POPEL—HOW:
Parallele, inkrementelle Generierung natürlichsprachlicher Sätze aus konzeptuellen Einheiten

Diplomarbeit
von Günter Neumann

nach einem Thema
von Prof. Dr. W. Wahlster

Fachbereich Informatik
Universität des Saarlandes

März 1989

Praxis ohne Theorie ist blind,
Theorie ohne Praxis ist unfruchtbar.

John Desmontes Bernal

Ich möchte mich an dieser Stelle ganz besonders bei folgenden Personen bedanken: Prof. Dr. W. Wahlster , Prof. Dr. H. Uszkoreit und J. Allgayer. Mein größter Dank gilt N. Reithinger, dem Vater von POPEL, und W. Finkler, für die gute Zusammenarbeit bei der Entwicklung von POPEL–HOW.

Contents

Chapter 1

Einleitung

1.1 Aufgabenstellung und Überblick

Aufgabenstellung

Die vorliegende Arbeit ist im Rahmen des Sonderforschungsbereichs 314, Teilprojekt N1: XTRA (eXpert TRAnslator), einem natürlichsprachlichen Zugangssytem zu Expertensystemen entstanden. Eine wichtige Aufgabe innerhalb dieses Projektes ist der Entwurf der Generierungskomponente POPEL (Production Of {Perhaps, Possibly, ...} Eloquent Language) [**?**], die die dialogspezifische Erzeugung sprachlicher Ausdrücke innerhalb von XTRA übernimmt. Die zwei zentralen Aufgaben hierbei sind:

- Festlegung, *was* gesagt und
- *wie* es gesagt werden soll.

Beide Aufgabenbereiche werden in POPEL durch voneinander getrennte, aber miteinander interagierende Komponenten – POPEL–WHAT und POPEL–HOW bearbeitet. Der Kommunikationsfluß zwischen diesen Komponenten ist bidirektional und erlaubt insbesondere einen inkrementellen und auf Wechselwirkung basierenden Generierungsprozeß zwischen Inhaltsfestlegung und – realisierung. Daraus und aus der Einbettung von POPEL in ein Dialogsystem ergeben sich für die Architektur und die Funktionsweise der *inhaltsrealisierenden* Komponente POPEL–HOW folgende spezielle Anforderungen:

- Es sollen möglichst viele Wissensquellen des Gesamtsystems eingesetzt werden.
- Es müssen dialogspezifische Äußerungen erzeugt werden.
- Die Ausgangsstrukturen sind Teilstrukturen oder Segmente der konzeptuellen Beschreibungsebene des Gesamtsystems.
- Der inkrementelle Generierungsprozeß erfordert es, daß die Segmente so schnell wie möglich über alle Darstellungsebenen des Systems (konzeptuelle, semantische, syntaktische und morphologische Ebene) transformiert werden, damit sie unmittelbar geäußert werden können.

- Da die erzeugten Teilstrukturen unterspezifiziert sein können – womit die rasche Abbildung gefährdet wäre – muß die fehlende Information von der übergeordneten Ebene angefordert werden können.

- Daraus folgt: Der Generierungsprozeß in POPEL–HOW ist auf jeder Darstellungsebene inkrementell und bidirektional.

- Insbesondere sind Rückwirkungen auf die inhaltsfestlegende Komponente POPEL–WHAT möglich, d.h. POPEL–HOW muß gezielt mit POPEL–WHAT kommunizieren können.

Die besondere Art des Generierungsprozesses – *inkrementell* und *bidirektional* – legt es nahe, als operationale Basis auf jeder Ebene ein *verteiltes paralleles* Modell einzusetzen. Dies erlaubt es, die einzelnen Segmente kooperierenden Prozessen zuzuordnen. Jeder Prozeß ist dann dafür verantwortlich, die Verbalisierung seines Segments durchzuführen. Dies setzt aber voraus, daß die verwendeten Wissensquellen sinnvoll segmentiert, verteilt und miteinander in Beziehung gebracht werden können.

POPEL–HOW ist das erste inhaltsrealisierende System, das einen inkrementellen und bidirektionalen Verbaliserungsprozeß über mehrere Ebenen durchführt und als operationale Basis ein verteiltes paralleles Modell verwendet. Bisherige inkrementelle Systeme, wie z.B. [?] modellieren keinen bidirektionalen Kommunikationsfluß oder behandeln inkrementelle Verbalisierung (vgl. [Appelt, 1985], [Hovy, 1987]) für die inhaltsrealisierende Komponente nicht explizit.

