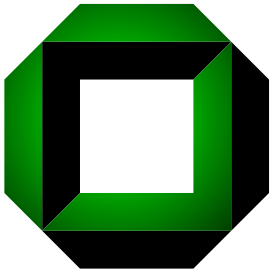


Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

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

Vorarbeiten für die Erweiterung des Graphersetzungssystems GrGen um dynamisch zusammengesetzte Muster

Studienarbeit von Edgar Jakumeit

September 2007

Betreuer:
Dipl.-Inform. 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

GrGen.NET ist ein Graphersetzungssystem, das es dem Nutzer erlaubt, sein Anwendungsgebiet durch gerichtete, typisierte und attributierte Multigraphen zu modellieren, sowie Vorgänge auf diesen durch Graphersetzungsregeln, kombiniert in Graphersetzungssequenzen zu spezifizieren. Bei der Anwendung einer Regel wird eine Instanz ihres Mustergraphen, so sie denn im Arbeitsgraphen gefunden wurde, durch eine Instanz ihres Ersetzungsgraphen überschrieben.

Die einfachen aus Knoten und Kanten bestehenden Muster haben sich für manche Aufgaben als zu unübersichtlich und unflexibel erwiesen; gewünscht wurden: eine Strukturierungsmöglichkeit für komplexe Muster, Muster mit variablen Bestandteilen und schließlich Muster mit einer statisch nicht festgelegten Anzahl von Wiederholungen von Teilmustern.

Um den oben genannten Unzulänglichkeiten entgegen zu wirken, wurde im Rahmen dieser Arbeit eine Erweiterung der Sprache von GrGen.NET um eingebettete und alternative Muster entwickelt und eine Anpassung des Mustersuchalgorithmus von GrGen.NET an die Erweiterung entworfen. Als Vorarbeit zu deren Umsetzung wurde die Codeerzeugung unter Einführung einer Zwischenschicht reimplementiert.

Aufbau: In Kapitel 2 wird die Regelbeschreibungssprache von GrGen.NET vorgestellt, in Kapitel 3 wird ihre Erweiterung beschrieben, in Kapitel 4 wird die Implementierung von GrGen.NET erläutert, in Kapitel 5 wird das Einziehen der Codeerzeugungszwischenschicht kurz angerissen und in Kapitel 6 schließlich wird der Entwurf für die Erweiterung vorgestellt.

Kapitel 2

Graphersetzung mit GrGen

Aufbau Graphen

Graphen sind eine abstrakte und dennoch intuitive Form der Datenorganisation, in der Beziehungen zwischen Entitäten durch Geflechte von durch Kanten verbundenen Knoten dargestellt werden.
Sie existieren in verschiedenen Formen, für uns interessant sind gerichtete, typisierte und attributierte Multigraphen, da diese das Metamodell bilden, das dem GrGen-System zugrunde liegt.

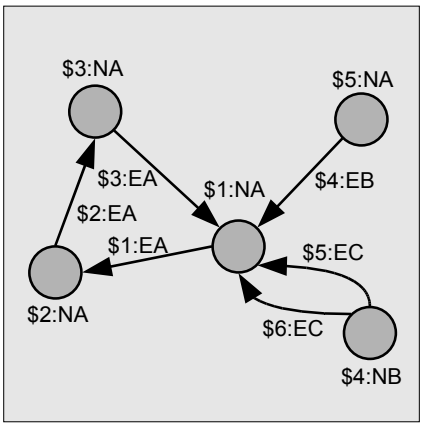
Gerichtet heißt, dass eine Kante von einem ausgezeichneten Quell- zu einem ausgezeichneten Zielknoten führt.

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.

Knoten und Kanten wollen wir zusammenfassend als Graphelemente bezeichnen. Die Typen sind in einer Hierarchie analog den Klassen in objektorientierten Programmiersprachen angeordnet; sämtliche Knotentypen sind vom Basistyp Node, sämtliche Kantentypen vom Basistyp Edge abgeleitet. Zur Veranschaulichung ist in Abbildung 2.1 links ein Graph skizziert, wie ihn die rechts stehende GrGen-Regel aus dem Nichts heraus materialisieren würde.




```
1  rule materialize {
2    pattern {
3    }
4    replace {
5      n1:NA-:EA->n2:NA;
6      n2-:EA->n3:NA;
7      n3-:EA->n1;
8      :NA-:EB->n1;
9      n4:NB-e1:EC->n1;
10     n4-e2:EC->n1;
11   }
12 }
```

Abbildung 2.1: Graph mit Regel

Graphverarbeitung

Nachdem wir nun den Aufbau der Graphen kennen, wenden wir uns ihrer Verarbeitung zu. Diese erfolgt durch die Anwendung von Regeln (Produktionen) $p : L \to R$, bestehend aus einem Mustergraphen L und einem Ersetzungsgraphen R. Bei der Anwendung einer Regel $L \to R$ wird im Arbeitsgraphen G eine Instanz von L gesucht und durch eine Instanz von R ersetzt, mit einem veränderten Arbeitsgraphen G' als Ergebnis. Das Vorgehen ist in Abbildung 2.2 abstrakt skizziert.

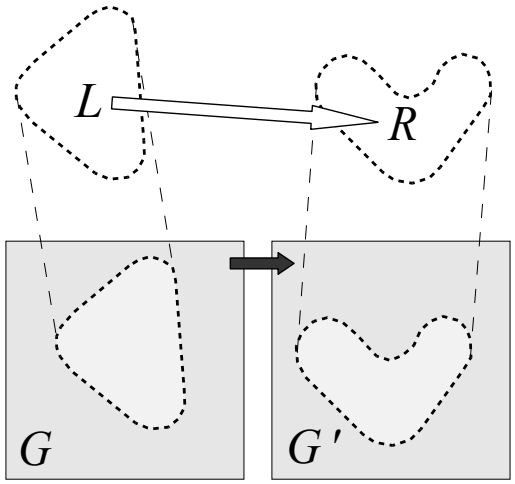


Abbildung 2.2: Graphersetzung abstrakt

Eine gefundene Instanz heißt Passung (match) m von L in G, formal wird sie durch einen Graphmorphismus vom Mustergraphen in den Arbeitsgraphen, genauer: einen strukturell gleichen und typverträglichen Teilgraphen des Arbeitsgraphen, beschrieben.

Die Instanz von R wird bestimmt mit Hilfe einer Entsprechen-sich-Zuordnung r von Graphelementen aus L nach R, einem partiellen Graphmorphismus: In G bleiben die, als sich entsprechend festgelegten Elemente aus der Passung m erhalten; die nur in L vorhandenen werden aus G gelöscht, und die nur in R vorhandenen werden zu G hinzuinstanziiert.

Abbildung 2.3 konkretisiert die abstrakte Abbildung 2.2, die zugehörige GrGen-Regel wird in Abbildung 2.4 aufgelistet und in 2.5 schließlich wird die Passung der Musterelemente auf die Arbeitsgraphelemente tabellarisch angegeben (Musterelemente in `Schreibmaschinenschrift`).

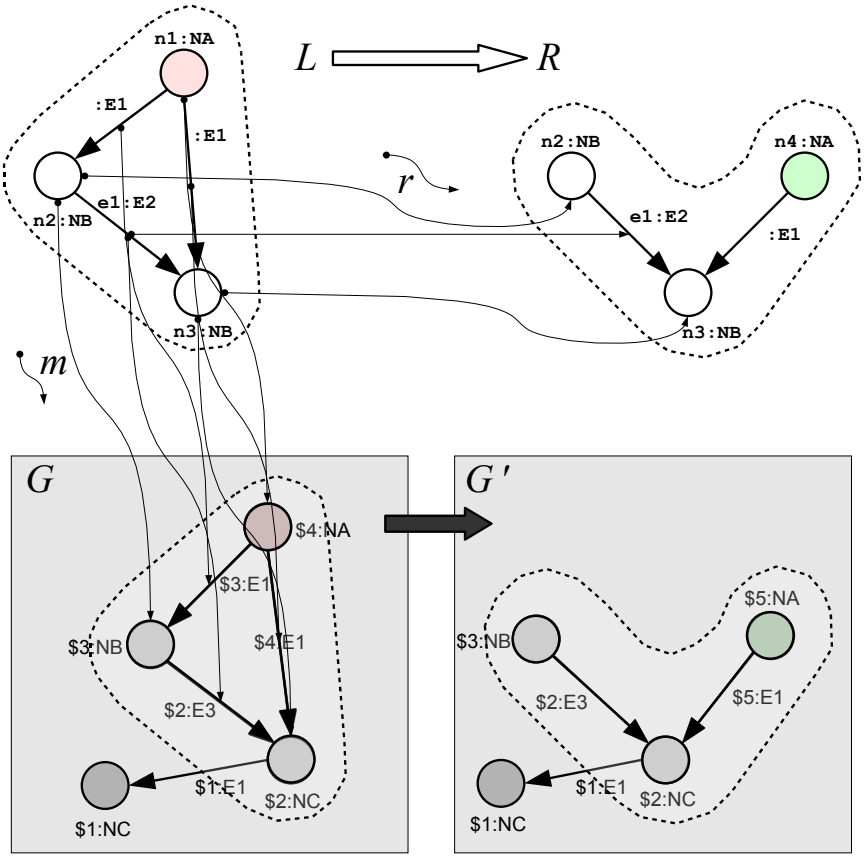


Abbildung 2.3: Graphersetzung konkret

```
1  rule foo {
2    pattern {
3      n1:NA;
4      n1-:E1->n2:NB;
5      n1-:E1->n3:NB;
6      n2-e1:E2->n3;
7    }
8    replace {
9      n4:NA;
10     n4-:E1->n3;
11     n2-e1:E2->n3;
12   }
13 }
```

Abbildung 2.4: 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 2.5: Passung Musterelemente auf Arbeitsgraphelemente

Konflikte

Beim skizzierten Vorgehen kann es zu folgenden Konflikten kommen:

- 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.

Diese Konflikte werden in GrGen gemäß dem Formalismus der Single Pushout Graphersetzung durch folgende Festlegungen gehandhabt:

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

Für diejenigen, denen der zweite Konfliktfall nicht ersichtlich ist, möchte ich hervorheben, dass die Passung nicht zwangsläufig injektiv ist – sie ist ein Homomorphismus, verschiedene Knoten des Mustergraphen können auf den gleichen Knoten des Arbeitsgraphen abgebildet werden. Dieses Verhalten ist bei vielen Knoten, die potentiell zusammenfallen können, schwierig zu überblicken, der Regelschreiber kann 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.

Weitere Sprachelemente

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.

