Seminar

Graph Grammatiken

Lehrstuhl für Informatik III

RWTH Aachen

Sommersemester 2004

Programmierte Graphersetzungssysteme

Michael Faber
Robert Heracles

Inhaltsverzeichnis

1 Einleitung

Graphersetzungssysteme werden in der Informatik oft eingesetzt, um praktische Probleme zu lösen, wobei sich ihre Einsatzgebiete in 3 verschiedene Teilbereiche aufteilen lassen:

- Die Beschreibung und Erzeugung von bereits bekannten Graphausdrücken, die graphentheoretische Eigenschaften modellieren
- Die Erkennung von Graphausdrücken, die beispielsweise die einem Satze in natürlicher Sprache zugrunde liegende logische Struktur modellieren
- Spezifikation von neuen Graphklassen und Transformationen von Graphen, die Datenbanken und Arbeitsschritte auf diesen Datenbanken ausdrücken können

Hier soll die dritte der obigen Klassen betrachtet und analysiert werden. In dieser werden Graphersetzungssysteme benötigt, mit denen man, wie mit einer hohen Programmiersprache umgehen kann. Dies erfordert den Einsatz von speziellen Kontrollstrukturen, so dass Graphersetzungsschritte innerhalb einer Anwendung gesteuert und überwacht werden können. Unter Einsatz solcher Kontrollstrukturen spricht man von programmierten Graphersetzungssystemen.

Die Implementationen dieser Idee, PAGG (Programmed Attributed Graph Grammars) und PROGRESS (Programmed Graph Replacement Systems), die zur Entwicklung von CAD Systemen oder Softwareentwicklungswerkzeugen eingesetzt werden, sind die bis dato einzigen dieser Art und werden im weiteren Verlauf der Arbeit noch ausführlicher vorgestellt.

Um solche Anwendungen zu verwirklichen, werden hohe Anforderungen an das Graphersetzungssystem gestellt. Man benötigt Möglichkeiten, komplexe Regeln für die Anwendung einer Ersetzungsregel oder der Bearbeitung von Knotenattributen zu definieren. So sollte man beispielsweise

- Die Beschreibung von Graphschemata und die Manipulation von Graphen mit Hilfe von Ersetzungen trennen.
- Ableitbare Attribute und Beziehungen in einer rein deklarativen Form beschreiben können.
- So genannte „Generate and Test“ Probleme mit Tiefensuche bzw. Backtracking in der Art lösen können, in der auch Prolog damit umgeht.

Bei der Erstellung von formalen Definitionen zu programmierten Graphersetzungssystemen wie PAGG oder PROGRESS stellt sich heraus, dass es insbesondere sehr schwierig ist, Kontrollstrukturen oder die Eigenschaft der ableitbaren Graphen mengen- oder klassentheoretisch auszudrücken. Deshalb soll im folgenden Kapitel zuerst ein Rahmenwerk geschaffen werden, mit dem die formale Definition von anwendungsorientierten Graphersetzungssystemen leichter fällt bzw. besser beschrieben werden kann. Die Entwicklung dieses Rahmenwerkes wurde hauptsächlich durch die folgenden Beobachtungen beeinflusst:

- Benötigt werden *relationale Strukturen*, da diese eine Verallgemeinerung von Graphen bzw. Hypergraphen sind.
- Um die benötigten Eigenschaften zu beschreiben eignen sich vor allem *logische Formeln.*
- *Nonmonotonic Reasoning* ist gut geeignet, um die benötigte Ableitungseigenschaft zu umschreiben und Integrität von Formelmengen zu prüfen.

- Für die Beschreibung der Semantik von nichtdeterministischen, rekursiven Graphersetzungsprogrammen bietet sich die Fixpunkttheorie an

Auf Grund dieser Beobachtungen werden zuerst (im folgenden Kapitel) logikbasierte Graphersetzungssysteme vorgestellt und mit ihrer Hilfe die Sprache PROGRESS definiert.

2 Grundlagen für Strukturersetzungssysteme

In diesem Kapitel soll ein allgemeines Rahmenwerk vorgestellt werden, das viele verschiedene Graphersetzungssysteme als Spezialfälle enthält und PROGRESS zugrunde liegt. Es kombiniert Ansätze von algebraischen Ersetzungssystemen und deduktiven Datenbanksystemen. Die so genannten Logik Basierten Strukturersetzungssysteme (Logic Based Structure Replacement Systems) liefern die Vorraussetzung für die Definition von Graphmodellen, die Spezialfälle von Relationalen Strukturen sind, und versorgen uns mit einer Fülle von Schreibweisen, die bei der Graphersetzung benötigt werden.

Die vorgestellten Definitionen und Formalismen werden durch ein Beispiel (Abb.1) veranschaulicht, das durch die gesamte Arbeit mitgeführt wird, und zeigen soll, wie mit Hilfe von LBSR Systemen ein Graphersetzungssystem beschrieben werden kann. Das Beispiel zeigt eine Personendatenbank, die mit Hilfe unseres Strukturmodells beschrieben und bearbeitet werden soll. Die Datenbank besteht aus *man* bzw. *woman* Knoten, die ein Attribut (*income)* besitzen. Jeweils ein *man* und *woman* Knoten können durch eine *wife* Kante verbunden sein. Weiterhin kann eine Person beliebig viele Kinder besitzen.

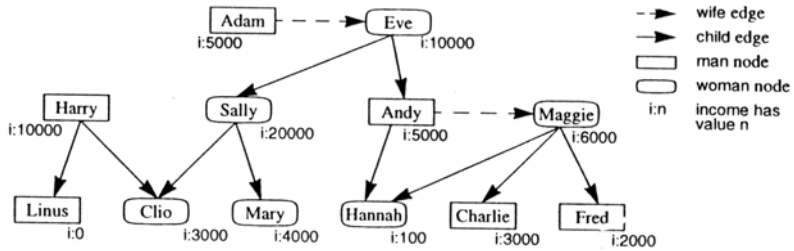

Abbildung 1: Beispiel Personendatenbank

2.1 Strukturen

Innerhalb dieses Kapitels soll beschrieben werden wie knoten- und kantenbeschriftete Graphen als Spezialfälle von schemakonsistenten Strukturen modelliert werden können, also als Menge von logischen Formeln mit bestimmten Eigenschaften.

Definition 2.1 (Signatur)

Ein 5 Tupel $\sum$:= (*Af, Ap, V, W, X*) wird als *Signatur* bezeichnet, wenn

- *Af* ein Alphabet von Funktionssymbolen ist
- *Ap* ein Alphabet von Prädikatssymbolen ist
- *V* ein Alphabet von Objektbezeichnern ist
- *W* ein Alphabet von Objektmengenbezeichnern ist
- *X* ein Alphabet von logischen Variablen ist, die zur Quantifizierung verwendet werden

Die Definition der Signatur führt mit *V* und *W* Mengen von Konstanten ein, die spezielle Objekte bzw. Mengen von Objekten benennen. Mit Hilfe dieser Bezeichner werden später in Ersetzungsregeln Matches von einzelnen Objekten oder Objektmengen einer Regel auf einer gegebenen Struktur beschrieben.

Beispiel 2.1

Die zu dem Graphen aus Abbildung 1 passende Signatur würde wie folgt aussehen

- *Af* := {child, woman, wife, man, person, income, integer, 0, ...,+}
- *Ap* := { node, edge, att, type}
- *V* := {Adam, Eve, ..., He, She, Value}
- *W* := {HerChildren, HisChildren}
- *X* := {x1, x2, ... }

Die Alphabete *V*, *W* und *X* bestehen aus willkürlich ausgesuchten Konstanten oder Variablen. Die Menge *Af* beinhaltet einfach alle Symbole, die für Knoten oder Kantenbeschriftungen, Attributstypen und –werte oder Auswertungsfunktionen benötigt werden. *Ap* besteht aus 4 Prädikatsymbolen, die für die von uns gewählte Modellierung von gerichteten, attributierten Graphen benötigt werden. Die gewählten Prädikate haben die folgende Bedeutung:

- node(*x*, *l*): Der Graph enthält einen Knoten *x* mit Beschriftung *l*
- edge(*x*, *e*, *y*): Der Graph enthält eine Kante mit Beschriftung *e* von Knoten *x* zu Knoten *y*
- attr(*x*, *a*, *v*): Das Attribut *a* an Knoten *x* hat den Wert *v*
- type(*v*, *t*): Der Attributswert *v* hat den Typ *t*

Im Folgenden kennzeichnet $\sum$ immer eine Signatur über den beschriebenen Alphabeten.

Definition 2.2 ($\sum$ - Term und Atom)

Tx($\sum$) ist die Menge aller $\sum$-*Terme,* die alle Funktionssymbole und Konstanten aus *Af*, freie Variablen aus *X* und zusätzliche Bezeichner aus *V* und *W* enthält.
A($\sum$) ist die Menge von $\sum$- *Atomen*, über *Tx($\sum$)*, die Prädikatsymbole aus *Ap* und das ‚=' Zeichen enthält (damit die Gleichheit von Termen ausgedrückt werden kann)

Definition 2.3 ($\sum$ - Formeln und Ableitung von $\sum$ - Formeln).

F($\sum$) ist die Menge aller Mengen von prädikatenlogischen Sätzen (Formeln ohne freie Variablen), die aus Elementen von *A($\sum$)*, $\wedge$, $\vee$ als logische Operatoren und $\exists$, $\forall$ als Quantoren besteht.
Des weiteren bedeutet für 2 Mengen Φ und Φ' bestehend aus $\sum$- *Formeln* die Schreibweise $\Phi \succ \Phi'$, dass alle Formeln der Menge Φ' aus der Menge Φ abgeleitet werden können.

Im Folgenden werden Elemente aus *F($\sum$)* benutzt, um Strukturen, Strukturschemata oder linke und rechte Regelseiten einer Ersetzungsregel zu beschreiben.

Definition 2.4($\sum$ - Strukturen)

Eine Menge von Sätzen $F \in F(\sum)$ heißt $\sum$ - *Struktur* ($F \in L(\sum)$) genau dann, wenn $F \subseteq A(\sum)$ und *F* keine Formel der Form „$\tau 1 = \tau 2$" enthält.

Mit Hilfe dieser Definition lassen sich nun Graphen als relationale Strukturen beschreiben. So können wir den Graphen aus unserem Beispiel durch folgende Formelmenge beschreiben:

Beispiel 2.2

F={node(Adam,man), node(Eve,woman), …, attr(Adam,income,5000), attr(Eve,income, 10000), ..., edge(Adam,wife,Eve), edge(Eve,child,Sally), ... }

Allerdings gilt für die Menge F, dass sie beliebig viele Graphen als Modell hat. Der Beispielgraph ist einer dieser Graphen, aber Graphen mit noch zusätzlichen Knoten oder Kanten werden genauso durch die Menge F beschrieben. Es muss also ein Operator eingeführt werden, mit dem man sicherstellen kann, dass nur ein einzelner Graph durch eine Formelmenge beschrieben wird, und der uns die Möglichkeit gibt, ungewollte Modelle für $\sum$ - *Strukturen* auszuschließen. Dafür erweitern wir die gegebene Formelmenge geeignet und erhalten dann eine Formelmenge, die ein minimales Graphmodell beschreibt. Wir unterscheiden zwischen Fakten und abgeleiteten Fakten. Die Erweiterung setzt sich dann aus den negierten Fakten der $\sum$ - *Struktur* zusammen, die nicht aus den Originalfakten ableitbar sind. Der benötigte Operator übernimmt genau diese Erweiterung der Formelmenge, so dass eine konsistente (also widerspruchsfreie) und hinreichend komplette Formelmenge entsteht.

Definition 2.5 ($\sum$ - Struktur Abschlussoperator)

Eine Funktion C: $L(\sum) \rightarrow F(\sum)$ heißt $\sum$ *Struktur Abschlussoperator* genau dann, wenn für alle Strukturen $F \in L(\sum)$ gilt: $F \subseteq C(F)$, $C(F)$ ist eine konsistente Formelmenge und
$C(p(F)) = p(C(F))$, wobei p: $F(\sum)$ -> $F(\sum)$ eine Umbenennung von Bezeichnern aus V, W ist.

Die Nützlichkeit des Abschlussoperators soll an einem kleinen Beispiel noch einmal demonstriert werden. Nehmen wir an, wir hätten die Definition einer Datenbank, die lediglich eine Person enthält. Diese ist beschrieben durch die Menge F={node(Adam, man)}. Ohne einen Abschlussoperator haben wir Schwierigkeiten zu zeigen, dass diese Datenbank wirklich nur eine Person enthält, da die Menge F sämtliche Graphen als Modell hat, die mindestens einen Knoten mit dem Attribut ‚man' besitzen. Um also mit F ein minimales Graphmodell zu beschreiben, definieren wir den Abschlussoperator C wie folgt (wobei ausgelassene Mengen Kanten usw. beschreiben):
$C(F) = F \cup \ldots \cup \{ \forall x, l$: node($x$, man) $\rightarrow$ (x= v1 $\vee$ x= v2 $\vee$ … $\vee$ x= vk) | v1, …, vk $\in V$ sind Bezeichner in F}.
Mit Hilfe von C sind wir in der Lage zu beweisen, dass F nur den ‚man' Knoten ‚Adam' enthält: $C(F) \succ (\forall x, l$: node(x, l) $\rightarrow$ x= Adam) $\wedge$ node(Adam, man).
Da Abschlussoperatoren strukturspezifische Definitionen enthalten können, benutzen wir sie bei der Definition eines Strukturschemas.

Definition 2.6 ($\sum$ - Strukturschema)

Ein Tupel S= (Φ , C) heißt $\sum$ - *Strukturschema* ($S \in S(\sum)$) genau dann, wenn

- $\Phi \in F(\sum)$ eine konsistente Formelmenge ohne Verwendung von speziellen Bezeichnern aus V, W ist.
- C ein $\sum$ - *Struktur Abschlussoperator ist*

Mit Bezug auf ein $\sum$- *Strukturschema* können wir direkt den Begriff der schemakonsistenten Struktur definieren, der besagt, dass eine Struktur genau dann nicht schemakonsistent ist, wenn wir einen Widerspruch zwischen den Formeln der abgeschlossenen Struktur und denen des Schemas ableiten können.

Definition 2.7 (schemakonsistente $\sum$ - Struktur)

Sei $S = (\Phi, C) \in S(\sum)$ ein Schema. Eine $\sum$ - *Struktur* $F \in L(S)$ ist *schemakonsistent* zu S genau dann, wenn $C(F) \cup \Phi$ eine konsistente Formelmenge ist.

Ein Strukturschema $S = (\Phi, C)$ für den Beispielsgraphen würde so aussehen:
$\Phi := F1 \cup F2 \cup F3$, wobei die Mengen *F1* bis *F3* wie folgt aufgebaut sind:
$F1 = \{ \forall x, e, y$: edge(x, e, y) $\rightarrow \exists xl, yl$: node($x, xl$) $\wedge$ node(y, yl),
$\forall x, a, v$: attr(x, a, v) $\rightarrow \exists l$: node(x, xl), ...$\}$
definiert Beschränkungen für das Graphmodell und schließt hängende Kanten und Attribute ohne Knoten aus.
$F2 = \{ \forall x$: node(x, *man*) $\rightarrow$ node(x, *personl*),
$\forall x, y$: edge(x, *wife*, y) $\rightarrow$ node(x, *man*) $\wedge$ node(y,*womanl*)
$\forall x, v$: attr(x, *income*, v) $\rightarrow$ type(v, *integer*), ...$\}$
beinhaltet alle Nebenbedingungen, die aus der Struktur der Datenbank kommen, z.B.

- ein ‚man' Knoten ist auch ein ‚person' Knoten oder
- eine ‚wife' Kante läuft immer von einem ‚man' zu einem ‚woman' Knoten

$F3 = \{ \forall x, y$: ancestor(x, y) $\Leftrightarrow \exists z$: edge($z$, *child*, x) $\wedge$ ($z = y \vee$ ancestor(z, y), ...$\}$
legt ableitbare Eigenschaften, wie die 'ancestor' Beziehung zwischen Knoten fest.

