrieben wurde. Die rekursive Regel hingegen wird in einem einzigen, die ganze Kette umfassenden Schritt ausgeführt; erst wird die gesamte Kette gepasst, dann wird sie umgeschrieben. Kann die Passungssuche vom Anfangszustand aus in eine Sackgasse verzweigen befindet sich der Graph nach dem Ausführen der GES in einem fehlerhaften Zustand. Dieser Umstand kann teilweise durch ein Umschließen der Sequenz mit Transaktionsklammern behoben werden, die den Arbeitsgraphen beim Fehlschlag der Sequenz in seinen Ausgangszustand zurückversetzen. Aber zum einen ist dieses Verhalten kostspieliger als den toten Zweig gar nicht erst umzuschreiben, und zum anderen kann mit den Transaktionsklammern zwar zurückgesetzt, aber nicht am letzten Entscheidungspunkt wieder aufgesetzt werden. Da bei mehreren Passungsmöglichkeiten die Passung indeterministisch bestimmt wird, gibt es keine Garantie, dass die Sequenz beim nächsten Versuch nicht wieder in die gleiche Sackgasse läuft. In der Passungssuche der rekursiven Regeln hingegen ist das Wiederaufsetzen am letzten Entscheidungspunkt eingebaut, so dass hier der Suchraum systematisch durchlaufen wird. Dies ermöglicht es einem auch, sich mit `[?ChainReverse]` sämtliche Passungen zurückgeben zu lassen, was bei der Sequenzenlösung nicht möglich ist, da die Allklammerung `[SomeRule]` nur

für einzelne Regeln und nicht für Sequenzen definiert ist; der Fragezeichen-Operator `?` vor einer Regel stuft diese zu einer Prüfung herab.

Isomorphe Passung der Teilmuster

Die Knoten und Kanten der Muster der in den GES verwandten Regeln werden unabhängig voneinander gepasst. Die Knoten und Kanten der Teilmuster der rekursiven Regeln hingegen werden isomorph zueinander gepasst. Wird die eingeführte Kette nur im Arbeitsgraphen gesucht (ohne selbigen umzuschreiben), kann die Unabhängigkeit der Passungen der einzelnen Regelanwendungen der Graphersetzungssequenz dazu führen, dass bei einem Arbeitsgraph, der eine Kette mit einer Schleife enthält, die Schleife unbestimmt oft erneut gepasst wird oder sogar die Sequenz nicht terminiert.

Multiple Teilmuster

Verzweigend zusammengesetzte Muster unbeschränkter Tiefe schließlich, wie dem Binärbaum in Abbildung 4.9, dessen Blätter alle mit einem gemeinsamen Knoten verbunden werden, kann man nicht mehr durch Graphersetzungssequenzen mit Parameterübergabe verarbeiten. Stattdessen müsste man eine Regel iterieren, die die Anwendungsstelle(n) im Arbeitsgraphen markiert (Einsatz von Schiffchen im Arbeitsgraphen), bis sie nicht mehr anwendbar ist.

Das Beispiel in Abbildung 4.10 mit seinen verzweigenden Strukturen unbeschränkter Breite und unbeschränkter Tiefe müsste man ebenso über das Propagieren von Schiffchen durch den Arbeitsgraphen behandeln, denn für jeden Schritt in die Tiefe würde man eine geschachtelte Iteration mehr benötigen, was bei Tiefe 3 schon zu folgender abstrakter Struktur des GES führen würde:

```
1 ( (o1)=Bla(o0) && ( (o2)=Bla(o1) && ( Bla(o2) )* )* )*
```

Prägnanz

Selbst wenn man nur die noch über die Sequenzen direkt handhabbaren Listenstrukturen bearbeitet und das vorzeitige Umschreiben wie auch das unabhängige Passen der Einzelstücke keine Probleme bereiten, wird man das Passen komplexer Strukturen nicht über die GES ausprogrammieren wollen – die Sequenzen werden nämlich umfangreich, wie an der Sequenz in Abbildung 4.11 zu dem Beispiel Transkription A.3 auf Seite 97 zu sehen ist. Die einzelnen Regeln sind recht simpel, ihre operationale Steuerung jedoch ist komplex und durch die Seiteneffekte jeder Regelanwendung auf den Arbeitsgraphen und die Parameter schwierig zu überblicken. Das gilt insbesondere

```
1  pattern BinTree(from:NA)
2  {
3    alternative {
4      Base {
5        negative {
6          from --> child1:NA;
7          from --> child2:NA;
8        }
9
10       modify(common:NA) {
11         from --> common;
12       }
13     }
14     Recursive {
15       from --> child1:NA;
16       from --> child2:NA
17       left:BinTree(child1);
18       right:BinTree(child2);
19
20       modify(common:NA) {
21         left(common);
22         right(common);
23       }
24     }
25   }
26
27   modify(common:NA) {
28   }
29 }
```

Abbildung 4.9: Binärbaum

```
pattern MultipleBla(n:Node)
{
   alternative
   {
      OneAndAgain {
         n --> offspring:Node;
         :MultipleBla(offspring);
         :MultipleBla(n);
      }
      NoBlaLeft {
         negative {
            n --> :Node;
         }
      }
   }
}
```

Abbildung 4.10: Verzweigende Struktur unbeschränkter Breite und Tiefe

```
<
 (prev,rprev) = TATABox
 && ( !TerminationSequence(prev)
     && (prev,rprev)=NextChainElement(prev,rprev)
     && (A(prev,rprev) || C(prev,rprev) || G(prev,rprev) || T(prev,rprev))
    )*
 && TerminationSequence(prev)
>
```

Abbildung 4.11: GES zur Transkription

dann, wenn die Lösungen solcher Teilaufgaben auch noch zu größeren Programmen kombiniert werden müssen, dem eigentlichen Aufgabengebiet der GES.

Fazit

Das Bearbeiten rekursiv strukturierter Graphen mit einfachen Regeln und einer *programmierten Steuerung des Such- und Umschreibeablaufs* (wie durch die Graphersetzungssequenzen) ist zwar prinzipiell möglich, aber unangemessen. Besser geeignet sind die hier vorgestellten rekursiven Regeln mit ihrer *deklarativen Beschreibung des Musteraufbaus und der davon abhängigen Ersetzung.*

4.7 Verwandte Arbeiten

Meinem Wissen nach ist GrGen das einzige Graphersetzungswerkzeug, das zusammengesetzte Regeln und damit Paargraphgrammatiken implementiert. In Viatra 2 scheinen sie angedacht [BV06], aber nie ausformuliert oder implementiert worden zu sein.

Kapitel 5

Implementierung von GrGen.NET

In diesem Kapitel beschreibe ich den Vorgang der Codegenerierung für die Passungssuchprogramme und die Passungssuche durch den generierten Code. Es bildet die Grundlage für die beiden folgenden Kapitel, in denen ich zuerst die Erweiterung des generierten Codes (d.h. der Passungssuchprogramme und der Ersetzungsprogramme) und dann die Erweiterungen am Codegenerator beschreiben werde.

5.1 Vorgang der Codegenerierung

Der Grundaufbau des GrGen.NET-Graphersetzungssystems wurde bereits in Abschnitt 2.6 kurz beschrieben und in Abbildung 2.10 skizziert, so dass ich hier nur darauf verweisen möchte. Im Folgenden werde ich vertiefend auf den Graphersetzungsregelgenerator von GrGen eingehen, getrennt nach Frontend und Backend.

5.1.1 Frontend des Generators

Der Vorgang der Spezifikationsverarbeitung im Frontend ist in der Grafik 5.1 dargestellt. Als Erstes werden die Graphmodell- und Regelspezifikationsdateien von einem Lexer und einem Parser analysiert und in einen abstrakten Syntaxbaum überführt. Lexer und Parser werden durch ANTLR2 aus einer Spezifikation der GrGen-Graphmodell- und Regelspezifikationssprache generiert, der AST-Aufbau erfolgt durch, an die Grammatikregeln annotierten Code. Es folgt eine ausprogrammierte Semantische Analyse bestehend aus einer Resolverphase zum Auflösen der Namen zu Zeigern auf ihr jeweiliges Deklarationsobjekt und einer anschließenden Checkerphase zum Durchführen der semantischen Prüfungen. AST u. und AST r. stehen für AST unresolved und AST resolved. Waren die Prüfungen erfolgreich, wird

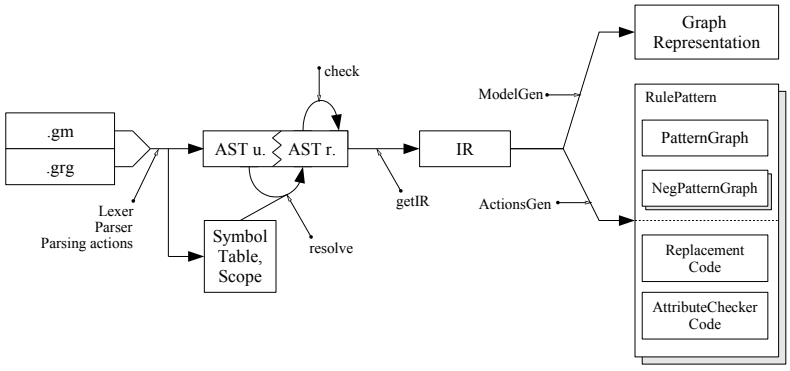


Abbildung 5.1: Ablauf der Spezifikationsverarbeitung im Frontend von GrGen

der AST in eine Zwischensprache umgewandelt, aus der heraus dann C#-Code generiert wird. Dabei wird für jede Regel eine Klasse, die das Muster der Regel beschreibt, erzeugt. Diese Regel- und Musterrepräsentation dient zum einen dem C# Backend als Ausgangspunkt der dort erfolgenden Generierung des Passungsfindungscodes, kann also als weitere Zwischensprache angesehen werden; zum anderen steht sie dem LibGr-Nutzer als Metainformation zur Verfügung. Außerdem wird im Frontend für jede Regel der Code zum Durchführen der Ersetzung erzeugt.

5.1.2 Backend des Generators

Der Vorgang der Codegenerierung im LGSP-Backend, der in der Grafik 5.2 dargestellt ist, läuft wie folgt ab: Durch das Java-Frontend wurde zu jeder Regel ein Regelmuster-Objekt erzeugt, bestehend aus einem Mustergraphen für das Muster selbst und einem Mustergraphen für jedes NAC. In den Mustergraphen werden die Knoten und Kanten des Musters aufgeführt sowie weitere Bedingungen angegeben.

Aus den Mustergraphen werden nun im C#-Backend Planungsgraphen aufgebaut. Die Knotenmenge eines Planungsgraphen besteht aus einem Wurzelknoten sowie den zu suchenden Elementen des Musters, seine Kanten sind die dazu anwendbaren Suchoperationen, versehen mit abgeschätzten Kosten. Im Anschluss wird zu jedem Planungsgraph ein minimaler spannender Arboreszent (knotenüberdeckender Baum aus gerichteten Kanten mit minimalen Kosten) bestimmt und in Form eines Suchplangraphen abgespeichert.

Die Bäume der Suchplangraphen werden dann im Suchplan, einer Folge von Suchoperationen, linearisiert und zusammengeführt; außerdem werden weitere Bedingungen zur Überprüfung eingeplant, das ganze wieder unter heuristischer Minimierung der Kosten des Suchplans.

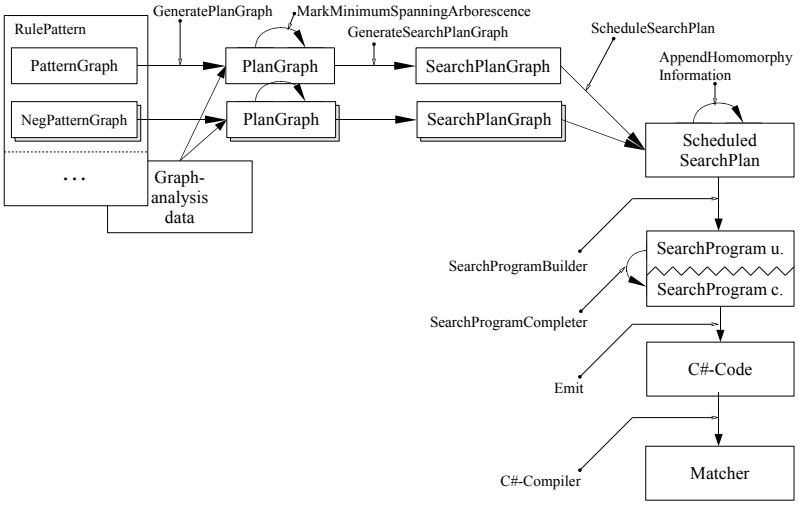


Abbildung 5.2: Ablauf der Suchplanung und Codeerzeugung im Backend von GrGen

Aus dem Suchplan wird ein Suchprogramm erzeugt, in dem die Suchplanoperationen feiner untergliedert werden, und das mit seiner Baumstruktur die Schachtelungsstruktur des Codes direkt widerspiegelt. Aus dem Suchprogramm wird schließlich C#-Code erzeugt, der zu guter Letzt übersetzt und in einer Action-Klasse innerhalb einer .NET-Assembly dem Nutzer zur Ausführung angeboten wird.

Die Planung der Suche, durch die die Reihenfolge bestimmt wird, in der die Musterelemente zu suchen sind, wird in [Bat06] und [BKG08] umfassend dargestellt, der Aufbau des GrGen.NET-Systems und der Vorgang der Codegenerierung werden in [Kro07] weitergehend erläutert und Aussagen zum danach eingeführten Suchprogramm finden sich in meiner Studienarbeit [Jak07].

5.2 Passungsvorgang und Ersetzung

Das generierte Programm führt eine *Ansatzerweiterung im Rücksetzverfahren* durch. Ich beginne dessen Erläuterung mit dem einfachsten Verfahren überhaupt, Versuch und Irrtum: In jedem Schritt wird an ein Musterelement ein beliebiges, noch nicht betrachtetes Element aus dem Graphen gebunden, und wenn dann am Ende alle Musterelemente einen Partner haben, wird überprüft, ob sie wirklich mit den Typen und der Struktur des Musters übereinstimmen. Tun sie es nicht, wird der letzte Schritt rückgängig gemacht und das nächste Element wird ausprobiert. Sind die möglichen Kandida-

ten des letzten Schrittes erschöpft, wird zum vorherigen zurückgesetzt und dort ein anderes Element ausprobiert, und so weiter, bis alle Möglichkeiten durchprobiert wurden. Dieses Vorgehen, bei dem zuerst alle Musterelemente Kandidaten aus dem Arbeitsgraphen erhalten und diese dann im Anschluss daraufhin überprüft werden, ob sie mit dem Muster übereinstimmen, ist ineffizient, da der baumartige Suchraum in seiner Gänze, jeder Zweig bis zu seiner maximalen Tiefe, durchlaufen wird.

Im derzeit generierten Verfahren wird deshalb sofort bei der Bindung eines Arbeitsgraphelementes an ein Musterelement geprüft, ob es sich überhaupt in die bereits gefundene partielle Passung integrieren lässt. Es wird also weiterhin eine Suche mit Rücksetzen zum vorherigen Entscheidungspunkt und der Neuauswahl eines Elementes durchgeführt – aber nur für einen dabei gefundenen Teilgraphen des Arbeitsgraphen, der mit dem Muster übereinstimmt, Ansatz genannt, wird der folgende Auswahlschritt ausgeführt. Der durchlaufene Zustandsraum dieser Suche hat ebenso Baumstruktur, besteht aber alleine aus den möglichen partiellen Passungen; Zweige in unpassende Erweiterungen werden abgeschnitten. Implementierungsnäher besehen sind zusätzlich zu den partiellen Passungen im Suchraum noch die Erweiterungen mit dem nächsten unpassenden Element enthalten – der Kandidat muss schließlich zuerst gewählt werden bevor er überprüft werden kann.