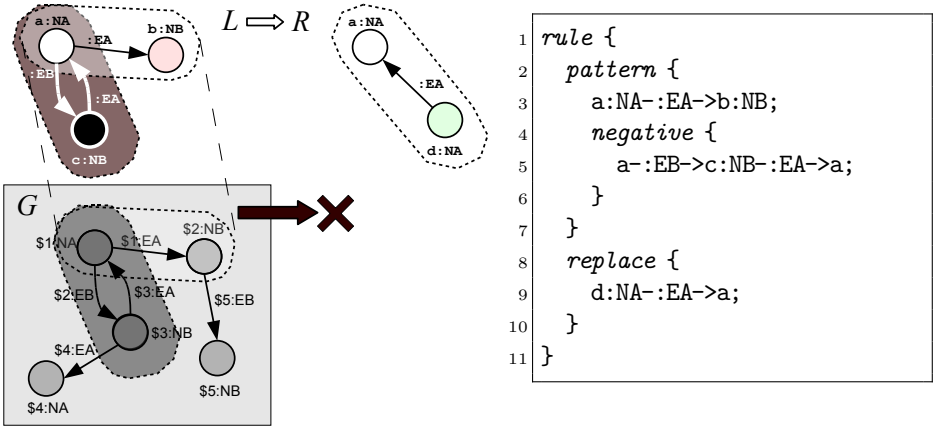


Abbildung 2.6: GrGen-Graph mit NAC grafisch und textuell

Die zulässigen Muster können in einem `if`-Block innerhalb des Musters mit zusätzlichen Typ- und Attributbedingungen eingeschränkt werden, des Weiteren können in einem `eval`-Block innerhalb der Ersetzung Attribute neu berechnet und zugewiesen werden.

Bei der Nutzung von GrGen für komplexere Graphverarbeitungsaufgaben werden Graphersetzungsregeln aus Graphersetzungssequenzen heraus aufgerufen, in diesen werden die einzelnen Regeln kombiniert, ihre jeweilige Anwendung wird in Abhängigkeit von Erfolg oder Fehlschlag der Passungssuche gesteuert. 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.

Operatoren S_1 *op* S_2**:** Mit Operatoren werden Sequenzen kombiniert, sie werten ihre Argumente von links nach rechts aus, man unterscheidet strikte und faule Operatoren. Die strikten werten alle ihre Argumente aus und bestimmen dann ihren Rückgabewert, die faulen brechen die Argumentauswertung ab, sobald ihr Rückgabewert aufgrund des Wertes von einem der Argumente feststeht. An Operatoren stehen unter anderem zur Verfügung: Das logische Und – die Operation schlägt

fehl, wenn eines der Argumente ein Fehlschlag war, ansonsten ist sie erfolgreich; && für faul, & für strikt. Das logische Oder – die Operation ist erfolgreich, wenn eines der Argumente ein Erfolg war, ansonsten schlägt sie fehl; || für faul, | für strikt.

Transaktionsklammern $< S >$**:** Die Effekte der erfolgreichen Regelanwendungen innerhalb der von den Transaktionsklammern umschlossenen Sequenz werden bei einem Fehlschlag der Sequenz rückgängig gemacht.

Iteration $S+$**:** Die zu iterierende Sequenz wird so lange ausgeführt, bis sie das erste mal fehlschlägt; die Iteration schlägt fehl, wenn die zu iterierende Sequenz keinmal erfolgreich ausgeführt wurde.

Variablen v**:** Regeln können mit Ein- und Ausgabeparametern versehen werden, über die eine Regel Knoten und Kanten vor ihrer Anwendung erhalten sowie nach ihrer Anwendung zurückgeben kann. Diese dienen der Kombination von Regeln, ihre Werte können zwischen den Anwendungen in Variablen zwischengespeichert werden.

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.

Für eine genaue Definition des in diesem Kapitel nur kurz gestreiften möchte ich den Leser auf [BG07] oder [Gei07] verweisen.

Kapitel 3

Erweiterungen

Mit den Sprachmitteln, die im vorangegangenen Kapitel vorgestellt wurden, lässt sich bereits erfolgreich Graphersetzung betreiben – dennoch sind bei der Nutzung verschiedene Unzulänglichkeiten zu Tage getreten. Diese wollen wir im Folgenden vorstellen, jeweils zusammen mit einer Erweiterung der Sprache zu ihrer Behebung.

3.1 Eingebettete Muster

Problem: Bei komplexen Mustern wird die Beschreibung ihres Aufbaus aus Knoten und Kanten schnell unübersichtlich, der Nutzer verliert den Überblick in all den Details.
Lösung: Eine weitere Abstraktionsebene einführen, auf der die Details hinter neuen Grundelementen verborgen werden können. Hierzu bietet sich das Abstraktionsmittel Teilgraph an, bei dem Knoten und Kanten zu einem Teilmuster zusammengefasst werden und dann einen Namen erhalten. Komplexe Muster können nun aus den Teilmustern, anhand der Nennung von deren Namen, zusammengesetzt werden.

Teilmusterspezifikation und Deklaration

Konkret umgesetzt wird das oben vorgestellte Vorgehen durch die Einführung der Sprachmittel Teilmusterspezifikation und Deklaration. Eine Teilmusterspezifikation, in EBNF
`SubPattern ::= "pattern" Name "{" PatternBody "}"`
entspricht weitgehend einer Musterspezifikation innerhalb einer Regel. Im Gegensatz zu dieser kommt sie aber nur außerhalb einer Regelspezifikation vor, weshalb sie in ihrem Rumpf auch keine Regeleingaben entgegennehmen und keine Regelausgaben aushändigen kann, dafür verfügt sie zusätzlich noch über einen Namen. Durch sie wird ein Teilmustertyp definiert.

```
pattern Foo {
  anf:NA-->i1:NB-->end:NC;
  anf-->i2:NB-->end;
}
```

Ein solchermaßen definierter Teilmustertyp kann dann in einer Teilmusterdeklaration zum Einführen eines Teilmusters verwendet werden, in EBNF:
`Declaration ::= EntityName ":" TypeName "(" ")"`
Eine Teilmusterdeklaration spezifiziert im Muster, welches sie enthält, das Auftreten einer Teilmusterentität mit dem angegebenen Namen und dem angegebenen Teilmustertypen. Somit können Teilmuster innerhalb einer Musterspezifikation als Bestandteile neben Knoten und Kanten auftreten, eingeführt durch Teilmusterdeklarationen neben den Knoten- und Kantendeklarationen.

```
pattern Bar {
  a:NA-->e:NC;
  foo:Foo();
}
```

Einbettung durch Anknüpfungen

Die leeren Klammern hinter den Teilmusternamen wecken die Erwartung nach Parametern - und in der Tat ist der Platz zwischen den Klammern für eine Art von Parametern vorgesehen. Für gewöhnlich sind die Teilmuster nämlich mit dem umschließenden Muster verbunden, und nicht nur, wie im obigen Beispiel, einfach im umschließenden Muster enthalten.

Um das Teilmuster mit seinem umgebenden Muster zu verbinden, benötigen wir Schnittstellenelemente, die in beiden bekannt sind. Diese können aber nur in einem von ihnen enthalten sein – als am praktikabelsten hat sich die Festlegung „Schnittstellenelemente sind im umgebenden Muster enthalten“ erwiesen, an die im Teilmuster dann angeknüpft wird.

Anknüpfungen werden in der Teilmusterspezifikation als Liste von Knoten- und Kantendeklarationen notiert:
`SubPattern ::= "pattern" Name "(" Connections ")" "{" Body "}"`
`Connections ::= (NodeDeclaration|EdgeDeclaration)||","`
Die deklarierten Graphelemente sind Verweise auf Graphelemente aus dem Musterkontext, das Teilmuster kann sie benutzen, indem es Verbindungen zu ihnen spezifiziert oder Attributbedingungen prüft, sie aber nicht im Ersetzungsteil verändern, da sie nicht in ihm enthalten sind.

```
pattern Foo(anf:NA, end:NC) {
  anf-->i1:NB-->end;
  anf-->i2:NB-->end;
}
```

Im umschließenden, die Teilmusterspezifikation über eine Teilmusterdeklaration verwendenden Muster, wird innerhalb der Klammern der Teilmusterdeklaration eine Liste von in diesem Muster sichtbaren Graphelementen angegeben, in EBNF

```
Declaration ::= EntityName ":" TypeName "(" Element||"," ")"
```

Die Verbindung erfolgt durch Bindung der Anknüpfungen aus der Teilmusterspezifikation an die, bei der Teilmusterdeklaration angegebenen Elemente aus dem, die Teilmusterdeklaration enthaltenden Muster. 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.

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

Zur Verdeutlichung: Wir haben hier keine Ein- und Ausgabeparameter, die Arbeitsgraphelemente enthalten, die in einer Vorgängerregel für eine Nachfolgerregel gepasst wurden, vor uns, sondern Teilmuster, die an ein umschließendes Muster anknüpfen und sich über dieses untereinander verbinden. Außerdem muß, im Gegensatz zu einer Regelanwendung, in der Teilmusterdeklaration jede Anknüpfung gebunden werden.

Häufig wird eine Kante als Anknüpfung benutzt um einen Knoten aus dem umschließenden Muster mit einem Knoten aus dem Teilmuster zu verbinden, dem Teilmuster können aber auch Knoten des umschließenden Musters genannt werden. Beides wird im folgenden Beispiel dargestellt. Darüber hinaus kann es notwendig werden, einen Knoten des Teilmusters in das umschließende Muster auszulagern; dieses Vorgehen ist erforderlich, wenn er in unterschiedlichen Kontexten mit einer unterschiedlichen Anzahl von Kanten verbunden werden soll.

```
1  pattern Foo {
2    a1:NA-e:Edge->;
3    bla:Bla(e, ci);
4    ci:NC;
5    :Blo(ci);
6  }
7  pattern Bla(-e:Edge->, end:NC) {
8    -e->beg:NA-->i1:NB-->end;
9    beg-->i2:NB-->end;
10 }
11 pattern Blo(a:NC) {
12   a-->:NA-->:NA-->a;
13 }
```

Semantik

Die Semantik eines spezifizierten Musters bei der Anwendung einer Regel bestand bisher aus dem Suchen nach einer Passung des Musters im Arbeitsgraphen. An der Passungsbestimmung ändert sich nichts, jedoch wird nun das zu passende Gesamtmuster erst davor aus den einzelnen Teilmustern zusammengesetzt. Das kann in einem eigenen Schritt vollständig vor der Passungssuche geschehen, aber auch mit dieser verwoben sein. Es wird – wenn existent – eine Passung des zusammengesetzten Gesamtmusters geliefert, bestehend aus der Teilpassung des umschließenden Musters und den Teilpassungen aller enthaltenen Teilmuster. Direkt oder indirekt rekursive Muster sind derzeit noch verboten (was sich in Kapitel 3.3 ändern wird), da mit den aktuellen Sprachmitteln nur ein unendlich großes Muster spezifiziert werden könnte, bzw. die Zusammensetzung nicht terminieren würde.