Die Entwicklung und Implementation von POPEL–HOW ist im Rahmen von zwei Diplomarbeiten von Wolfgang Finkler und mir durchgeführt worden. In der Arbeit von [?] wird im wesentlichen das parallele Basismodell und der verteilte Abbildungsprozeß vorgestellt. Daneben wird gezeigt, wie konzeptuelle auf semantische Strukturen abgebildet werden. Die zugrundeliegenden Wissensquellen hierfür waren bereits gegeben und ließen sich recht einfach in dem parallelen Modell verteilt einsetzen. Allerdings waren noch keine Abbildungsvorschriften formuliert, so daß diese im Rahmen von POPEL–HOW entwickelt werden mußten.

In der vorliegenden Arbeit werde ich zeigen, wie syntaktische Teilstrukturen in POPEL–HOW parallel und inkrementell konstruiert, flektiert und linearisiert werden. Für diese Aufgabe wurde eine eigene deklarative Grammatik entwickelt, da die Grammatik der Analyse hierfür nicht geeignet ist und auch bisher in keinem Generierungssystem eine deklarativ formulierte Grammatik eingesetzt wird, die die besonderen Anforderungen an die inkrementelle und verteilte parallele Verarbeitung erfüllt.

Aufgrund der Komplexität der Aufgabenstellung war es das zentrale Ziel beider Arbeiten, einen Prototypen zu implementieren. Daher stehen in dieser Arbeit die Prinzipien und Verfahren im Vordergrund, die für den inkrementellen, bidirektionalen und parallelen Generierungsprozeß in POPEL–HOW entwickelt wurden, weniger die Erläuterung spezieller linguistischer Phänomene. Es wurde aber von vornherein darauf geachtet, daß durch die deklarative Darstellung des Wissens unter Verwendung von mächtigen Repräsentationsformalismen eine "beliebige" Erweiterung und damit eine entsprechende Behandlung spezieller Phänomene möglich ist.

Zur Zeit werden mit dem in dieser Arbeit beschriebenen Prototypen einfache Haupt–, Imperativ– und Fragesätze, sowie elliptische Sätze erzeugt. Die Konstruktion dieser Sätze umfaßt sämtliche Tempora– , Modalitätsformen sowie Aktiv– und Passivtransformationen. Dabei werden fast alle erlaubten Variationen in der linearen Abfolge zugelassen. Aufgrund des Einsatzes der Morphologiekomponente MORPHIX sind bezüglich der Flexion keine Beschränkungen vorhanden. Bei der Generierung von Nominalphrasen wird zwischen der Erzeugung von einfachen indefiniten, definiten und pronominalisierten Phrasen unterschieden. Nominalphrasen können auch elliptifiziert werden.

Der Prototyp ist auf einer Symbolics Lispmaschine 3640 in Commonlisp unter Release 7.1 implementiert worden. Die Simulation des parallelen Basismodells wurde mit Hilfe des Multiprocessingsystems unter Einbeziehung des Flavorsystems der Lispmaschine realisiert.

Überblick

Die Arbeit ist in sieben Kapitel aufgeteilt. Das erste Kapitel geht auf allgemeine Aspekte natürlichsprachlicher Generierungssysteme ein. Danach wird die Einbettung von POPEL–HOW in den Systemen POPEL bzw. XTRA erläutert. Das zweite Kapitel beschreibt die Architektur und prinzipielle Funktionsweise von POPEL–HOW. Dieses Kapitel ist eine Zusammenfassung der Arbeit von [**?**]. Am Schluß des zweiten Kapitels werden die besonderen Anforderungen und Aufgaben, die sich für die syntaktische Beschreibungsebene ergeben, beschrieben. Bevor im vierten Kapitel die zentralen Wissensquellen für diesen Bearbeitungsschritt vorgestellt und erläutert werden, sind im dritten Kapitel die für das Verständnis notwendigen linguistischen und formalen Grundlagen eingeführt. Im fünften und sechsten Kapitel kann dann im Detail beschrieben werden, wie in POPEL–HOW die syntaktische Struktur inkrementell und parallel aufgebaut wird. Das letzte Kapitel enthält eine Wertung und einen Ausblick auf noch zu lösende Probleme.

Anmerkungen zur Notation

In den Kapiteln, in denen die entwickelten Algorithmen ausführlich erläutert werden, werde ich durch Implementationshinweise Anmerkungen zur programmtechnischen Umsetzung interessanter Punkte geben.