Alle weiteren Regeln des LBSR Systems müssen die oben angeführten Bedingungen erfüllen, müssen also eine schemakonsistente Struktur in eine zweite, ebenfalls schemakonsistente, Struktur überführen. Weiterhin können ableitbare Eigenschaften in Vor- oder Nachbedingungen einer Ersetzung benutzt werden. Für die Anwendung von Ersetzungsregeln ist die Auswahl einer Substruktur aus einer gegebenen Struktur maßgebend, deshalb werden im folgenden Kapitel Substrukturen sowie Methoden zur Auswahl von Substrukturen vorgestellt.

2.2 Substrukturen

In diesem Kapitel wird festgelegt, welche Substrukturen einer gegebenen Struktur mögliche Ziele eines Ersetzungsschrittes werden können, welche also auf die linke Seite einer Strukturersetzungsregel passen. Hierfür werden Morphismen verwendet, die der zentrale Punkt der Definition einer Strukturersetzungsregel sind. LBSR Systeme müssen außerdem in der Lage sein, Mengen von Kanten sowie Knotenattribute in einem Ersetzungsschritt zu bearbeiten. Erstere werden mit Hilfe der Objektmengenbezeichner behandelt, Attribute dagegen werden von $\sum$- *Termen* repräsentiert.

Definition 2.8 ($\sum$ - Term Relation)

Seien $F, F' \in F(\sum)$ zwei Formelmengen mit *Vf, Wf* und *Vf', Wf'*. Weiterhin sei $T(\sum)$ die Menge aller variablen- und bezeichnerfreien $\sum$ - *Terme*. Eine Relation
$u \subseteq (Vf \times (Vf' \cup T(\sum)) \cup (Wf \times (Wf' \cup Vf' \cup T(\sum))$ heißt $\sum$- *Term Relation* genau dann, wenn

- Für alle $v \in Vf$ gilt: $|u(v)| = 1$ wobei $u(x) = \{y | (x, y) \in u\}$
- Für alle $w \in Wf$ gilt: $|u(w)| = 1$ oder $u(w) \subseteq Vf'$ oder $u(w) \subseteq T(\sum)$

Eine Relation ist also genau dann eine $\sum$ - *Term Relation*, wenn jeder Objektbezeichner auf genau einen Objektbezeichner oder genau einen $\sum$ *Term* abgebildet wird und jeder Objektmengenbezeichner auf genau einen Objektmengenbezeichner oder eine Menge von Objektbezeichnern oder einen $\sum$ - *Term* abgebildet wird.
Da die Definition nur Beziehungen zwischen Bezeichnern und $\sum$ - *Termen* herstellt, muss diese noch erweitert werden, um Beziehungen zwischen beliebigen Formeln ausdrücken zu können. Damit können wir dann Beziehungen zwischen Strukturen herstellen, die durch Formelmengen ausgedrückt sind.

Definition 2.9 ($\sum$ - *Relation)*
Sei u eine $\sum$ - *Term Relation*. Die Erweiterung u^* von u auf den Bereich der $\sum$ - *Formeln* heißt $\sum$ - *Relation*.
$u^*(\Phi, \Phi') :\Leftrightarrow \exists \psi \in F(\sum)$ mit freien Variablen $x1, \ldots, xn$ ohne Bezeichner aus V, W
$\wedge \exists v1, \ldots, vn \in V \cup W$ mit $vi \neq vj$ für $i,j \in \{1, \ldots, n\}$
$\wedge \exists \tau 1, \ldots, \tau n \in V \cup W \cup T(\sum)$ mit $u(v1, \tau 1), \ldots, u(vn, \tau n)$:
$\Phi = \psi [v1/x1, \ldots, vn/xn] \wedge \Phi' = [\tau 1/x1, \ldots, \tau n /xn]$
Im Folgenden wird die hier angegebene, abkürzende Schreibweise für $\Phi \in F(\sum)$ verwendet: $u^*(\Phi) := \{\phi' \in F(\sum) | \phi \in \Phi \text{ und } (\phi, \phi') \in u^*\}$.

Die Definition besagt, dass zwei Formeln ϕ und ϕ' durch die Relation u^* in Beziehung stehen, wenn sie sich nur in Termen unterscheiden, die ihrerseits durch u in Beziehung gesetzt sind.

Auf diesen zwei Definitionen aufbauend können wir nun Morphismen zwischen Formelmengen oder Strukturen definieren. Morphismen sind $\sum$ - *Relationen*, die bestimmten Eigenschaften genügen.
Mit Hilfe dieser Morphismen werden wir dann Strukturersetzungsschritte definieren.

Definition 2.10 ($\sum$ - (Struktur)Morphismus)
Seien $F, F' \in F(\sum)$. Eine $\sum$ - *Relation* u von F nach F' ist ein $\sum$ - *Morphismus* von F nach F' ($u: F \xrightarrow{\sim} F'$) genau dann, wenn $F' \succ u^*(F)$.
Wenn $F, F' \in L(\sum) \subseteq F(\sum)$ $\sum$ - *Strukturen* sind, heißt u ein $\sum$ - *Struktur Morphismus*.

Für den Beispielgraphen, der in Beispiel 2.2 formalisiert wurde, kann man so $\sum$ - *Relationen* angeben, die auch $\sum$ - *Struktur Morphismen* sind.
Sei F' die Struktur aus Beispiel 2.2 und eine zweite Struktur gegeben durch F = { node(He, man), node(She, woman), attr(She, income, Value), edge(He, wife, She), edge(He, child, HisChildren), edge(She, child, HerChildren) }, dann ist ein $\sum$ - *Struktur Morphismus* gegeben durch:
u ={ (HerChildren, Hannah), (HerChildren, Charlie), (HerChildren, Fred), (He, Andy), (She, Maggie), (Value, 6000), (HisChildren, Hannah) }.

In diesem Beispiel ist $u^*(F)$ lediglich eine Teilmenge von F', was dem einfachsten Morphismus zu einer Struktur entspricht. Man kann erkennen, dass zwei verschiedene Bezeichner auf einen einzigen anderen Bezeichner abgebildet werden können. Weiterhin kann jeder Objektmengenbezeichner auf eine beliebige Menge von Objektbezeichnern oder Termen abgebildet werden (hierzu gehört ebenfalls die leere Menge).
Um letztlich eine Strukturersetzungsregel vollständig definieren zu können, benötigen wir noch die genaue Definition von Substrukturen, die evtl. noch zusätzlichen Bedingungen unterliegen können. Daher wird im Folgenden nicht nur die Substruktur definiert, sondern weiterführend auch noch Substrukturen mit zusätzlichen Nebenbedingungen.

Definition 2.11 (Substrukturen)
Seien F und $F' \in L(\sum)$ Strukturen. F heißt *Substruktur* von F' unter der $\sum$- *Relation* u ($F \subseteq_u F'$) genau dann, wenn u: $F \xrightarrow{\sim} F'$.

Eine Struktur F ist also eine Substruktur einer zweiten Struktur F', wenn F durch einen Morphismus zu F' in Beziehung gesetzt werden kann. Hierbei ist es durchaus möglich, dass mehrere Objektbezeichner aus F auf genau einen Objektbezeichner aus F' abgebildet werden.

Definition 2.12 (Substruktur mit Nebenbedingungen)
Sei $S = (\Phi, C)$ ein Strukturschema, $F, F' \in L(\sum)$ Strukturen und $\Psi \in F(\sum)$ eine Menge von Nebenbedingungen, die sich auf die Bezeichner von F beziehen. F ist eine *bedingte Substruktur* von F' unter der $\sum$- *Relation* u und zusätzlichen Nebenbedingungen Ψ ($F \subseteq_{u,\Psi} F'$) genau dann, wenn

- $F \subseteq_u F'$, also $F' \succ u^*(F)$
- u: $\Psi \cup F \xrightarrow{\sim} \Phi \cup C(F')$, also $\Phi \cup C(F') \succ u^*(\Psi \cup F) = u^*(\Psi) \cup u^*(F)$

Wir sagen also, dass eine Struktur F eine bedingte Substruktur einer Struktur F' ist, wenn wir mit Hilfe aller Fakten und zusätzlichen Formeln, die unter Abschluss aus F' entstanden sind, zeigen können, dass die Formeln aus F und Ψ ableitbar sind. Dann ‚erfüllt' F die gestellten Bedingungen.

Die Auswahl einer geeigneten Substruktur für die anzuwendende Strukturersetzungsregel soll an einem Beispiel aus der Anwendung PROGRESS noch weiter ausgeführt werden. Wir betrachten einen in PROGRESS geschriebenen Subgraphentest und werden diesen mit Hilfe von Formelmengen ausdrücken. Abbildung 2 zeigt einen PROGRESS Test mit der Bedeutung: „finde ein beliebiges Paar von unverheirateten Personen, die nicht Geschwister sind, und deren Einkünfte der angegebenen Bedingung genügen, und sämtliche ihrer Kinder".

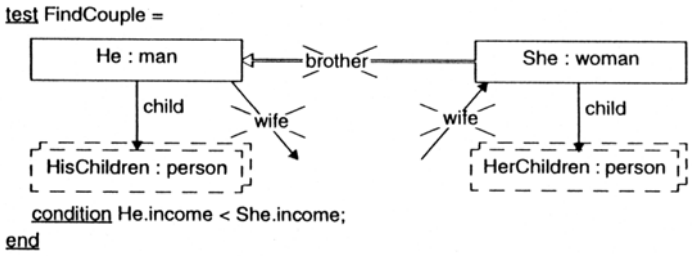

Abbildung 2 (Subgraphtest in PROGRESS)

Formalisiert man den Test, so erhält man die Formelmengen

F:={node(He, man), node(She, woman), attr(He, income, HisValue), attr(She, income, HerValue), edge(He, child, HisChildren), edge(She,child,HerChildren)}

Ψ :={$\neg$ brother(He, She), $\neg \exists x$: edge(He, wife, x), $\neg \exists x$: edge(x, wife, She), HisValue < HerValue, node(HisChildren, person), node(HerChildren, person)},

wobei F den gegebenen Graphen und Ψ die benötigten Nebenbedingungen formalisiert. Unterscheiden muss man Objektbezeichner wie *He* oder *She* und Objektmengenbezeichner wie *HisChildren* oder *HerChildren*, die auf eine beliebige Anzahl von Objektbezeichnern oder $\sum$- *Termen* abgebildet werden dürfen. Unter diesen Vorraussetzungen definiert das unten angegebene u eine korrekte $\sum$- *Relation* zwischen F und F', so dass $F \subseteq_{u,\Psi} F'$ gilt wobei wir davon ausgehen, dass F' für die Datenbank aus Beispiel 2.2 steht.

u:={(He, Harry), (She, Sally), (HisValue, 10000), (HerValue, 20000), (HisChildren, Linus), (HisChildren, Clio), (HerChildren, Clio), (HerChildren, Mary)}.

Wir gehen davon aus, dass der Abschlussoperator C uns Formeln erstellt, in denen ausgedrückt ist, dass Harry und Sally nicht Geschwister sind (Es gibt keinen gemeinsamen Vaterknoten zu Harry und Sally) bzw. dass beide noch nicht verheiratet sind (von Harry geht keine ‚wife' Kante aus; Sally ist nicht das Ziel einer ‚wife' Kante). Weiterhin werden die Bezeichner HisValue und HerValue an die Werte 10000 und 20000 gebunden, so dass die angegebene Bedingung „HisValue < HerValue" erfüllt ist. Unter zu Hilfenahme des Graphschemas $S = (\Phi, C)$ aus unserem Beispiel, können wir zeigen, dass Linus, Clio und Mary zu den Klassen ‚man' und ‚woman' und somit auch zu der Klasse ‚person' gehören. Anzumerken ist noch, dass im Beispiel die Anzahl der Objektbezeichner, die an die Objektmengenbezeichner aus F gebunden sind, maximal ist. Diese Eigenschaft ist wichtig, falls eine $\sum$- *Relation* für das Überschreiben von Strukturen eingesetzt werden soll, ist aber in unserem Beispiel eher nebensächlich.

Im folgenden Kapitel werden nun mit Hilfe der obigen Definitionen Strukturersetzungsregeln definiert und an einem weiteren PROGRESS Beispiel veranschaulicht.

2.3 Strukturersetzung

Mit dem Wissen der beiden vorausgehenden Kapitel können wir an dieser Stelle den Begriff der Strukturersetzungsregel einführen. Hierfür werden wir ein Quadrupel von Formelmengen wählen, das die Strukturersetzungsregel formal beschreibt.

Definition 2.13 (Strukturersetzungsregel)
Ein Quadrupel p:= (AL, L, R, AR) mit AL, AR $\in F(\sum)$ und L, R $\in L(\sum)$ heißt *Strukturersetzungsregel* oder *Produktion* über der Signatur $\sum$ ($p \in P(\sum)$) genau dann, wenn

- AL nur Bezeichner aus der Menge L enthält
- AR nur Bezeichner aus der Menge R enthält
- Jeder Objektmengenbezeichner aus L ebenfalls in R enthalten ist und umgekehrt

Die ersten zwei Eigenschaften einer Produktion stellen sicher, dass in Formelmengen, die die Nebenbedingungen definieren, nur die Bezeichner der jeweiligen Seite verwendet werden. Nebenbedingungen für die linke Seite dürfen sich also nur auf Objekte beziehen, die auch wirklich auf der linken Seite vorhanden sind. Die dritte Eigenschaft einer Produktion stellt sicher, dass Objektmengenbezeichner nur benutzt werden, um von der veränderten Struktur Verbindungen zu neuen Substrukturen zu erstellen, die von der rechten Seite einer Regel erstellt wurden.

Als Beispiel für eine Produktion werden wir das obige Beispiel eines Substrukturtests zu einer Strukturersetzungsregel erweitern. Wir definieren F und Ψ wie im obigen Beispiel und können dann die Produktion *Marry* folgendermaßen angeben: *Marry*:= (AL, L, R, AR) mit

- AL= Ψ
- L= F
- R= F \ {attr(She, income, Value)}
  $\cup$ {edge(He, wife, She), attr(She, income, Value + taxReduction(Value)),
  edge(He, child, HerChildren), edge(She, child, HisChildren)}.
- AR= {}

Abbildung 3 zeigt die Produktion geschrieben in PROGRESS. Der Ablauf einer jeden Strukturersetzungsregel ist wie folgt (in Klammern der Bezug zum Beispiel):

- Suche passende Matches für die Objektbezeichner aus L (She und He), so dass die Nebenbedingungen aus AL berücksichtigt und eingehalten werden
- Erweitere den gefundenen Match auf die Objektmengenbezeichner aus L (HisChildren, HerChildren). Die Bezeichner müssen zu der maximalen Anzahl an Objekten in Beziehung gesetzt werden, so dass die Nebenbedingungen nicht verletzt sind
- Entferne das Bild von L ohne R (das alte *income* Attribut von *She*) aus der Datenbank
- Füge das Bild von R ohne L (das neue *income* Attribut von *She*, eine *wife* Kante zwischen *He* und *She*, *child* Kanten zu den Kinderknoten) der Datenbank hinzu
- Teste die Nebenbedingungen AR. Falls eine der Bedingungen verletzt ist, mache alle Veränderungen am Graphen rückgängig und suche im ersten Schritt einen anderen möglichen Match
- Teste auf Inkonsistenzen in der Datenbank. Falls Inkonsistenzen vorliegen, suche ebenfalls einen anderen möglichen Match im ersten Schritt. Mache alle Veränderungen am Graphen rückgängig

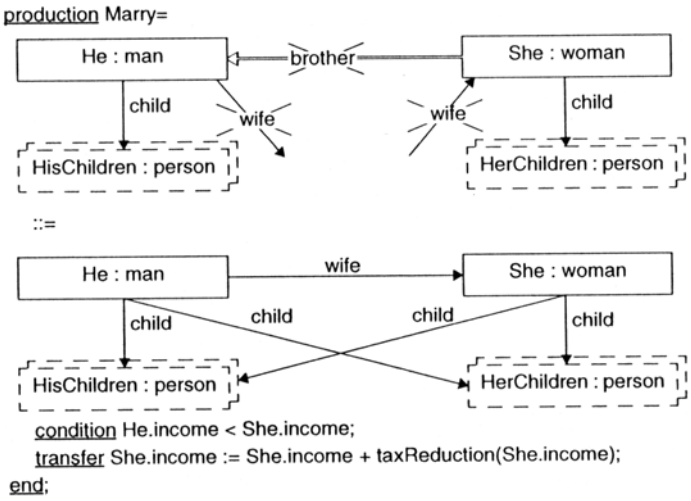

Abbildung 3 (Produktion in PROGRESS)

Anzumerken bleibt, dass die Möglichkeit besteht, Nebenbedingungen zu straffen oder auszuweiten, je nachdem welche Bedingungen schon an die Datenbank selber gestellt sind (man könnte beispielsweise den Test, ob Personen schon verheiratet sind fallenlassen, wenn die Datenbank Bedingungen enthält, die Polygamie verhindern). Hier liegt jedoch auch ein weiterhin offenes Problem, nämlich die Frage, ob und wie beliebige Nachbedingungen (Bedingungen der rechten Regelseite) in äquivalente Vorbedingungen (Bedingungen der linken Seite) umgeformt werden können.