Eingebettete Muster konnten bisher schon über Graphersetzungssequenzen realisiert werden. Dazu wird die Ersetzungsregel mit dem komplexen Muster entlang der Teilmuster in mehrere Teilregeln aufgespalten; durch Schiffchen im Arbeitsgraphen oder die neueren Regelparameter ist dabei sicherzustellen, dass die Teilregeln den gleichen Ansatz bearbeiten. Die Teilregeln werden dann als Abfolge innerhalb einer Transaktion ausgeführt, beispielsweise Muster A mit Teilen B und C in der GRS `<A&B&C>`.

Gegenüber der Erweiterung der Musterbeschreibungssprache weist dieses Vorgehen mehrere Nachteile auf:

- Es basiert auf einer operationalen Abfolge von Regelanwendungen, wohingegen wir uns eine deklarative Spezifikation des Musteraufbaus wünschen.
- Die Teilregeln müssen immer in der festgelegten Reihenfolge aufgerufen werden, für die die Parameterübergabe ursprünglich entwickelt wurde.
- Die Ersetzung wird gleich nach dem erfolgreichen Passen des Musterteiles angewandt, dabei können Knoten, die eigentlich zum Verbinden der Regelteile benötigt werden, verloren gehen; außerdem müssen diese vorschnellen Ersetzungen zurückgerollt werden, wenn sich später herausstellt, dass die Regelabfolge nicht erfolgreich angewendet werden kann.

Kurz: Teilmuster über Graphersetzungssequenzen zu realisieren ist ineffizient – sowohl für den Menschen als auch die Maschine.

Eingebettete Teilmuster verbessern nicht nur den Überblick, sondern ermöglichen darüber hinaus auch noch die einfache Wieder- und Mehrfachverwendung von einmal spezifizierten Teilmustern über Teilmusterdeklarationen, anstelle des sonst bei jeder Nutzung nötigen Kopierens und Anpassens der Teilmusterspezifikation von Hand.

Ein längeres, anwendungsnäheres Beispiel befindet sich im Anhang in Abschnitt A.1, ein weiteres Beispiel mit Kantenanknüpfungen in A.3, dieses enthält aber auch bis jetzt noch nicht spezifizierte Erweiterungen.

3.2 Alternative Muster

Problem: Feste Muster sind zu unflexibel.
Lösung: Muster mit variablen Bestandteilen.

Derzeitige Musterbeschreibungen sind starr, mit ihnen wird jeweils genau ein Muster spezifiziert, von homomorphem Zusammenfallen abgesehen. In der Anwendung jedoch werden flexiblere Musterbeschreibungen benötigt, mit denen man auch Varianten eines Musters abdecken kann.

Muster mit Varianten können, ebenso wie Teilmuster, über einen Umweg mit Regeln und Graphersetzungssequenzen realisiert werden. Dazu stellt man für jede gewünschte Variante eine eigene Regel auf und führt diese dann innerhalb einer Graphersetzungssequenz disjunktiv verknüpft aus, beispielsweise als GRS `A|B|C` für die Varianten A,B,C. Doch ebenso wie bei den eingebetteten Mustern ist eine deklarative Musteraufbaubeschreibung einer operationalen Graphersetzungssequenz vorzuziehen.

Um Muster mit Varianten beschreiben zu können, führen wir das Sprachmittel der Alternative ein, die Syntax in EBNF ist gegeben durch
`Alternative ::= "alternative" [Name] "{" Case* "}"`
`Case ::= Name "{" PatternBody "}"`
Die Alternative kann an einer beliebigen Stelle im Muster eingeführt werden, auf das Schlüsselwort `alternative` folgt der Name der Alternativen, wird er weggelassen, wird „Alternative“ als Vorgabe angenommen. Eine Alternative 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 }
```

Angedacht waren Varianten als unterschiedliche Rümpfe einer Musterspezifikation, entsprechend der Nutzung der Alternative im obigen Beispiel, aber auf expliziten Nutzerwunsch hin sind sie ein eigenständiges Sprachmittel, das eingebettet innerhalb eines Muster auftreten 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 mög-

lich, 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 }
```

Ein längeres, anwendungsnäheres Beispiel befindet sich im Anhang in Abschnitt A.2.

Zur Semantik: Aus einer Musterspezifikation mit Alternativen werden nun, vollständig vor oder stückweise während der Passungssuche, unterschiedliche resultierende Gesamtmuster zusammengesetzt. Mit jedem Alternativenzweig mehr wird auch ein zusammengesetztes Gesamtmuster mehr gebildet, bei mehreren Alternativen entsprechend der Multiplikation der Anzahl der Zweige der einzelnen Alternativen. Jedes der resultierenden Gesamtmuster 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).

3.3 Wiederholte Muster

Problem: Muster mit einer statisch nicht bekannten Anzahl von Wiederholungen eines Teilmusters können nicht spezifiziert werden.
Lösung: Rekursive Muster.

Muster mit einer im Voraus bekannten Anzahl k von Wiederholungen eines Teilmusters können durch Ausfalten von Hand angegeben werden – das ist zwar nur für ganz kleine k praktikabel, aber zumindest theoretisch immer möglich. Bei Mustern mit einer unbekannten Anzahl n von Wiederholungen muss das Ausfalten hingegen irgendwann abgebrochen werden, sagen wir bei K – und sobald wir auf einen Graphen mit $K + 1$ Wiederholungen treffen versagt das Muster.

Doch mit den eingebetteten und den alternativen Mustern halten wir bereits alle Mittel in der Hand um das Problem zu lösen: die Interaktion der beiden Sprachmerkmale erlaubt rekursive Muster. 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.

```
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

Mit rekursiven Mustern kann der Nutzer nun unendlich viele zusammengesetzte Muster spezifizieren, zu denen Passungen zu suchen sind. Beim Zusammensetzen, statisch vor oder dynamisch während der Passungsbestimmung, werden die Gesamtmuster rekursiv aufgezählt. Da der Arbeitsgraph endlich ist, ist die Aufgabe dennoch *effektiv* lösbar, 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*.

Bemerkungen

Ein explizites Iterationskonstrukt der Art „Zulässig ist eine Kette aus dem gegebenen Teilmuster, verschweißt an den ausgezeichneten Anfangs- und Endpunkten der Kettenglieder“ war der eigentliche Ausgangspunkt dieser Arbeit. Im ihrem Verlauf kam aber der Wunsch nach eingebetteten und alternativen Mustern hinzu, und wie gesehen sind bereits durch deren Interaktion Wiederholungen möglich. Da diese auch noch allgemeiner als das Iterationskonstrukt sind (verzweigende Muster, vom Kettenstück unterschiedenes Endglied), wurde der iterative Ansatz zu Gunsten des rekursiven Ansatzes fallen gelassen.

Sind für eine Kette (so wie sie im letztaufgeführten Beispiel spezifiziert wurde) Anfang und Ende über Anknüpfungen gegeben, erhalten wir als Passung eine vollständige Kette zwischen diesen Punkten oder überhaupt keine Passung. 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 möge die folgende Kette ohne vorzeitigen Abbruch dienen.

```
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 }
```

Nach diesem Beispiel für die Interaktion von eingebetteten, alternativen und negativen Mustern wollen wir noch einen kurzen 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. Den Effekt, dass auf ein Graphelement in verschiedenen Teilmustern (deren Instanzen) über unterschiedliche Namen zugegriffen werden kann, haben wir bereits durch die Anknüpfungen erhalten.

Elemente innerhalb eines Teilmusters können weiterhin ebenso wie Elemente innerhalb eines Regelmusters über die `hom`-Deklaration als homomorph zueinander spezifiziert werden. Das Element wird aber nur in dem Muster, in dem es deklariert wurde, gepasst und kann auch nur in dessen Ersetzung verändert werden.

Wir wollen diesen Abschnitt mit einem kleinen Beispiel, welches das Zusammenspiel von Anknüpfungen und rekursiven Mustern beleuchtet, schließen. Im Gegensatz zur vorangegangenen Kette, in der die Anknüpfung mit jedem Ausfalten weitergesetzt wird, um das Ende des vorangehenden Gliedes mit dem Anfang des folgenden Gliedes zu verbinden, wird hier die Anknüpfung konstant gehalten, um ausgehend von ihr wiederholt ein Teilmuster anzubringen, versinnbildlicht im Kopf einer Pusteblume.

```
pattern Blowball(head:Node) {
  alternative {
    Base {
      negative {
        head-->:Node;
      }
    }
    Recursive {
      head-->:Node;
      :Blowball(head);
    }
  }
}
```

3.4 Ersetzungsgraph

Ich habe in den vorangegangenen Abschnitten Erweiterungen der Beschreibung des Mustergraphen vorgestellt, im Folgenden möchte ich deren Auswirkungen auf die Beschreibung des Ersetzungsgraphen vorstellen; beginnen werde ich meine Ausführungen jedoch mit einer Bemerkung zum syntaktischen Verhältnis zwischen einer Musterbeschreibung und einer Ersetzungsbeschreibung.

Blockschachtelung

Problem: Verwirrung der blockschachtelungsgewohnten Nutzer durch die GrGen-Syntax.

Der Namensraum des Musters einer Regel mit den darin definierten Graphelementen wird in der Ersetzung eingeblendet – ohne dass die Ersetzung im Muster syntaktisch geschachtelt wäre. Diese Fernwirkung widerspricht der Intuition der Nutzer, insbesondere da die aus den Programmiersprachen der C-Familie bekannten geschweiften Klammern eine Deutung als Block geradezu aufzwingen, und da ihr sonstiges Auftreten in einer Regelspezifikation auch tatsächlich einen Block anzeigt.

Lösung: Die Ersetzung syntaktisch im Muster schachteln.

Auf diese Weise wird klar ersichtlich, welche Elemente die Ausgangsbasis für die Veränderung darstellen. Prinzipiell kann der `replace`-Teil an jeder Stelle im Muster auftreten, aus Gründen der Lesbarkeit wird aber seine Positionierung am Ende des Musters empfohlen, wie im folgenden Beispiel.

```
rule R {
  pattern {
    beg:NA-->i1:NB-->end:NC;
    beg-->i2:NB-->end;

```

```
6     replace {
7       beg-->end;
8     }
9   }
10 }
```

Zusammengesetzte Ersetzung

Im Ersetzungsteil einer Regel konnte bisher spezifiziert werden, dass gepasste Musterelemente gelöscht werden oder erhalten bleiben sollen, zusätzlich konnten neue Elemente hinzugefügt werden. Diese Möglichkeiten wollen wir ebenfalls für Graphelemente eines Teilmustertyps anbieten, die Syntax soll der bereits bekannten folgen, 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    pattern {
3      foo:Foo();
4      bar:Bar();
5      replace {
6        foo; // bleibt erhalten
7        // bar nicht aufgeführt, wird gelöscht
8        blub:Blub(); // wird neu eingeführt
9      }
10   }
11 }
```

Die Verallgemeinerung der Ersetzungsmöglichkeiten Erhalten, Löschen, und Hinzufügen von Knoten und Kanten auf Teilmuster wird aber für ganze Teilmuster häufig zu grobschlächtig sein. Wir erlauben deshalb in einem Teilmuster selbst, so wie im Regelmuster, die Angabe einer `replace-` oder `modify`-Ersetzung, die das Teilmuster durch Ersetzungsoperationen auf seinen Elementen feinkörnig verändern kann.