Der aktuelle Suchzustand ist der Pfad von der Wurzel bis zum aktuellen Zustandsknoten. Ein Schritt in die Tiefe entspricht einer Erweiterung des Musters, ein Schritt in der Breite entspricht der Neuauswahl eines Arbeitsgraphelementes.

Genauer betrachtet besteht die das Verfahren implementierende Suchprozedur aus einer Schachtelung von Schleifen, wobei in jeder die zu einem Musterelement in Frage kommenden Arbeitsgraphelemente durchprobiert werden. Die Entscheidungspunkte sind die Schleifen, der Kandidat befindet sich in der Iterationsvariable der aktuellen Schleife. Das Rücksetzen erfolgt durch das Verlassen der aktuellen Schleife und das Ausführen der nächsten Iteration der umschließenden Schleife. Der aktuelle Zweig des Suchbaumes mit seinen Zustandsknoten der partiellen Passungen ist somit direkt im Code repräsentiert, durch die Position des Befehlszeigers und den Zustand der lokalen Variablen der Suchprozedur bis zu dieser Position hin. Seine maximale Tiefe erlangt er beim Erreichen der innersten Schachtel. Für große Muster wird das Suchprogramm somit durchaus umfangreich und tief verschachtelt.

Passungsobjekt und Ersetzung

Wurde durch den generierten Suchcode eine Passung gefunden, wird ein Passungsobjekt erzeugt, mit den nun bestätigten Kandidaten aus den lokalen Variablen gefüllt, und in eine Liste von Passungsobjekten einfügt. Ein Passungsobjekt besteht aus einer Reihung von Knoten und einer Reihung von Kanten, die Zuordnung zwischen Musterelement und Reihungsindex erfolgt

durch generierte Aufzählungen, die zu einem Musterelementnamen seinen Index festlegen. Negative Teilmuster kommen nicht vor, da ihr Vorhandensein gerade die Passung verhindert.

Die Ersetzung wird durch einen Aufruf des vom Frontend erzeugten Ersetzungscodes ausgeführt, wobei die Passung durch Übergabe des Passungsobjektes festgelegt wird.

Kapitel 6

Erweiterungen im generierten Code

In diesem Kapitel beschreibe ich, wie die in den Kapiteln 3 und 4 eingeführten Erweiterungen im generierten Code umgesetzt werden, getrennt in die Passungssuche und in die Ersetzung, mit dem erweiterten Passungsobjekt als Schnittstelle dazwischen.

Die Haupterkenntnis besteht darin, dass die Passungssuche (selbstverständlich) nicht im Stil der abstrakten Semantik erfolgt, bei der zuerst alle Muster und Regeln – möglicherweise sogar unendlich viele – über ihre Zusammensetzungsgrammatik erzeugt und dann auf den Arbeitsgraphen angewendet werden, sondern der Vorgang der Musterzusammensetzung mit der Passungssuche *verwoben* ist. Dabei werden von jedem (Teil-)Muster zuerst die terminalen Bestandteile zu passen versucht, bevor mit dem Ausfalten der geschachtelten Teilmuster fortgefahren wird; genauer gesagt werden zuerst die terminalen Bestandteile gepasst, dann die negativen Muster zu passen versucht (zuerst terminale, dann nichtterminale Bestandteile), und erst wenn letzteres nicht erfolgreich war, werden die nichtterminalen Bestandteile des positiven Musters gepasst.

6.1 Passungsobjekt und Ersetzung

Um die Passung von eingebetteten Mustern zurückgeben zu können, wurden die flachen Passungsobjekte mit ihren Knoten- und Kantenreihungen um eine Reihung für eingebettete Passungsobjekte erweitert, womit die Passungsobjekte (siehe A.2) nun in einer hierarchischen Baumstruktur die Musterzusammensetzung widerspiegeln. Jedem Teilmuster wird statisch ein Platz in der Reihung zugewiesen, an dem sich nach erfolgter Passung das gefundene Passungsobjekt befindet. Die alternativen Muster werden ähnlich den eingebetteten Mustern behandelt: Für die gesamte Alternative wird statisch ein Platz in der Reihung reserviert, der dann dynamisch mit dem Passungs-

objekt des Teilmusters des gewählten Alternativenzweiges gefüllt wird. Zur Identifikation welcher Zweig gewählt wurde, verweist jedes Passungsobjekt zusätzlich auf die Repräsentation seines zugehörigen Teilmusters.

Erweiterung der Ersetzungsdurchführung

Die Ersetzung wird während eines rekursiven Laufes über die Schachtelstruktur des Passungsobjektes durchgeführt, als Kombination der Ersetzungen der Teilregeln.

Hierzu werden zu jeder Teilregel die Ersetzungsfunktionen für ihre abhängige Ersetzung und zu jedem Teilmuster die Ersetzungsfunktionen für das Erzeugen und Löschen dieses Teilmusters generiert. Anhand der spezifizierten Teilmusterdeklarationen und Teilmusterersetzungsnutzungen werden die generierten Ersetzungsfunktionen aus der umschließenden Regel heraus aufgerufen, bei Alternativen wird anhand des gepassten Zweiges die passende Ersetzungsfunktion ausgewählt.

Die Veränderungen der geschachtelten Muster werden vor den Veränderungen der sie enthaltenden Muster ausgeführt, also beim Aufstieg im Passungsbaum, weil angeknüpfte Elemente im Muster, in dem sie deklariert wurden, gelöscht oder verändert werden können und dann den Ersetzungsfunktionen der geschachtelten Mustern nicht mehr zur Verfügung stünden.

6.2 Wiederverwendung der Passungssuche

Wie bereits in Kapitel 5 beschrieben, werden die Passungen durch *Ansatzerweiterung im Rücksetzverfahren* bestimmt, in einer generierten Suchprozedur aus geschachtelten Schleifen, in denen Kandidaten für Bindungen von Arbeitsgraphelementen an Musterelemente bestimmt und überprüft werden.

Das Generieren eines Suchprogramms zu einem Muster, mit geschachtelten Schleifen zum Finden der Knoten und Kanten, hat sich bewährt. Wir wollen diese Technik auch auf die Teilmuster anwenden.

Die Generierung eines einzigen Suchprogramms durch vollständiges Aussubstituieren der Teilmuster in das Gesamtmuster wäre für nichtrekursive Muster noch prinzipiell möglich, wenn auch wegen des Codeumfangs nicht unbedingt empfehlenswert. Für die unbeschränkt großen rekursiven Muster hingegen müsste – bei einem unbekannten Arbeitsgraphen – ein unendlich großes Suchprogramm generiert werden. Bei einem bekannten Arbeitsgraphen immerhin könnte das rekursive Aufzählen der Mustergraphen und damit das Generieren der Suchprogramme abgebrochen werden, wenn das Muster die im Arbeitsgraphen vorhandene Anzahl von Knoten und Kanten (modulo Homomorphie) erreicht. Doch ist dieses statische, im voraus Zusammensetzen völlig ineffizient, außerdem wollen wir Suchprogramme ohne Kenntnis des Arbeitsgraphen generieren können, so dass wir die Forderung

nach einem dynamischen Zusammensetzen während der Suche, entlang dessen, was tatsächlich im Arbeitsgraphen vorhanden ist, stellen müssen.

Wir wollen somit für die einzelnen Teilmuster Suchprogramme generieren und das Enthaltensein in einem umgebenden Muster durch Aufrufe abbilden. Bei diesem Vorgehen treten zwei wesentliche Fragen im Entwurf auf:

Kombination von Aufrufen: Wo und wann erfolgen die Aufrufe zum Durchwandern des Suchraumes?

Kombination von Teilpassungen: Wo und wann werden die Passungsobjekte erzeugt und zusammengesetzt?

Hierzu werde ich im folgenden Abschnitt den Entwurf aus meiner Studienarbeit rekapitulieren, den ich im Rahmen dieser Diplomarbeit implementiert habe.

6.3 Erweiterung der Passungssuche

Das Hauptproblem bei der Suche nach zusammengesetzten Mustern, die Diskrepanz zwischen der Struktur des Suchvorganges und der Struktur der Muster und damit der Passungsobjekte, wird durch das Linearisieren der Suche auf einen Zweig des Suchzustandsbaumes gelöst; die partiell gefundene Passung wird vollständig auf dem Aufrufkeller vorgehalten, auch beim Suchen nach mehrfachen, verzweigenden Teilmustern.

Das wird dadurch erreicht, dass bei einem aus mehreren Teilmustern zusammengesetzten Muster das Passungssuchprogramm des Gesamtmusters nur das Passungssuchprogramm des ersten enthaltenen Teilmusters direkt aufruft, diesem aber dafür als Auftrag mitgibt, anstelle seiner mit den noch offenen Teilmustern fortzufahren. Mit dieser Fortsetzungsweitergabe wird eine verzweigende Passung auf einen Zweig des Suchbaumes linearisiert, dadurch kann der gesamte Suchzustand in den lokalen Variablen der Suchprogrammschachteln auf dem Aufrufkeller vorgehalten werden – fast der gesamte Suchzustand: auf dem Stapel liegt die bisher gefundene partielle Passung, doch die noch offenen Aufträge müssen ebenfalls gemerkt werden. Auf dem Aufrufkeller ist das nicht möglich, da die noch folgenden Funktionen, die die Aufträge zu bearbeiten haben, keinen Zugriff auf diesen haben. Und da die Anzahl der offenen Aufträge mit jedem Abstieg in ein verzweigendes Teilmuster wächst, reicht eine Variable fester Größe nicht aus. Wir wollen das am tiefsten geschachtelte Teilmuster – dessen Auftrag – zuerst behandeln, deshalb bietet sich ein Auftragskeller an.

Der durch das Rücksetzverfahren zu verwaltende Suchzustand besteht somit zusätzlich zum Aufrufkeller mit der gefundenen partiellen Passung aus dem Auftragskeller mit den noch abzuarbeitenden Teilmustern. Er wird in den Passungsfunktionen folgendermaßen verwaltet:

1. Wird eine Passungsfunktion aufgerufen, entfernt sie zuerst ihren Auftrag vom Keller und fängt dann mit der Suche nach ihrem lokalen Muster an.

2. Hat sie erfolgreich ihre lokalen Knoten und Kanten gefunden,
   a) legt sie Aufträge für alle ihre Teilmuster auf dem Auftragskeller ab und
   b) ruft die Passungsfunktion zum obersten Auftrag des Kellers auf. Wenn sie keine Teilmuster besitzt werden einfach keine Aufträge erteilt, dann wird nach dem Fund ihrer lokalen Elemente ein früherer, von einem umschließenden Muster erteilter Auftrag abgearbeitet.

3. Kehrt der Aufruf zum obersten Auftrag zurück, nimmt sie die abgelegten Aufträge wieder vom Keller (diese wurden zwischenzeitlich entfernt, aber jede Passungsfunktion fügt ihren Auftrag vor der Rückkehr wieder ein). Dann wird lokal weitergesucht, Verhalten wie 2., d.h. Aufträge ablegen

4. Haben sich die Entscheidungspunkte der lokalen Suche erschöpft, kehrt die Passungsfunktion zu ihrem Aufrufer zurück; davor legt sie wieder ihren ursprünglichen Auftrag auf dem Keller ab.

Das Zurückschreiben des eigenen Auftrags auf den Keller vor der Rückkehr ist notwendig, damit beim Rücksetzen keine Aufträge von umschließenden Mustern, die nicht dem direkten Aufrufer entstammen, verloren gehen. Das Entfernen der Aufträge und wiederablegen in 3. ist eigentlich nur bei der Enumeration von Alternativen notwendig, bei diesen können sich die Aufträge ändern. Die Passung wurde gefunden, wenn der Auftragskeller bei 2.b) leer ist; sämtliche bei der Suche entstandenen Aufträge liegen zu dem Zeitpunkt bearbeitet auf dem Aufrufkeller vor.

Damit ist der Suchvorgang abgedeckt, über das Zusammensetzen der Passungsobjekte müssen wir uns aber noch weitergehendere Gedanken machen, weil ein Suchprogramm keinen direkten Zugriff auf die Passungsobjekte seiner Teilmuster hat, und weil unser Ziel darin besteht, einen Passungsbaum erst dann auf der Halde anzulegen, wenn die Passung vollständig gefunden wurde. Es gilt: Wenn die Passung gefunden wurde, befindet sie sich, zwar in ihren Einzelteilen, aber doch vollständig, auf dem Aufrufkeller.

Ein Problem ist nun, dass das Suchprogramm, welches die Passung gerade vervollständigt hat, nicht auf den Aufrufkeller zugreifen kann, um sie auf der Halde zu replizieren. Wir lösen es, indem wir nach dem Finden einer Passung, beim Wiederaufstieg aus den Teilmustersuchprogrammen, bei jedem Aufstiegsschritt den gefunden Teil auf der Halde auskristallisieren.

Ein anderes Problem lautet: Wie übergeben wir dem Suchprogramm die Passungsobjekte seiner Teilmuster zum Zusammenbauen des eigenen Passungsobjektes? Eine Rückgabe über den Aufrufkeller bietet sich nicht an,

weil das aufgerufene Suchprogramm nicht weiß, was für Rückgabewerte von seinem Aufrufer erwartet werden. Es weiß nur lokal, welche Rückgabewerte es selbst erwartet (ein Teilmuster kann in mehreren Mustern vorkommen, sein Suchprogramm damit von unterschiedlichen Suchprogrammen aus aufgerufen werden, selbst wenn man von den Problemen durch die Fortsetzungsweitergabe absieht). Die Lösung besteht im Einführen eines Rückgabekellers von Passungsobjekten. Jedes Suchprogramm entnimmt so viele Passungsobjekte wie es Teilmuster besitzt, baut aus diesen plus den lokalen Knoten und Kanten sein eigenes Passungsobjekt auf und legt es hernach auf dem Rückgabekeller ab.

Eine gefundene Passung befindet sich während des Suchvorganges zum einen Teil unfixiert in den lokalen Variablen auf dem Aufrufkeller, zum anderen Teil bereits in einem Passungsteilbaum fixiert auf der Halde, verwiesen durch die Einträge auf dem Rückgabekeller. Beim Erreichen des Gesamtmusters schließlich wird das Passungsobjekt des Gesamtmusters zusammengesetzt, danach ist der Rückgabekeller wieder leer. Ist der Rückgabekeller vorher leer, bedeutet das, dass mit dem auf dem Aufrufkeller befindlichen Zustand keine Passung gefunden wurde, die Suche wird dann mit dem nächsten Kandidaten fortgesetzt.

Das oben skizzierte gilt wenn genau eine Passung gefunden werden soll – aber was ist, wenn mehr als eine Passung, oder gar alle vorhandenen gewünscht werden? In dem Fall werden so viele Rückgabekeller benötigt wie Passungen von einem Zustand des top-level-Musters aus vorhanden sind. Wir führen eine Liste mit Rückgabekellern ein; immer dann, wenn eine vollständige Passung gefunden wird, wird ein Keller erzeugt und mit dem Passungsobjekt der passungsabschließenden Suchfunktion initial belegt.