Um im weiteren Verlauf die Sprache eines LBSR Systems erklären zu können, benötigen wir den Begriff der schemaerhaltenden Strukturersetzung, der angibt wann eine Struktur aus einer anderen mit Hilfe der Produktion p direkt ableitbar ist. Mit diesem Ableitungsbegriff und einer gegebenen Ausgangsstruktur definiert sich dann die Sprache des LBSR Systems.

Definition 2.14 (schemaerhaltende Strukturersetzung)
Sei $S = (\Phi, C) \in S(\sum)$ ein Strukturschema und $F, F' \in L(\sum)$ zwei schemakonsistente Strukturen. Sei p:= (AL, L, R, AR) $\in P(\sum)$ eine Strukturersetzungsregel. Die Struktur F' ist *direkt ableitbar* aus der Struktur F durch die Anwendung von p ($F \Rightarrow_p F'$) genau dann, wenn

1. Es gibt einen Morphismus u: $L \xrightarrow{\sim} F$ mit $L \subseteq_{u, AL} F$
2. Es gibt keinen Morphismus u': $L \xrightarrow{\sim} F$ mit $L \subseteq_{u', AL} F$ und $u \subseteq u'$
3. Es gibt einen Morphismus w: $R \xrightarrow{\sim} F'$ mit $R \subseteq_{u, AR} F'$
4. Der Morphismus w bildet jeden neuen Objektbezeichner aus R, der nicht in L definiert ist, auf einen neuen separaten Objektbezeichner in F' ab, der nicht in F definiert ist
5. Unter $K = L \cap R$ gilt: v:= $\{(x, y) \in u \mid x$ ist ein Bezeichner in $K\} = \{(x, y) \in w \mid x$ ist ein Bezeichner in $K\}$
6. Es gibt eine Struktur $H \in L(\sum)$: $F \setminus (u^*(L) \setminus v^*(K)) = H = F' \setminus (w^*(R) \setminus v^*(K))$

Die ersten beiden Punkte der Definition sagen aus, dass die durch *u* ausgewählte Substruktur in *F* maximal ist und die gegebenen Vorbedingungen einhält. Der nächste Punkt besagt, dass die durch *w* gewählte Substruktur in *F'* die gegebenen Nachbedingungen einhält. Der vierte Punkt garantiert uns, dass alle von der Produktion neu erzeugten Objekte (Objektbezeichner, die erstmals in R auftauchen) auch wirklich erst in der Zielstruktur vorhanden sind, und nicht schon in der Ausgangsstruktur. Im fünften Punkt wird sichergestellt, dass *u* und *w* in Bezug auf die Bezeichner aus K (dem Schnitt aus L und R) identisch sind, d.h. die Teile der Struktur, die erhalten bleiben, beziehen sich auf die gleichen Bezeichner in L und R. Mit dem letzten Punkt der Definition wird die Existenz einer Zwischenstruktur des Ersetzungsschritts beschrieben, die man aus der Ausgangsstruktur erhält, nachdem die alte Substruktur entfernt wurde und bevor die neue Substruktur eingefügt wird.

Diese Art der Strukturersetzung verbietet es uns nicht, homomorphe Matches von zwei Bezeichnern auf ein und dasselbe Objekt in *F* durchzuführen. Diese Art der Abbildung ist also durchaus erlaubt und findet im Beispiel auch Anwendung (Die Objektmengenbezeichner *HisChildren* und *HerChildren* matchen auf Knotenmengen, die beide den Knoten *Clio* enthalten). Auch muss mit Hilfe von geeigneten Formeln noch dafür gesorgt werden, dass ein Knoten mit seinem kompletten „Anhang“ (‚edge’, ‚attr’ Formeln) gelöscht wird.
Abschließend werden in diesem Kapitel noch LBSR Systeme und die von ihnen erzeugte Sprache definiert. Dafür werden wir noch konkrete $\sum$- *Strukturen* durch abstrakte ersetzen und abstrakte $\sum$- *Strukturen* sowie abstrakte schemakonsistente $\sum$ *Strukturen* definieren.

Definition 2.15 (abstrakte $\sum$- Struktur)

Die Menge der *abstrakten* $\sum$- *Strukturen* $AL(\sum) := L(\sum)/\cong$ besteht aus allen Äquivalenzklassen von $\sum$- *Strukturen*, die der folgenden Äquivalenzrelation $\cong$ genügen:
$\forall F, F' \in L(\sum): F \cong F' :\Leftrightarrow \exists$ Isomorphismus $u: F \xrightarrow{\sim} F'$

Definition 2.16 (abstrakte schemakonsistente $\sum$- Struktur)

Sei $S \in S(\sum)$ ein Schema und *L(S)* die Menge aller *S*-Schemakonsistenten Strukturen. Die Menge der *abstrakten schemakonsistenten* $\sum$- *Strukturen* wird definiert durch
$AL(\sum) := L(S)/\cong$.

In Bezug auf die letzten beiden Definitionen definieren wir nun ein LBSR System und dessen Sprache wie folgt:

Definition 2.17 (LBSR System)

Ein Tupel LBSR := (*AS, P*) heißt logikbasiertes Strukturersetzungssystem (logic based structure replacement system) in Bezug auf ein Strukturschema $S \in (\sum)$, wenn

- $AS \in AL(S)$ eine schemakonsistente Struktur ist, die Ausgangsstruktur
- $P \in P(\sum)$ eine Menge von Produktionen ist.

Definition 2.18 (Sprache eines LBSR Systems)
Sei LBSR := (*AS, P*) ein LBSR System. Die Sprache $AL(\text{LBSR}) \subseteq AL(S)$ ist dann definiert als $AF \in AL(\text{LBSR}) :\Leftrightarrow AS \Rightarrow^* AF$.
Dabei ist $\Rightarrow^*$ die transitive, reflexive Hülle von $\Rightarrow$ und es gilt
$AF \Rightarrow AF' \subseteq AL(\sum)$ x $AL(S) :\Leftrightarrow \exists F \in AF, F' \in AF', p \in P: F \Rightarrow F'$.

Die Sprache eines LBSR Systems besteht also aus allen abstrakten Strukturen, die aus der Ausgangsstruktur mit Hilfe der Produktionen abgeleitet werden können. Die Produktionen können dabei beliebig oft angewendet werden.

Als Ausgangspunkt für die programmierte Strukturersetzung haben wir in diesem Kapitel erst einmal mit den LBSR Systemen ein Rahmenwerk geschaffen, in dem wir verschiedene Strukturersetzungssysteme miteinander vergleichen können. Der logikbasierte Ansatz eignete sich hierfür gut, da logischen Formeln gut zu verwenden sind, um ableitbare Eigenschaften von Datenstrukturen zu beschreiben. Ebenso können Systeme mit zusätzlichen Nebenbedingungen, wie sie bei der Produktion z.B. eingesetzt werden, durch logische Formeln gut formuliert werden. Mit der Strukturersetzung erhält man zusätzlich eine bequeme Möglichkeit, um z.B. auf deduktiven Datenbanken zu arbeiten.
Im folgenden Kapitel wollen wir nun an Hand dieses Rahmens drei Graphersetzungsansätze vergleichen.

3 Graphersetzungssysteme

3.1 Praktische Ansätze

In diesem Kapitel wollen wir nun, im Gegensatz zu Kapitel 2, von den allgemeinen Formulierungen Abstand nehmen, und uns mit drei konkreten kontextbezogenen Graphersetzungssystemen beschäftigen. Genauer werden wir die im vorigen Kapitel angeführten Eigenschaften eines Ersetzungssystems benutzen, um uns die algorithmischen Ansätze PAGG und PROGRESS, sowie den algebraischen Ansatz DELTA anzusehen und zu vergleichen. Verglichen werden die drei Ansätze im Bezug auf die Anwendung von Ersetzungsregeln, d.h. wie können Ersetzungsregeln eingeschränkt oder zusätzliche Bedingungen an die Regel formuliert werden, und ihre Behandlung von Knotenattributen. Der erste Punkt wird im nächsten Abschnitt unter dem Oberbegriff kontextbezogene Ersetzungsregeln behandelt.

3.2 Kontextbezogene Ersetzungsregeln

Zu Beginn dieses Abschnitts wollen wir die drei praktischen Ansätze der Graphersetzung, PAGG, PROGRESS und DELTA, in Bezug auf die Suche nach einem passenden Match für die linke Regelseite und den Ersetzungsschritt betrachten. Ruft man sich den Ablauf einer Ersetzungsregel noch einmal ins Gedächtnis:

- Finde in einem Ausgangsgraphen ein Bild für den Graphen der linken Regelseite
- Entferne diesen Match aus dem Ausgangsgraph
- Binde ein Bild des Graphen der rechten Regelseite in den Ausgangsgraph ein

so stellt man fest, dass der Teilgraph der rechten Regelseite ohne jede Bindung zum Ausgangsgraphen in diesem vorliegt. Es wird also ein Vorgehen benötigt, das uns

garantiert, dass der neue Teilgraph mit dem Ausgangsgraphen verbunden ist, bzw. Kanten von entfernten Knoten nicht ‚frei' im Graph auftauchen. Dieser Ansatz wird *einbettende Ersetzung (embedding transformation)* genannt und basiert seinerseits auf dem so genannten *gluing approach*. Die Idee hinter dem *gluing approach* besteht darin, einzelne Knoten im Ausgangsgraphen auszuzeichnen und diese weder zu löschen noch neu zu erstellen, sondern einfach mit all ihren Kanten zu nicht ausgezeichneten Knoten während der Ersetzung beizubehalten. Ein Knoten wird zu einem ausgezeichneten Knoten (*preserved node)*, falls er ein Match zu einem Knoten in einem der Graphen der linken oder rechten Regelseite ist.
Zum Vergleich der Ansätze dienen vor allem zwei Fragen. Zum einen die Frage, wie mit nichtisomorphen Matches umgegangen wird (z.B. zwei Knoten im Graphen der linken Regelseite matchen auf genau einen Knoten im Ausgangsgraph). Zum anderen die Frage was passiert, wenn Kanten eines gematchten Knotens nicht durch die linke Regelseite gematcht werden. Bei diesen Überlegungen spielt das Prinzip der ausgezeichneten Knoten eine wichtige Rolle. Die drei angesprochenen Ansätze verhalten sich hier zwar ähnlich, unterscheiden sich aber dennoch in Einzelheiten.
In PAGG wird die Idee der ausgezeichneten Knoten verwendet und es wird gefordert, dass ein isomorpher Match des Graphen der linken Regelseite mit dem Ausgangsgraphen vorliegt. D.h. hier ist es nicht möglich, dass zwei Knoten auf einen einzelnen Knoten im Ausgangsgraphen gematched werden. Weiterhin werden in PAGG alle Kanten eines zu entfernenden Knotens, die durch die linke Seite nicht gematched werden, ebenso entfernt.
PROGRESS benutzt explizite Anweisungen um ein nichtisomorphes Matching zu erzeugen. Nichtisomorphe Matches sind also im Allgemeinen möglich, allerdings sollten so genannte Auszeichnungsbedingungen (*identification conditions)* angegeben sein, die verhindern, dass evtl. ein Knoten ausgezeichnet und gelöscht werden soll (wenn er z.B. durch einen ausgezeichneten Knoten und einen Knoten gematcht wird, der gelöscht werden soll). Ansonsten verhält sich PROGRESS wie der oben beschriebene PAGG Ansatz.
Der DELTA Ansatz funktioniert prinzipiell wie PROGRESS. Auch hier ist nichtisomorphes Matching möglich, muss aber ebenso durch explizite Anweisungen erzeugt werden. Auch das Problem, dass ausgezeichnete Knoten gelöscht werden sollen, kann auftreten, kann aber ebenfalls durch Auszeichnungsbedingungen behoben werden.

Mit den oben genannten Maßnahmen ist die Möglichkeit gegeben, einfache einbettende Ersetzungen durchzuführen. Dennoch können keine Ersetzungen durchgeführt werden, in denen z.B. Kanten von Vorgängern eines gelöschten Knotens zu dessen Nachfolgern gezogen werden sollen. Hierfür müsste der Kontext eines Graphen noch mitberücksichtigt werden. Um solche Eigenschaften realisieren zu können, brauchen wir komplexere Regeln für die einbettende Ersetzung. Für diese Regeln benötigen wir

- Bezeichner der linken Regelseite, deren Match im Ausgangsgraphen eine beliebige Anzahl von verbundenen, nicht gematchten, Knoten besitzt.
- Beschreibungen eines Pfades durch den Ausgangsgraphen, der als Startpunkt einen gematchten Knoten und als Endpunkt eine Menge von ungematchten Knoten hat.

- Knoten der rechten Regelseite, die im Ausgangsgraphen dann Ziel oder Startpunkt einer neuen Kante sind sowie einen Teil, der Richtung und Beschriftung dieser neuen Kante angibt

Um solche komplexen einbettenden Regeln beschreiben zu können, benutzen PAGG und DELTA eine graphische Regelschreibweise. PROGRESS dagegen bietet graphische sowie textuelle Regelbeschreibungen an. Wir beschränken uns hier jedoch auf die graphische Schreibweise. In Abbildung 4 sind diese Art von graphischen einbettenden Regeln abgebildet.

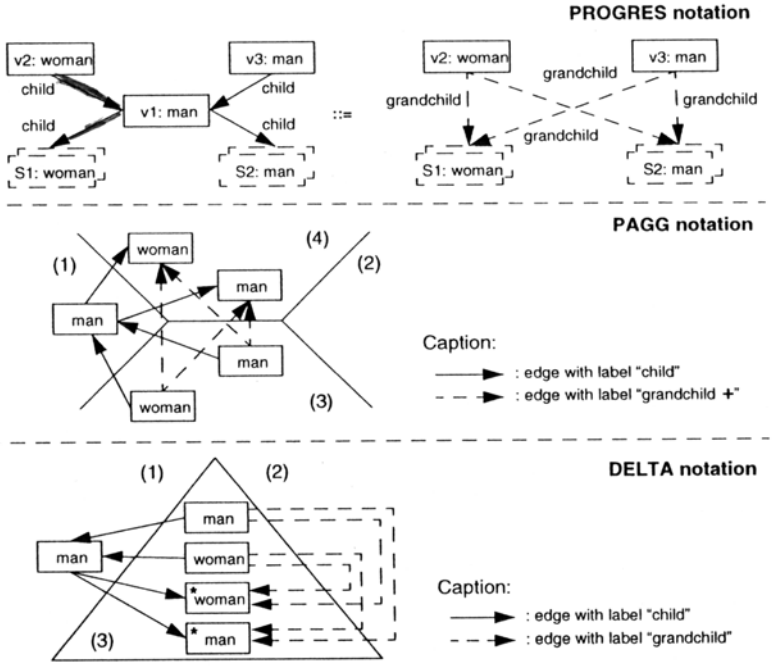

Abbildung 4 (Graphische einbettende Regeln)

Alle 3 graphischen Regeln stellen eine Produktion dar, in der die "mittlere" Generation aus der Personendatenbank entfernt wird und die dritte Generation mit Hilfe von *grandchild* Kanten mit der ersten Generation verbunden wird. In den verschiedenen Ansätzen werden Knoten bzw. Kanten, die eine beliebige Anzahl von Knoten oder Kanten im Ausgangsgraphen matchen können, auf eine andere Weise dargestellt. Unterstützt wird diese Eigenschaft in unserem in Kapitel 2 definierten Rahmenwerk durch die Objektmengenbezeichner, die eine beliebige Anzahl an Objekten matchen können. PROGRESS verwendet zur Kennzeichnung dieser speziellen Knoten doppelt gestrichelte Rechtecke, im Gegensatz zu einfachen Rechtecken, die normale Knoten repräsentieren.
Bei PAGG Regeln unterscheidet man zwischen 4 Sektionen:

- Die linke Seite des X kennzeichnet Knoten und Kanten, die im Verlauf der Ersetzung zusammen mit allen inzidenten Kanten gelöscht werden.
- Knoten und Kanten der rechten Seite werden neu erstellt
- Im unteren Viertel der Regel stehen die ausgezeichneten Knoten und Kanten, die während der Ersetzung nicht verändert werden.