Das Muster, welches das Teilmuster enthält, verwendet die im Teilmuster spezifizierte Ersetzung, indem es in seiner Ersetzung den Namen des Teilmusters, gefolgt von einer geöffneten und einer geschlossenen Klammer anführt (Intuition: Ersetzungsaufruf auf gepasstem Musterobjekt).

```
1  pattern Foo
2  {
3    beg:NA-->i1:NB-->end:NC;
4    beg-->i2:NB-->end;
5    replace {
6      beg-->end;
7    }
8  }
9  rule R {
10   pattern {
11     foo:Foo();
12     replace {
13       foo();
14     }
15   }
16 }
```

Wir folgen damit der Semantik der zusammengesetzten Muster und setzen den Ersetzungsgraph – in Abhängigkeit von der Struktur des Mustergraphen – zusammen. Die Teilmusterstruktur kann in einem Ersetzungsschritt nicht geändert werden, wohl aber der Inhalt eines jeden Teilmusters.

Ersetzungsanknüpfungen

Analog den Musteranknüpfungen in den Teilmustern können auch die Ersetzungen der Teilmuster Anknüpfungen erhalten, über die die einzelnen Ersetzungsteile miteinander verbunden werden. Die Anknüpfungen des Musters sind in der Ersetzung sicht-, aber nicht löschbar; sie können zum Hinzufügen von Kanten zu einem Knoten oder Ändern des Quell/Zielknotens verwendet werden. Im folgenden Beispiel werden zwei Muster durch eine Ersetzungsanknüpfung miteinander verbunden.

```
1  pattern Foo(beg:NA)
2  {
3    beg-->i1:NB-->end:NC;
4    beg-->i2:NB-->end;
5    replace(b:NB) {
6      beg-->b-->end;
7    }
8  }
9  pattern Bar
10 {
11   a:NA;
12   replace(b:NB) {
13     b-->a-->b;
14   }
15 }
16
```

```
rule R {
  pattern {
    a:NA-->a;
    foo:Foo(a);
    bar:Bar();
    replace {
      foo(b:NB);
      bar(b);
    }
  }
}
```

Ein längeres, anwendungsnäheres Beispiel befindet sich im Anhang in Abschnitt A.3.

Alternativen

Die Teilmuster in den Zweigen der Alternative sind, jedes für sich gesehen, gewöhnliche, geschachtelte Teilmuster, die sich potentiell in die Zusammensetzung des Gesamtmusters einbringen. Dementsprechend können sie auch einen lokal geschachtelten Ersetzungsteil erhalten, mit dem der Ersetzungsgraph zusammengebaut wird, wenn im Gesamtmuster dieser Zweig der Alternative gewählt wurde.

```
pattern Foo {
  beg:NA-->i1:NB-->end:NC;
  beg-->i2:NB-->end;

  alternative {
    Attachment {
      end-->e1:NA;
      end-->e2:NA;
      replace {
        end-->e1;
        end-->e2;
        e1-->e2; e2-->e1;
      }
    }
    Empty {
      replace {
      }
    }
  }

  replace {
    anf-->end;
  }
}
```

Im obigen Beispiel wurde das Spezifizieren von Ersetzungen zu den Mustern in den Zweigen einer Alternative gezeigt. Nun wollen wir, in gewisser Weise umgekehrt, die Benutzung einer Alternative aus einer Ersetzung heraus betrachten. Direkt innerhalb einer Ersetzung kann eine Alternative nicht auftreten, aber innerhalb einer Ersetzung können Teilmuster eingeführt werden, und ein solches Teilmuster kann potentiell Alternativen enthalten. Dieser als Ersetzungsoperation Hinzufügen bezeichnete Fall ist bei Teilmustern mit Alternativen komplexer als bei Mustern ohne solche, da zusätzlich zur Deklaration des einzufügenden Teilmusters noch die Entscheidung, welcher Alternativenzweig zu wählen ist, kommuniziert werden muss.

Sie soll über folgende Syntax ermöglicht werden:
`Decision ::= SubpatternName "." AlternativeName "." CaseName`
Zur Erinnerung: Wurde kein Alternativenname vergeben, wird als Voreinstellung „Alternative“ angenommen, wie es im folgenden Beispiel „Hinzufügen einer Kette mit einem Zwischen- und dem Endglied“ zu sehen ist:

```
1 pattern Foo {
2   replace {
3     beg:Node-->end:Node;
4     ch:Chain(beg,end);
5     ch.Alternative.Recursive;
6     ch.Alternative.Recursive.rest.Alternative.Base;
7   }
8 }
```

Im obigen Beispiel wurde in Zeile 4 über eine Teilmusterdeklaration innerhalb der Ersetzung das aus 3.3 bekannte Muster Kette zum Hinzufügen spezifiziert. Die Auswahl, welcher Zweig der Alternative in den Arbeitsgraph instanziiert werden soll ist hier allerdings noch offen. In Zeile 5 wird dann mit der Entscheidung für Recursive die offene Alternative geschlossen – aber im gewählten Zweig tritt ein weiteres Teilmuster mit Alternative auf, erneut das Kette-Muster vermittels der `rest`-Deklaration. In Zeile 6 wird schließlich diese Alternative mit der Auswahl des Kettenendstücks geschlossen.

Multiple Ersetzungen

Problem: Nur eine Ersetzung pro Muster.
Bei der Nutzung eines Teilmusters aus mehreren Mustern heraus werden für gewöhnlich unterschiedliche Ersetzungen benötigt, um die Ersetzung in ihrem jeweils unterschiedlichen Kontext anzuknüpfen. Es ist dem Nutzer derzeit nicht möglich, zu einem eingebetteten Muster mehrere Ersetzungen anzugeben und sich dann im Ersetzungsteil der Regel für eine zu entscheiden. Diese Unzulänglichkeit konnte bisher nur durch die Spezifikation mehrerer gleichartiger Teilmuster umgangen werden, sprich Spezifikationsduplikation von Hand.
Lösung: Multiple benannte Ersetzungen zu einem Muster ermöglichen.

Bei der Ersetzungsdefinition kann auf das `replace-` oder `modify`-Schlüsselwort folgend ein Name angegeben werden.
Die Ersetzung wird von der Funktionsaufruf-auf-Musterobjekt-Syntax
`NameOfSubpattern "(" ReplConnections ")"`
der unbenannten Ersetzung zu einer Methodenaufrufsyntax
`NameOfSubpattern "." NameOfReplacement "(" ReplConnections ")"`
erweitert, mit der die durchzuführende Ersetzung ausgewählt werden kann.

```
1  subpattern BinTree(from:NA) {
2    alternative {
3      Base {
4        negative {
5          from-->child1:NA;
6          from-->child2:NA;
7        }
8        modify LinkLeafesTo(a:NA) {
9          from-->a;
10       }
11       replace LeftArm {
12       }
13     }
14     Recursive {
15       from-->child1:NA;
16       from-->child2:NA
17       left:BinTree(child1);
18       right:BinTree(child2);
19
20       modify LinkLeafesTo(a:NA) {
21         left.LinkLeafesTo(a);
22         right.LinkLeafesTo(a);
23       }
24       replace LeftArm {
25         from-->child1;
26         left.LeftArm();
27       }
28     }
29   }
30 }
```

Ich möchte an dieser Stelle auf die Unterschiede zwischen der Ersetzungsspezifikation in einer Regel und der Ersetzungsspezifikation innerhalb eines Teilmusters hinweisen. Im `replace`-Teil der Regel sind weder Ersetzungsanknüpfungen noch benannte Ersetzungen erlaubt, dafür aber die der Regel eigenen Ein-/Ausgabeparameter. Im `replace`-Teil eines Teilmusters existieren keine Regelparameter, dafür aber die Ersetzungsanknüpfungen sowie benannte Ersetzungen. Durch diese Trennung bleibt die Schnittstelle der Regeln unverändert, und damit die Syntax der Regelanwendung in den Graphersetzungssequenzen gleich.

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 [Via07].

Kapitel 4

Implementierung von GrGen

In diesem Kapitel beschreibe ich als Grundlage für die folgenden beiden Kapitel den Aufbau von GrGen, den Vorgang der Codegenerierung für die Passungssuchprogramme und den Passungsvorgang bei der Ausführung des generierten Codes.

4.1 Aufbau GrGen

Der Grundaufbau des GrGen-Graphersetzungssystems ist in folgender Grafik skizziert.

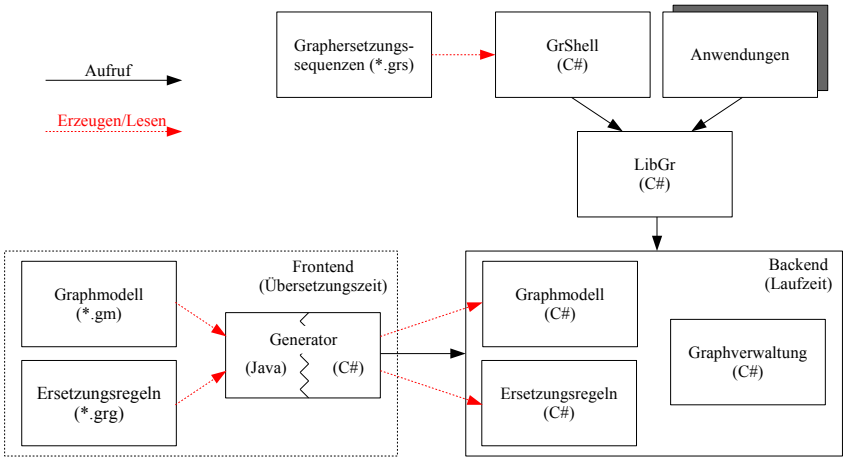


Abbildung 4.1: Grundaufbau GrGen

Die im nächsten Kapitel beschriebenen Umbauarbeiten konzentrierten sich auf den Graphersetzungsgenerator, genauer auf das LGSP-Backend, das derzeit einzige Backend von GrGen.

4.2 Vorgang Codegenerierung

Der Vorgang der Codegenerierung im LGSP-Backend ist in folgender Graphik dargestellt.