Für die Beschreibung der Algorithmen verwende ich einen Pseudocode. Ich gehe zwar davon aus, daß die Syntax intuitiv verstanden werden kann, möchte aber dennoch zwei Punkte hervorheben. Zum einen werden Mengen immer durch großgeschriebene Bezeichner benannt (z.B. `DAG` für die Menge von gerichteten azyklischen Graphen) und Elemente von Mengen durch kleingeschriebene Bezeichner (z.B. `dag` $\in$ `DAG`). Um mich auf Teilstrukturen eines Elementes beziehen zu können, verwende ich oft eine funktionale Schreibweise. Beispielsweise schreibe ich `cat(d)` zur Benennung der Kategorie eines DAGs und nehme an, daß `cat` die Funktion ist, die den Wert der Kategorie des DAGs `d` berechnet. Der Name der Funktion drückt aus, von welchem Typ das Ergebnis ist (z.B. hat `cat` (für category) als Wert ein Element vom Typ Kategorie).

1.2 Einführung in den Problembereich "Generierung natürlichsprachlicher Ausdrücke"

Im Rahmen der Entwicklung natürlichsprachlicher Systeme (NSS) wird versucht, die den natürlichen Sprachen zugrundeliegenden Prozesse der Analyse und Generierung, zu erforschen und zu implementieren. Bisher ist der Verarbeitungsschritt "Generierung" in den meisten Systemen im Vergleich mit dem Verarbeitungsschritt "Analyse" weniger stark beachtet worden. Einer der Gründe mag sein, daß bei der Bearbeitung natürlicher Sprache dem *Verstehensprozeß* eine größere Komplexität zugesprochen wird, als dem *Erzeugungsprozeß*. Ein NSS muß bei der Analyse aufgrund der *Kompetenz* des Benutzers mit einem unbeschränkten und nicht vorhersehbaren Wortschatz rechnen.

In der Generierungsrichtung eines NSS ist dagegen die Frage von zentralem Interesse, *wie* sich konzeptuelle Strukturen in eine sprachliche Form überführen lassen. Im Vordergrund der Entwicklung von Generierungsverfahren steht damit die *Sprachverwendung*. Da die Qualität und Quantität des konzeptuellen Wissens und des zugrundeliegenden Repräsentationsformalismuses in früheren Systemen sehr beschränkt war, erschien es vertretbar, primitive und wenig flexible Generierungsverfahren zu entwickeln und einzusetzen.

Die vorherrschenden Methoden waren (vgl. [**?**])[1]:

- Ausgabe vorgefertigter Textteile (*canned text*)
- Kontextsensitive Instanziierung von vorgefertigten Textschemata (*direct replacement*)

Diese Verfahren haben bezüglich der sprachlichen Adäquatheit ihrer Ergebnisse erhebliche Mängel. Wenn zur Sprachausgabe vorgefertigte Textteile verwendet werden, muß z.B. bei der Implementation eines Frage–Antwort–Systems bereits sehr genau überlegt werden, welche Fragen vom Benutzer gestellt werden können und wie die Antworten zu formulieren sind. Damit ist das System in seiner Flexibilität zu antworten erheblich eingeschränkt. Hinzukommt, daß das System kein Wissen darüber hat, was es äußert. Der Benutzer kann daher nicht sicher sein, ob die Äußerung des Systems tatsächlich mit dem übereinstimmt, was das System als Reaktion auf die Benutzereingabe hin berechnet hat.

Werden vorgefertigte Textschemata für die Sprachausgabe eingesetzt (wie zum Beispiel in dem System SHRDLU [**?**]), in denen jeweils direkt die konzeptuellen Strukturen abgebildet werden, so hängt die Komplexität und die Flexibilität der Äußerungen unmittelbar von der Komplexität der Wissensbasis des Systems ab. Ist der zu verbalisierende Ausschnitt der Wissensbasis sehr groß, dann ist das Generierungssytem nicht in der Lage, sich auf wesentliche Teile zu beschränken. Die Folge davon ist, daß die Ausgabe für den Benutzer viel zu umfangreich und damit unüberschaubar werden kann. Zudem wird bei der Instanziierung der Textschemata nur sehr wenig linguistisches Wissen eingesetzt, so daß die Sprachqualität sehr gering ist.