Bei der Rückkehr in eine Passungsfunktion eines Teilmusters wird folgendermaßen vorgegangen:

- Existiert kein Rückgabekeller, geht die Suche mit dem nächsten Kandidaten weiter.
- Gibt es weniger Rückgabekeller als gewünschte Passungen, wird der aktuelle Kellerinhalt auskristallisiert, dann geht die Suche mit dem nächsten Kandidaten weiter.
- Gibt es so viele Rückgabekeller wie gewünschte Passungen, wird der aktuelle Kellerinhalt auskristallisiert, dann wird zum Aufrufer zurückgekehrt.

Den Kellerinhalt auskristallisieren bedeutet bei mehreren Rückgabekellern, das lokale Passungsobjekt erzeugen und mit den Knoten und Kanten füllen, dann für jeden Rückgabekeller einzeln die benötigten Teilmusterpassungen entnehmen, mit dem Passungsobjekt verbinden, und auf dem jeweiligen Rückgabekeller ablegen. Man könnte auch für jeden Keller einzeln ein eigenes lokales Passungsobjekt erzeugen, was aber ohne weiteren Nutzen nur unnötig Speicher kosten würde. Dieses Vorgehen bereitet keine Probleme,

weil die bereits auskristallisierten Teile der Passungen alle zum gleichen Zustand der aktuellen Funktion und ihrer Vorgänger auf dem Aufrufkeller, also der gleichen partiellen Passung, gefunden wurden.

Der Suchzustand nach dem Finden mehrere Passungen, in einer aktuellen Suchfunktion in der Mitte eines Suchraumzweiges, besteht aus

- dem gemeinsamen Teil der bisher gefundenen Passungen auf dem Aufrufkeller, der auch speichereffizient in identische Passungsobjekte auskristallisiert werden wird, für alle bereits gefundenen Passungen und alle weiteren, die noch von dieser Funktion aus gefunden werden,
- dem Zustand der aktuellen Suchfunktion in der obersten Kellerschachtel,
- den noch abzuarbeitenden Aufträgen auf dem Aufgabekeller,
- und dem Zustand der bereits gefundenen Passungen auf den Rückgabekellern, mit dem Passungsobjekt des lokalen Objektes zuoberst für die bereits gefundenen Passungen, zusammengesetzt aus den da aktuellen Werten der lokalen Variablen und den auf dem Rückgabekeller gelieferten Passungen der Teilmuster.

Zum besseren Verständnis habe ich am Ende des Kapitels eine Abfolge von Situationen der 3-Keller-Maschine beim Passen eines Beispielmusters skizziert, beginnend mit der leeren Maschine in Abbildung 6.1, endend mit der gefundenen Passung in Abbildung 6.12. Vom Beispielmuster wird nur die Schachtelstruktur der Teilmuster gezeigt, da ihr terminaler Inhalt für die Steuerung der 3-Keller-Maschine nicht von Bedeutung ist. Der Arbeitsgraph ist isomorph zum Beispielmuster mit der Alternative D2.

6.4 Behandlung Isomorphie

Die Prüfung der Elemente eines Musters auf Isomorphie erfolgt durch Lesen und Schreiben von „Ist bereits gebunden worden“-Bits, die in den *Arbeitsgraphelementen* in einem `flags`-Bitvektor enthalten sind. Isomorphieprüfungen sind damit billig möglich, Prüfungen auf homomorphes Zusammenfallen müssen zusätzlich gegen die bereits zu den Musterelementen gepassten Graphelementen in den lokalen Variablen verglichen werden.

Die Prüfung auf globale Isomorphie zwischen den Elementen der Teilmuster wird über ein `IS_MATCHED_BY_ENCLOSING_PATTERN`-Bit bewerkstelligt. Das Bit wird geschrieben, nachdem das lokale Muster vollständig gefunden wurde und bevor der Aufruf der Suchfunktion für das folgende, zu verarbeitende Muster erfolgt; nach der Rückkehr des Aufrufes ist es dann entsprechend zu entfernen.

Die Prüfung auf lokale Isomorphie zwischen den Elementen innerhalb eines (Teil-)Musters erfolgt durch ein `IS_MATCHED`-Bit für das positive (Teil-)-Muster und durch weitere Bits für geschachtelte negative Muster, da ja jedes negative Muster unabhängig von seinen umgebenden Mustern ist und somit

einen eigenen Isomorphieprüfungsraum besitzt. Die aktuelle Schachtelungstiefe und damit zu prüfende Position innerhalb des Bitvektors wird in der Variablen `negLevel` mitgezählt, beginnend mit 0 für das positive Gesamtmuster aus der Regel; sie wird mit Betreten eines negativen Musters um eins erhöht und mit Verlassen eines negativen Musters um eins erniedrigt. Ein positives Teilmuster kann während des Passungsvorganges durchaus mit einem höheren `negLevel` als 0 gepasst werden, nämlich dann, wenn es als Teilmuster in einem negativen Muster verwendet wird. Wurde ein Kandidat akzeptiert, wird in seinem Bitvektor das `IS_MATCHED`-Bit der aktuellen Tiefe gesetzt, beim Verwerfen des Kandidaten ist es wieder zu entfernen. Die Teilmustertiefe, die über die Bits behandelt werden kann ist beschränkt, wenn `negLevel` das `MAX_NEG_LEVEL` übersteigt, werden die Homomorphieinformationen dieser Tiefen in C#-Dictionaries verwaltet.

Jeder Kandidat des lokalen Musters muss gegen das *globale* Bit geprüft werden, ist es nicht gesetzt, wird fortgefahren, ist es gesetzt, wird er verworfen. Jeder Kandidat des lokalen Musters muss gegen das *lokale* Bit geprüft werden, ist es nicht gesetzt, wird fortgefahren, ist es gesetzt und kann er mit anderen Musterelementen zusammenfallen, muss auf korrektes Zusammenfallen geprüft werden, kann er nicht zusammenfallen, wird er verworfen.

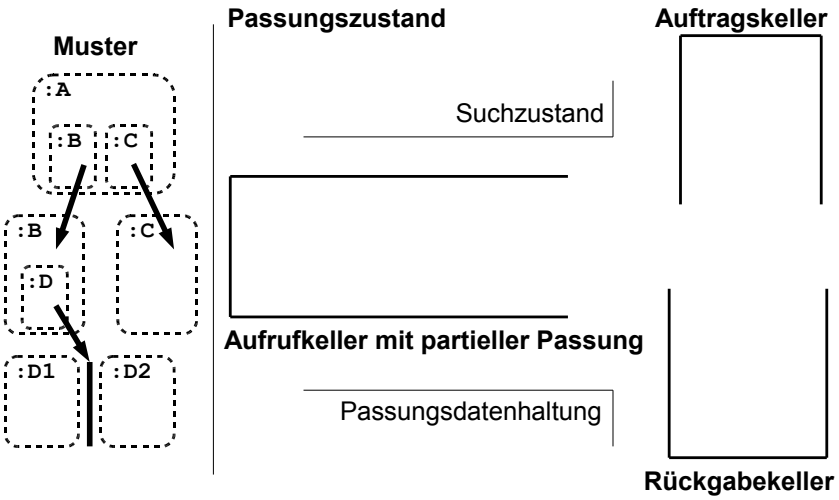


Abbildung 6.1: 1. Startzustand

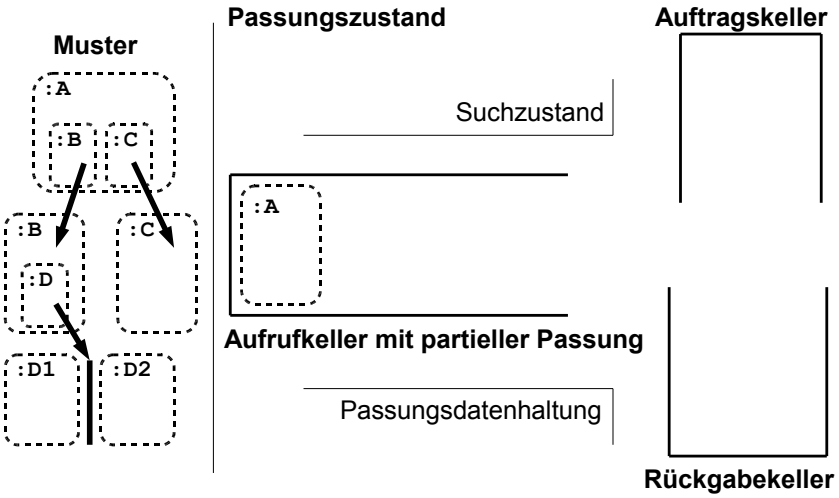


Abbildung 6.2: 2. Der terminale Teil des Gesamtmusters A wurde gepasst

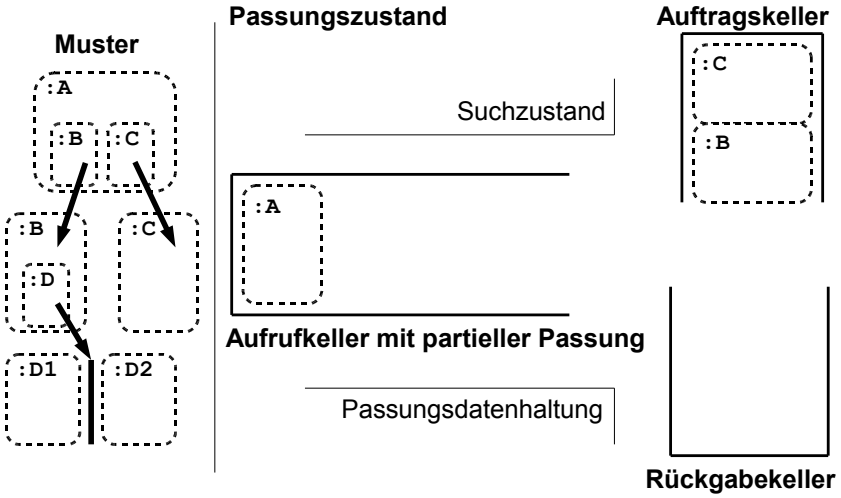


Abbildung 6.3: 3. Die Aufträge für die Teilmuster B und C werden abgelegt

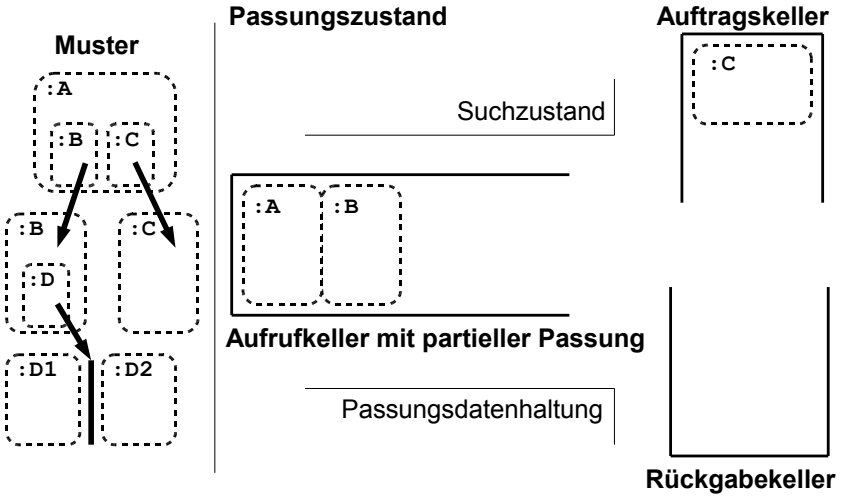


Abbildung 6.4: 4. Der Auftrag für B wird abgearbeitet, der terminale Teil von B wurde gepasst

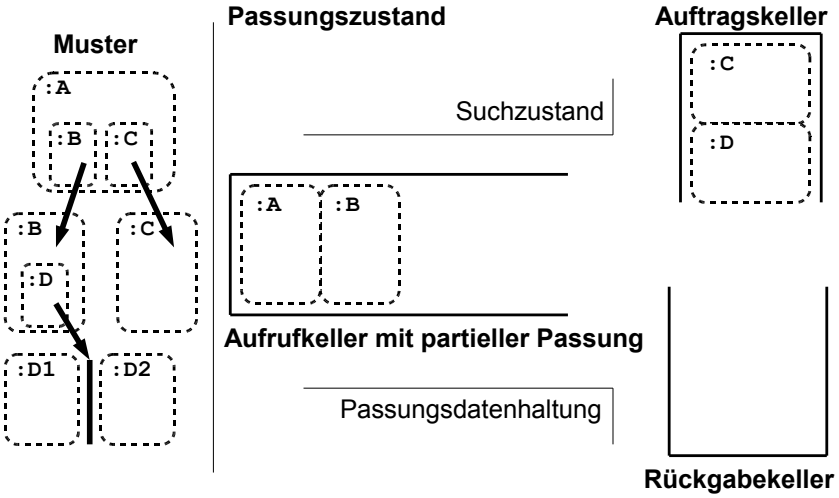


Abbildung 6.5: 5. Der Auftrag für die Alternative D wird abgelegt

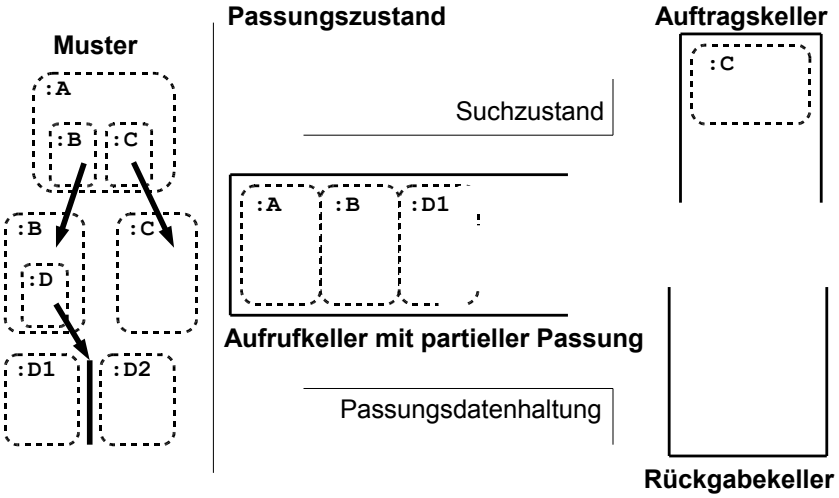


Abbildung 6.6: 6. Der Auftrag für D wird abgearbeitet, D1 wird ausprobiert, schlägt aber fehl