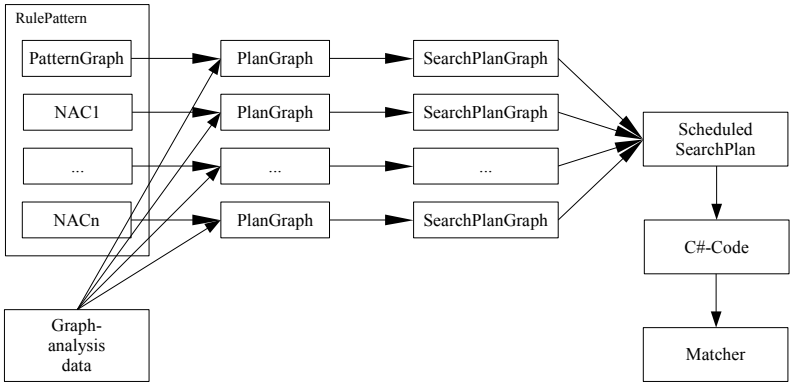


Abbildung 4.2: Aufbau der Suchplanerzeugung im LGSPBackend von GrGen.NET

Jede Regel, zu der ein Mustersuchprogramm erzeugt werden soll, wird durch den vorderen Java-Teil des Frontends in Form eines Regelmuster-Objektes gegeben, 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. Zudem wird im vorderen Teil des Frontends der Code für die Ersetzung generiert.

Aus den Mustergraphen werden nun im hinteren C#-Teil des Frontends 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 (überdeckender 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. Aus dem Suchplan wird schließlich C#-Code erzeugt, der zu guter Letzt übersetzt und in einem LGSPAction-Objekt dem Nutzer zur Ausführung angeboten wird.

Der Aufbau des GrGen.NET-Systems und der Vorgang der Codegenerierung werden in [Kro07] ausführlicher erläutert.

4.3 Passungsvorgang

Der generierte Code erzeugt bei seiner Anwendung, genauer beim Finden einer Passung, ein Passungsobjekt des Typs LGSPMatch, das er in eine Liste von Passungsobjekten einfügt. Ein Passungsobjekt besteht aus einer Reihung von Knoten und eine Reihung von Kanten. Negative Teilmuster kommen nicht vor, da ihr Vorhandensein gerade die Passung verhindert.

Die Passungsobjekte, d.h. die Passungen des Mustergraphen im Arbeitsgraphen werden durch *Ansatzerweiterung im Rücksetzverfahren* bestimmt.

Ich beginne dessen Erläuterung aus didaktischen Gründen 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 Kandidaten 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. Rücksetzen erfolgt durch Verlassen der Schleife und 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.

Kapitel 5

Umbau der Codeerzeugung

Im Codegenerator von GrGen.NET wurde bisher aus dem Suchplan direkt Code erzeugt. Der dazu notwendige Steuerzustand war komplex, und, auch mangels Dokumentation, schwer verständlich. Zudem wurde an verschiedenen Stellen immer wieder gleichartiger Code ausgeführt und erzeugt, dem Umstand geschuldet, dass viele Suchplanoperationen gemeinsame Teiloperationen wie Prüfung auf Isomorphie oder Verbundenheit enthalten. Als Vorbereitung auf eine folgende Erweiterung um eingebettete und alternative Muster habe ich mich deshalb entschlossen, zwischen Suchplan und Code eine weitere Zwischensprache einzuführen, das Suchprogramm, welches die Abstraktionsebene der Teiloperationen in einer Datenstruktur direkt repräsentiert.

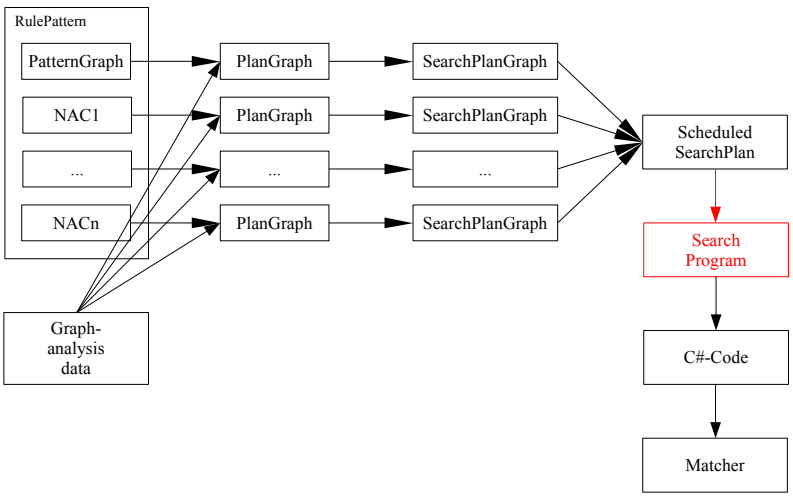


Abbildung 5.1: Aufbau der Suchplanerzeugung mit der neuen Zwischenschicht

Mit ihr wird der Code verständlicher sowie leichter zu warten und zu erweitern, außerdem werden durch die explizite Repräsentation des Suchprogramms in einer Datenstruktur globale Optimierungen auf diesem möglich. Aber wie immer beim Einführen einer weiteren Abstraktionsebene wird der Code auch langsamer in der Ausführung.

Im neuen Codegenerator wird der Suchplan, d.h. die Folge der Suchplanoperationen, auf einen Baum von Suchprogrammoperationen abgebildet, der direkt die Schachtelungsstruktur des Codes widerspiegelt, wie sie primär durch die Typ- und Kandidateniterationsschleifen, sekundär durch die Operationen zum Umgang mit fehlgeschlagenen Prüfungen gegeben ist.

Danach werden in einem Baumdurchlauf die Abbruchoperationen eingefügt, die von der Struktur des Baumes und der Position der Prüfoperation in diesem abhängen. (Hier werden globale Zusammenhänge berücksichtigt, wohingegen das Aufbauen des Baumes ein lokaler Vorgang abhängig von der gerade aktuellen Suchplanoperation war.)

Schließlich wird der Baum durchlaufen und entsprechender Code erzeugt. Für den interessierten Leser sowie Entwickler, die GrGen erweitern wollen, befinden sich im Anhang B eine Auflistung der Suchplanoperationen, eine Auflistung der Suchprogrammoperationen und die exemplarische Umsetzung einer Suchplanoperation in ihre Suchprogrammoperationen.

Kapitel 6

Entwurf erweiterter Passungsvorgang

Im Laufe dieses Kapitels möchte ich eine Passungsmaschine für die in Kapitel 3 spezifizierten eingebetteten und alternativen Muster entwickeln, beginnend mit einer Beschreibung des erweiterten Passungsobjektes, über einen ersten, noch nicht zufriedenstellenden Entwurf hin zu einer 3-Keller-Maschine, abgeschlossen mit ein paar Bemerkungen zur Anpassung der Codegenerierung.

6.1 Erweitertes Passungsobjekt

Um die Passung von eingebetteten Mustern zurückgeben zu können, werden die bisherigen flachen Passungsobjekte, bestehend aus den gefundenen Knoten und Kanten, um eine Reihung für eingebettete Passungsobjekte erweitert. Jedem Teilmuster wird statisch ein Platz in der Reihung zugewiesen, an dem sich nach erfolgter Passung das gefundene Passungsobjekt befindet – dieses ist somit von einer hierarchischen Baumstruktur, die die Musterzusammensetzung widerspiegelt. 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 Passungsobjekt des Teilmusters des gewählten Alternativenzweiges gefüllt wird. Zur Identifikation welcher Zweig gewählt wurde, verweist jedes Passungsobjekt nun zusätzlich auf die Repräsentation seines zugehörigen Teilmusters in der LibGr (diese Metainformation wurde zum zugänglich machen für den Nutzer mit einer Schnittstelle versehen).

Ersetzen erfolgt in einem Durchlauf durch den Passungsbaum, unter Anwendung des Ersetzungsmusters, das zum jeweils gepassten Teilmuster gehört (nach Auswahl der anzuwendenden Ersetzung, wenn mehrere mögliche Ersetzungen spezifiziert wurden). So viel zur Struktur des Suchergebnisses; nun zur Struktur der Suche selbst.

6.2 Wiederverwendung der Passungssuche

Wie bereits in Kapitel 4 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 ersichtlich 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 gibt es zwei wesentliche Entwurfsentscheidungen,

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?

mit denen wir uns im Folgenden auseinandersetzen werden.

6.3 Erweiterung Passungssuche, Erster Entwurf

Im Zuge eines kurzen Vorgedankens möchte ich erwähnen, dass die Vorgehensweise, die Suchfunktion bei ihrem Aufruf alle partiellen Lösungen finden zu lassen und erst danach aus ihr zurückzukehren, um die gefundenen Teillösungen in der Suchfunktion des Gesamtmusters zu kombinieren, praktisch nicht durchführbar ist: sie verbietet sich wegen des Speicherverbrauchs. Abgesehen davon müsste ein neues Verfahren zur Isomorphieprüfung entwickelt werden, das keine Informationen in den Graphen selbst schreibt.

Lassen wir also die Suchfunktion zurückkehren, sobald ein Ergebnis gefunden wurde, und lassen wir sie das Ergebnis in Form einer Referenz auf das Match-Objekt (damit auf der Halde) liefern. Die Suchfunktion für das Teilmuster wird aus dem umschließenden Muster heraus in einer Schleife aufgerufen, so lange bis sie vom gegebenen Ansatz aus nicht mehr zur Passung gebracht werden kann oder die gewünschte Maximalanzahl Passungen erreicht wird.
Doch woher weiß die Teilmustersuchfunktion bei ihrem zweiten Aufruf an welcher Stelle im Suchraum sie mit der Suche fortfahren muss?
Ihre gesamten lokalen Variablen und die lokalen Variablen ihrer Teilmuster existieren nicht mehr. Aber die in diesen Variablen enthaltenen Informationen gibt es noch, sie finden sich im Passungsobjekt (dem Passungsbaum) wieder. Wenn wir das Passungsobjekt einlesen und die entsprechenden Aufrufe der Suchfunktionen erzeugen, können wir den alten Suchzustand auf dem Aufrufkeller wieder rematerialisieren und schließlich mit der Suche fortfahren. Wir benötigen also einen zweiten Modus für die Suchfunktionen, in dem sie ein Passungsobjekt wieder einlagern.

Bei diesem Verfahren befinden sich in den Schachteln der Suchfunktionen auf dem Aufrufkeller:

- die Wurzeln der Passungsbäume bereits gefundener Teilmuster,
- der aktuell bearbeitete Zweig des Passungsobjektes,
- und die Information welche Teilmuster noch zu bearbeiten sind.

Die gefundenen und zusammengesetzten Passungsobjekte hingegen stehen auf der Halde.