Der entscheidende Nachteil beider Vorgehensweisen ist das Erzeugen von einförmigen und kaum zusammenhängenden Ausdrücken, die bei längerem Arbeiten mit dem System vom Benutzer eine hohe Akzeptanzbereitschaft erfordern. Damit natürlichsprachliche Systeme vom menschlichen Benutzer bezüglich ihrer Kommunikationsfähigkeit als akzeptable Partner angenommen werden können, müssen sie in der Lage sein, *kommunikativ adäquate*, *verständliche* und *kohärente* Äußerungen hervorzubringen [**?**]. Hierfür ist es erforderlich,

[1]Auch heute noch werden diese Methoden in vielen NSS und vor allem in Erklärungskomponenten von Expertensystemen für die Sprachausgabe verwendet.

die vom System gemachten Äußerungen den Erwartungen des Benutzers anzupassen und in den bisherigen Text oder Dialog zusammenhängend zu integrieren [?]. Das natürlichsprachliche Generierungssystem muß als Komponente eines NSS bei der Beantwortung von Fragen den Wissensstand seines Dialogpartners berücksichtigen und versuchen, die Aufmerksamkeit des Partners auf das Wesentliche zu konzentrieren. Alle Teile der zu äußernden Systemreaktion müssen dabei syntaktisch, semantisch und pragmatisch korrekt zueinander in Beziehung gebracht werden.

1.2.1 Allgemeine Aspekte natürlichsprachlicher Generierungssysteme

Seit Anfang der achtziger Jahre verstärken sich die Forschungsaktivitäten in der Konzeption, Entwicklung und Implementierung von Komponenten zur natürlichsprachlichen Generierung von Systemreaktionen, um das Ziel – Erzeugung kommunikativadäquater Äußerungen – zu erreichen. Allerdings zeigen aktuelle Beiträge bei grundlagenorientierten Tagungen (vgl. u.a. [?]), daß mit der Entwicklung von leistungsfähigen Generierungssystemen erst begonnen wird. Es wird daher im Folgenden erläutert, welche allgemeinen Aspekte bei der Konzeption von natürlichsprachlichen Generierungssystemen von besonderem Interesse sind[2].

Ein zentraler Gesichtspunkt bei der Entwicklung von natürlichsprachlichen Generierungssystemen (NGS) ist es, den Verbalisierungsprozeß in erster Linie als ein spezielles *Planungsproblem* aufzufassen [McDonald, 1986]. Hiermit ist die Auffassung verbunden, Sprache als ein Hilfsmittel zu verstehen, mit dem der Sprecher übergeordnete *kommunikative Ziele* erreichen will [Appelt, 1985]. Um diese Ziele zu erreichen, muß die zu verbalisierende Äußerung erst geplant werden, wobei der Planungsprozeß durch die linguistischen Fähigkeiten des Äußernden beeinflußt und eingeschränkt wird. Während des Planungsprozesses ist es notwendig, eine Fülle von Entscheidungen zu treffen.

Als ein wesentliches Prinzip hat sich die Aufteilung des Generierungsprozesses (und damit die Aufteilung von Generierungssystemen) in zwei große Teilbereiche ergeben:

1. In einen *inhaltsfestlegenden* Teil, in dem entschieden wird, was gesagt wird und
2. in einen *inhaltsrealisierenden* Teil, in dem die sprachliche Form der Äußerungen bestimmt wird.

1.2.1.1 Der inhaltsfestlegende Teil

Im inhaltsfestlegenden Teil, er wird auch *What–To–Say* (WTS) Teil genannt, werden *konzeptuelle* Entscheidungen getroffen. Die zentralen Aufgaben innerhalb dieses Prozesses sind die *Auswahl* der Information, die für das aktuelle Diskursziel (z.B. Beantwortung einer Frage) relevant ist und das Festlegen einer sinnvollen *Reihenfolge* der zu verbalisierenden Teile [McKeown, 1985].

Ausgangspunkt für den Generierungsprozeß ist die dialogunabhängige konzeptuelle Darstellung des Ergebnisses, das aufgrund einer Eingabe (u.a. textuelle, graphische oder

[2]Bisher implementierte Systeme (wie z.B. Text [McKeown, 1985], Mumble [McDonald, 1980] oder KAMP [Appelt, 1985]) berücksichtigen dabei jeweils spezielle Teilprobleme.

sensorische Eingabe) berechnet wurde[3]. Im inhaltsfestlegenden Teil eines NGS wird geplant und entschieden, ob das Ergebnis der gesamten Systemreaktion oder nur ein relevanter Teil davon geäußert werden soll oder sogar zusätzliche Teile berechnet werden müssen, die ursprünglich gar nicht vorgesehen waren [?]. Entscheidet sich der WTS Teil dafür, nur wenige Konzepte zu verbalisieren (weil das Ergebnis z.B. sehr umfangreich ist), muß es beurteilen können, wie *wichtig* das damit verbundene Wissen für das NSS *und* für den Benutzer ist.