- Die obere Seite des X enthält Knoten, die auf beliebig viele andere Knoten im Ausgangsgraphen matchen können, und macht damit die komplexen einbettenden Ersetzungen möglich.

Kanten im Diagramm, die zusätzlich mit einem ‚-' markiert sind, müssen vor Anwendung der Regel im Ausgangsgraphen vorhanden sein, müssen aber während der Ersetzung entfernt werden. Dagegen kennzeichnen ‚+' Zeichen Kanten, die vor der Ersetzung vorhanden sein können, nach der Ersetzung aber vorhanden sein müssen.
DELTA Regeln ähneln denen aus PAGG sehr stark, nur dass hier ein Dreieck zur Abtrennung der einzelnen Bereiche gewählt wurde. Im Einzelnen entspricht der Bereich rechts des Dreiecks dem rechts des X, der links des Dreiecks dem links des X und der innere Bereich des Dreiecks dem oberen Teil des X in PAGG. Weiterhin werden ausgezeichnete Knoten, die auf mehrere Knoten matchen können, mit Hilfe eines ‚*' bezeichnet.
Zusätzlich können in PROGRESS und DELTA noch weitere Bedingungen an den Ausgangsgraphen formuliert werden. Hierzu können beliebig große Graphen, so genannte Kontextgraphen, noch zusätzlich an den Graphen der linken Seite angehängt werden, die dann einen gewissen Kontext für die Ersetzung entweder fordern oder nicht zulassen. Diese Option wurde in unserem Rahmenwerk durch die zusätzlichen Nebenbedingungen an die Substruktur beschrieben. Der zusätzliche Graph darf natürlich nur ausgezeichnete Knoten und Kanten enthalten, da er während der Ersetzung ja unverändert bleibt und nur der Bestimmung von Kontextbedingungen dient.

3.3 Knotenattribute

Die Behandlung von Knotenattributen in den einzelnen Ansätzen soll nun in diesem Kapitel vorgestellt werden. Knotenattribute sind zusätzliche Informationen, die in den Knoten gespeichert werden und natürlich auch verändert und angepasst werden sollen. Daher unterstützen die drei vorgestellten Ansätze die Definition von Knotenattributen, deren Typen meist Strings oder Zahlen sind. Diese Attribute werden wie ein eigenes Objekt behandelt, das an den Knoten angehängt wird. Um auf solchen Attributen zu arbeiten, kann in den graphischen Ersetzungsregeln zusätzlich textuell beschrieben werden, wie mit den Knotenattributen umzugehen ist (vgl. Abb. 3). Bei Knotenattributen gilt es zwei verschiedene Typen zu unterscheiden. Zum einen die *intrinsische Attribute* (intrinsic attributes), die externe Eigenschaften eines Knotens im Graphen beschreiben, und die nur durch explizite Zuweisungen verändert werden können. Zum anderen die *abgeleiteten Attribute* (derived attributes), die über Attribute anderer Knoten definiert sind, und somit ständig konsistent gehalten werden müssen, sich also auch implizit ändern können.

In PROGRESS können beide Arten von Attributen behandelt werden. Die PROGRESS Produktion in Abbildung 3 manipuliert z.B. das intrinsische Knotenattribut ‚income'. Intrinsische Attribute können also innerhalb einer solchen Ersetzungsregel manipuliert werden, abgeleitete Attribute dagegen sind in ihrer Handhabung komplexer. Sie müssen erst einmal als Teil des Graphschemas angegeben sein und es müssen Referenzen zu allen notwendigen Attributen aus anderen Knoten angelegt werden. Von einer genaueren Betrachtung der ableitbaren Attribute wird hier abgesehen, da dies den Rahmen dieser Ausführung sprengen würde.

PAGG macht in seiner Behandlung von Knotenattributen keinen Unterschied zwischen intrinsischen und abgeleiteten Attributen. Es werden Gleichungen zu den Ersetzungsregeln hinzugefügt, in denen die Berechnung der jeweiligen Attribute eines Knotens angegeben ist. Hierbei wird, wie gesagt, nicht unterschieden, ob ein Attribut einmal einen berechneten Wert bekommt und diesen dann behält (intrinsische Attribute) oder, ob man ein Attribut betrachtet, dessen Wert ebenfalls durch andere Produktionen implizit verändert werden kann (abgeleitete Attribute). Bei dieser Art der Attributbehandlung kann es natürlich zu Inkonsistenzen kommen, allerdings ist diese Vorgehensweise um einiges einfacher zu realisieren.
DELTA schließlich bietet nicht die Möglichkeit mit abgeleiteten Attributen umzugehen.

3.4 Vergleich der Ansätze

Abschließend sollen noch einmal die drei diskutierten Ansätze, PROGRESS, PAGG und DELTA, in Form einer zusammenfassenden Tabelle gegenübergestellt werden. Diese soll zeigen, ob und wie die Ansätze mit den unterschiedlichen Punkten der Strukturersetzung umgehen.

Eigenschaft/Ansatz	**PAGG**	**PROGRESS**	**DELTA**
Intrinsische Knotenattribute	kein Unterschied zu ableitbaren Attributen	einzeln möglich	möglich
Ableitbare Knotenattribute	kein Unterschied zu inneren Attributen	einzeln möglich	Nicht möglich
Homo/Isomorpher Match	nur isomorph	homo/isomorph	homo/isomorph
Match auf Knotenmengen	einbettender Graph	Knoten, die auf Mengen matchen	Knoten, die auf Mengen matchen
Löschen von ausgezeichneten Knoten	unmöglich	Verboten	verboten
Behandlung von ‚freien' Kanten	automatisch gelöscht	automatisch gelöscht	automatisch gelöscht
Graphische/textuelle Ersetzungsregeln	graphisch	graphisch sowie textuell	graphisch
Verwendung von ausgez. Knoten	ja	ja	ja

Tabelle 1 (Übersicht der praktischen Ansätze)

Im zweiten Teil der Ausarbeitung werden zuerst theoretisch die für die eigentliche programmierte Graphersetzung notwendigen Kontrollstrukturen, wie z.B. BCF Operatoren, eingeführt und schließlich die bereits hier betrachteten Ansätze bezüglich der verwendeten Kontrollstrukturen verglichen.

4 Kontrollstrukturen

Die Theorie der Strukturersetzung, die in Kapitel 2 erläutert wurde, bietet die Möglichkeit, schemakonsistente Strukturersetzungsregeln zu definieren und diese auf eine gegebene Struktur anzuwenden. Um jedoch komplexere Vorgänge wie z.B. die Hintereinanderausführung von Ersetzungsregeln zu definieren, benötigen wir Mechanismen zur Steuerung und Kontrolle. Es gibt in der Literatur verschiedene Ansätze, um solche Mechanismen umzusetzen.

In diesem Kapitel wollen wir die *Basic Control Flow Operatoren* (BCF-Operatoren) vorstellen, eine minimale Menge von Kontrollstrukturen, welche grundlegende Aktionen bezüglich der Ersetzungsregeln definieren. Die BCF-Operatoren sind ein wichtiges Werkzeug, das im Kontrollmechanismus von PROGRESS angewandt wird. Zum einen kann dadurch eine imperative Kontrolle verwirklicht werden, die es ermöglicht, die Abfolge von Ersetzungsregeln selbst vorzugeben und zu programmieren. Zum anderen kann die imperative Kontrolle zum Mechanismus der *Control Flow Graphen* weiterentwickelt werden, wodurch sich eine lesbare und leicht zugängliche graphische Darstellung ergibt, wie sie in PROGRESS verwendet wird.

4.1 Voraussetzungen

Die Kontrollstrukturen sollen es ermöglichen, die bezüglich einer Struktur ausgewählten Ersetzungsregeln einfach anordnen und kombinieren zu können. Eine so programmierte Transformation der Struktur muss die folgenden Eigenschaften besitzen:

1. *Atomarer Charakter:* Eine programmierte Transformation besteht aus einzelnen atomaren Ersetzungsschritten, die sequentiell hintereinanderausgeführt werden. Eine gegebene Struktur wird demzufolge genau dann verändert, wenn alle Ersetzungsschritte erfolgreich sind.
2. *Boolesche Eigenschaft:* Wird eine programmierte Transformation ausgeführt, so kommen für das Resultat nur zwei Möglichkeiten in Betracht: entweder schlägt die gesamte Transformation fehl (*abort*) und die gegebene Struktur wird nicht verändert oder sie ist erfolgreich (*commit*) und die gegebene Struktur wird transformiert. Je nach Resultat wird die Anwendung die nachfolgende Ersetzungsregel des commit - oder des abort - Pfades im Kontrollmechanismus verfolgen. Diese Eigenschaft soll sowohl für Transformationen als auch für einzelne Ersetzungsregeln gelten.
3. *Nichtdeterminismus:* Es kann der Fall sein, dass eine Ersetzungsregel bezüglich der gegebenen Struktur verschiedene Anwendungsmöglichkeiten hat. D.h. es gibt mehr als eine Substruktur, die der linken Seite der Ersetzungsregel entspricht. Ist dies der Fall, so muss eine nichtdeterministische Auswahl gewährleistet werden, z.B. durch Anwendung von Backtracking.
4. *Bewahrung der Konsistenz:* Die Anwendung einer programmierten Transformation darf die Konsistenz der gegebenen Struktur nicht verletzen. Diese kann durch eine Menge von Integritätsbeschränkungen gegeben sein (siehe Kapitel 2.2).
5. *Rekursion:* Der rekursive Aufruf von programmierten Transformationen ist erlaubt. Diese Eigenschaft verleiht dem Kontrollmechanismus mehr Ausdrucksstärke und erleichtert den Umgang mit rekursiv definierten Strukturen.

Im Folgenden werden wir programmierte Graph-Transformationen, die die geforderten Eigenschaften erfüllen, als *Transaktionen* bezeichnen.

```
<Transaction> ::=     <TransactionId> "=" <BCFExpr> ;
<BCFExpr> ::=         <BasicAction> | <ActionCall> | <BCFTerm> ;
<BasicAction> ::=     "skip" | "loop" ;
<ActionCall> ::=      <RuleId> | <TransactionId> ;
<BCFTerm> ::=         "def" "(" <BCFExpr> ")" | "undef" "(" <BCFExpr> ")" |
                      "(" <BCFExpr> ";" <BCFExpr> ")" |
                      "(" <BCFExpr> "[]" <BCFExpr> ")"
                      "(" <BCFExpr> "&" <BCFExpr> ")";
```

Abbildung 5: Syntax von Graph-Transaktionen und BCF-Ausdrücken

4.2 BCF Operatoren und Semantik

4.2.1 BCF - Operatoren

Die Definition der BCF-Operatoren (siehe Abb. 5) folgt dem Gedanken, dass wir nur eine minimale Menge an Kontrollstrukturen benötigen, die jedoch untereinander beliebig kombiniert werden können. Eine solche Kombination bezeichnen wir als *BCF-Ausdruck*. Eine *Graph-Transaktion* ist definiert als funktionale Abstraktion eines BCF-Ausdrucks.

Es gibt drei verschiedene Ausprägungen eines BCF-Ausdrucks. Falls es sich um eine *einfache Aktion* (*BasicAction*) handelt, so gibt es zwei Möglichkeiten:

- skip - entspricht der identischen Abbildung des gegebenen Graphen auf sich selbst. Dieser Operator ist stets erfolgreich.
- loop - entspricht einer nichtterminierenden Berechnung auf dem gegebenen Graphen, die weder fehlschlägt noch erfolgreich ist.

Des weiteren kann es sich bei dem BCF-Ausdruck um einen *Aktionsaufruf* (*ActionCall*) handeln, der über die entsprechenden Identifikationssymbole entweder eine einfache Ersetzungsregel oder eine andere Graph-Transaktion aufruft. Die dritte Ausprägung des BCF-Ausdrucks ist der *BCF-Term*, welcher aus den folgenden fünf *BCF-Operatoren* zusammengesetzt sein kann:

- def(*a*) testet, ob die Aktion *a* auf der gegebenen Struktur F erfolgreich ist. Falls ja, so ist der Operator erfolgreich und F wird unverändert zurückgegeben.
- undef(*a*) testet, ob die Aktion *a* auf der gegebenen Struktur fehlschlägt. Falls ja, so ist der Operator erfolgreich und F wird unverändert zurückgegeben.
- (*a* ; *b*) bezeichnet die Hintereinanderausführung der Aktionen *a* und *b*, wobei zuerst *a* auf die gegebene Struktur angewandt wird und danach *b* „auf ein passendes Resultat“ der Anwendung von *a*.
- (a b) repräsentiert die nichtdeterministische Auswahl bezüglich der Anwendung von Aktion *a* oder Aktion *b*.
- (a & b) berechnet das Resultat der gemeinsamen Anwendung von *a* und *b* auf der gegebenen Struktur und gibt als Ergebnis die Struktur nach erfolgreicher paralleler Anwendung beider Aktionen zurück. (hierfür wird ein Operator benötigt, der die Gleichheit bzw. Isomorphie zweier Graphen testet)

Durch die Forderung der booleschen Eigenschaft (siehe Kapitel 4.1) für Ersetzungsregeln und Transaktionen muss im Folgenden nicht zwischen Ausdrücken

unterschieden werden, die Seiteneffekte auf den gegebenen Graphen haben und solchen, die den Graph unverändert lassen. Ein Ausdruck von der Form

$$(Cond_1 \rightarrow Body_1 \quad ... \quad Cond_n \rightarrow Body_n)$$

kann ersetzt werden durch

$$(\underline{def}(Cond_1); Body_1 \quad ... \quad \underline{def}(Cond_n) ; Body_n).$$

In diesem Fall wird eine Änderung an dem Graphen ($Body_i$) nur dann ausgeführt, falls die Anwendung von ($Cond_i$) terminiert. Ist dies nicht der Fall, so wird der Graph ohne Änderung zurückgegeben.

In der Beschreibung der BCF-Operatoren gibt es noch eine unsauber formulierte Aussage bezüglich der gewünschten Semantik der Hintereinanderausführung (Konkatenation) zweier Aktionen *a* und *b*. Die Aktion *b* soll "auf ein passendes Resultat" angewandt werden. Dies hat mit der geforderten Eigenschaft des Nichtdeterminismus der Operatoren zu tun (siehe Kapitel 4.1). Angenommen die Ersetzungsaktion *a* auf einer gegebenen Struktur F hat drei verschiedene Anwendungsmöglichkeiten und kann somit drei verschiedene Resultate F1, F2 und F3 zurückgeben. Es kann jedoch sein, dass die Ersetzungsaktion *b* bei Anwendung auf F1 fehlschlägt, jedoch bei Anwendung auf F2 und F3 erfolgreich ist. In diesem Fall sind also nur die Strukturen F2 und F3 passende Resultate der Anwendung von *a*, nicht aber die Struktur F1. Es muss also gewährleistet sein, dass bei der Anwendungsfolge *a* auf F und *b* auf F1 die gesamte Transformation nicht als Fehlschlag verworfen wird, sondern dass ein geeigneter Mechanismus vorhanden ist, der die zweite Anwendung wieder rückgängig machen kann. Wir müssen aufgrund dieses Nichtdeterminismus also schon im Vorhinein Informationen über das Resultat der Anwendung einer Aktion haben. Dies lässt sich durch *Tiefensuche* in Verbindung mit *Backtracking* erreichen.