Dieses Verfahren würde seinen Zweck erfüllen, weist aber deutliche Schwächen auf: Zum einen wäre da das Erzeugen von Passungsbäumen zu gefundenen partiellen Passungen auf der Halde, von denen nicht sicher ist, ob sie zu einer vollständigen Passung erweitert werden können. Ist dies nicht möglich, was bei verzweigenden Mustern häufig vorkommen wird, war der Aufwand für die Speicherreservierungen umsonst, ebenso wie der angeforderte Speicher selbst. Zum anderen kostet das Rematerialisieren des Suchzustandes aus dem Suchergebnis Zeit. Können wir uns diesen Aufwand ersparen? Bei höchstens einem Teilmuster pro Muster könnten wir dies – indem wir in jeder Suchfunktion beim Finden einer Passung mit den lokalen Variablen ein Passungsobjekt erzeugen und dieses der Teilmustersuchfunktion zum Füllen ihrer noch offenen Stelle übergeben. Die Suchfunktionen würden dann kein Passungsobjekt mehr zurückgeben, nur noch die letztaufgerufene Blattprozedur würde die vollständige Passung speichern. Doch damit würde das Speicheranforderungsproblem noch verschlimmert werden, und abgesehen davon: die Möglichkeit mehrere Teilmuster angeben zu können ist ein wesentliches Merkmal der spezifizierten Spracherweiterung und als solches auch zu realisieren.

Unsere Probleme bei der Suche nach zusammengesetzten Mustern entste-

hen durch die Diskrepanz zwischen der Struktur des Suchvorganges und der Struktur der Passungsobjekte. Das Finden und Zusammensetzen des Passungsbaumes muss linearisiert auf einem Zweig des Suchzustandsbaumes stattfinden. Wir geben den intuitiven Gedanken, dass ein Mustersuchprogramm bei seiner Rückkehr ein Passungsobjekt abliefert und die Suchprogramme der Teilmuster aus dem umschließenden Muster heraus aufgerufen werden auf, und gelangen so zum zweiten Entwurf.

6.4 Erweiterung Passungssuche, Zweiter Entwurf

Ist ein Muster aus mehreren Teilmustern zusammengesetzt, ruft das Passungssuchprogramm des Gesamtmusters nur noch das Passungssuchprogramm des ersten enthaltenen Teilmusters direkt auf, gibt diesem aber als Auftrag mit, 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 zudem 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.

Das eine Problem ist, dass das Suchprogramm, welches sie 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.

Das andere 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 Rückgabe-

kellern, 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.

Eine solche Situation der 3-Keller-Maschine ist in Abbildung 6.1 skizziert.

6.5 Anpassung Generierung

Der Suchplan ist um eine Operation für das Suchen nach einem Teilmuster sowie um eine Operation zum Abarbeiten einer Alternative (mit den darin geschachtelten Suchplänen der Alternativenzweige) zu erweitern.

Im Suchprogramm müssen für die Teilmuster Operationen zum Schreiben von Aufträgen auf den Auftragskeller hinzugefügt werden, zum Auslesen des Auftragskellers und Aufrufen des folgenden, dem Auftrag entsprechenden Suchprogramms, und Operationen zum Zusammensetzen des Passungsobjektes. Alternative Muster können durch einfaches hintereinander anführen des Passungscodes der Teilmuster der Fälle gehandhabt werden. Bei mehreren Alternativen in einem Muster ist ein Herausziehen der Alternativen in eigene Suchprozeduren in Erwägung zu ziehen, um die kombinatorische Explosion der Varianten im Code zu vermeiden.

Behandlung Isomorphie

Im derzeit bestehenden GrGen.NET-System enthält jedes *Arbeitsgraphelement* eine mapped-Variable, in die eine Homomorphie-Id geschrieben wird, wenn das Graphelement an ein Musterelement gebunden wird, und auf die in den „Homomorphie nur wenn spezifiziert“-Prüfungen, mit dem Spezialfall reine Isomorphie, getestet wird. Zusätzlich enthält jedes *Arbeitsgraphelement* eine negMapped-Variable für Homomorphieprüfungen innerhalb von NACs, da diese eigene, geschachtelte Homomorphiebedingungen angeben dürfen.

Die Vorgehensweise, die gesamte Homomorphieinformation in den Arbeitsgraphelementen zu speichern wird aufgegeben. Sie erlaubt zwar sehr schnelle Homomorphieprüfungen, verbraucht aber mehr Speicherplatz als für die Aufgabe angemessen ist, zumal homomorphes Zusammenfallen selten spezifiziert wird. Außerdem müßte, aufgrund der Erweiterung auf mehrfach geschachtelte Blöcke von NACs und Alternativen bei weiterhin gegebener Möglichkeit einer `hom(a,b)`-Deklaration, mit `a` außen und `b` innen, das negMapped-Feld auf eine Liste von der Tiefe der Schachtelung verallgemeinert werden, was den Speicherplatzverbrauch noch einmal massiv vergrößern würde.

Statt dessen wird die Homomorphieinformation pro Arbeitsgraphelement auf ein isMapped-Bit reduziert, das angibt, ob das entsprechende Element bereits an ein Musterelement gebunden wurde. Isomorphieprüfungen sind

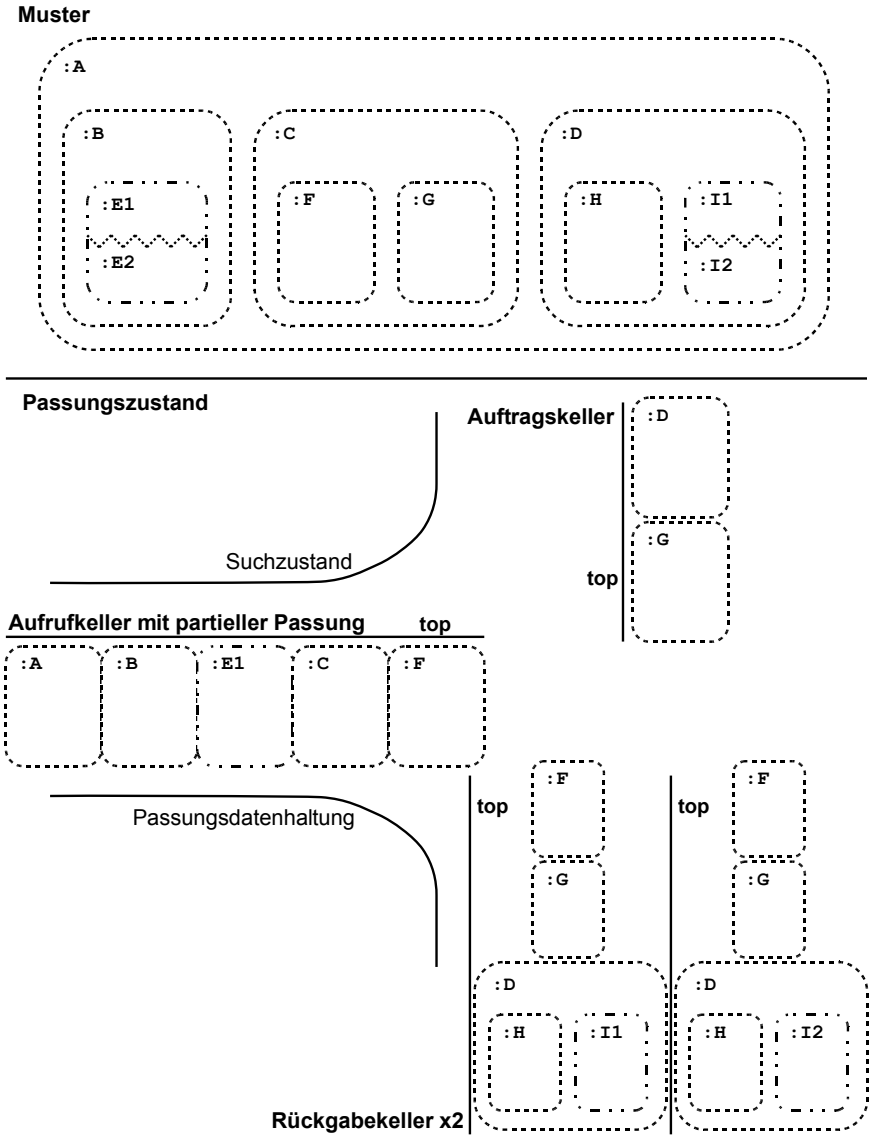


Abbildung 6.1: Suchzustand der 3-Keller-Maschine

damit weiterhin billig möglich, Prüfungen auf homomorphes Zusammenfallen müssen statt eines Vergleiches gegen das mapped-Feld des gepassten Arbeitsgraphelementes (gegen die darin enthaltene Homomorphie-Id) gesondert außerhalb des Graphen erfolgen. Hierzu muss das betrachtete Element mit im schlechtesten Fall allen bereits zu den Musterelementen gepassten Graphelementen in den lokalen Variablen verglichen werden.

Die Prüfmethode mit dem isMapped-Bit lässt sich einfach auf Teilmuster, die wie in Kapitel 3 spezifiziert vollständig isomorph zueinander gepasst werden, erweitern: zusätzlich zu dem „Ist im gerade aktuellen Teilmuster bereits gebunden worden"-Bit wird ein „Ist in einem beliebigen, bereits gepassten Teilmuster gebunden worden"-Bit aufgenommen. 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.
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.

Optimierungen

Die Codeerzeugung für die 3-Keller-Maschine kann noch auf verschiedene Weisen optimiert werden:

Offenes Einbauen von Mustern in des Gesamtmuster, analog dem Inlining von Code im Übersetzer. Anwendbarkeit, Vor- und Nachteile sind die gleichen wie beim Inlining von Code.

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.

Anknüpfungen durch Ausgabeparameter implementieren. Anknüpfungen werden derzeit im umschließenden Muster gepasst und dann als Eingabeparameter an die Teilmuster weitergereicht – unter gewissen Umständen kann ein Bestimmen des Anknüpfungselementes im Teilmuster und Hochreichen ins umschließende Muster lohnend sein, auch über eine Kette von Rekursivschritten hinweg.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurden eingebettete und alternative Muster erörtert und als eine Erweiterung von GrGen.NET spezifiziert. Zur Suche nach ihnen im Arbeitsgraphen wurde ein effizienter Algorithmus entworfen, als Vorarbeit zu dessen Implementierung wurde die Codeerzeugung von GrGen.NET umgeschrieben.

Für eine folgende Diplomarbeit offen geblieben ist die eigentliche Implementierung: Im Frontend das Einlesen der Spezifikation, das Prüfen von Syntax und statischer Semantik sowie das Erzeugen von Datenstrukturen für das Backend. Im Backend die Suchplanung anhand der erhaltenen Datenstrukturen und am Ende die Codegenerierung für die Passungssuchprogramme. Eine Realisierung der im vorigen Kapitel vorgestellten Optimierungen würde die Implementierung schließlich vervollkommnen.

Darüber hinaus ist noch das Verhältnis der eingebetteten und alternativen Muster zu kontextfreien Graphgrammatiken zu untersuchen.