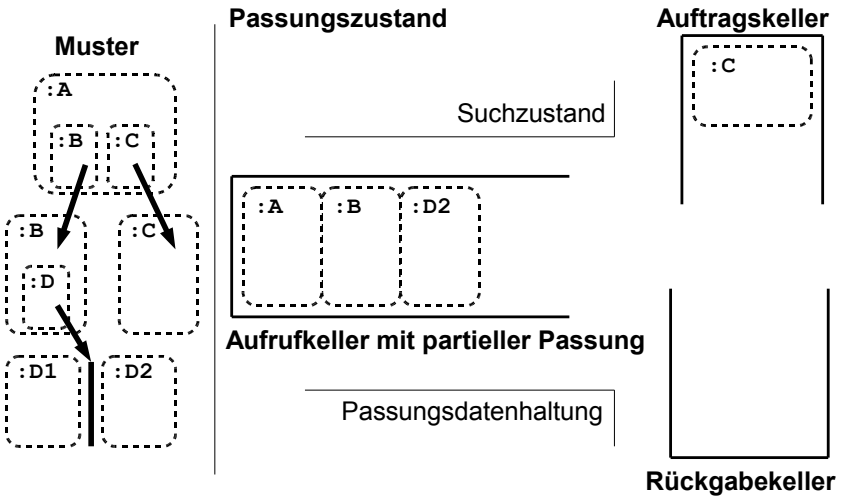


Abbildung 6.7: 7. Der Auftrag für D wird abgearbeitet, D2 wurde gepasst

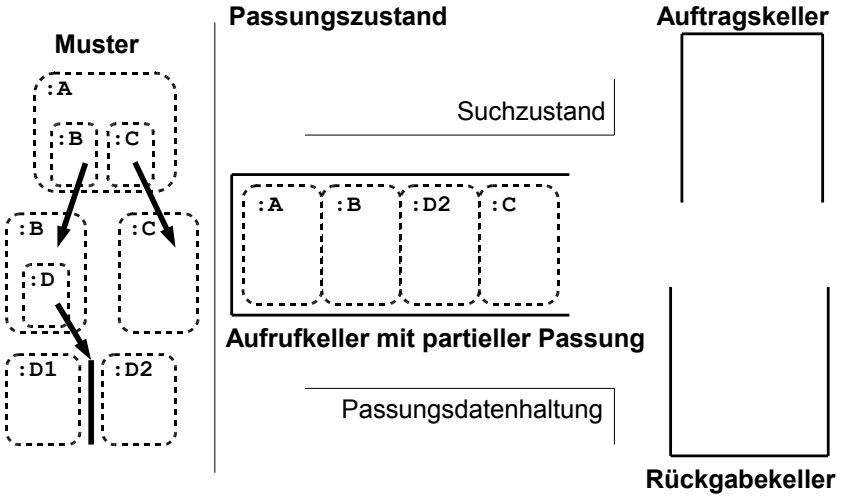


Abbildung 6.8: 8. Der Auftrag für C wird abgearbeitet, C wurde gepasst, die Passung für das zusammengesetzte Muster wurde gefunden, befindet sich aber noch komplett auf dem Aufrufkeller

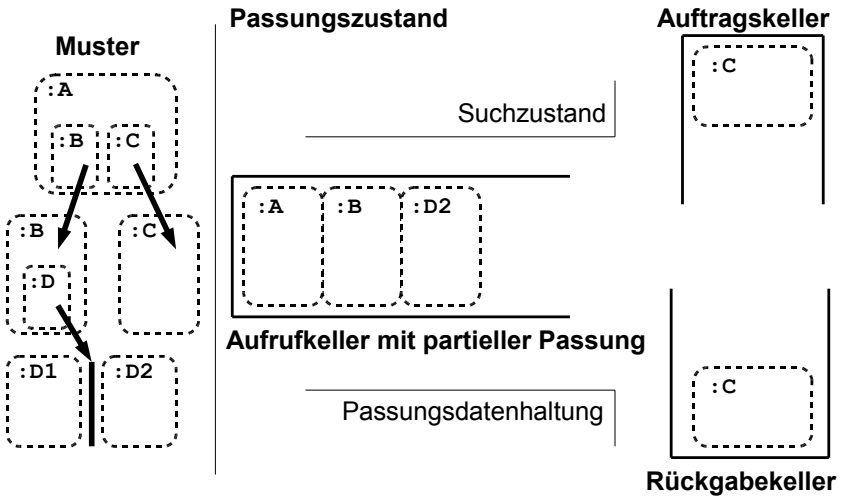


Abbildung 6.9: 9. Die Passung von C wird dem Aufrufkeller entnommen und auf dem Rückgabekeller abgelegt

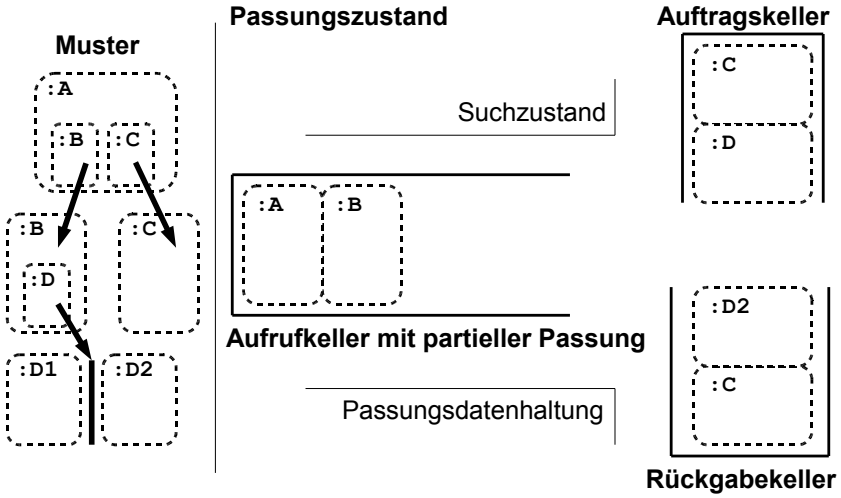


Abbildung 6.10: 10. Die Passung von D2 wird dem Aufrufkeller entnommen und auf dem Rückgabekeller abgelegt

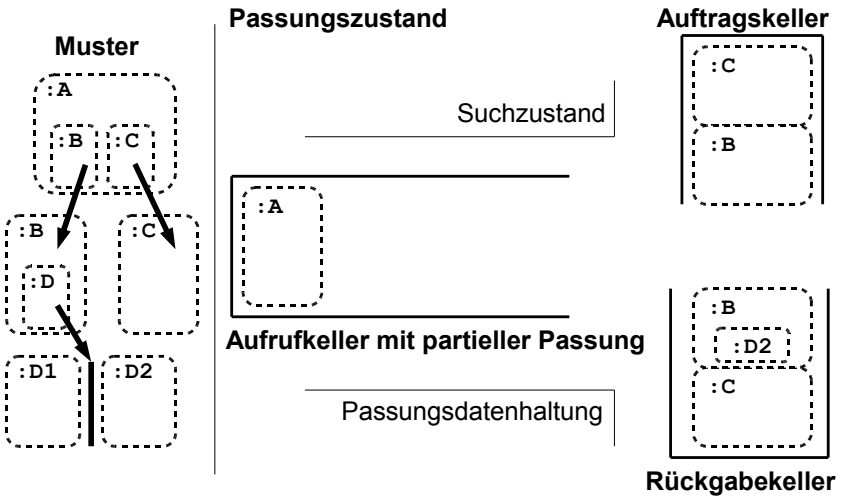


Abbildung 6.11: 11. Die Passung von B wird dem Aufrufkeller entnommen, D2 vom Rückgabekeller wird eingefügt, die Zusammensetzung wird auf dem Rückgabekeller abgelegt

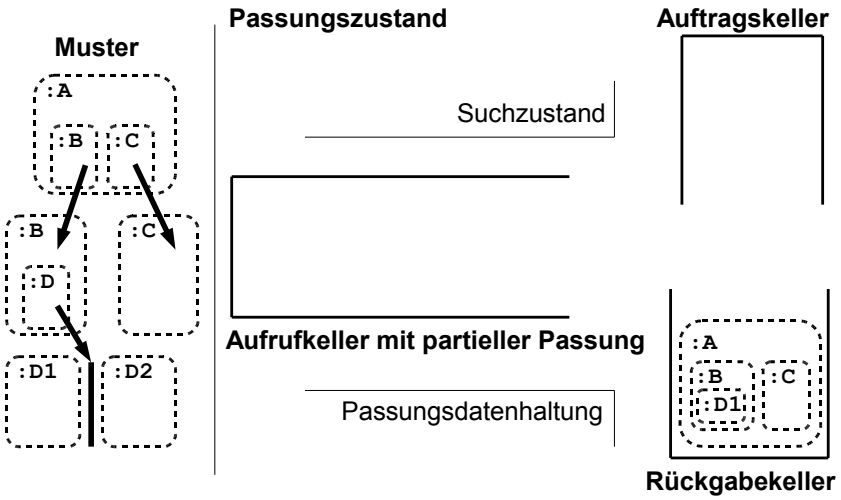


Abbildung 6.12: 12. Die Passung von A wird dem Aufrufkeller entnommen, B und C vom Rückgabekeller wird eingefügt, wir verfügen jetzt über das Passungsobjekt des zusammengesetzten Musters

Kapitel 7

Erweiterungen des Codegenerators

Nach der Beschreibung der Umsetzung der Erweiterungen im generierten Code im vorangegangenen Kapitel möchte ich in diesem Kapitel die hierzu notwendigen Änderungen am Vorgang der Codegenerierung vorstellen, entsprechend des Aufbaus des Graph Rewrite GENerators getrennt in die Anpassungen im Frontend und die im Backend.

7.1 Anpassungen des Generator-Frontends

Der Aufbau der Spezifikationsverarbeitung im Frontend des Generators nach den Erweiterungen ist in Abbildung 7.1 dargestellt. Verglichen mit Abbildung 5.1 wird ersichtlich, dass keine Änderungen an der Phasenstruktur notwendig waren, es mussten lediglich die Datenstrukturen und die Algorithmen zwischen ihnen erweitert bzw. angepasst werden. Dafür mussten allerdings *sämtliche* Phasen angepasst werden. Ich möchte im Folgenden die Änderungen entlang der Phasenstruktur beschreiben.

Syntax und statische Semantik

In der Lexerspezifikation wurden neue Schlüsselwörter eingeführt, in der Parserspezifikation wurde die Syntax um Teilmusterspezifikationen, Teilmusterdeklarationen und Teilmusterersetzungsnutzungen sowie Alternativen erweitert. Durch die Entscheidung, mehr zu akzeptieren als eigentlich zulässig wäre, konnte der bestehende Graphanalysecode (Muster und Ersetzung) in der Teilmusterspezifikation und in den Alternativenzweigen wiederverwendet werden; die überakzeptierten ungültigen Spezifikationen werden in der semantischen Analyse erkannt und zurückgewiesen. Für die Alternativenzweige wurde eine neue Symboltabelle angelegt, Teilmusterspezifikationen werden in der Symboltabelle der Typen, Teilmusterdeklarationen in

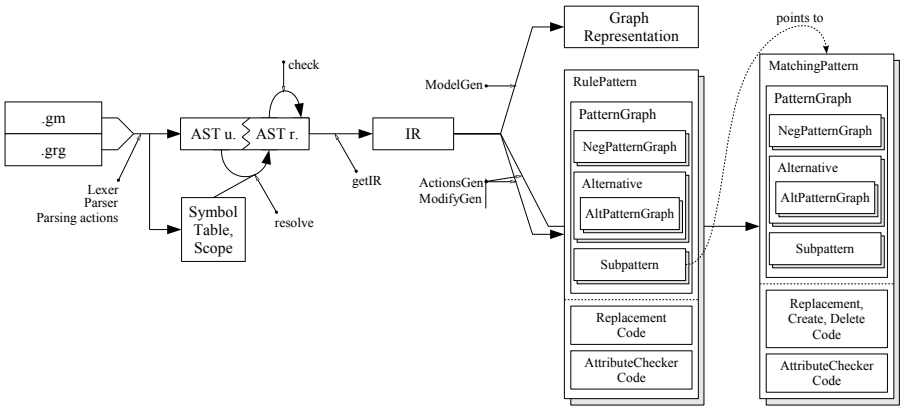


Abbildung 7.1: Ablauf der Codeerzeugung im Frontend von GrGen mit den Erweiterungen

der Symboltabelle der Entitäten abgelegt. Der abstrakte Syntaxbaum wurde um Klassen für Alternativen und Alternativenzweige sowie um Klassen für Teilmusterspezifikationen, Teilmusterdeklarationen und Teilmustererersetzungsnutzungen erweitert. In der Namensauflösung werden Teilmusterdeklarationen ihren Teilmusterspezifikationen und Teilmusterersetzungsnutzungen ihren Teilmusterdeklarationen zugeordnet. Während der semantischen Analyse werden diverse Konsistenzbedingungen geprüft, z.B. die Einhaltung der Signatur einer Teilmusterspezifikation bei ihrer Nutzung in einer Teilmusterdeklaration (Anzahl und Obertypbeziehung der formalen gegenüber der aktuelle Anknüpfungen) oder Äquivalenz der Ersetzungsanknüpfungen in den Alternativenzweigen zu denen im umschließenden Muster. Außerdem wurden, mit dem Parser beginnend und durch alle Phasen hindurch, die negativen Muster aus dem `RulePattern` in den `PatternGraph` verschoben, was in negativen und alternativen Mustern geschachtelte negative Muster ermöglicht hat.

Vor diesen Erweiterungen musste jedoch die semantische Analyse erst erweiterbar gemacht werden. In ihrem Ausgangszustand war ihr Ablauf – was wann wo durch wen aufgelöst oder typgeprüft wird – kaum vorhersagbar, und die Konsequenzen einzelner Eingriffe kaum abschätzbar. Sebastian Buchwald und ich haben deshalb einige Refaktorisierungen durchgeführt, um sie verständlicher zu machen:

- Mehrere nahezu semantikleere Hilfsklassen wurden aus dem Ablauf entfernt, mit dem Effekt von deutlich verkürzten Aufrufkaskaden.

- Die statischen Typen wurden wesentlich geschärft, anstelle der Unmengen an `instanceof` und cast-Operatoren.

- Durch die Vereinfachungen war es dann möglich, die semantische Analyse so zu erweitern, dass ein deutlich kürzerer, überakzeptierender Parser verwendet werden konnte, dessen Sprache dann in der semantischen Analyse auf das zulässige beschränkt wird.

Das Ablaufschema der semantischen Analyse besteht jetzt aus je einer generischen Tiefensuche über den AST durch die `resolve` und die `check`-Methode, die auf den einzelnen AST-Knoten die lokalen `resolveLocal` oder `checkLocal`-Methoden aufruft, die von den Klassen, die Namen auflösen oder Typen prüfen wollen, überschrieben werden.

Zwischensprache

Die neuen Sprachelemente im AST werden nach den Analysen in ihre Repräsentation in der Zwischensprache IR abgebildet, wobei die eigenständigen Klassen für Alternativenzweige und Teilmusterspezifikationen aufgehoben werden, sie werden als (Teil-)Regeln gespeichert; das vereinfacht den späteren rekursiven Lauf über die IR-Struktur in der Generierung ungemein.

Erwähnenswert ist noch die neu eingeführte Funktion `ensureDirectlyNestingPatternContainsAllNonLocalElementsOfNestedPattern`, über sie wird durch das Einfügen von Elementen in Mustern sichergestellt, dass von dem Muster aus, das ein bestimmtes Element zum ersten mal enthält, bis zu einem geschachtelten Muster, das dieses Element ebenfalls enthält, alle dazwischen liegenden Muster dieses Element auch enthalten. Damit wird sichergestellt, dass dieses Element als Vorbelegung vom Muster vor der tiefsten Nutzung zum Muster mit der tiefsten Nutzung heruntergereicht werden kann. Dies geschieht in einem rekursiven Lauf über die geschachtelten Muster in der IR-Struktur. Als Parameter wird jedem Muster die Menge der bereits bekannten Elemente heruntergereicht, vor dem Abstieg werden die im aktuellen Muster enthaltenen Elemente der Menge der bereits bekannten Elemente hinzugefügt, dann erfolgt der rekursive Abstieg in die direkt geschachtelten Muster mit der Übergabe der Menge der bereits bekannten Elemente. Beim Aufstieg werden die Elemente, die in den direkt geschachtelten Mustern, aber nicht im aktuellen Muster enthalten sind, jedoch dort bereits bekannt sind, dem aktuellen Muster hinzugefügt.