Ein weiterer Aspekt des Nichtdeterminismus ist die Unterscheidung von nichtterminierenden und fehlschlagenden Berechnungen. Angenommen es soll bezüglich einer gegebenen Struktur F durch den BCF-Operator (*a* *b*) die Auswahl einer Aktion getroffen werden, wobei *a* auf F eine nichtterminierende Berechnung ausführt, *b* jedoch eine Menge an definierten Resultaten bzgl. F liefert. In diesem Fall soll bei der Tiefensuche eine fehlschlagende Berechnung per Backtracking übersprungen werden, jedoch muss eine nichtterminierende Berechnung als mögliches Resultat festgehalten werden.

Im folgenden Beispiel wird die Graph-Transaktion *Marry** definiert, um in einer gegebenen Datenbank alle Personen einer *Marry* - Relation zuzuordnen. Dabei wird die Ersetzungsregel *Marry*, wie sie in Kapitel 2.3 definiert wurde, verwendet. Außerdem verwenden wir eine Ersetzungsregel *MarkUnmarried*, welche in dem gegebenen Graphen alle einzelnen Personen auswählt und markiert, die nicht in einer *Marry* - Relation stehen. Falls es keine solche Person gibt, endet sie mit einem Fehlschlag.

Beispiel 4.1 (Transaktion Marry für eine Personendatenbank)*

Marry* = (((def(MarkUnmarried) ; Marry) ; Marry*)
undef(MarkUnmarried))

Die Transaktion *Marry** prüft zunächst die Existenz von nichtverheirateten Personen in der Datenbank. Falls solche gefunden werden, so wird die Ersetzungsregel *Marry* ausgeführt und *Marry** wird erneut aufgerufen. Andernfalls existiert keine unverheiratete Person in der Datenbank und der undef - Operator ist erfolgreich. Die gesamte Transaktion ist genau dann erfolgreich, wenn alle Personen der Datenbank am

Ende in einer *Marry* - Relation stehen. Die Anwendung von *Marry** benötigt eine Tiefensuche mit Backtracking. Falls z.B. nur eine ungerade Anzahl an Personen noch verheiratet werden müssen, so wird der ausführende Mechanismus alle gültigen Kombinationen ausprobieren, dann aber mit Fehlschlag abbrechen müssen, weil immer eine Person übrigbleibt, die nicht verheiratet werden kann. In diesem Fall bricht die Anwendung ab und der Graph wird unverändert zurückgegeben.

Wendet man *Marry** auf die Personendatenbank aus Kapitel 2.1 an, so kann der ausführende Mechanismus z.B. damit beginnen, zuerst Linus und Mary und danach Harry und Sally zu verheiraten. Wurden diese zwei Ersetzungsregeln ausgeführt, so ist kein gültiger Partner für Hannah mehr übrig, da der Partner kein Bruder sein darf (siehe Definition 2.12: Struktur mit Nebenbedingungen). Der Mechanismus wendet also Backtracking an und sucht eine Alternative zu der *Marry*-Relation von Harry und Sally. Es kann z.B. versucht werden, Harry und Hannah zu verheiraten.

4.2.2 Semantik der Transaktionen

Um ein Programmiertes Strukturersetzungssystem generell definieren zu können, benötigen wir weitere Definitionen, um vor allem die Semantik von Hintereinanderausführungen mit Rekursion behandeln zu können.

In der Literatur [Schürr97] wird eine Fixpunktsemantik verwendet, um rekursiv definierte Transaktionen zu behandeln. Zum einen kann die Existenz eines kleinsten Fixpunktes für jede beliebige rekursive Menge von Transaktionen nachgewiesen werden, und zum anderen kann ein Algorithmus erstellt werden, der die Resultate von terminierenden Transaktionen berechnet und die Resultate von potentiell nichtterminierenden Transaktionen approximiert. Das Aufstellen dieser *Berechnungsfunktion* bedarf der Anwendung eines Fixpunkt-Theorems (*Proposition 7.3.1* in [Schürr97]) und der Formalisierung der Semantischen Domäne von Transaktionen, sowie der Einführung einer partiellen Ordnung auf dieser Domäne.

Im Folgenden werden diese Voraussetzungen kurz erläutert.

Definition 4.1 (Semantische Domäne von Transaktionen)

Sei $S \in S(\Sigma)$ ein Schema und sei $\mathcal{AL}(S)$ eine S-konsistente Klasse von abstrakten Strukturen. Dann ist die *Semantische Domäne* D definiert als Potenzmenge von binären Relationen:

$$\mathsf{D} := 2^{AL(S)\times(AL(S)\times\{\infty\})}.$$

Eine betrachtete Transaktion kann also dargestellt werden als eine Menge von Tupeln (F, F'), welche aus zwei Strukturen bestehen, wobei F die Eingabe und F' die Ausgabe der Ersetzung bezeichnet. Falls es bei Anwendung einer Transaktion zu einer nichtterminierenden Berechnung kommt, wird dies in der zweiten Komponente des Tupels mit dem ∞ - Symbol gekennzeichnet. Eine Tupelmenge aus D wird nun so interpretiert, dass sie das Resultat einer rekursiven Transaktion approximieren soll. Um die Qualität einer solchen Approximation mit anderen möglichen Approximationen vergleichen zu können, benötigen wir eine partielle Ordnung auf der Semantischen Domäne D.

Definition 4.2 (Partielle Ordnung auf der Semantischen Domäne)
Seien R, $R' \in \mathsf{D}$ zwei Mengen von Transaktionen (Tupeln) bezüglich eines Strukturschemas S. Dann ist eine *partielle Ordnung* „$\leq$“ definiert wie folgt:

$$R \leq R' :\Leftrightarrow \forall F, F' \in \mathcal{L}(S): \ (F, F') \in R \Rightarrow (F' = \infty \vee (F, F') \in R') \\ \wedge (F, \infty) \notin R \Rightarrow ((F, F') \in R \vee (F, F') \notin R') .$$

Die partielle Ordnung besagt also, dass $R_1 \leq R_2$ genau dann gilt, wenn die Tupelmenge R_2 die gegebene rekursive Transaktion besser approximiert als R_1. Dabei steht jede Eingabe F sowohl in R_1 als auch in R_2 entweder zu der selben Menge von Ausgaben F' in Relation, oder in R_1 gibt es wegen (F, ∞) keine Approximation, während in R_2 eine potentiell größere Menge an Ausgaben existiert. Für die Menge aller *potentiell nichtterminierenden Berechnungen* $R_\infty := AL(S) \times \{\infty\}$ gilt demzufolge: $R_\infty \leq$ R' für alle R' $\in \mathsf{D} \setminus R_\infty$, da diese Menge jede Transaktion schlechter approximiert als eine gegebene Menge, die mindestens eine terminierende Berechnung enthält.
Zur Anwendung des Fixpunkt-Theorems muss schließlich noch gelten, dass für jede Folge von Tupelmengen aus D ein größtes Element R existiert, welches eine bessere Approximation liefert als jedes Element der Tupelmenge, und welches in der Semantischen Domäne D das geringste Element mit dieser Eigenschaft ist. Dies wird in [Schürr97] mit dem Beweis des *Lemma 7.3.4* gezeigt.

Definition 4.3 (BCF-Ausdrücke und Transaktionen)
Sei $S \in S(\sum)$ ein Schema, D die Semantischen Domäne und $P \subseteq \mathcal{P}(\sum)$ eine Menge von Strukturersetzungsregeln. $\mathsf{E}(P)$ bezeichnet die Menge aller BCF-Ausdrücke und $\mathcal{T}(P)$ die Menge aller *Transaktionen* über der Menge P der Strukturersetzungsregeln. Die Syntax dieser Mengen ist definiert wie in Abbildung 5.
Mit den abstrakten Strukturen F, F', $F'' \in \mathcal{AL}(S)$ und mit a, $b \in \mathsf{E}(P)$ ist die *semantische Funktion für BCF-Ausdrücke* $\mathsf{R}_P : \mathsf{E}(P) \rightarrow \mathsf{D}$ definiert wie folgt:

(1) $(F, F') \in \mathsf{R}_P$ skip $:\Leftrightarrow F = F'$
(2) $(F, F') \in \mathsf{R}_P$ loop $:\Leftrightarrow F' = \infty$
(3) $(F, F') \in \mathsf{R}_P\ p :\Leftrightarrow F \ {}_p\!\Rightarrow F'$ für eine Produktion $p \in \mathcal{P}(\Sigma)$ oder
$F' = \infty$ falls die Anwendung von p nicht terminiert
(4) $(F, F') \in \mathsf{R}_P$ def(a) $:\Leftrightarrow \exists F'' \neq \infty : (F, F'') \in \mathsf{R}_P\ a \ \wedge \ F = F'$
oder $(F, \infty) \in \mathsf{R}_P\ a \ \wedge \ F' = \infty$
(5) $(F, F') \in \mathsf{R}_P$ undef(a) $:\Leftrightarrow (\neg\exists F'' : \ (F, F'') \in \mathsf{R}_P\ a\ ) \wedge \ F = F'$
oder $(F, \infty) \in \mathsf{R}_P\ a \ \wedge \ F' = \infty$
(6) $(F, F') \in \mathsf{R}_P\ (a\,;b) :\Leftrightarrow \exists F'' \neq \infty : (F, F'') \in \mathsf{R}_P\ a \ \wedge (F'', F') \in \mathsf{R}_P\ b$
oder $(F, \infty) \in \mathsf{R}_P\ a \ \wedge \ F' = \infty$
(7) $(F, F') \in \mathsf{R}_P\ (a \ \ b) :\Leftrightarrow (F, F') \in \mathsf{R}_P\ a \ \vee (F, F') \in \mathsf{R}_P\ b$
(8) $(F, F') \in \mathsf{R}_P\ (a \,\&\, b) :\Leftrightarrow F' = \infty \wedge (\,(F, \infty) \in \mathsf{R}_P\ a \ \wedge (F, \infty) \in \mathsf{R}_P\ b\ )$
oder $(F, F') \in \mathsf{R}_P\ a \ \wedge (F, F') \in \mathsf{R}_P\ b$

Betrachtet man die Definition der Semantik von def und undef genauer, so stellt sich heraus, dass die Ausdruckskraft des undef - Operators stärker ist als die des def - Operators. Für beide Operatoren gilt, dass sie eine unendliche Berechnung ausführen ($F' = \infty$), falls die Anwendung von a auf der gegebenen Struktur F unendlich lange läuft. def(a) terminiert erfolgreich und gibt die gegebene Struktur F zurück, falls zumindest eine erfolgreiche Berechnung von a auf F existiert. Falls a auf F fehlschlägt, so schlägt auch def(a) fehl. Andererseits terminiert undef(a) erfolgreich, falls keine erfolgreiche Berechnung von a auf F gefunden werden kann. Falls für a jedoch eine erfolgreiche Berechnung auf F gefunden wird, die nicht unendlich lange läuft, so schlägt undef(a) fehl. Der undef - Operators berücksichtigt also stärker als der def - Operator die Möglichkeit einer unendlich langen Berechnung.

Um auch der Definition von def(a) eine stärkere Ausdruckskraft zu verleihen, können wir jedoch auf den undef - Operator zurückgreifen und diesen doppelt anwenden, damit auch bei einer Überprüfung der erfolgreichen Berechnung von a auf F unendliche Berechnungen ausgeschlossen werden können.

Definition 4.4 (der def $^{\infty}$ *- Operator)*

Seien $F, F', F'' \in \mathcal{AL}(S)$ abstrakte Strukturen und sei $a \in \mathsf{E}(P)$ ein BCF-Ausdruck. Eine stärkere Version des def - Operators kann wie folgt definiert werden:

$$
\begin{aligned}
(F, F') \in \mathsf{R}_P\ \underline{\mathrm{def}}^{\infty}(a) \quad : \quad & (F, F') \in \mathsf{R}_P\ \underline{\mathrm{undef}}(\underline{\mathrm{undef}}(a)) \\
& \Big(\quad F'' : \ (F, F'') \in \mathsf{R}_P\ \underline{\mathrm{undef}}(a) \ \Big) \quad F = F' \\
& \quad \text{oder} \qquad (F, \infty) \in \mathsf{R}_P\ \underline{\mathrm{undef}}(a) \qquad F' = \infty \\
& \Big(\quad F'' \neq \infty : \ (F, F'') \in \mathsf{R}_P\ a \\
& \qquad\qquad\quad (F, \infty) \in \mathsf{R}_P\ a \qquad F = F' \Big) \\
& \quad \text{oder} \qquad (F, \infty) \in \mathsf{R}_P\ a \qquad F' = \infty
\end{aligned}
$$

Der $\underline{\mathrm{def}}^{\infty}$- Operator ist geeignet, um den korrekten Ablauf einer Transaktion sicherzustellen. Er wird in Kapitel 5.3 im Kontrollmechanismus von Control Flow Graphen angewandt.

Mit der Semantik aus Definition 4.3 kann nun ein BCF-Ausdruck E, der allgemein n Aufrufe von Transaktionen $t_1, ..., t_n$ beinhalten kann, wie folgt als Funktion notiert werden:

$$\mathsf{R}_P\ E : \mathsf{D}^n \rightarrow \mathsf{D}.$$

Da die Transaktionen $t_1, ..., t_n$ eine Semantik $\mathsf{R}_P\ t_1, \ldots, \mathsf{R}_P\ t_n$ besitzen, ist die Semantik von E nach Definition 4.3 wie folgt gegeben:

$$\mathsf{R}_P\ E\ [\mathsf{R}_P\ t_1, \ldots, \mathsf{R}_P\ t_n\ ].$$

Auf diese Weise kann die Semantik jedes BCF-Ausdrucks und jeder Transaktion dargestellt werden, solange sie keine rekursiven Aufrufe beinhalten. Um jedoch eine rekursive Transaktion - wie z.B. *Marry** aus Beispiel 4.1 - semantisch korrekt zu interpretieren, muss mit Hilfe des Fixpunkt-Theorems ein geringstes Element $R \in \mathsf{D}$ berechnet werden, damit die folgende Gleichung angewandt werden kann:

$$R = \mathsf{R}_P\ (((\underline{\mathrm{def}}(\ \mathrm{MarkUnmarried}\ )\ ;\ \mathrm{Marry})\ ;\ \mathrm{Marry}^*)\ \underline{\mathrm{undef}}(\ \mathrm{MarkUnmarried}))\ [R].$$

Um somit eine Fixpunktsemantik für Transaktionen zu erhalten, muss aufgrund der Vorraussetzungen des Fixpunkt-Theorems (*Proposition 7.3.1*) noch gezeigt werden,

dass BCF-Operatoren - und damit alle BCF-Ausdrücke - monoton sind bezüglich der in Definition 4.2 festgelegten partiellen Ordnung. Dies geschieht mit *Lemma 7.3.7* in [Schürr97], womit es möglich ist, das folgende Korollar zu zeigen.

Korollar 4.5 (Fixpunktsemantik für Transaktionen)
Sei T $\{t_1, ..., t_n\}$ $\mathcal{T}(P)$ eine Menge von Transaktionen, die innerhalb ihrer zugehörigen BCF-Ausdrücke $E_1, ..., E_n$ $\mathsf{E}(P)$ keine anderen Transaktionen als t_1 bis t_n und keine anderen Strukturersetzungsregeln als solche aus P $\mathcal{P}(\Sigma)$ aufrufen. D.h.
$t_1 = E_1[t_1, ..., t_n], ..., t_n = E_n[t_1, ..., t_n]$.
Mit $\mathcal{AL}(\Sigma)$ als zugrundeliegende Klasse von abstrakten, schemakonsistenten Strukturen gilt die folgende Behauptung, die eine *semantische Funktion für Transaktionen* R_P^* : T $\rightarrow$ D definiert:

(1) $t_1, ..., t_n$ haben eindeutige, geringste Fixpunkte $\mathsf{R}_P^* t_1, ..., \mathsf{R}_P^* t_n$ D
(2) Diese Fixpunkte lassen sich wie folgt approximieren:
R^0_1 $\mathcal{AL}(S) \times \{\infty\}; ...; R^0_n$ $\mathcal{AL}(S) \times \{\infty\}$
R^{k+1}_1 $\mathsf{R}_P^* E_1 [R^k_1, \ldots, R^k_n]; \ldots; R^{k+1}_n$ $\mathsf{R}_P^* E_n [R^k_1, \ldots, R^k_n]$ wobei
$\mathsf{R}_P^* E_i : \mathsf{D}^n \rightarrow \mathsf{D}$ Approximationen der Transaktionen $t_1, ..., t_n$ als Eingabe erhält und eine neue Approximation t_i durch Anwendung von Definition 4.3 auf den BCF-Ausdruck E_i berechnet.