Abseits der angesprochenen Diplomarbeitsaufgaben harren noch andere angedachte Erweiterungen, wie

- eine `copy`-Operation zum Kopieren eines Knotens mitsamt der anliegenden Kanten,
- eine `exact`-Deklaration zur Festlegung, dass an einem Knoten keine weiteren als die angegebenen Kanten anliegen dürfen,
- durch Angabe einer Minimal- und einer Maximalanzahl von Zusammensetzschritten beschränkte Muster,
- ungerichtete Kanten, um die Modellierung von ungerichteten Graphersetzungsproblemen angenehmer zu gestalten,

ihrer Ausarbeitung und Realisierung.

Anhang A

Beispiele

A.1 Beispiel Volladdierer

```
1  pattern pMOSTransistor(gate:VkPt, source:VkPt, drain:VkPt, vdd:VkPt) {
2    t:CMOSTransistor;
3    gate-:Gate->t;
4    t-:Source->source;
5    t-:Drain->drain;
6    t-:Substrat->vdd;
7  }
8  pattern nMOSTransistor(gate:VkPt, source:VkPt, drain:VkPt, gnd:VkPt) {
9    t:CMOSTransistor;
10   gate-:Gate->t;
11   t-:Source->source;
12   t-:Drain->drain;
13   t<-:Substrat-gnd;
14 }
15 pattern NAND(a:VkPt, b:VkPt, out:VkPt, vdd:VkPt, gnd:VkPt) {
16   :pMOSTransistor(a, vdd, out, vdd);
17   :pMOSTransistor(b, vdd, out, vdd);
18   :nMOSTransistor(b, out, negdrainsourcelink, gnd);
19   :nMOSTransistor(a, negdrainsourcelink, gnd, gnd);
20   negdrainsourcelink:VkPt;
21 }
22 pattern XOR(a:VkPt, b:VkPt, out:VkPt, vdd:VkPt, gnd:VkPt) {
23   :NAND(a, b, a_nand_b:VkPt, vdd, gnd);
24   :NAND(a, a_nand_b, a__nand__a_nand_b:VkPt, vdd, gnd);
25   :NAND(b, a_nand_b, b__nand__a_nand_b:VkPt, vdd, gnd);
26   :NAND(a__nand__a_nand_b, b__nand__a_nand_b, out, vdd, gnd);
27 }
28 pattern Volladdierer{a:VkPt, b:VkPt, carry_in:VkPt, sum:VkPt, carry_out:VkPt,
29                      vdd:VkPt, gnd:VkPt) {
30   :XOR(a, b, a_xor_b:VkPt, vdd, gnd);
31   :XOR(a_xor_b, carry_in, sum, vdd, gnd);
32   :NAND(a, b, a_nand_b:VkPt, vdd, gnd);
```

```
33   :NAND(a_xor_b, carry_in, a_xor_b__nand__carry_in:VkPt, vdd, gnd);
34   :NAND(a_nand_b, a_xor_b__nand__carry_in, carry_out, vdd, gnd);
35 }
```

A.2 Beispiel Carbocyclisches Aromat

```
1  pattern Funktionsgruppe(anker:C) {
2    alternative {
3      Hydrogen {
4        anker-->:H;
5      }
6      Hydroxyl {
7        anker-->:O-->:H;
8      }
9      Methyl {
10       anker-->c:C;
11       c-->:H;
12       c-->:H;
13       c-->:H;
14     }
15     Nitro {
16       anker-->n:N;
17       n-->:O;
18       n-->:O;
19     }
20   }
21 }
22 pattern CarbocyclischesAromat {
23   // Benzolring
24   c1:C-->c2:C-->c3:C-->c4:C-->c5:C-->c6:C-->c1;
25   c1-->c2; c3-->c4; c5-->c6;
26
27   f1:Funktionsgruppe(c1);
28   f2:Funktionsgruppe(c2);
29   f3:Funktionsgruppe(c3);
30   f4:Funktionsgruppe(c4);
31   f5:Funktionsgruppe(c5);
32   f6:Funktionsgruppe(c6);
33 }
```

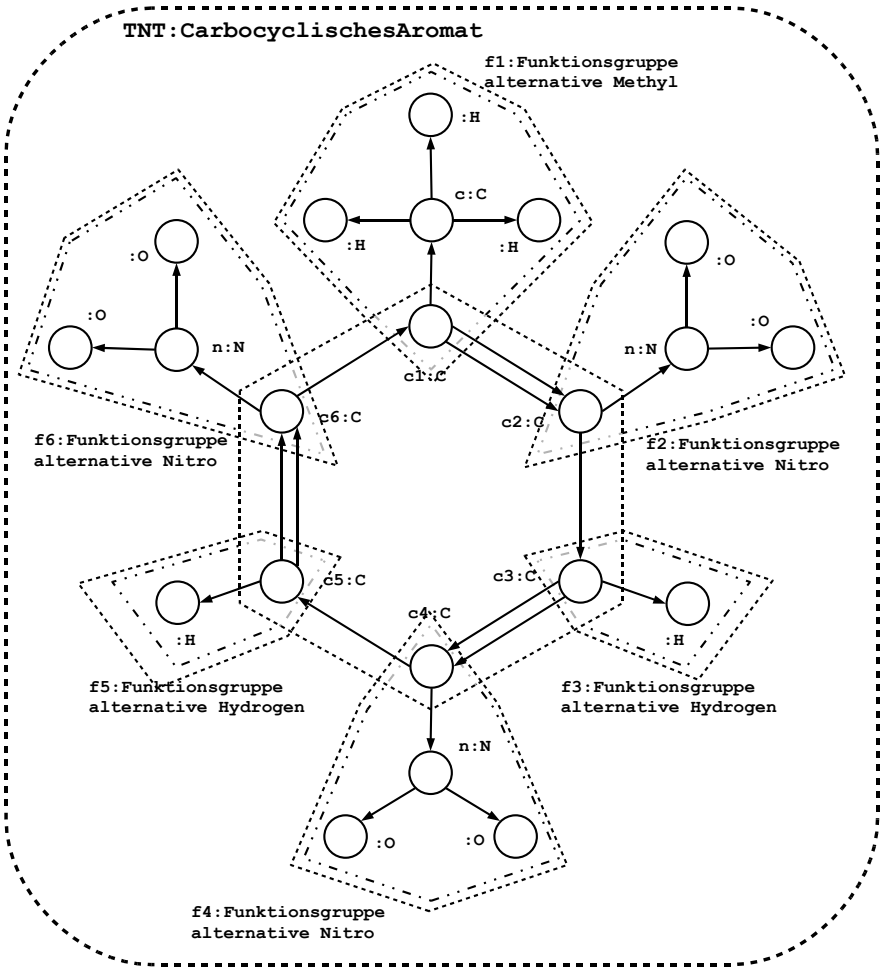


Abbildung A.1: Eine der möglichen Zusammensetzungen des Musters CarbocyclischesAromat: Trinitrotoluol(TNT)

A.3 Beispiel Transkription DNA in RNA

```
rule Transkription
{
  pattern {
    -to_next:Edge->;
    b:DNAChainBegin(to_next);
    c:DNAChain(to_next);
    modify {
      -rna_to_next:Edge->;
      b(rna_to_next);
      c(rna_to_next);
    }
  }
}
pattern DNAChain(-from_prev:Edge->)
{
  alternative {
    Chain {
      -to_next:Edge->;
      e:DNAChainElement(from_prev, to_next);
      c:DNAChain(to_next);
      modify(-rna_from_prev:Edge->) {
        e(rna_from_prev, rna_to_next);
        c(rna_to_next);
      }
    }
    End {
      e:DNAChainEnd(from_prev);
      modify(-rna_from_prev:Edge->) {
        e(rna_from_prev);
      }
    }
  }
}
pattern DNAChainBegin(-to_next:Edge->)
{
  :H-->:O-->c3;
  :DesoxyriboseCore(c1_nb, c3:C, c5:C);
  -c1_nb:Edge->;
  nb:NukleoBase(c1_nb);
  c5-->:O-->p:P-to_next->;
  p-->do:O; p-->do; p-->:O-->:H;
  modify(-rna_to_next:Edge->) {
    :RNAChainBegin(rna_to_next, rna_c1_nb);
    -rna_c1_nb:Edge->;
    nb(rna_c1_nb);
  }
}
```

```
48 pattern DNAChainElement(-from_prev:Edge->, -to_next:Edge->)
49 {
50   -from_prev->:O-->c3;
51   :DesoxyriboseCore(c1_nb, c3:C, c5:C);
52   -c1_nb:Edge->;
53   nb:NukleoBase(c1_nb);
54   c5-->:O-->p:P-to_next->;
55   p-->do:O; p-->do; p-->:O-->:H;
56   modify(-rna_from_prev:Edge->, -rna_to_next:Edge->) {
57     :RNAChainElement(-rna_from_prev->, -rna_to_next->, rna_c1_nb);
58     -rna_c1_nb:Edge->;
59     nb(rna_c1_nb);
60   }
61 }
62 pattern DNAChainEnd(-from_prev:Edge->)
63 {
64   -from_prev->:O-->c3;
65   :DesoxyriboseCore(c1_nb, c3:C, c5:C);
66   -c1_nb:Edge->;
67   nb:NukleoBase(c1_nb);
68   c5-->:O-->p:P-->:O-->:H;
69   p-->do:O; p-->do; p-->:O-->:H;
70   modify(-rna_from_prev:Edge->) {
71     :RNAChainEnd(rna_from_prev, rna_c1_nb);
72     -rna_c1_nb:Edge->;
73     nb(rna_c1_nb);
74   }
75 }
76 pattern RNAChainBegin(-to_next:Edge->, -c1_nb:Edge->)
77 {
78   :H-->:O-->p:P;
79   p-->do:O; p-->do; p-->:O-->:H;
80   p-->:O-->c5;
81   :RiboseCore(c1_nb, c3:C, c5:C);
82   c3-->:O-to_next->;
83 }
84 pattern RNAChainElement(-from_prev:Edge->, -to_next:Edge->, -c1_nb:Edge->)
85 {
86   -from_prev->p:P;
87   p-->do:O; p-->do; p-->:O-->:H;
88   p-->:O-->c5;
89   :RiboseCore(c1_nb, c3:C, c5:C);
90   c3-->:O-to_next->;
91 }
92 pattern RNAChainEnd(-from_prev:Edge->, -c1_nb:Edge->)
93 {
94   -from_prev->p:P;
95   p-->do:O; p-->do; p-->:O-->:H;
96   p-->:O-->c5;
```

```
97    :RiboseCore(c1_nb, c3:C, c5:C);
98    c3-->:O-->H;
99  }
100 pattern NukleoBase(c1_nb:Edge)
101 {
102   alternative {
103     A {
104       a:Adenin(c1_nb);
105       modify(rna_c1_nb:Edge) {
106         :Uracil(rna_c1_nb);
107       }
108     }
109     C {
110       c:Cytosin(c1_nb);
111       modify(rna_c1_nb:Edge) {
112         :Guanin(rna_c1_nb);
113       }
114     }
115     G {
116       g:Guanin(c1_nb);
117       modify(rna_c1_nb:Edge) {
118         :Cytosin(rna_c1_nb);
119       }
120     }
121     T {
122       t:Thymin(c1_nb);
123       modify(rna_c1_nb:Edge) {
124         :Adenin(rna_c1_nb);
125       }
126     }
127   }
128 }
129 pattern RiboseCore(-c1_nb:Edge->, c3:C, c5:C)
130 {
131   o:O-->c1:C-->c2:C-->c3-->c4:C-->o;
132   c1-->:H; c2-->:H; c3-->:H; c4-->H;
133   c1-c1_nb->;
134   c2-->:O-->:H;
135   c4-->c5;
136   c5-->:H; c5-->:H;
137 }
138 pattern DesoxyriboseCore(-c1_nb:Edge->, c3:C, c5:C)
139 {
140   o:O-->c1:C-->c2:C-->c3-->c4:C-->o;
141   c1-->:H; c2-->:H; c3-->:H; c4-->H;
142   c1-c1_nb->;
143   c2-->:H;
144   c4-->c5;
145   c5-->:H; c5-->:H;
```

```
146 }
147 pattern Adenin(-c1_nb:Edge->) {
148   -c1_nb->n1:N-->c1:C-->n2:N-->c2:C-->c3:C-->n1;
149   c1-->n2; c2-->c3;
150   c1-->:H;
151   c2-->c4:C-->n3:N-->c5:C-->n4:N-->c3;
152   c4-->n3; c5-->n4;
153   c5-->:H;
154   c4-->n5:N; n5-->:H; n5-->:H;
155 }
156 pattern Cytosin(-c1_nb:Edge->) {
157   -c1_nb->n1:N-->c1:C-->c2:C-->c3:C-->n2:N-->c4:C-->n1;
158   c1-->c2; c3-->n2;
159   c1-->:H;
160   c2-->:H;
161   c3-->n3:N; n3-->:H; n3-->:H;
162   c4-->o:O; c4-->o;
163 }
164 pattern Guanin(-c1_nb:Edge->) {
165   -c1_nb->n1:N-->c1:C-->n2:N-->c2:C-->c3:C-->n1;
166   c1-->n2; c2-->c3;
167   c1-->:H;
168   c2-->c4:C-->n3:N-->c5:C-->n4:N-->c3;
169   c5-->n4;
170   c4-->:O; c4-->:O;
171   n3-->H;
172   c5-->n5:N; n5-->:H; n5-->:H;
173 }
174 pattern Thymin(-c1_nb:Edge->) {
175   -c1_nb->n1:N-->c1:C-->c2:C-->c3:C-->n2:N-->c4:C-->n1;
176   c1-->c2;
177   c1-->:H;
178   c2-->c5:C; c5-->:H; c5-->:H; c5-->:H;
179   c3-->o:O; c3-->o;
180   n2-->:H;
181   c4-->o2:O; c4-->o2;
182 }
183 pattern Uracil(-c1_nb:Edge->) {
184   -c1_nb->n1:N-->c1:C-->c2:C-->c3:C-->n2:N-->c4:C-->n1;
185   c1-->c2;
186   c1-->:H;
187   c2-->:H;
188   c3-->o:O; c3-->o;
189   n2-->:H;
190   c4-->o2:O; c4-->o2;
191 }
```