Generierung

Die Regelrepräsentations- und Ersetzungscodegenerierung erfolgt in mehreren rekursiven Läufen über die Schachtelstruktur der Teilmuster in der IR (`IR::PatternGraph`). In einem Vorlauf werden Typenreihungen für die zulässige Typen der Musterelemente generiert (wobei die Reihungen nur gefüllt werden, wenn der Typ durch Typausdrücke eingeschränkt wurde), zudem Indizierungsenums, über die die Musterelementenamen auf ihren Index in

den Reihungen der gefundenen Arbeitsgraphelemente des Passungsobjektes abgebildet werden.

Regelrepräsentationslauf

Im Regelrepräsentationslauf werden, von `genSubpattern` für die Teilmuster und `genAction` für die Regeln und Prüfungen aus, in einem Wechselspiel von `genPatternGraph` und `genElementsRequiredByPatternGraph` die Teilmuster- und Regelrepräsentationen vom Typ `MatchingPattern` und `RulePattern` erzeugt, mit den enthaltenen `PatternGraph`-Objekten für die einzelnen (geschachtelten) Mustergraphen.

Über `genElementsRequiredByPatternGraph` werden zu einem Graphen seine Bestandteile wie `PatternNode` für die Musterknoten, `PatternEdge` für die Musterkanten, `PatternGraphEmbedding` für die verwendeten Teilmuster oder `Alternative` für die enthaltenen Alternativen generiert, wobei die Repräsentationen von enthaltenen negative Teilmustern und Alternativenzweigen wiederum durch `genPatternGraph` vor dem enthaltenden Muster erzeugt werden. Das `RulePattern` ist dabei vom `MatchingPattern` abgeleitet, es erweitert es um Ausgabeparameter und eine fest erwartete `Modify`-Methode.

Ein Graphelement, das in einem geschachtelten Muster enthalten ist, aber einem der Obermuster entstammt, wird im Untermuster als Verweis auf das Element aus dem Obermuster abgespeichert. Durch eine `PointOfDefinition`-Membervariable wird festgelegt, in welchem `PatternGraph` es zuerst definiert wurde. Im alten Code wurden Quell- und Zielknoten einer Kante noch direkt in der `PatternEdge` gespeichert, was zu dem Problem führte, dass eine Kante, die in der Luft hing, aber in einem negativen Muster mit einem Knoten verfeinerte wurde, nicht richtig repräsentiert werden konnte. Ich habe es wie folgt gelöst: Anstelle der Kante speichert jeder `PatternGraph` in zwei Streutabellen die Zuordnung einer Kante zu ihrem Quell- und ihrem Zielknoten, genauer gesagt speichert es die in ihm deklarierten Zuordnungen. Quell- und Zielknoten einer Kanten werden durch eine Anfrage an den aktuellen Mustergraphen bestimmt, von dem aus in der Schachtelstruktur bis zu ihrer Deklaration aufgestiegen wird (analog der Behandlung von geschachtelten Gültigkeitsbereichen im Übersetzerbau).

Da alle Graphelemente, auch die der geschachtelten Teilmuster flach in der `initialize`-Methode des `MatchingPattern` erzeugt werden, muss durch Namenspräfixe für Eindeutigkeit gesorgt werden; dies wird während des rekursiven Laufs über die Schachtelstruktur durch den Parameter `pathPrefix` gesichert, die richtige Benennung bereits deklarierter Elemente wird über eine `alreadyDefinedEntityToName`-Streutabelle sichergestellt. Die Trennung in Konstruktor und `initialize`-Methode ist nötig, weil ein rekursives `MatchingPattern` bei seiner Konstruktion auf sich selbst verweisen muss.

Des Weitern werden noch lokale und globale Homomorphietabellen für

die Knoten und für die Kanten erzeugt, durch die festgelegt wird, welche Elemente zusammenfallen dürfen, außerdem wird der Code für die Attributprüfungen generiert (die globalen Tabellen spezifizieren die Homomorphie zwischen Elementen aus dem `PatternGraph`en eines Alternativenzweiges und einem `PatternGraph`, der diesen umschließt).

Ersetzungslauf

Im Ersetzungslauf wird durch `genModify` von `ModifyGen` der Ersetzungscode generiert, die Schachtelung von negativen Mustern muss in diesem nicht berücksichtigt werden (da ein negatives Muster keine Ersetzung besitzt, es verhindert ja gerade das Zustandekommen einer Passung).

Wie in Abschnitt 6.1 beschrieben, werden für die einzelnen Teilmuster und ihre Ersetzungsrollen (abhängige Ersetzung, Hinzufügen, Löschen) lokale Ersetzungsfunktionen erzeugt, die durch Aufrufe kombiniert die Gesamtersetzung bewerkstelligen. Mein Ziel war es, den vorhandenen Code zum Generieren des Ersetzungscodes einer `IR::Rule` für das Generieren des Codes zum Erzeugen und Löschen von Teilmustern sowie der Durchführung ihrer abhängigen Ersetzung wiederzuverwenden. Dazu habe ich zuerst seine Zustandsabhängigkeiten über die neu eingeführte Klasse `ModifyGenerationState` offen gelegt und anschließend als Eingabe an den Ersetzungsgenerierungscode Hilfsobjekte vom Typ `ModifyGenerationTask` eingeführt, in denen der linke und der rechte Graph unabhängig von der spezifizierten Regel angegeben werden können. Der Code zum Erzeugen eines Teilmusters wird generiert, indem als linker ein leerer Graph und als rechter der Mustergraph verwendet wird, der Codes zum Löschen eines Teilmusters wird generiert, indem als linker der Mustergraph und als rechter ein leerer Graph verwendet wird, für eine abhängige Ersetzung werden einfach die Graphen aus der `IR::Rule` referenziert, für eine Prüfung wird der Mustergraph als linker *und* rechter Graph genutzt. Für Alternativen werden Weiterleitungsfunktionen erzeugt, die anhand des gepassten Zweiges entscheiden, welche Ersetzungsfunktion aufzurufen ist. Ersetzungsanknüpfungen werden für die generierten Funktionen und in den generierten Funktionen auf lokale Parameter abgebildet.

Die Kernfunktion der Ersetzungscodegenerierung `genModifyRuleOrSubrule` wurde um Hilfsfunktionen zum Einfügen von Aufrufen der generierten Ersetzungsfunktionen der Teilmuster erweitert (`genSubpatternModificationCalls`, `genAlternativeModificationCalls`, `genNewSubpatternCalls`, `genDelSubpatternCalls`). Da die Ersetzungsmethoden der Teilmuster flach in ihrem `MatchingPattern` oder `RulePattern` erzeugt werden, erhalten sie ihren Schachtelungspfad als Präfix, ihre Rolle wird durch `Modify`, `Create` und `Delete` als Suffix unterschieden; für das Erhalten werden selbstverständlich keine Methoden benötigt.

7.2 Anpassungen des Generator-Backends

Im Gegensatz zum Frontend waren im Backend auch Änderungen an der Gesamtstruktur notwendig, die Codegenerierung aus den `MatchingPattern` und den abgeleiteten `RulePattern` heraus geschieht jetzt in drei aufeinanderfolgenden Schritten, die ich im Folgenden vorstellen möchte.

1. Schritt

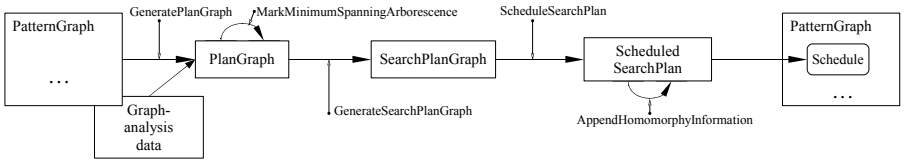


Abbildung 7.2: Ablauf der Codeerzeugung im Backend von GrGen mit den Erweiterungen – 1. Schritt

Im ersten Schritt (dessen Ablauf wird in Abbildung 7.2 gezeigt), wird durch `GenerateScheduledSearchPlans` in einem rekursiven Lauf über die Schachtelstruktur der `PatternGraph`en in den `MatchingPattern` (und `RulePattern`) zu jedem Mustergraph eine Liste von Suchoperationen, der `ScheduledSearchPlan` berechnet. Die Abfolge ist zuerst so wie die in Abschnitt 5.1 beschrieben: Aus dem `PatternGraph` wird in `GeneratePlanGraph` ein `Plangraph` und darin durch `MarkMinimumSpanningArborescence` der minimale spannende Arboreszent berechnet, dieser wird durch `GenerateSearchPlanGraph` in Form eines `SearchPlanGraph` abgespeichert. Der `SearchPlanGraph` wird durch `ScheduleSearchPlan` in den `ScheduledSearchPlan` linearisiert und durch Einfügen der Homomorphieprüfungsoperationen mittels `AppendHomomorphyInformation` vervollständigt. Im Gegensatz zu Abschnitt 5.1 wurden die negativen Suchpläne in diesem Schritt noch nicht in die positiven eingefügt, statt dessen wird der berechnete `ScheduledSearchPlan` als `Schedule` in dem `PatternGraph` zwischengespeichert, der ihn erzeugt hat.

2. Schritt

Im zweiten Schritt, visualisiert in Abbildung 7.3, werden durch `MergeNegativeSchedulesIntoPositiveSchedules` in einem rekursiven Lauf über die Schachtelstruktur der `PatternGraph`en in den `MatchingPattern` (und `RulePattern`) die `Schedules` der negativen `PatternGraph`en in den `Schedule` des nächstgelegenen, sie umschließenden positiven `PatternGraph`en integriert. Aufgrund geschachtelter negativer Graphen kann das

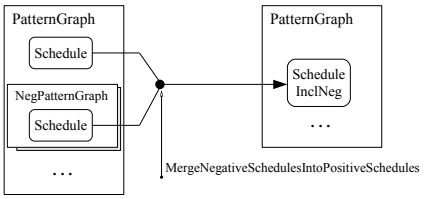


Abbildung 7.3: Ablauf der Codeerzeugung im Backend von GrGen mit den Erweiterungen – 2. Schritt

über mehrere Ebenen hinweg geschehen, das Ergebnis wird in den positiven `PatternGraph`en in `ScheduleIncludingNegatives` abgespeichert.

3. Schritt

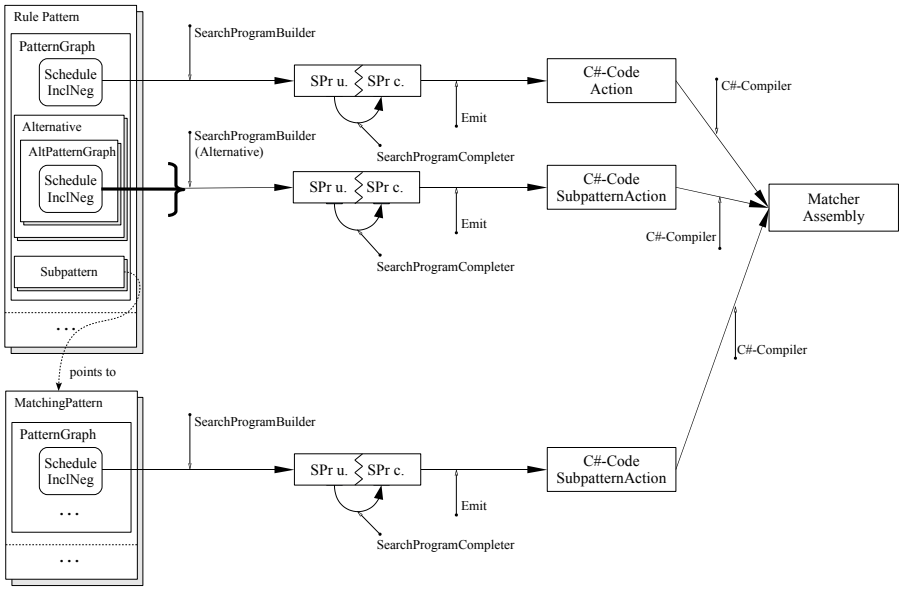


Abbildung 7.4: Ablauf der Codeerzeugung im Backend von GrGen mit den Erweiterungen – 3. Schritt

Im dritten Schritt schließlich wird durch `GenerateMatcherSourceCode` Code erzeugt, wieder aus einem rekursiven Lauf über die Schachtelstruktur der `PatternGraph`en heraus, anhand der in diesen gespeicherten `Schedule-IncludingNegatives`. Der Ablauf ist in Abbildung 7.4 dargestellt.

Zu jedem `RulePattern` wird eine `Action`-Klasse erzeugt, und zu jedem `MatchingPattern` wird eine `SubpatternAction`-Klasse erzeugt, in die der Passungsfindungscode hinein generiert wird. Zusätzlich wird zu jeder `Alternative`, die in den `RulePattern` oder `MatchingPattern` geschachtelt auftritt eine `SubpatternAction`-Klasse erzeugt.

Aus den in den `PatternGraph`en gespeicherten `ScheduleIncludingNegatives` werden über `GenerateSearchProgram` und den `SearchProgramBuilder` die Suchprogramme aufgebaut und über den `SearchProgramCompleter` vervollständigt; je ein Suchprogramm für jeden `PatternGraph`en und je ein Suchprogramm für jede `Alternative` mit allen direkt geschachtelten `PatternGraph`en ihrer Zweige.

Die `SubpatternAction`-Klassen enthalten nicht nur den Passungscode, sondern sind zugleich die Aufgaben der 3-Keller-Maschine, die auf dem Aufgabenkeller abgelegt werden; als Membervariablen enthalten sie ihre Anknüpfungen.

Ansteuerungscode 3-Keller-Maschine

Die höheren Schichten der Codegenerierung sind – wie gesehen und wie gewünscht – in weiten Teilen unabhängig von der Ansteuerung der 3-Keller-Maschine. Diese tritt erst auf der Ebene der Suchprogramme verstärkt in Erscheinung, durch das Einfügen einer `InitializeSubpatternMatching`-Suchprogrammoperation am Anfang eines Suchprogramms und einer `FinalizeSubpatternMatching`-Suchprogrammoperation am Ende eines Suchprogramms, hauptsächlich jedoch nach dem Umsetzen des `ScheduleIncludingNegatives` in Suchprogrammoperationen durch die Hilfsfunktion `buildMatchComplete` an der innersten Stelle des Suchprogramms. Das ist der Ort im Suchprogramm, an dem bei Ausführung des Suchprogramms das lokale Muster gerade gefunden wurde; der Code für die jetzt zu findenden Teilmuster und die ebenso behandelten Alternativen wird durch `insertPushSubpatternTasks` eingefügt, zum Abarbeiten des Auftragsstapels wird `MatchSubpatterns` in das Suchprogramm eingefügt, gefolgt von `insertPopSubpatternTasks`, das den Code zum Aufräumen des Auftragsstapels einfügt. Suchprogrammoperationen zum Handhaben von Erfolg und Misserfolg werden dann durch `insertCheckForSubpatternsFound` in das Suchprogramm eingefügt, der Code zum Verwalten des Rückgabestapels wird, aus dieser Funktion heraus aufgerufen, durch `insertMatchObjectBuilding` eingefügt.

Kapitel 8

Evaluation

Um die Leistungsfähigkeit der Implementierung abschätzen und mit anderen Graphersetzungssystemen vergleichen zu können, habe ich den Benchmark Transkription entwickelt, der sich an der, aus der Genetik bekannten, Transkription einer DNA-Sequenz in eine RNA-Sequenz orientiert. Ich werde zuerst die Aufgabe Transkription allgemein beschreiben, dann den Benchmark Transkription konkret vorstellen und im Anschluss auf ihm basierende Messergebnisse zu GrGen und Viatra 2 R2 präsentieren. Die zugehörigen GrGen-Spezifikationen befinden sich im Anhang A.3 (Beispiel Transkription DNA in RNA abstrakt).