Beweis: siehe [Schürr97]

Das Korollar 4.5 liefert eine Funktion, die das Resultat von terminierenden Transaktionen berechnet.

Basierend auf den Überlegungen zur Semantik von Transaktionen kann nun allgemein ein *PLSR-System* definiert werden. Ein solches System beschreibt die grundlegenden Eigenschaften des Kontrollmechanismus, der im Rahmen eines Strukturersetzungssystems verwendet wird. Es dient uns im nachfolgenden Kapitel zum Vergleich der verschiedenen Ansätze, Kontrollmechanismen umzusetzen.

Definition 4.6 (PLSR System)
Ein *programmiertes logik-basiertes Strukturersetzungs-System (PLSR-System)* unter Verwendung einer Signatur Σ ist ein 5-Tupel PLSR (S, A, P, T, t). Es gilt:

(1) $S \in S(\Sigma)$ ist ein Strukturschema.
(2) A $\mathcal{AL}(S)$ ist die schemakonsistente Ausgangsstruktur.
(3) P $\mathcal{P}(\Sigma)$ ist eine Menge von Strukturersetzungsregeln.
(4) T $\mathcal{T}(P)$ ist eine Menge von (rekursiv definierten) Transaktionen über P.
(5) t T ist die *Haupt-Transaktion* des PLSR-Systems.

Definition 4.7 (Sprache eines PLSR Systems)
Die Sprache $\mathcal{AL}$ (PLSR) eines gegebenen PLSR (S, A, P, T, t) ist wie folgt definiert:
F $\mathcal{AL}$(PLSR) : (A, F) R_P^* t $F \neq \infty$.

Die Sprache eines PLSR lässt sich also bestimmen, indem man eine Haupt-Transaktion t auf die Ausgangsstruktur A anwendet. Die Transaktion t kann nun

Strukturersetzungsregeln aus *P* anwenden oder andere Transaktionen aus der Menge *T* aufrufen, für die dasselbe gilt. Die Menge $\mathcal{AL}$ (PLSR) umfasst alle Strukturen, die sich auf diese Weise durch terminierende Berechnungen ermitteln lassen.

5 Verwendung der Kontrollstrukturen

In diesem Kapitel sollen verschiedene Ansätze untersucht werden, die einen Mechanismus zur Anwendung von Kontrollstrukturen liefern. Generell kann man drei Arten von Ansätzen unterscheiden:

- Der *deklarative* Ansatz, der in Abschnitt 5.1 beschrieben wird, geht von einer Regel-basierten Sichtweise aus, wobei z.B. Prioritäten für die Ausführung von Strukturersetzungsregeln definiert werden können, um den Ablauf einer Transaktion zu steuern.
- Abschnitt 5.2 erläutert den *imperativen* Ansatz, der von einer Anwendungs-orientierten Sichtweise ausgeht und auf die Kontrollmechanismen von imperativen Programmiersprachen zurückgreift.
- Im Abschnitt 5.3 soll schließlich mit den *Control Flow Graphen* eine Erweiterung des imperativen Ansatzes vorgestellt werden, die ebenfalls anwendungsorientiert ist, die jedoch den Ablauf einer Transaktion nicht rein textuell, sondern graphisch, mit Hilfe von Diagrammen steuert.
- Der Abschnitt 5.4 befasst sich mit den in PROGRESS verwendeten Control Flow Diagrammen, einer Darstellung, die teils graphisch, teils textuell die Kontrollstrukturen des Ersetzungssystems definiert.

Um die vorgestellten Ansätze vergleichen zu können, wird das folgende Beispiel für die komplexere Transformation einer Personendatenbank betrachtet. Die hierbei verwendete Ersetzungsregel *DeleteUnmarried* ist so definiert, dass sie fehlschlägt, wenn alle Personen in der Datenbank verheiratet sind. Andernfalls wählt sie nichtdeterministisch eine nichtverheiratete Person aus und löscht diese.

Beispiel 5.1 (Komplexe Transformation für eine Personendatenbank)

```
Marry*= ((( def( MarkUnmarried ) ; Marry) ; Marry*)   undef( MarkUnmarried )).
Main   = ( Marry*   (undef( Marry*); (DeleteUnmarried; Main)) ).
```

Die Ersetzungsregeln *Marry**, *Marry* und *MarkUnmarried* sind wie in Beispiel 4.1 gegeben. Zusätzlich ist die Regel *Main* eingeführt, welche zunächst versucht, per Aufruf von *Marry** alle noch nicht verheirateten Personen in die *Marry*-Relation zu stellen. Falls das nicht gelingt, d.h. falls die *Marry**-Regel mit einem Fehlschlag terminiert, wird die Alternative ausgeführt. Die Ersetzungsregel *DeleteUnmarried* löscht eine unverheiratete Person aus der Datenbank, welche existieren muss, da im Falle eines Fehlschlags von *Marry** noch nicht alle Personen verheiratet sind. Anschließend ruft sich *Main* rekursiv auf und wiederholt die Transaktion. Der gesamte Prozess endet also, wenn entweder alle Personen verheiratet oder aus der Datenbank gelöscht sind.

5.1 Deklarativer Ansatz

Die Ansätze, die in diesem Abschnitt vorgestellt werden, gehen alle davon aus, dass eine übergeordnete Suchschleife ausgeführt wird, welche die Ersetzungsregeln solange auf die gegebene Struktur anwendet, wie es möglich ist. Damit wird strikt die

Vorgehensweise aus Definition 4.7 übernommen, in der - beginnend mit einer Ausgangsstruktur - mit Hilfe einer anfänglichen Haupt-Transaktion t alle möglichen resultierenden Strukturen abgeleitet werden. Das bedeutet allerdings, dass die sukzessive Anwendung von Ersetzungsregeln im Vordergrund steht und es für den Anwender keine Möglichkeit gibt, eigene Bedingungen zu definieren, welche die Berechnung terminieren lassen. Zudem kann bei diesem Ansatz nicht ohne weiteres auf erfolgreiche oder fehlschlagende Transaktionen reagiert werden. Um Kontrollstrukturen wie den def- oder undef-Operator definieren zu können, sind weitere Bedingungen und neue Ersetzungsregeln notwendig. Aus diesem Grund sind die deklarativen Ansätze nicht geeignet, um eine komplexere Transaktion wie die *Main*-Transaktion aus Beispiel 5.1 zu realisieren. Im Folgenden werden daher drei deklarative Ansätze nur vorgestellt, aber nicht zur Umsetzung des Beispiels verwendet.

Definition 5.1 (Graph Grammatik mit Regel-Prioritäten)
Eine *Graph Grammatik mit Regel-Prioritäten* besteht aus einem Axiom und einer Menge *P* von Graphersetzungsregeln, auf denen eine partielle Ordnung definiert ist, so dass:

$P \quad \{ p_{1,1}, ..., p_{1,k(1)} \} > ... > \{ p_{n,1}, \ldots, p_{n,k(n)} \}$

mit n N und k: $N \rightarrow N$. Mit N $\{1, 2, ... \}$ bezeichnet k die Anzahl der Regeln auf Prioritätsstufe i.
Die Semantik der Prioritäten ist durch folgende Transaktion definiert:

$t \quad ((t'; t) \quad \underline{skip})$

wobei

$t' \quad (t_1 \quad (\underline{undef}(t_1) ; t_2) \quad ... \quad (\underline{undef}(t_1) \,\&\, ... \,\&\, \underline{undef}(t_{n-1})) ; t_n))$

$t_i \quad (p_{i,1} \quad ... \quad p_{i,k(i)})$ für i = 1, ..., n.

Per Definition wird also in dieser Graph Grammatik die Ausführung einer Regel mit geringerer Priorität verhindert, falls es eine Regel mit höherer Priorität gibt. Durch Anwendung der Transaktion t wird immer zuerst die Menge von Regeln betrachtet, welche die höchste Priorität hat. Dadurch ergibt sich allerdings ein generelles Problem des deklarativen Ansatzes mit Vergabe von Prioritäten: Durch das Bevorzugen einer bestimmten Regel *p* kann es dazu kommen, dass zwar innerhalb der Anwendung von *p* eine große Anzahl an Subgraphen (*Matches* im Graphen) ersetzt werden kann, diese *Matches* aber auf Grund der ausgeführten Ersetzung für eine Regel *p'* mit geringerer Priorität nicht mehr in Betracht kommen können. Aufgrund dieser Eigenschaft hat die Grammatik eine eher schwache Ausdruckskraft.

Definition 5.2 (Matrix - Graph Grammatik)
Eine *Matrix - Graph Grammatik* besteht aus einem Axiom und einer Menge von Graphersetzungsregeln, die wie folgt in Sequenzen eingeteilt werden:

$\{(p_{1,1}, ..., p_{1,k(1)}), ..., (p_{n,1}, ..., p_{n,k(n)})\}$

mit n N und k: $N \rightarrow N$.
Die Semantik dieser Menge von Sequenzen ist durch folgende Transaktion definiert:

$t \quad ((t'; t) \quad \underline{skip})$

wobei

$t' \quad (t_1 \quad ... \quad t_n)$

$t_i \quad ((p_{i,1} \quad (\underline{undef}(p_{i,1})) ; ...; (p_{i,k(i)} \quad \underline{undef}(p_{i,k(i)})))$ für i = 1, ..., n.

Die Matrix - Graph Grammatik arbeitet also mit nichtdeterministisch ausgewählten Sequenzen von Ersetzungsregeln. In einer Sequenz selbst werden dann nacheinander die Regeln auf den Graphen angewendet, falls dies möglich ist, oder übersprungen, falls es nicht möglich ist.

Definition 5.3 (Programmierte Graph Grammatik)
Eine *Programmierte Graph Grammatik* über einer Menge von Graphersetzungsregeln P besteht aus einem Axiom und einer Menge von Tripeln (p_i, T_i, F_i) für i = 1, ..., n. Die Komponenten sind wie folgt definiert:

(1) p_i P ist die Ersetzungsregel, die angewendet werden soll.

(2) T_i $\{ p_{i,1}, ..., p_{i,k(i)} \}$ P ist die Menge von Ersetzungsregeln, die nach erfolgreicher Ausführung von p_i angewendet werden können.

(3) F_i $\{ p'_{i,1}, ..., p'_{i,k(i)} \}$ P ist die Menge von Ersetzungsregeln, die nach fehlgeschlagener Ausführung von p_i angewendet werden können.

Die Semantik dieser Graph Grammatik ist durch folgende Transaktion definiert:

t $(t_1 \quad ... \quad t_n)$

wobei

t_i $(p_i ; (t_{i,1} \quad ... \quad t_{i,k(i)}))$ $(\underline{\text{undef}}(p_i) ; (t'_{i,1} \quad ... \quad t'_{i,k'(i)}))$.

Des weiteren gilt:

t_i bezeichnet $(p_i , \{ p_{i,1} , ..., p_{i,k(i)} \}, \{ p'_{i,1} , ..., p'_{i,k'(i)} \})$,

$t_{i,j}$ bezeichnet $(p_{i,j} , ...)$ für j 1, ..., k(i),

$t'_{i,j}$ bezeichnet $(p'_{i,j} , ...)$ für j 1, ..., k'(i).

Der Ansatz der Programmierten Graph Grammatik erweitert den rein deklarativen Mechanismus, indem eine Fallunterscheidung für die Ausführung einer Ersetzungsregel p_i gemacht wird. Je nachdem, ob p_i erfolgreich war oder fehlschlägt, werden hier unterschiedliche Verzweigungen in der Programmstruktur gewählt. Doch auch dieser Mechanismus eignet sich nicht zur Darstellung von komplexen Transaktionen, da er nicht die Implementierung von abstrakteren Funktionen ermöglicht. Eine Verbesserung dieses Ansatzes bieten die Control Flow Graphen, die in Abschnitt 5.3 beschrieben sind.

5.2 Imperativer Ansatz

Im Gegensatz zu den deklarativen Ansätzen des vorangegangenen Abschnitts besitzen die imperativen Ansätze eine weitaus stärkere Ausdruckskraft, wenn es darum geht, komplexe Transaktionen auf Graphen auszuführen. Hier gehen die Kontrollstrukturen von einer anwenderorientierten Steuerung aus, die sich aus den wesentlichen Befehlen einer imperativen Programmiersprache zusammensetzt. Im Folgenden werden die zwei wesentlichen Programmierten Graphersetzungssysteme PAGG und PROGRESS näher betrachtet, welche durch die Umsetzung imperativer Kontrollstrukturen beide geeignet sind, um das Beispiel 5.1 zu implementieren.

Definition 5.4 (Kontrollstrukturen in PROGRESS)
Die Kontrollstrukturen von PROGRESS sind wie folgt definiert:

(1) fail undef(skip),
bezeichnet die stets fehlschlagende Anweisung.

(2) do *a* or *b* end (*a* *b*),

bezeichnet die nichtdeterministische Auswahl zwischen den Anweisungen *a* und *b*.

(3) if def *a* then *b* else *c* end ((def(*a*) ; *b*) (undef(*a*) ; *c*)),
bezeichnet die deterministische, bedingte Auswahl, welche je nach Anwendbarkeit von *a* auf den Graphen entweder *b* oder *c* ausführt.

(4) atom *a* ; *b* end (*a* ; *b*),
bezeichnet die deterministische Hintereinanderausführung der Anweisungen *a* und *b*. Endet schon die erste Anweisung mit Fehlschlag, so schlägt der gesamte Ausdruck fehl.

(5) do *a* and *b* end ((*a* ; *b*) (*b* ; *a*)),
bezeichnet die nichtdeterministische Hintereinanderausführung der Anweisungen *a* und *b*.

(6) while def *a* do *b* end Aufruf der Transaktion
t = ((def(*a*) ; (*b* ; *t*)) undef(*a*)),
bezeichnet die bedingte Iteration, welche die Anweisung *b* solange ausführt, wie die Anwendung von *a* auf dem Graphen definiert ist.

Mit Hilfe dieser Kontrollstrukturen lässt sich die Transformation *Main* wie in Beispiel 5.2 darstellen. Eine anschauliche Definition der Kontrollstrukturen von PROGRESS wird in Abschnitt 5.3 gezeigt.

Beispiel 5.2 (PROGRESS - Implementierung der Transformation Main)

Marry* = while def MarkUnmarried do Marry end .
Main = if def Marry* then Marry* else atom DeleteUnmarried ; Main end end .

Das Programmierte Graphersetzungssystem PAGG verfolgt bezüglich der Anwendbarkeit der Kontrollstrukturen eine andere Strategie als PROGRESS. Die BCF-Operatoren def und undef, wie sie in Kapitel 4 angegeben sind, werden nicht isoliert, sondern nur kontextbezogen verwendet, daher kann nicht ohne Änderung an dem Graphen getestet werden, ob eine Anweisung *a* erfolgreich ist oder fehlschlägt. Aus der Definition der Kontrollstrukturen von PAGG wird klar, dass hier eine deterministische Anwendung von Ersetzungsregeln im Mittelpunkt steht. Handelt es sich bei der betrachteten Menge von Ersetzungsreglen um eine *konfluente* Menge, d.h. die Reihenfolge der Anwendung spielt keine Rolle, so ist PAGG für Transformationen, die über einer solchen Menge definiert sind, bestens geeignet. Für die Übersetzung von Beispiel 5.1 muss jedoch dessen Semantik leicht verändert werden.