Anhang B

Details Suchprogramm

Suchplanoperationen

MaybePreset Musterelement wurde als Parameter in die Regel hineingereicht, wenn nicht: danach suchen

NegPreset Musterelement ist bereits im umschließenden positiven Muster bestimmt worden, jetzt noch negative Bedingungen prüfen

Lookup Musterelement bestimmen, dazu durch alle Elemente des Graphen vom passendem Typ durchiterieren, den Kandidaten prüfen

Outgoing Musterkante bestimmen, dazu durch alle ausgehenden Kanten eines Knotens durchiterieren, den Kandidaten prüfen

Incoming Musterkante bestimmen, dazu durch alle eingehenden Kanten eines Knotens durchiterieren, den Kandidaten prüfen

ImplicitSource Musterknoten bestimmen, dazu den Quellknoten einer Kante nehmen und prüfen

ImplicitTarget Musterknoten bestimmen, dazu den Zielknoten einer Kante nehmen und prüfen

NegativePattern Negatives Muster prüfen

Condition Bedingung prüfen

Suchprogrammoperationen

GetTypeByIteration Typ für folgendes GetCandidateByIteration bestimmen, durch Iteration über Datenstruktur mit zulässigen Typen, Weitergabe über Variable, folgender Code in Iteration geschachtelt

GetTypeByDrawing Nur ein Typ für folgendes GetCandidateByIteration zu betrachten, diesen bestimmen, Weitergabe über Variable, folgender Code schließt an

GetCandidateByIteration Kandidat durch Iteration über die Typenliste bestimmen, Variable mit ihm beschreiben, folgender Code in Iteration geschachtelt

GetCandidateByDrawing Kandidat aus den Eingabeparametern oder als Quell/Ziel-Knoten einer Kante bestimmen, Variable mit ihm beschreiben, folgender Code schließt an

CheckCandidateForType Prüfen, ob Kandidat in Variable einen gemäß Muster zulässigen Typ hat, Code für Prüfung schlug fehl geschachtelt, so wie bei den übrigen Check-Operationen, Code für Erfolg schließt an

CheckCandidateFailed Prüfung von Kandidat hat sich bereits statisch als Fehlschlag herausgestellt

CheckCandidateForConnectedness Prüfen, ob Kandidat mit den Musterelementen verbunden ist, mit denen er verbunden sein sollte

CheckCandidateForIsomorphy Prüfen, ob der gegebenen Kandidat für das Musterelement mit einem anderen Musterelement homomorph Zusammenfallen würde

CheckCandidateForPreset Prüfen, ob ein gültiger Kandidat in den Eingabeparametern übergeben wurde

CheckPartialMatchByNegative Prüfen, ob ein negatives Muster bei der bisherige partiellen Passung zuschlägt

CheckPartialMatchByCondition Prüfen, ob eine Bedingung bei der bisherigen partiellen Passung zuschlägt

AcceptIntoPartialMatchWriteIsomorphy Kandidat hat Prüfungen bestanden, in die partielle Passung aufnehmen und Isomorphiedaten für spätere Prüfungen auf homomorphes Zusammenfallen mit ihm in den Graph schreiben

WithdrawFromPartialMatchRemoveIsomorphy Rücksetzschritt, Kandidat wieder aus partieller Passung, und geschriebene Isomorphiedaten wieder aus Graph entfernen

PartialMatchCompletePositive Partielle Passung hat alle Prüfungen bestanden, Muster wurde gefunden

PartialMatchCompleteNegative Partielle Passung hat alle Prüfungen bestanden, das negative Muster wurde gefunden

PartialMatchCompleteBuildMatchObject Partielle Passung hat alle Prüfungen bestanden, jetzt alle gefundenen Kandidaten aus ihren lokalen Variablen in ein Passungsobjekt schreiben

AdjustListHeads Position der zuletzt gefundenen Passung in Graph selbst schreiben, so dass nächste Suche mit der Folgeposition beginnt

CheckContinueMatchingMaximumMatchesReached Prüfung ob mit Passungssuchen fortzufahren ist anhand der gewünschten Maximalzahl an Passungen

CheckContinueMatchingFailed Prüfug, ob mit Passungssuchen fortzufahren ist, hat sich bereits statisch als Fehlschlag herausgestellt (negatives Muster, weiter in positivem mit nächstem Kandidaten)

ContinueOperation Weiter mit nächstem Kandidat

GotoLabel Sprungmarke für ContinueOperation über goto

RandomizeListHeadsTypes Zufällige Auswahl des ersten Elementes der folgenden Iteration

Umsetzungsbeispiel

Die Lookup-Suchplanoperation wird nach folgendem Schema in eine Schachtelung von Suchprogrammoperationen umgesetzt:

Lookup	
1a	GetTypeByIteration, wenn das Musterelement einen Typ hat, der auf mehrere Typen im Graph zutreffen kann, der weitere Code ist innerhalb dieser Schleife geschachtelt
1b	GetTypeByDrawing, wenn das Musterelement einen Typ hat, der nur auf einen Typ im Graph zutreffen kann, der weitere Code schließt sich an diese Operation an
2	GetCandidateByIteration, die Liste der Graphelemente von diesem Typ durchmustern, jedes Element ist ein Kandidat für die Passung, der weitere Code ist innerhalb dieser Schleife geschachtelt
3	CheckCandidateForConnectedness * n, so viele Verbundenheitsprüfungen, wie es Musterelemente gibt, mit denen der Musterkandidat verbunden sein sollte und die bereits bestimmt wurden, Code für fehlgeschlagene Prüfung geschachtelt, Code für Erfolg schließt an
4	CheckCandidateForIsomorphy, wenn notwendig Isomorphie sicherstellen durch Prüfen gegen Zusammenfallen mit bereits gepassten Elementen, Code für fehlgeschlagene Prüfung geschachtelt, Code für Erfolg schließt an
5	AcceptIntoPartialMatchWriteIsomorphy, wenn notwendig Isomorphieinformationen zu aktuellem Kandidat schreiben, weitere Code schließt an
6	Das Suchprogramm zu den folgenden Suchplanoperationen einbauen, dessen Struktur ist hier unbekannt, aber der weitere Code schließt an das zurückgegebene Ende an
7	WithdrawFromPartialMatchRemoveIsomorphy, wenn notwendig Isomorphieinformationen entfernen, weiterer Code schließt an
8	Schachtelung der Kandidateniteration schließen, d.h. Anschlusspunkt für weiteren Code auf 2 setzen
9	Schachtelung der Typeniteration schließen, wenn sie notwendig war, d.h. Anschlußpunkt für weiteren Code auf 1 setzen

Tabelle B.1: Umsetzung

Literaturverzeichnis

[BG07] Blomer, Jakob ; Geiss, Rubino: *The GrGen.NET User Manual*. Benutzerhandbuch. `http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-V1_0-2007-07-02.pdf`. Version: 2007

[Gei07] Geiss, Rubino: *Graphersetzung mit Anwendungen im Übersetzerbau*. Dissertation(eingereicht), 2007

[Kro07] Kroll, Moritz: *GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen*. Studienarbeit, 2007

[Via07] Viatra Development Team: *The VIATRA 2 Model Transformation Framework Users Guide*. Benutzerhandbuch. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/viatratut_October2006.pdf`. Version: 2007