8.1 Aufgabe Transkription

Als Transkription wird in der Genetik das Erzeugen einer RNA-Sequenz anhand einer DNA-Sequenz, dem Träger der Erbinformation, bezeichnet. RNA und DNA sind Nukleinsäuren, das sind Makromoleküle, die aus Ketten von Nukleotiden bestehen. Jedes Nukleotid besitzt einen ringförmigen Zucker als Kern, bei der DNA der Desoxyribose, bei der RNA der Ribose. Die Zucker und damit die Nukleotide sind über eine Phosphatgruppe untereinander verbunden, zudem sind sie mit jeweils einer der Nukleinbasen Adenin, Cytosin, Guanin, Thymin, Uracil oder Hydroxyguanin verbunden, im Folgenden mit ihrem Anfangsbuchstaben abgekürzt. Ein DNA-Nukleotid darf nur eine Bindung mit einer der Nukleinbasen A,C,G oder T aufweisen; ein RNA-Nukleotid nur eine Bindung mit einer der Nukleinbasen A,C,G oder U. Ansonsten ist das Nukleotid, und damit die gesamte Sequenz, in der es vorkommt, als beschädigt anzusehen. Auf einer DNA-Kette befinden sich mehrere zu transkribierende Einheiten, die Gene; sie sind innerhalb der Kette jeweils durch eine Start- und eine Endsequenz kenntlich gemacht. Die Transkription beginnt nach der TATABox, das ist die Startsequenz mit der Nukleinbasenabfolge TATAAA, sie endet mit bzw. genau vor dem ersten Auftreten der Terminierungssequenz CCCACTNNNNNNAGTGGGAAAAAA,

wobei die N für eine beliebige Nukleinbase stehen. Ein solchermaßen eingegrenzter Abschnitt der DNA-Kette wird in eine RNA-Kette umgeschrieben, indem schrittweise für jedes DNA-Nukleotid ein RNA-Nukleotid erzeugt und an die bereits bestehende RNA-Kette angehängt wird. Dabei werden die Nukleinbasen nicht direkt übernommen, sondern von A in U, C in G, G in C und T in A umgeschrieben. Bei dieser Beschreibung handelt es sich um eine Vereinfachung für den Zweck des Benchmarks, die aber den tatsächlichen Gegebenheiten hinreichend weit entspricht.

Graphmodell

Für den Benchmark wird das Graphmodell wie folgt festgelegt: Es besteht aus den Knotentypen `D` und `R` für die Zucker Desoxyribose und Ribose sowie den Knotentypen der Nukleinbasen, `A` für Adenin, `C` für Cytosin, `G` für Guanin, `T` für Thymin, `U` für Uracil und `H` für Hydroxyguanin, die alle vom Knotentyp `N` (für Nukleinbase) abgeleitet sind. Des Weiteren enthält es einen Kantentyp `PG` für die Phosphatgruppen, außerdem wird der Kantenbasistyp `Edge` verwendet. Eine DNA-Kette besteht aus einer Folge von `D`-Knoten, die über `PG`-Kanten untereinander verbunden sind. Von den Zucker-Knoten führt eine `Edge`-Kante zu dem jeweiligen Nukleinbasenknoten; in einer RNA-Kette wird anstelle des `D`-Knotens ein `R`-Knoten verwandt. Zu einer DNA-Sequenz eines Genes ist wie weiter oben beschrieben eine RNA-Sequenz als Abschrift schrittweise aufzubauen. Für den Benchmark wird festgelegt, dass ein beschädigtes Gen durch die Transkription nicht verarbeitet werden darf (selbst wenn das erste beschädigte Nukleotid erstmals an letzter Stelle vor der Terminierungssequenz auftritt).

Mögliche Verfeinerung

Das oben vorgestellte abstrakte Modell mit seinen Knoten und Kanten, die große Molekülteile repräsentieren, kann auf ein zugrundeliegendes Modell aus Atomen verfeinert werden. Dazu wird ein `R/D`-Knoten, der über je eine `PG` Kante mit dem vorherigen und dem nachfolgenden `R/D`-Knoten in der Kette sowie über eine Kante mit dem Nukleotidknoten verbunden ist, durch sein Ribose/Desoxyribose-Molekül ersetzt; dessen c1,c3,c5-Atome stellen die Verbindungspunkte dar, über die das Molekül des Nukeotids und das Molekül der Phosphatgruppe, die mit dem vorhergehenden Zucker verknüpft, und das Moleküls der Phosphatgruppe, die mit dem nachfolgenden Zuckers verknüpft, angebunden werden. Bei dieser Modellierung tritt der Graphcharakter deutlicher hervor als bei der vorherigen. Da die Geschwindigkeit der Suche nach den Knoten und Kanten der einzelnen Teilgraphen für mich aber weniger interessant ist als der Aufwand für die Verwaltung der Teile, habe ich das abstrakte Graphmodell als Basis für den Benchmark gewählt. Es ist in A.3 zu finden, die Transkription auf Atomen wird in A.4 vorgestellt (aller-

dings ohne Beachtung von Anfangs- und Endsequenzen), und in A.5 wird ein rekursive Regel zur Umwandlung einer abstrakt modellierten DNA-Kette in eine mit Atomen modellierten DNA-Kette aufgeführt.

8.2 Benchmark Transkription

Der Graph des Benchmarks besteht aus 2 korrekt aufgebauten und 2 beschädigten DNA-Ketten, die durch ein Programm generiert werden. Jede dieser Ketten ist wie folgt aufgebaut (entlang der Kette, vom Anfang zum Ende):

1. DNA-Kettenstück mit 100 Kettengliedern
2. TATA-Box (6 Kettenglieder) von Gen 1
3. DNA-Kettenstück von Gen 1 mit n Kettengliedern
4. Terminierungssequenz (24 Kettenglieder) von Gen 1
5. DNA-Kettenstück mit 100 Kettengliedern
6. TATABox (6 Kettenglieder) von Gen 2
7. DNA-Kettenstück von Gen 2 mit n Kettengliedern
8. Terminierungssequenz (24 Kettenglieder) von Gen 2
9. DNA-Kettenstück mit 100 Kettengliedern

Die Nukleotide werden zufällig gewählt, für die beschädigten Ketten ist n immer 100, für die korrekt aufgebauten Ketten wird über n die Problemgröße variiert. Die Aufgabenstellung besteht darin, 10-mal alle vorkommenden, korrekt aufgebauten Gene im Graphen zu finden und zu transkribieren.

8.3 Messergebnisse

Der Vergleich der Leistungsfähigkeit mit anderen Graphersetzungssystemen beschränkt sich auf Viatra 2, da nur dieses Graphersetzungswerkzeug neben GrGen die nötigen alternativen und rekursiven Muster implementiert. Da Viatra 2 aber nur rekursive Muster und keine rekursiven Regeln anbietet, beschränkt sich der Vergleich zusätzlich auf das Finden der umzuschreibenden DNA-Sequenzen. Die Messungen erfolgten auf einem AthlonXP 2400+ mit 1GiB Speicher unter WindowsXP SP3, Java 6u6, Eclipse 3.3.2 und Microsoft .NET 2 SP 1. Die Ergebnisse sind in Abbildung 8.1 dargestellt; ich bitte die logarithmische Skala für die Suchdauer zu beachten.

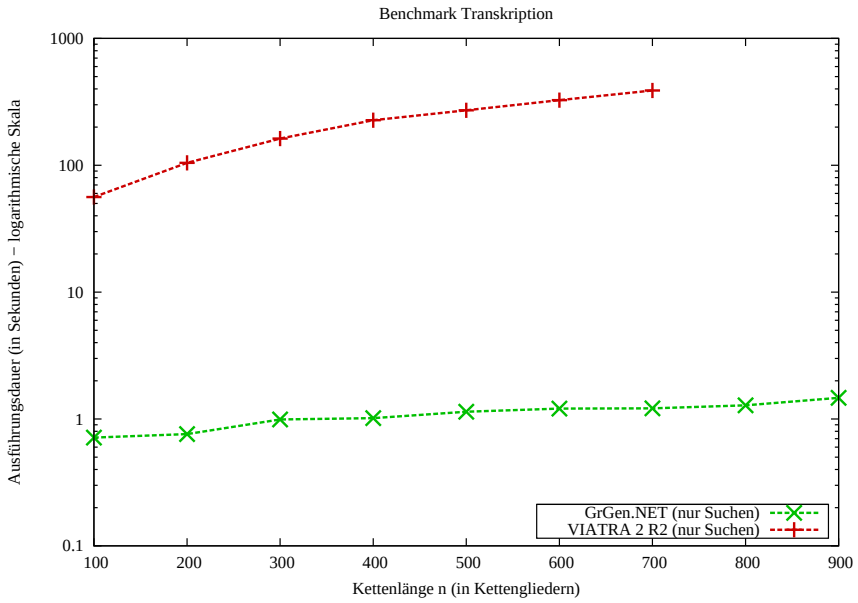


Abbildung 8.1: Benchmark GrGen gegen Viatra 2 R2

Aus ihr kann entnommen werden, dass meine Implementierung von rekursiven Mustern in GrGen um 2 Größenordnungen schneller ist als die von Viatra 2 R2. Ketten einer Länge ≥ 800 konnten mit Viatra 2 R2 aufgrund einer dann auftretenden `StackOverflowException` nicht mehr gepasst werden. Ein weiteres Problem an Viatra 2 R2 ist der $O(n^2)$-Algorithmus zum Importieren von Graphen, durch den das Einlesen des Benchmarkgraphen mit 800 Kettengliedern (20.000 Graphelemente) bereits 20 Minuten in Anspruch nimmt (gegenüber 4 Sekunden in GrGen). Augenfällig ist der hohe Sockelbetrag von ca. 700ms bei GrGen, der erst bei Kettenlängen von 900 verdoppelt wird. Das Durchmustern des Graphen auf der Suche nach den Kettenanfängen geht hier nur in geringem Umfang ein, im Wesentlichen entsteht der Sockelbetrag durch die Just-In-Time-Übersetzung des MSIL-Codes der generierten Suchprogramme.

In Abbildung 8.2 ist das Verhalten von GrGen beim Steigern der Problemgröße hin zu Ketten bis zu 10.000 Kettengliedern (entsprechend einem Benchmarkgraphen mit mehr als 165.000 Elementen) dargestellt, wobei zusätzlich zur reinen Suche noch die Messwerte für Suchen und Ersetzen, und somit dem eigentlichen Benchmark Transkription, gezeigt werden.

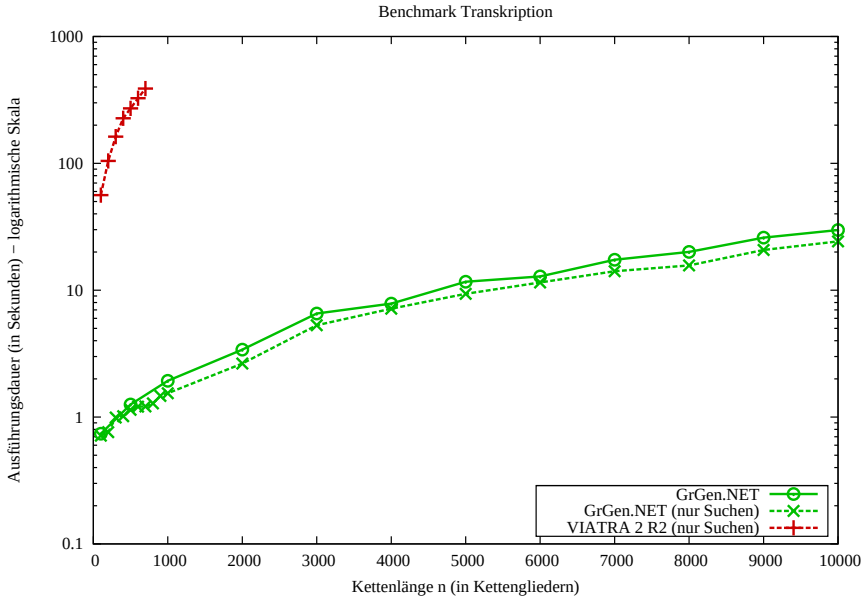


Abbildung 8.2: Benchmark GrGen (mit Viatra 2 R2)

Kapitel 9

Zusammenfassung und Ausblick

9.1 Zusammenfassung

Mit den rekursiven Regeln, die durch das Zusammenspiel von Teilregeln und alternativen Regeln zur Verfügung stehen, können komplexe, rekursiv strukturierte Graphmuster innerhalb *einer* Regelanwendung gefunden und umgeschrieben werden. Ich hatte dies bereits in meiner Studienarbeit vorgestellt, in Form einer Erweiterung von GrGen. In der vorliegenden Diplomarbeit habe ich sie zusätzlich über Muster- und Regelzusammensetzungsgrammatiken, aufbauend auf Sterngraphgrammatiken und Paargraphgrammatiken formal fundiert, und, zusammen mit geschachtelten negativen Mustern, implementiert.

Hierzu musste der Graphersetzungsgenerator, im Frontend wie im Backend, über seine gesamte Länge hinweg erweitert werden: vom Prüfen von Syntax und statischer Semantik über das Aufbauen der Zwischensprache hin zum Generieren des Ersetzungscodes, und über die Suchplanung hin zur Codegenerierung für die Passungssuchprogramme.

Die Suche wurde grundsätzlich effizient über eine 3-Keller-Maschine, die Ersetzung über einen rekursiven Durchlauf durch den Passungsbaum unter Durchführung der Teilersetzungen implementiert; die Effizienz der Implementierung wurde in einem Vergleich mit Viatra 2 aufgezeigt, dem einzigen Graphersetzungswerkzeug, das ebenfalls rekursive Muster beherrscht.

Um sich mit rekursiven Mustern und Regeln vertraut zu machen, möchte ich den Leser zum Stöbern in den `examples` und `tests`-Verzeichnissen des GrGen.NET-Programmpakets einladen, das unter der Adresse `www.grgen.net` frei heruntergeladen werden kann; sie enthalten einige Beispielspezifikationen zu GrGen allgemein sowie zu meinen Erweiterungen, wie das Beispiel Transkription oder eine Modellierung der Refaktorisierung PullUpMethod auf Programmgraphen [DHM08].

Vor dem Ausblick möchte ich die Zusammenfassung abschließen: Hat GrGen vor meiner Arbeit bereits Zeichen mit seiner Verarbeitungsgeschwindigkeit [GBG+06] gesetzt, so ist es mit den hier vorgestellten Erweiterungen nun zudem wegweisend, was die Ausdrucksmächtigkeit von Graphersetzungssystemen anbelangt [HJG08].

9.2 Ausblick

Die Passungssuche ist bereits 2 Größenordnungen schneller als die des einzigen Konkurrenzwerkzeugs, das ebenfalls rekursive Muster unterstützt – das ist beachtlich angesichts der grundständigen Implementierung ohne weitergehende Optimierungen. Ich gehe davon aus, dass die Leistung mit teilmusterübergreifenden Analysen und darauf aufbauenden Optimierungen noch deutlich gesteigert werden könnte. Mögliche Verbesserungen sind:

Offenes Einbauen von Mustern in das Gesamtmuster, analog dem Inlining von Code im Übersetzer, mit den gleichen Vor-, und, wenn zu exzessiv betrieben, Nachteilen.