Definition 5.5 (Kontrollstrukturen in PAGG)

Die Kontrollstrukturen von PAGG sind wie folgt definiert:

(1) begin *a* ; *b* end ((def(*a*) def(*b*)) ; (*a* undef(*a*)) ; (*b* undef(*b*))),
(2) try *a* else *b* end (*a* (undef(*a*) ; *b*)),
(3) repeat *a* end Aufruf der Transaktion
t = (*a* ; (*t* undef(*a*))).

Dabei gilt die folgende Semantik:

- begin *a* ; *b* end ist die Kontrollstruktur für die Hintereinanderausführung, welche zwei Anweisungen *a* und *b* sequentiell auf den gegebenen Graphen anwendet.

Im Unterschied zum atom - Ausdruck von PROGRESS kommt es nur dann zum Fehlschlag, wenn beide Anweisungen auf dem Graph nicht ausführbar sind. Andernfalls wird die fehlschlagende Anweisung übersprungen und in der Sequenz wird die nachfolgende Anweisung ausgeführt.

- try *a* else *b* end entspricht dem PROGRESS - Ausdruck if *a* then *b* else *c* end. Anstelle von *a* können dort auch mehrere Argumente stehen.
- repeat *a* end wendet die Anweisung *a* wiederholt auf den Graphen an und endet erfolgreich, wenn *a* mindestens einmal angewandt werden konnte.

Wie schon oben angesprochen, wird in PAGG ein anderer Ansatz verfolgt, wodurch auch die Forderung des *Atomaren Charakters* und der *Booleschen Eigenschaft* nicht erfüllt werden. Das heißt, im Falle eines Fehlschlags z.B. der Anweisung *a* innerhalb der Ausdrücke (2) und (3), wird möglicherweise der gegebene Graph verändert. Dadurch arbeiten die nachfolgenden Anweisungen nicht mit dem erwarteten Resultat und auch nicht mit dem Graph, so wie er vor der Anwendung von *a* vorlag, sondern mit einem Graphen, auf dem teilweise Veränderungen ausgeführt wurden.

Eine Implementierung von PAGG - Kontrollstrukturen verwendet kein Backtracking und macht somit auch keine Änderungen am gegebenen Graphen rückgängig. Aus diesem Grund lässt sich die Transaktion *MarkUnmarried*, die in Beispiel 5.1 ausschließlich als Argument des def - bzw. undef - Operators vorkommt, nicht in eine PAGG - Implementierung einfügen. *MarkUnmarried* wird als Bedingung zur Anwendung von *Marry* nur überprüft und in jedem Fall nach Überprüfung wieder rückgängig gemacht. Eine Implementierung von Beispiel 5.1 könnte wie folgt aussehen:

Beispiel 5.3 (PAGG - Implementierung der Transformation Main)

```
Marry* = begin repeat Marry end ;
           <Fehlschlag, falls unverheiratete Person existiert> end .
Main   = repeat try Marry* else DeleteUnmarried end end .
```

Der semantische Unterschied ergibt sich also daraus, dass bei der Anwendung von *Marry** ohne vorherigen Test in jedem Fall mindestens einmal die Transaktion *Marry* ausgeführt wird. Kommt es also zum Fehlschlag von *Marry**, so arbeitet *DeleteUnmarried* möglicherweise auf einem veränderten Graphen, bei dem schon ein oder mehrere Paare gebildet wurden, deren zugehörige Personen dann nicht mehr gelöscht werden können.

5.3 Control Flow Graphen

In diesem Abschnitt soll zunächst auf das generelle Konzept der Control Flow Graphen eingegangen werden. Bei diesen Konstrukten handelt es sich um die graphische Darstellung von Kontrollstrukturen, die Ersetzungsregeln und komplexere Ausdrücke über einen call - Befehl aufrufen können, und somit Transaktionen auf Graphen definieren.

Daran anknüpfend sind im Kontrollmechanismus von PROGRESS die Control Flow Diagramme definiert, die die Anwendung einer Ersetzungsregel (bezeichnet als *Produktion*) gleichzeitig graphisch und textuell beschreiben. Diese Diagramme werden in Abschnitt 5.4 genauer beschrieben.

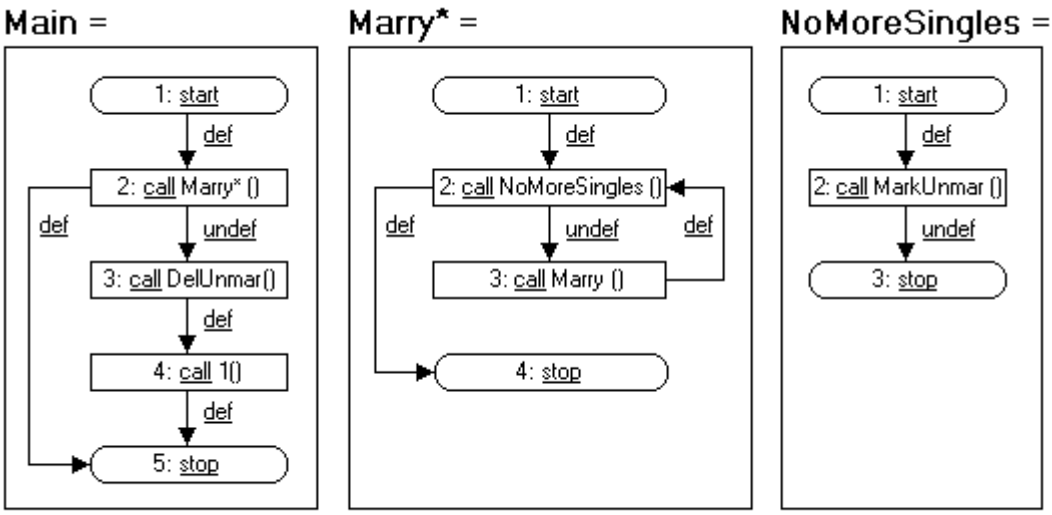

Abbildung 6 : Übersetzung der Transaktion *Main* in einen Control Flow Graphen

Grundsätzlich ist die Semantik der Control Flow Graphen ähnlich definiert wie die Semantik einer Programmierten Graph Grammatik (siehe Abschnitt 5.1). Der wesentliche Unterschied besteht darin, dass die Control Flow Graphen als abstrakte Funktionen betrachtet werden können, d.h. innerhalb eines Control Flow Graphen können andere (Sub-) Graphen aufgerufen werden, welche entweder erfolgreich oder mit Fehlschlag terminieren. Dadurch werden auch die Forderungen der *Booleschen Eigenschaft* und des *Atomaren Charakters* erfüllt.

Die Übersetzung der Transaktion *Main* in einen Control Flow Graphen ist in Abbildung 6 gegeben. Generell folgt ein Control Flow Graph dem Aufbau eines Syntaxdiagramms, wobei die folgenden Eigenschaften gelten sollen:

- Jeder Knoten $v \quad V$ ist mit einer Anweisung $l \quad$ {nop, call a, start , stop} beschriftet, wobei a eine einfache Ersetzungsregel oder den start-Knoten eines anderen Control Flow Graphen bezeichnet. Die Beschriftung entspricht einer Anweisung
- In jedem Subgraph existiert genau ein Knoten mit l = start und genau ein Knoten mit l = stop. Jeder Knoten v im Subgraph liegt auf einem Pfad vom start- zum stop-Knoten.
- Der Durchlauf bzw. die Ausführung eines Subgraphen beginnt in dessen start-Knoten und endet in dessen stop-Knoten.
- Jede Kante $e \quad E$ im Graphen hat eine Beschriftung $m \quad$ {def, undef }.
- Endet die Ausführung der Anweisung eines Knoten v erfolgreich, so folgt der Durchlauf einer ausgehenden Kante mit m = def . Der Nichtdeterminismus der Kontrollstruktur ist dadurch gegeben, dass v eine, mehrere oder keine ausgehenden def - Kanten haben kann.
- Schlägt die Ausführung der Anweisung eines Knoten v fehl, so folgt der Durchlauf einer ausgehenden Kante mit m = undef . Auch in diesem Fall kann v eine, mehrere oder keine ausgehenden undef - Kanten haben.

Die Semantik eines Knoten mit ausgehenden def- und undef-Kanten lässt sich mit Hilfe von BCF-Ausdrücken beschreiben.

Definition 5.6 (Semantik von Control Flow Graphen)
Die Semantik einer Menge von Control Flow Graphen über einer Menge *P* von Graphersetzungsregeln und einer Menge von Control Flow Knoten *V* ist wie folgt definiert:

Jeder Knoten $v \in V$ mit der Beschriftung $l \in \{ \underline{nop}, \underline{call}\ a, \underline{start} \}$ mit k ausgehenden def-Kanten zu den Knoten $v_1, ..., v_k$ und k' ausgehenden undef-Kanten zu den Knoten $v'_1, ..., v'_{k'}$ lässt sich in folgende BCF-Ausdrücke übersetzen:

v	$((x ; (v_1 \ ... \ v_k)) \ (\underline{undef}(x) ; (v'_1 \ ... \ v'_{k'})))$,	falls k,k' > 0
v	$(x ; (v_1 \ ... \ v_k))$,	falls k>0, k'=0
v	$(\underline{undef}(x) ; (v'_1 \ ... \ v'_{k'}))$,	falls k'>0, k=0
v	$\underline{fail} = \underline{undef}(\underline{skip})$	falls k=k'=0

Dabei ist

x	a	falls v die Beschriftung „$\underline{call}(a)$" mit $a \in P \in V$ hat
x	$\underline{skip}$	sonst

Jeder Knoten $v \in V$ mit der Beschriftung „$\underline{stop}$" lässt sich in den BCF-Ausdrück

$v \quad \underline{skip}$

übersetzen, wodurch die Ausführung des Control Flow Graphen bzw. Teilgraphen erfolgreich terminiert.

Da sich anhand dieser Definition jeder Knoten direkt in eine Transaktion übersetzen lässt, die mit Hilfe der ausgehenden def - und undef - Kanten andere Transaktionen aufruft, kann die Äquivalenz von BCF-Ausdrücken und Control Flow Graphen leicht gezeigt werden.

Behauptung 5.7 (Äquivalenz von BCF-Ausdrücken und Control Flow Graphen)
Eine beliebige Menge von Transaktionen mit ihren zugrundeliegenden BCF-Ausdrücken - ohne den Operator & für parallele Ausführung von Ausdrücken auf einer Struktur und mit dem stärkeren $\underline{def}^{\infty}$- Operator anstelle des ursprünglichen def - Operators - kann in eine äquivalente Menge von Control Flow Graphen übersetzt werden und umgekehrt.

Der Beweis von Behauptung 5.7 folgt direkt aus Definition 5.6. Um von BCF-Ausdrücken zu äquivalenten Control Flow Graphen zu gelangen, wird z.B. eine Anweisung $\underline{undef}(a ; ...)$ durch einen Knoten v_1 mit Beschriftung *a* und einer einzelnen ausgehenden Kante mit Beschriftung $m = \underline{undef}$, welche in einen Knoten v_2 führt, dargestellt. v_2 repräsentiert dabei die in der Hintereinanderausführung folgenden Berechnungsschritte.

Wie in Kapitel 4.2.2 erläutert, wird der $\underline{def}^{\infty}$- Operator verwendet, um bei der Überprüfung der Anwendbarkeit eines Ausdrucks *a* im erfolgreichen Fall mögliche unendlich lange laufenden Berechnungen auszuschließen. Ein Ausdruck der Form $\underline{def}^{\infty}(a)$ wird gemäß Definition 4.4 zunächst in den Ausdruck $\underline{undef}(\underline{undef}(a))$ umgeformt und kann dann in einen äquivalenten Control Flow Graphen übersetzt werden.

5.4 Control Flow Diagramme in PROGRESS

Die Verwendung von Control Flow Graphen ist geeignet, um einerseits eine semantisch ausdrucksstarke Definition und andererseits graphisch anschauliche Darstellung von Transaktionen zu geben. In den Kontrollstrukturen von PROGRESS wurden diese Graphen eingesetzt und es wurde ein imperativer Befehlssatz definiert, welcher die Verknüpfung mehrerer einfacher Ersetzungsregeln (Produktionen), die mit Hilfe von Diagrammen dargestellt werden, erlaubt.

Diese Diagramme sind also den in Kapitel 5.3 vorgestellten Control Flow Graphen sehr ähnlich, jedoch können diese nun unterstützend in Textbefehle eingearbeitet werden, um z.B. eine Relation von Knoten zueinander einfacher zu definieren.

Zunächst wollen wir uns die grundlegenden Befehle von PROGRESS und ihre Diagramm-Darstellung ansehen. Der Control Flow Graph im Diagramm hat dieselben Eigenschaften wie in Kapitel 5.3, wobei nun eine Kante die Beschriftung commit anstelle von def und abort anstelle von undef tragen soll. Eine Anweisung *a*, also ein einfacher BCF - Ausdruck, wird im Folgenden als Produktion mit P() bezeichnet.

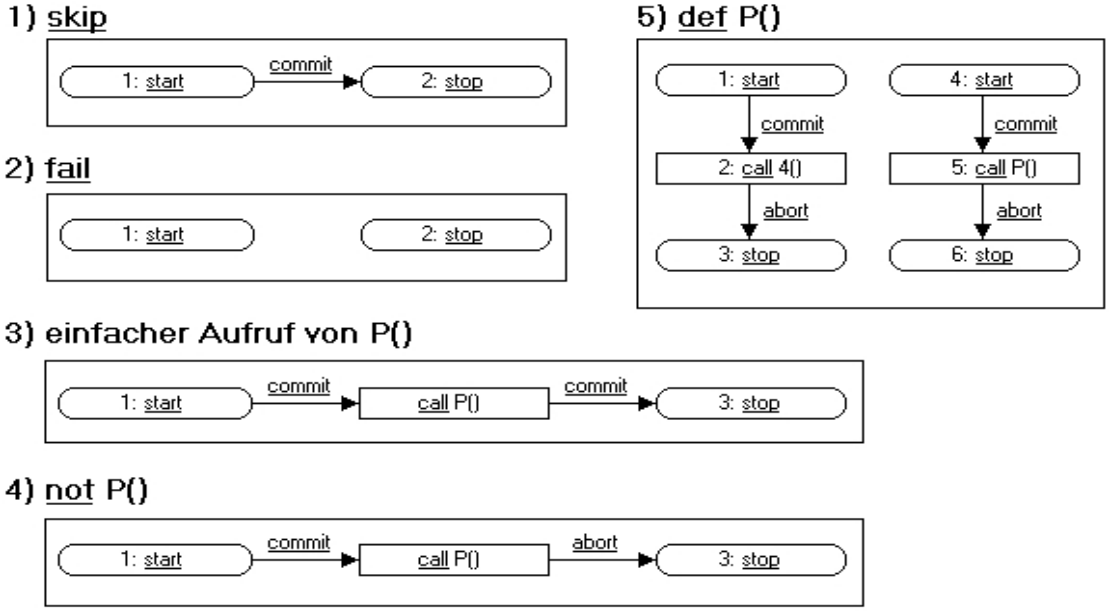

Abbildung 7 : Control Flow Diagramme für skip, fail, einfachen Aufruf, not und def

In Abbildung 7 sind die elementaren Befehle als Diagramme dargestellt.

1. Die Umsetzung des skip - Befehls entspricht der Semantik in Definition 4.3. Der gegebene Graph, auf dem der Befehl ausgeführt wird, bleibt unverändert und der Befehl selbst ist immer erfolgreich, d.h. im aufrufenden Diagramm wird nach Ausführen von skip eine commit - Kante verfolgt.
2. Die Ausführung von fail ist eine immer fehlschlagende Berechnung, die den gegebenen Graphen nicht verändert. Sie wird im aufrufenden Diagramm mit einer abort - Kante fortgesetzt.
3. Der einfache Aufruf einer Produktion P() wird auf dem gegebenen Graphen ausgeführt und falls es einen Match für P() gibt, wird der Graph entsprechend

verändert und das aufrufende Diagramm folgt einer commit - Kante. Andernfalls schlägt die Ausführung von P() fehl und der Graph wird unverändert zurückgegeben. In diesem Fall wird im aufrufenden Diagramm eine abort - Kante verfolgt.