Gemeinsame Alternativenteilmuster müssen nur einmal gepasst werden, nicht in jedem Zweig gesondert. Besonders interessant ist hierbei das Einbeziehen von NACs: bei rekursiven Mustern mit einem `negative` des Rekursivfalls als Basisfall deckt nämlich das Ergebnis einer Passungssuche beide Alternativenzweige ab. Es muss nur noch anhand des Suchergebnisses über den vorliegenden Fall entschieden werden.

Umdrehen der Suchreihenfolge, um eine Kette vom Ende aus in entgegengesetzter Richtung, oder allgemeiner ein zusammengesetztes Muster von innen nach außen zu suchen. In diesem Zusammenhang ist auch ein **Umdrehen der Parameterübergaberichtung** mancher Anknüpfungen von Ein- nach Ausgabe interessant; insbesondere von Graphelementen, die in einem Muster nur auftauchen, weil sie als Schnittstelle für zwei Teilmuster dienen, womit sie im terminalen Teil des umschließenden Musters unverbunden aus dem Graph herausgegriffen werden, um dann in den Teilmustern anhand der fehlenden Verbindungen verworfen zu werden.

Entfernen unnötiger Flexibilität, wie bei der Passungssuche nach iterierten Pfaden. Der Entwurf der 3-Keller-Maschine ist für verzweigende Muster vorgesehen, für einfache Ketten kann effizienterer Code generiert werden. Ebenso könnte der Code von Alternativen bei nur einer Alternative direkt im Code des enthaltenden Musters eingefügt werden, das Abbilden auf Teilmuster vermeidet eine Codeexplosion im allgemeinen Fall mehrerer Alternativen.

Mit meinen Erweiterungen habe ich einige Punkte von der Liste der gewünschten Merkmale für GrGen streichen können, es sind aber auch Neue hinzugekommen und manche Alte erhalten geblieben:

Das `independent` Schlüsselwort für negative Muster oder auch für positive Teilmuster, die damit unabhängig von den bereits gepassten Teilmustern gesucht werden.

Unterschiedliche abhängige Ersetzungen zu einem Teilmuster, wie in meiner Studienarbeit beschrieben.

Rückgaben aus Alternativenzweigen heraus, so dass unterschiedliche Graphelemente in Abhängigkeit der gewählten Alternative zurückgegeben werden können.

Hinzufügen von Teilmustern mit Alternativen in einer Ersetzung mit der Auswahl des zu instanziierenden Alternativenzweiges durch den Nutzer. Die Herausforderung besteht hier in der Vollständigkeitsprüfung der Auswahl in der semantischen Analyse.

Beschränkte Rekursion durch die Unterstützung von Wert-Parametern in den Teilmustern und deren Berechnung bei der Teilmusterinstanziierung.

Eine Kopieroperation für Knoten, die anliegende Kanten mit vervielfältigt, ähnlich dem Sesqui-Pushout-Ansatz; besonders interessant ist hier dann die Erweiterung auf ganze Teilmuster.

Mengenwertige Anknüpfungen, die über die festen Anknüpfungen des gewählten Stern- bzw. Hyperkantenansatzes hinausgehen und die Ausdrucksmächtigkeit in Richtung Knotenersetzungsgrammatiken bzw. adaptive Sterngrammatiken erhöhen würden.

Veränderung der Teilmusterstruktur in einem Ableitungsschritt, im Gegensatz zum derzeit nur möglichen lokalen Umschreiben Teilmuster für Teilmuster, um z.B. einen Baum in eine Liste umzuschreiben; es ist hierbei allerdings noch völlig offen, ob das überhaupt sinnvoll möglich ist.

Danksagung

Hey Freaks, danke für das tolle Arbeitsklima!

Ich danke meinem Betreuer Rubino Geiß für die Betreuung, die (wenn auch manchmal zähen) Diskussionen über die Erweiterungen sowie für die Seelennahrung. Außerdem den weiteren Mitarbeitern und Studenten am IPD Goos: Moritz Kroll, Sebastian Buchwald, Christoph Mallon, Andreas Schösser, Michael Beck, Matthias Braun und Andreas Zwinkau. Dafür, dass ihr das SVN zerschossen und wieder repariert habt, für die mal netten, mal fiesen Kommentare, für Blicke in Seelen, die nach Weltherrschaft streben oder sie gleich zerstören wollen, für den lausigen Code, mit dem ich mich herumschlagen musste (na gut, der geht hauptsächlich auf das Konto von Ex-Studenten, die hier nicht aufgezählt sind; ah, noch Danke an Veit Batz für die vielen Testfälle) und für die Kekse. Außerdem wart ihr gute Opfer, lange lebe der Erdbeben-Kampfbahn-Dienstgeber ;)

Noch was anderes: Ich danke allen, die für die elektronische Halde und nicht allein die Papier(müll)halde produzieren. Die Literaturrecherche auf Ersterer ist dank des allwissenden Orakels so viel effizienter als das Durchwühlen von Letzterer, dass ich mir diese Zumutung auch nur einmal angetan habe. Deshalb als Tipp an alle, die etwas publizieren, von dem sie auch wollen, dass es gelesen wird: Was nicht frei im Netz ist, existiert nicht!

Anhang A

Definitionen und Beispiele

A.1 EBNF

Die in dieser Arbeit verwandte EBNF ist definiert durch:

- ϵ steht für das leere Wort
- Nichtterminale werden durch Zeichenketten dargestellt, Bsp: `Id`
- Terminale Zeichen und Zeichenketten werden von An- und Ausführungszeichen `""` umschlossen, Bsp: `"bar"`
- Eine Produktion wird als ein Nichtterminal, gefolgt von einem wird-zu `::=`, gefolgt von einer Kette von Terminalen und Nichtterminalen geschrieben, Bsp: `Foo ::= "bar" Foo "bar"`
- Ein vertikaler Strich | beschreibt die Auswahl aus mehreren Möglichkeiten, Bsp: `Bin ::= "0" | "1"`
- Klammern `()` gruppieren Konstrukte,
  Bsp.: `Bin ::= ("0" | "1") Bin` | ϵ.
- Eckige Klammern `[]` umschließen optionale Konstrukte,
  Bsp.: `Foobar ::= ["foo "] "bar "`
- Eckige Klammern gefolgt von einem $*$ kennzeichnen eine – auch 0-malige – Wiederholung, Bsp: `[Id]`$*$
- Eckige Klammern gefolgt von einem + kennzeichnen eine mindestens 1-malige Wiederholung, Bsp: `[Id]`+, es gilt `[Id]`+ $\equiv$ `Id [Id]`$*$
- Das Konstrukt `A`||`B` beschreibt eine Folge von `A` mit `B` dazwischen,
  Bsp: Id || ", "paßt auf `Id,Id,Id` oder `Id,Id` oder `Id` oder ϵ,
  aber nicht auf `Id,Id,`

A.2 Passungsobjekt

Passungsobjekte geben in Form eines Ableitungsbaumes die Schachtelstruktur der Teilmuster wieder, ein Passungsobjekt ist wie folgt definiert:

```
1   /// An object representing a match.
2   public interface IMatch
3   {
4       /// The match object represents a match of this pattern
5       IPatternGraph Pattern { get; }
6
7       /// An array of all nodes in the match.
8       /// The order is given by the Nodes array of the IPatternGraph.
9       INode[] Nodes { get; }
10
11      /// An array of all edges in the match.
12      /// The order is given by the Edges array of the IPatternGraph.
13      IEdge[] Edges { get; }
14
15      /// An array of variables given to the matcher method.
16      /// The order is given by the Variables array of the IPatternGraph.
17      object[] Variables { get; }
18
19      /// An array of all submatches due to subpatterns and alternatives.
20      /// First subpatterns in order of EmbeddedGraphs array
21      /// of the according IPatternGraph,
22      /// then alternatives in order of Alternatives array
23      /// of the according IPatternGraph.
24      /// You can find out which alternative case was matched
25      /// by inspecting the Pattern member of the submatch.
26      IMatch[] EmbeddedGraphs { get; }
27  }
28
29  /// A pattern graph.
30  public interface IPatternGraph
31  {
32      /// The name of the pattern graph
33      String Name { get; }
34
35      /// An array of all pattern nodes.
36      IPatternNode[] Nodes { get; }
37
38      /// An array of all pattern edges.
39      IPatternEdge[] Edges { get; }
40
41      ...
42  }
```

A.3 Beispiel Transkription DNA in RNA abstrakt

Die folgende GrGen-Spezifikation beschreibt auf abstrakter Ebene die Transkription von DNA nach RNA, wie sie in Kapitel 8 für den Benchmark Transkription vorgestellt wurde; sie ist ebenfalls im `examples`-Verzeichnis des GrGen.NET-Programmpaketes zu finden, dieses enthält zusätzlich die Viatra 2-Version des Beispiels. Die Regel `transcription` sowie die Teilregeln `DNAChain` und `DNANucleotide` sind zudem noch in den Abbildungen A.1, A.2 und A.3 grafisch dargestellt.

```
node class N; // Nucleotide
node class A extends N; // Adenin
node class G extends N; // Guanin
node class C extends N; // Cytosin
node class T extends N; // Thymin
node class U extends N; // Uracil
node class H extends N; // Hydroxyguanin

node class S; // Sugar
node class D extends S; // Desoxyribose
node class R extends S; // Ribose

edge class PG; // Phosphate Group
```

```
rule transcription()
{
   // start transcription at TATABox: TATAAA.
   d1:D -:PG-> d2:D -:PG-> d3:D -:PG-> d4:D -:PG-> d5:D -:PG-> d6:D;
   d1 --> :T;
   d2 --> :A;
   d3 --> :T;
   d4 --> :A;
   d5 --> :A;
   d6 --> :A;

   d2r:DNAChain(d6);

   modify {
      r:R; // new starting point for rna
      d2r(r);
   }
}

pattern DNAChain(prev:D)
{
   alternative {
       Chain {
         negative {
            :TerminationSequence(prev);
```

```
26          }
27
28          prev -:PG-> next:D;
29          n:DNANucleotide(next);
30          d2r:DNAChain(next);
31
32          modify(rprev:R) {
33             rprev -:PG-> rnext:R;
34             n(rnext);
35             d2r(rnext);
36          }
37       }
38       End {
39          :TerminationSequence(prev);
40
41          modify(rprev:R) {
42          }
43       }
44    }
45
46    modify(rprev:R) {
47    }
48 }
49
50 pattern DNANucleotide(d:D)
51 {
52    alternative {
53       A {
54          d --> a:A;
55
56          modify(r:R) {
57             r --> u:U;
58          }
59       }
60       C {
61          d --> c:C;
62
63          modify(r:R) {
64             r --> g:G;
65          }
66       }
67       G {
68          d --> g:G;
69
70          modify(r:R) {
71             r --> c:C;
72          }
73       }
74       T {
```

```
75          d --> t:T;
76
77          modify(r:R) {
78             r --> a:A;
79          }
80       }
81    }
82
83    modify(rprev:R) {
84    }
85 }
86
87 pattern TerminationSequence(nprev:D)
88 {
89    // CCCACT
90    nprev -:PG-> d1:D -:PG-> d2:D -:PG-> d3:D;
91    d3:D -:PG-> d4:D -:PG-> d5:D -:PG-> d6:D;
92    d1 --> :C;
93    d2 --> :C;
94    d3 --> :C;
95    d4 --> :A;
96    d5 --> :C;
97    d6 --> :T;
98    // NNNNNN
99    d6 -:PG-> d7:D -:PG-> d8:D -:PG-> d9:D;
100   d9:D -:PG-> d10:D -:PG-> d11:D -:PG-> d12:D;
101   d7 --> :N;
102   d8 --> :N;
103   d9 --> :N;
104   d10 --> :N;
105   d11 --> :N;
106   d12 --> :N;
107   // AGTGGG = inverse and mirrored CCCACT
108   d12 -:PG-> d13:D -:PG-> d14:D -:PG-> d15:D;
109   d15:D -:PG-> d16:D -:PG-> d17:D -:PG-> d18:D;
110   d13 --> :A;
111   d14 --> :G;
112   d15 --> :T;
113   d16 --> :G;
114   d17 --> :G;
115   d18 --> :G;
116   // AAAAAA
117   d18 -:PG-> d19:D -:PG-> d20:D -:PG-> d21:D;
118   d21:D -:PG-> d22:D -:PG-> d23:D -:PG-> d24:D;
119   d19 --> :A;
120   d20 --> :A;
121   d21 --> :A;
122   d22 --> :A;
123   d23 --> :A;
```

```
124     d24 --> :A;
125 }
```

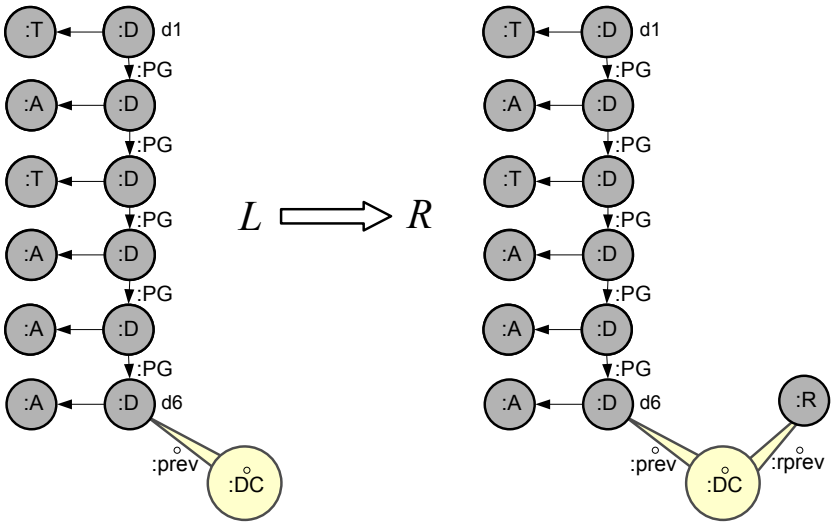


Abbildung A.1: Regel transcription

A.4 Beispiel Transkription DNA in RNA

Die folgende GrGen-Spezifikation beschreibt auf Ebene der Atome und Bindungen die Transkription von DNA nach RNA; aus Platzgründen wird nur ein Teil gezeigt, die vollständige Spezifikation befindet sich im `examples`-Verzeichnis des GrGen.NET-Programmpakets.

```
1 node class C;
2 node class H;
3 node class O;
4 node class N;
5 node class P;
```

```
1 rule Transkription {
2     pLink:P;
3     b:DNAChainBegin(pLink);
4     c:DNAChain(pLink);
5
6     modify {
7         oOfNext:O;
```

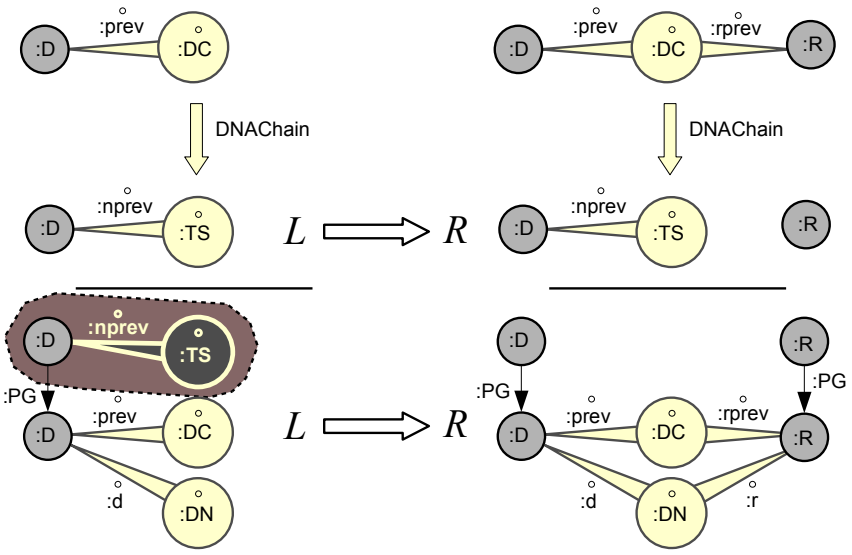


Abbildung A.2: Teilregel DNAChain

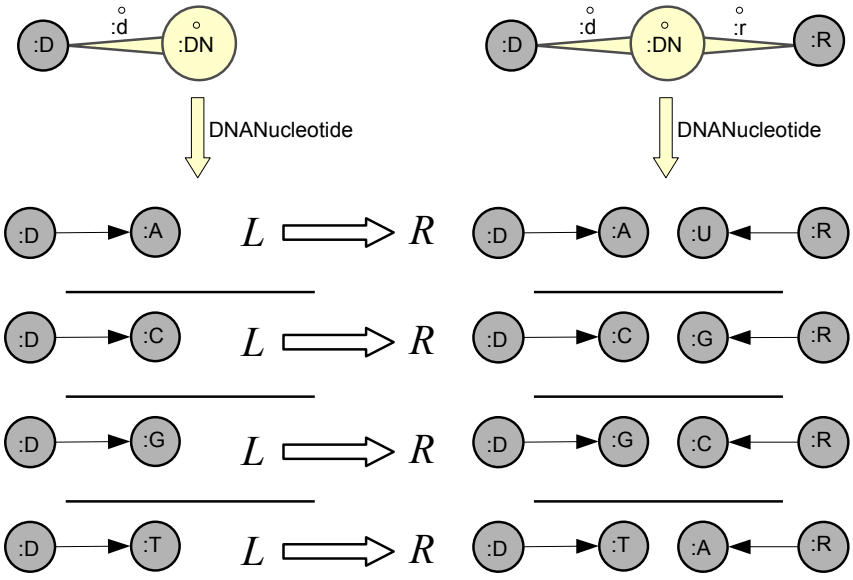


Abbildung A.3: Teilregel DNANucleotide

```
8        b(oOfNext);
9        c(oOfNext);
10    }
11 }
12
13 pattern DNAChainBegin(pOfNext:P)
14 {
15    :H --> :O --> c3;
16
17    // Desoxyribose core
18    o:O --> c1:C --> c2:C --> c3:C --> c4:C --> o;
19    c1 --> :H;  c2 --> :H;  c3 --> :H;  c4 --> :H;
20    c4 --> c5:C;
21    c5 --> :H;  c5 --> :H;
22    c2 --> :H;
23
24    n:Nukleinbase(c1);
25
26    c5 --> :O --> pOfNext;
27    pOfNext --> do:O;  pOfNext --> do;  pOfNext --> :O --> :H;
28
29    modify(oOfNext:O) {
30       rep_c1:C;
31       :RNAChainBegin(oOfNext, rep_c1);
32       n(rep_c1);
33    }
34 }
35
36 pattern DNAChain(pOfPrev:P) {
37    alternative {
38       Chain {
39          e:DNAChainElement(pOfPrev, pOfNext);
40          pOfNext:P;
41          c:DNAChain(pOfNext);
42
43          modify(oOfPrev:O) {
44             e(oOfPrev, oOfNext);
45             oOfNext:O;
46             c(oOfNext);
47          }
48       }
49       End {
50          e:DNAChainEnd(pOfPrev);
51
52          modify(oOfPrev:O) {
53             e(oOfPrev);
54          }
55       }
56    }
```

```
57
58    modify(oOfPrev:O) {
59    }
60 }
61
62 pattern DNAChainElement(pOfPrev:P, pOfNext:P)
63 {
64    pOfPrev --> :O --> c3;
65
66    // Desoxyribose core
67    o:O --> c1:C --> c2:C --> c3:C --> c4:C --> o;
68    c1 --> :H;  c2 --> :H;  c3 --> :H;  c4 --> :H;
69    c4 --> c5:C;
70    c5 --> :H;  c5 --> :H;
71    c2 --> :H;
72
73    n:Nukleinbase(c1);
74    c5 --> :O --> pOfNext;
75    pOfNext --> do:O;  pOfNext --> do;  pOfNext --> :O --> :H;
76
77    modify(oOfPrev:O, oOfNext:O) {
78       :RNAChainElement(oOfPrev, oOfNext, rep_c1);
79       rep_c1:C;
80       n(rep_c1);
81    }
82 }
83
84 pattern RNAChainElement(oOfPrev:O, oOfNext:O, c1:C)
85 {
86    oOfPrev --> p:P;
87    p --> do:O;  p --> do;  p --> :O --> :H;
88    p --> :O --> c5;
89
90    // Ribose core
91    o:O --> c1 --> c2:C --> c3:C --> c4:C --> o;
92    c1 --> :H;  c2 --> :H;  c3 --> :H;  c4 --> :H;
93    c4 --> c5:C;
94    c5 --> :H;  c5 --> :H;
95    c2 --> :O --> :H;
96
97    c3 --> oOfNext;
98 }
99
100 pattern Nukleinbase(c1:C) {
101    alternative {
102       A {
103          a:Adenin(c1);
104
105          modify(rc1:C) {
```

```
106             :Uracil(rc1);
107          }
108       }
109       C {
110          c:Cytosin(c1);
111
112          modify(rc1:C) {
113             :Guanin(rc1);
114          }
115       }
116       G {
117          g:Guanin(c1);
118
119          modify(rc1:C) {
120             :Cytosin(rc1);
121          }
122       }
123       T {
124          t:Thymin(c1);
125
126          modify(rc1:C) {
127             :Adenin(rc1);
128          }
129       }
130    }
131
132    modify(rc1:C) {
133    }
134 }
135
136 pattern Adenin(rc1:C) {
137    rc1 --> n1:N --> c1:C --> n2:N --> c2:C --> c3:C --> n1;
138    c1 --> n2;  c2 --> c3;
139    c1 --> :H;
140    c2 --> c4:C --> n3:N --> c5:C --> n4:N --> c3;
141    c4 --> n3;  c5 --> n4;
142    c5 --> :H;
143    c4 --> n5:N;  n5 --> :H;  n5 --> :H;
144 }
145
146 pattern Cytosin(rc1:C) {
147    rc1 --> n1:N --> c1:C --> c2:C --> c3:C --> n2:N --> c4:C --> n1;
148    c1 --> c2;  c3 --> n2;
149    c1 --> :H;
150    c2 --> :H;
151    c3 --> n3:N;  n3 --> :H;  n3 --> :H;
152    c4 --> o:O;  c4 --> o;
153 }
154
```

```
155 ...
```

A.5 Umwandlung DNA von abstrakt nach konkret

Durch die nachfolgende GrGen-Spezifikation werden DNA-Ketten in der abstrakten Modellierung von Beispiel A.3 in DNA-Ketten in der konkreten Modellierung auf Ebene der Atome und Bindungen von Beispiel A.4 umgewandelt. Aus Platzgründen wird nur ein Teil gezeigt, die vollständige Spezifikation ist im `examples`-Verzeichnis des GrGen.NET-Programmpaketes zu finden. Zur Umgehung von Namenskollisionen wurden die Knotenklassen der Atome mit dem Präfix A_ versehen.

```
1  rule A2K // transform abstract dna chain to atoms
2  {
3     start:D;
4
5     negative {
6        -:PG-> start;
7     }
8
9     n:A2KNucleotide(start);
10    a2k:A2K(start);
11
12    modify {
13       :A_H --> :A_O --> c3:A_C;
14       c1:A_C; c5:A_C;
15       :DesoxyriboseCore(c1, c3, c5);
16       n(c1);
17       a2k(c5);
18    }
19 }
20
21 pattern A2K(prev:D)
22 {
23    alternative
24    {
25       Chain {
26          prev -:PG-> next:D;
27          n:A2KNucleotide(next);
28          a2k:A2K(next);
29
30          modify(prevC5:A_C) {
31             :PhosphateGroup(prevC5, p);
32             p:A_P --> :A_O --> c3:A_C;
33             c1:A_C; c5:A_C;
34             :DesoxyriboseCore(c1, c3, c5);
```

```
35              n(c1);
36              a2k(c5);
37           }
38        }
39        End {
40           negative {
41              prev -:PG-> next:D;
42           }
43
44           modify(prevC5:A_C) {
45              :PhosphateGroupEnd(prevC5);
46           }
47        }
48     }
49
50     modify(prevC5:A_C) {
51     }
52  }
53
54  pattern A2KNucleotide(d:D)
55  {
56     alternative {
57        A {
58           d --> :A;
59
60           modify(c1:A_C) {
61              :Adenin(c1);
62           }
63        }
64        C {
65           d --> :C;
66
67           modify(c1:A_C) {
68              :Cytosin(c1);
69           }
70        }
71        G {
72           d --> :G;
73
74           modify(c1:A_C) {
75              :Guanin(c1);
76           }
77        }
78        T {
79           d --> :T;
80
81           modify(c1:A_C) {
82              :Thymin(c1);
83           }
```

```
84        }
85        U { // corrupt DNA
86           d --> :U;
87
88           modify(c1:A_C) {
89              :Uracil(c1);
90           }
91        }
92        H { // corrupt DNA
93           d --> :H;
94
95           modify(c1:A_C) {
96              :HydroxyGuanin(c1);
97           }
98        }
99     }
100
101    modify(c1:A_C) {
102    }
103 }
104
105 pattern DesoxyriboseCore(c1:A_C, c3:A_C, c5:A_C)
106 {
107    o:A_O --> c1 --> c2:A_C --> c3 --> c4:A_C --> o;
108    c1 --> :A_H;  c2 --> :A_H;  c3 --> :A_H;  c4 --> :A_H;
109    c4 --> c5;
110    c5 --> :A_H;  c5 --> :A_H;
111    c2 --> :A_H;
112 }
113
114 pattern PhosphateGroup(c5:A_C, p:A_P)
115 {
116    c5 --> :A_O --> p;
117    p --> do:A_O;  p --> do;  p --> :A_O --> :A_H;
118 }
119
120 pattern Adenin(rc1:A_C) {
121    rc1 --> n1:A_N --> c1:A_C --> n2:A_N --> c2:A_C --> c3:A_C --> n1;
122    c1 --> n2;  c2 --> c3;
123    c1 --> :A_H;
124    c2 --> c4:A_C --> n3:A_N --> c5:A_C --> n4:A_N --> c3;
125    c4 --> n3;  c5 --> n4;
126    c5 --> :A_H;
127    c4 --> n5:A_N;  n5 --> :A_H;  n5 --> :A_H;
128 }
129
130 pattern Cytosin(rc1:A_C) {
131    rc1 --> n1:A_N --> c1:A_C --> c2:A_C --> c3:A_C -->n2:A_N-->c4:A_C-->n1;
132    c1 --> c2;  c3 --> n2;
```

```
133     c1 --> :A_H;
134     c2 --> :A_H;
135     c3 --> n3:A_N;  n3 --> :A_H;  n3 --> :A_H;
136     c4 --> o:A_O;  c4 --> o;
137 }
138
139 ...
```

Literaturverzeichnis

[Bat06] Batz, Gernot V.: *An Optimization Technique for Subgraph Matching Strategies*. Internal Report 2006-7, 2006. – "http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf"

[BG08] Blomer, Jakob ; Geiss, Rubino: *The GrGen.NET User Manual*. http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf. Version: 2008

[BKG08] Batz, Gernot V. ; Kroll, Moritz ; Geiss, Rubino: *A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching*. Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings, 2008. – "http://www.info.uni-karlsruhe.de/papers/agtive_2007_search_plan.pdf"

[Buc08] Buchwald, Sebastian: *Erweiterung von GrGen.NET um DPO-Semantik und ungerichtete Kanten*. Studienarbeit, 2008. – "http://www.info.uni-karlsruhe.de/papers/sa_buchwald.pdf"

[BV06] Balogh, András ; Varró, Dániel: *Pattern composition in graph transformation rules*. European Workshop on Composition of Model Transformations, 2006. – "http://home.mit.bme.hu/~varro/publication/2006/cmt2006_bv.pdf"

[CEH+97] Corradini, Andrea ; Ehrig, Hartmut ; Heckel, Reiko ; Korff, Martin ; Löwe, Michael ; Ribeiro, Leila ; Wagner, Anika: *Algebraic Approaches to Graph Transformation - Part I: Single Pushout Approach and Comparison with Double Pushout Approach*. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, World Scientific, 1997

[Den07] Denninger, Oliver: *Erweiterung des Kantenkonzepts deklarativer Graphersetzungssysteme von Einfachkanten über Hyperkanten zu „Superkanten"*. Diplomarbeit, 2007. – "http://www.ipd.uka.de/Tichy/uploads/arbeiten/149/diplomarbeit.pdf"

[DHJ+06] Drewes, Frank ; Hoffmann, Berthold ; Janssens, Dirk ; Minas, Mark ; Eetvelde, Niels V.: *Adaptive Star Grammars.* Proceedings 3rd Int. Conf. on Graph Transformation (ICGT'06), 2006. – "http://www.informatik.uni-bremen.de/~hof/papers/06-ICGT.pdf"

[DHJ+08] Drewes, Frank ; Hoffmann, Berthold ; Janssens, Dirk ; Minas, Mark ; Eetvelde, Niels V.: *Shaped Generic Graph Transformation.* Applications of Graph Transformation with Industrial Relevance (AGTIVE'07) Selected Papers, 2008. – "http://www.informatik.uni-bremen.de/~hof/papers/07-AGTIVE.pdf"

[DHM08] Drewes, Frank ; Hoffmann, Berthold ; Minas, Mark: *Adaptive Star Grammars for Graph Models.* Proceedings 4th Int. Conf. on Graph Transformation (ICGT'08), 2008. – "http://www.informatik.uni-bremen.de/~hof/papers/08-ICGT.pdf"

[Ehr79] Ehrig, Hartmut: *Introduction to the Algebraic Theory of Graph Grammars (A Survey).* Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology, 1979

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam M.: *GrGen: A Fast SPO-Based Graph Rewriting Tool.* Proceedings 3rd Int. Conf. on Graph Transformation (ICGT'06), 2006. – "http://www.info.uni-karlsruhe.de/papers/grgen_icgt2006.pdf"

[Gei08] Geiss, Rubino: *Graphersetzung mit Anwendungen im Übersetzerbau.* Dissertation, 2008

[Hab92] Habel, Annegret: *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science, 1992

[HJG08] Hoffmann, Berthold ; Jakumeit, Edgar ; Geiss, Rubino: *Graph Rewrite Rules with Structural Recursion.* 2008. – "http://www.info.uni-karlsruhe.de/papers/GCM2008.pdf"

[Jak07] Jakumeit, Edgar: *Vorarbeiten für die Erweiterung des Graphersetzungssystems GrGen um dynamisch zusammengesetzte Muster.* Studienarbeit, 2007. – "http://www.info.uni-karlsruhe.de/papers/sa_jakumeit.pdf"

[Kro07] Kroll, Moritz: *GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen.* Studienarbeit, 2007. – "http://www.info.uni-karlsruhe.de/papers/sa_kroll.pdf"

[Pra71] Pratt, Terrence W.: *Pair Grammars, Graph Languages and String-to-Graph Translations*. 1971. – "Journal of Computer and System Sciences, Volume 5, pages 560–595"

[Via08] Viatra Development Team: *The VIATRA 2 Model Transformation Framework Users Guide*. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/viatratut_October2006.pdf`. Version: 2008