4. Der Befehl not P() lässt sich durch eine kleine Änderung des einfachen Aufrufs definieren. Falls P ausführbar ist, so kann nach dem Aufruf in Knoten 2 keine commit - Kante verfolgt werden, daher bricht die Berechnung ab und not P() schlägt fehl. Falls P nicht ausführbar ist, so wird nach dem Aufruf in Knoten 2 die abort - Kante verfolgt, was zu einer erfolgreichen Ausführung von not P() führt.
5. Der Befehl def P() testet die Ausführbarkeit von P auf dem Graphen, verändert aber nicht dessen Struktur. Falls P ausführbar ist, so kann nach Aufruf in Knoten 5 keine commit - Kante verfolgt werden, wodurch der rechte Teilgraph fehlschlägt und nach dessen Aufruf in Knoten 2 eine abort - Kante verfolgt werden kann. Falls P fehlschlägt, so muss nach Aufruf in Knoten 5 die abort - Kante verfolgt werden und der Teilgraph ist erfolgreich, doch da es von Knoten 2 aus keine commit - Kante gibt, schlägt das gesamte Diagramm fehl.

Die Definition von not P() garantiert, dass der gegebene Graph unverändert zurückgegeben wird. Wenn not P() fehlschlägt, wobei also alle möglichen Matches im Graphen gefunden und ausprobiert wurden, dann werden alle Änderungen, die durch P() im Graphen gemacht werden, verworfen. Wenn andererseits not P() erfolgreich ist, dann muß die Ausführung von P() fehlgeschlagen sein, und es wurde auch in diesem Fall nichts am Graphen verändert. Somit entspricht der Befehl not P() dem undef - Operator aus Definition 4.4. Demzufolge ist der hier beschriebene Befehl def P() aus zwei not Befehlen aufgebaut und somit entspricht def P() dem strengeren BCF-Operator $\underline{\text{def}}^{\infty}$.

In Abbildung 8 sind die Control Flow Diagramme für die deterministische Sequenz und für die nichtdeterministische Auswahl dargestellt.

1. Der Befehl & entspricht der deterministischen Sequenz von zwei oder mehreren Produktionen. Die allgemeine Schreibweise ist P_1 & P_2 & ... & P_n. Dabei ist eine Reihenfolge der Produktionen vorgegeben, d.h. um P_2 ausführen zu können, muss unbedingt P_1 zuerst erfolgreich angewendet worden sein. Gibt es mehrere Möglichkeiten für die Anwendung von P_1, so wird im Falle eines Fehlschlags Backtracking angewandt, bis der Erfolg oder Fehlschlag der gesamten Produktion P_1 feststeht.
2. Der Befehl or entspricht der nichtdeterministischen Auswahl von zwei oder mehreren Produktionen. Die allgemeine Schreibweise ist P_1 or P_2 or ... or P_n. Dabei schlägt der Aufruf des Diagramms also nur dann fehl, falls keine der Produktionen über eine commit - Kante erfolgreich terminiert.

Um das Resultat der Sequenz zweier oder mehrerer Produktionen freier zu gestalten, kann eine Kombination der Befehle & und or verwendet werden. Der Befehl and realisiert wie folgt die nichtdeterministische Sequenz, so dass die Reihenfolge der Produktionen keine Rolle spielt:

$$P_1 \text{ and } P_2 \text{ and } \ldots \text{ and } P_n$$
$$= (P_1 \,\&\, P_2 \,\&\, \ldots \,\&\, P_n) \text{ or } (P_2 \,\&\, P_1 \,\&\, \ldots \,\&\, P_n) \text{ or } \ldots \text{ or } (P_n \,\&\, P_{n-1} \,\&\, \ldots \,\&\, P_1)$$

Es genügt also für die erfolgreiche Ausführung eines and - Befehls, dass alle verwendeten Produktionen ausgeführt werden, egal in welcher Reihenfolge.

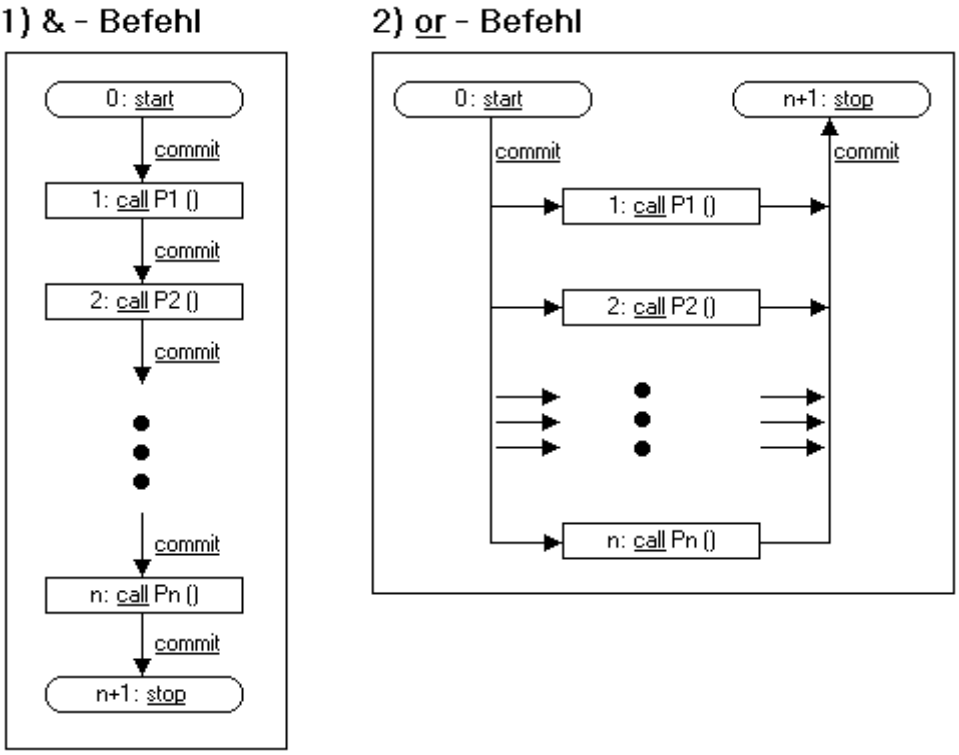

Abbildung 8 : Control Flow Diagramme für & und or

In Abbildung 9 sind die Control Flow Diagramme für die bedingte Alternative und die Iteration dargestellt.

1. Der choose - Befehl behandelt vor jeder möglichen Alternative zunächst die vorangestellte Bedingung, und führt die zugehörige Produktion nur dann aus, wenn die Bedingung erfolgreich über die commit - Kante terminiert. Bei einem Fehlschlag wird die nächste Bedingung geprüft und nur falls keine der Bedingungen erfüllt werden und somit keine Produktion ausgeführt werden kann, schlägt das gesamte Diagramm fehl. Die allgemeine Schreibweise des Befehls ist

choose when Q_1 then P_1
else when Q_2 then P_2
...
else when Q_n then P_n
end

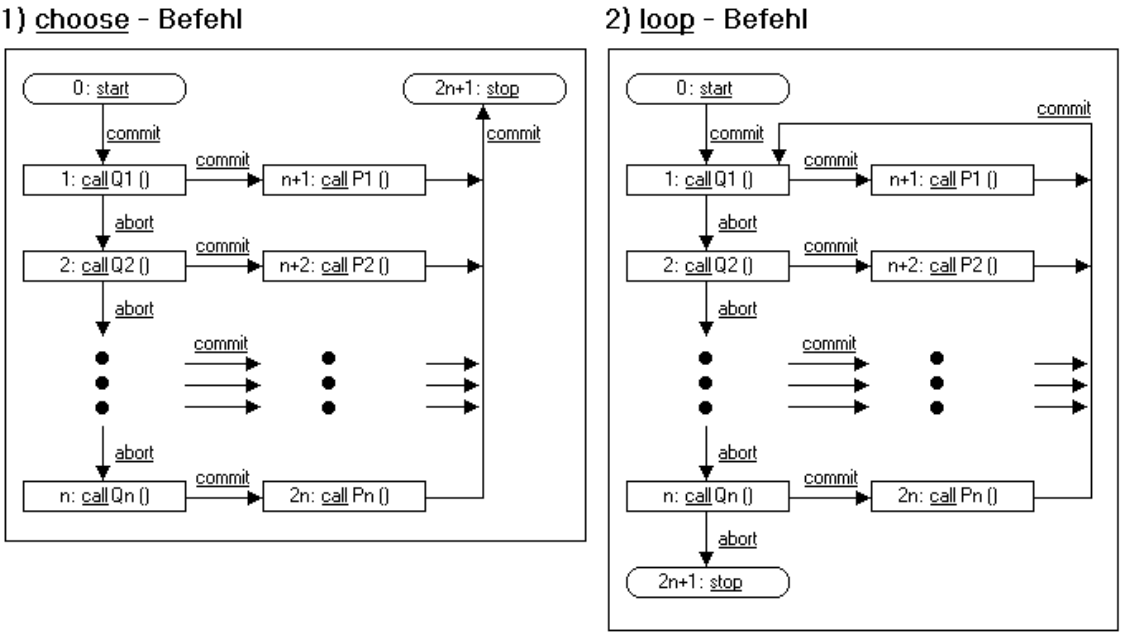

Abbildung 9 : Control Flow Diagramme für choose und loop

2. Der loop - Befehl besteht ähnlich wie der choose - Befehl zunächst aus einer Folge von bedingten Alternativen. Ist die Bedingung Q_i erfolgreich und ebenso die Produktion P_i, so startet die Iteration wieder mit der Überprüfung der ersten Bedingung Q_1. Schlägt jedoch nach erfolgreicher Überprüfung von Q_i die Produktion P_i fehl, so startet das Backtracking und es wird eine zuvor angewandte Produktion P_{i-1} erneut auf mögliche Matches geprüft. Falls eine solche nicht existiert, und somit keine erfolgreiche Transformation des Graphen mit dem loop - Konstrukt möglich ist, terminiert die Berechnung mit Fehlschlag und alle Änderungen im Graphen werden rückgängig gemacht. Kommt es während eines Iterationsschrittes dazu, dass keine der Bedingungen Q_i für i = 1, ... , n erfolgreich ausgeführt werden kann, so folgt die Berechnung nach der letzten Bedingung Q_n der abort - Kante und das loop - Konstrukt terminiert erfolgreich. Die allgemeine Schreibweise des Befehls ist

loop when Q_1 then P_1
else when Q_2 then P_2
...
else when Q_n then P_n
end

Mit der Definition dieser Befehle sind nun die Kontrollstrukturen von PROGRESS festgelegt. Für die drei wichtigen Kontrollstrukturen Sequenz, Alternative und Iteration gibt es jeweils eine deterministische als auch eine nichtdeterministische Variante, wie Tabelle 2 zeigt. Der nichtdeterministische

	deterministisch	nichtdeterministisch
Hintereinanderausführung	&	and
Alternative	choose (optional mit Bedingungen)	or
Iteration (*tail recursion*)	loop (optional mit Bedingungen)	loop (... or ...)

Tabelle 2 : Kontrollstrukturen von PROGRESS

loop - Befehl kommt zustande, indem man innerhalb der Schleife mit einem or - Befehl arbeitet, der eine nichtdeterministische Auswahl an Alternativen ermöglicht:

```
loop  (when Q1 then P1)
      or    (when Q2 then P2)
      or          …
      or    (when Qn then Pn)
    end
```

6 Zusammenfassung

Wir haben nun Graphersetzungssysteme kennengelernt, die von verschiedenen Ansätzen ausgehen, teils deklarativ arbeiten, teils imperativ. Die allgemeine Definition von Kontrollstrukturen und daraus resultierend die formale Notation von Graph-Transaktionen haben es ermöglicht, die verschiedenen Ansätze in Hinsicht auf elementare Eigenschaften zu vergleichen. Darunter fallen unter anderem der Nichtdeterminismus in der Auswahl von Ersetzungsregeln, die Fähigkeit einzelne Ersetzungsregeln als atomar zu betrachten und somit wie Funktionsaufrufe an jeder Stelle einsetzen zu können und die Eigenschaft, für eine angewendete Ersetzungsregel stets sagen zu können, ob sie erfolgreich war oder fehlschlug. Die boolesche Eigenschaft überträgt sich ebenso wie der atomare Charakter von den formal definierten Ersetzungsregeln auf die Kontrollstrukturen, die BCF-Operatoren, und somit auch auf ganze Transaktionen, welche mit Hilfe der Operatoren definiert werden können.

Mit Hilfe der in Kapitel 4 definierten Fixpunktsemantik für Transaktionen konnten schließlich auch rekursive Aufrufe behandelt werden, um auch komplexe Transaktionen auf Graphen mit Kontrollstrukturen umsetzen zu können. Die Definition des allgemeinen PLSR-Systems (Definition 4.6) und dessen Sprache (Definition 4.7) als deklarative Ableitung, welche mit einer Ausgangstruktur und einer Haupt-Transaktion beginnt und damit den gesamten Ersetzungsprozess startet, ermöglicht schließlich mit der semantischen Funktion R_P^* die Anwendung der Kontrollstrukturen.

Im Folgenden soll noch einmal ein kurzer Vergleich der in Kapitel 5 vorgestellten Ansätze die Zusammenfassung abschließen. Tabelle 3 zeigt, dass die meisten der vorgestellten Kontrollmechanismen die fünf in Kapitel 4 geforderten Eigenschaften nicht oder nur teilweise erfüllen. Bei den Programmierten Graph Grammatiken und bei PAGG wird die boolesche Eigenschaft z.B. nur in den einzelnen Ersetzungsregeln, nicht aber für Transaktionen oder komplexere Ausdrücke gewährleistet. Des weiteren

verwirklichen diese beiden Kontrollmechanismen nicht die Forderung nach einem atomaren Charakter der Hintereinanderausführung. Dadurch wird, wie schon in Kapitel 5.1 und 5.2 erwähnt, ein Backtracking unmöglich, da die Auswirkungen von Ersetzungsregeln, die nur zum Teil angewandt wurden und während der Berechnung mit Fehlschlag terminiert haben, auf dem Graphen nicht rückgängig gemacht werden.

Da PROGRESS auf dem Konzept der BCF-Operatoren entwickelt wurde und diese unter Berücksichtigung der fünf geforderten Eigenschaften definiert sind, wird in diesem Kontrollmechanismus eine hohe Ausdruckskraft erreicht. Ebenso bei den Control Flow Graphen, welche in PROGRESS zusammen mit dem imperativen, textuellen Befehlssatz als Diagramme eingesetzt werden und auf diese Art eine intuitive Programmierung erlauben.

	sequentielle, atomare Kontrolle	nichtdet. Kontrolle	boolesche Eigenschaft	Rekursion	Ansatzart + Notation
GG mit Regel-Prioritäten	nein	nein	nein	nein	deklarativ + textorientiert
Matrix GG	atomare Hintereinanderausführung	nein	nein	nein	dekl./imp. + textorientiert
Progr. GG	nicht-atomare Hintereinanderausführung	nein	nur in den Regeln	*tail recursion*	dekl./imp. + textorientiert
PAGG	nicht-atomare Hintereinanderausführung	nein	nur in den Regeln	ja	imperativ + textorientiert
PROGRESS mit Control Flow Diagrammen	ja	ja	ja	ja	imperativ + textuell und graphisch
Control Flow Graphen	ja	ja	ja	ja	imperativ + graphisch

Tabelle 3 : Vergleich verschiedener Kontrollmechanismen

7 Literatur

[Schürr97] Ehrig, H. / Kreowski, H.-J. / Montanari, U. / Rozenberg G. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1 - Chapter 7*. World Scientific

[Schürr91] Zündorf, Albert / Schürr, Andy. *Nondeterministic Control Structures for Graph Rewriting Systems*. Aachener Informatik-Berichte Nr. 91-17. Erscheint in: Proceedings of "17th International Workshop on Graph-Theoretic Concepts in Computer Science WG 91", LNCS, Springer Verlag