Ist das NGS in einem Dialogsystem eingebettet, muß die Äußerung in den bisherigen Dialog eingebettet werden. Die Auswahl des relevanten Teils wird dabei beeinflußt:

1. vom Wissen über den Kenntnisstand des Benutzers,
2. vom Wissen über eigene generelle und spezielle Ziele sowie
3. über generelle und spezielle Ziele des Dialogpartners;
4. von der gegenwärtigen Kommunikationssituation und
5. dem bisherigen Dialogverlauf.

Um den Kenntnisstand des Benutzers einbeziehen zu können, sollte das natürlichsprachliche Gesamtsystem über eine *Benutzermodellierungskomponente* verfügen. Die Funktion solch einer Komponente ist der inkrementelle Aufbau eines Benutzermodells. Als Benutzermodell bezeichnet man die Wissensquelle, die explizite Annahmen und Fakten über Aspekte des Benutzers (oder einer Benutzerklasse) enthält, die für das Dialogverhalten von Interesse sein könnten [?].

Wenn das Generierungssystem Zugriff auf Wissen darüber hat, was es beim Benutzer im gegenwärtigen Diskursbereich an Bekanntem und Selbstverständlichen annehmen kann oder welche Bewertungskriterien für den Dialogpartner gelten, ist es in der Lage, auf Eingaben in Abhängigkeit vom Benutzer unterschiedlich zu reagieren. Stellt der Benutzer z.B. Fragen über die Wissensbasis des Systems (vgl. [Kaplan, 1983], [McKeown, 1985]), so entscheidet das NGS darüber, wie umfangreich der zu verbalisierende Ausschnitt der Wissensbasis sein soll. Bei einem Benutzer, der mit dem Diskursbereich wenig vertraut ist, muß das NGS eine detailliertere Erklärung liefern als bei einem Experten.

Die Festlegung der *Reihenfolge* der zu verbalisierenden Teile hängt sehr stark von der gegenwärtigen Dialogsituation und von den kommunikativen Zielen des Benutzers und des NGS ab. Hierfür wird in Form von Textschemata [McKeown, 1985] oder Dialogstrukturen [Mann, 1988] die *argumentative* Struktur der Äußerung festgelegt, mit der dem Benutzer das kommunikative Ziel des NGS mitgeteilt werden soll. Die Basiselemente solcher Schemata konstituieren durch ihre lineare Folge die elementaren Ordnungsprinzipien von Äußerungen. Die Schemata bzw. Diskursstrukturen steuern demnach, was als nächstes zu sagen ist.

[3]Unter konzeptueller Darstellung verstehe ich die Repräsentation des Ergebnisses einer Berechnung in Ausdrücken einer Wissensrepräsentationssprache, wie z.B. KL–ONE, FRL usw. Dies soll aber nicht Systeme ausschließen, die z.B. als Ausgangsstrukturen Ausdrücke einer Datenbank haben (vgl. z.B. [McKeown, 1985])

1.2.1.2 Der inhaltsrealisierende Teil

Die Aufgabe, die ausgwählten konzeptuellen Einheiten in eine sprachliche Form zu überführen, wird vom *inhaltsrealisierenden* Teil, er wird auch *How–To–Say* (HTS) Teil genannt, übernommen. Das Ziel des HTS Teils ist es, die konzeptuellen Einheiten korrekt mit den angenommenen sprachlichen Ebenen (Semantik, Syntax, Morphologie) des natürlichsprachlichen Systems in Beziehung zu bringen. Das Ergebnis dieses Prozesses – der geäußerte Satz oder Text – muß dabei widerspiegeln, was die entsprechenden konzeptuellen Einheiten ausdrücken.

Zu den zentralen Aufgaben, die in diesem Teil zu lösen sind, zählen:

1. Die Abbildung von semantischem auf syntaktisches Wissen,
2. die Wahl geeigneter Lexeme für die konzeptuellen Einheiten,
3. die Wahl der Lexeme, die die Beziehungen dieser Einheiten widerspiegeln,
4. die Konstruktion der syntaktischen Struktur,
5. die Flexion der Lexeme und die Linearisierung der syntaktischen Struktur.

Die zentralen Wissensquellen für die Bearbeitung dieser Aufgaben sind ein Lexikon und eine Grammatik. Mit Hilfe des Lexikons werden die Lexeme ausgewählt. Sind mehrere Lexeme für die Verbalisierung eines Konzeptes möglich, muß entschieden werden, welches Lexem im gegenwärtigen Zustand am besten geeignet ist. Ein Hauptaspekt hierbei ist die Bestimmung relevanter Entscheidungspunkte, die den Auswahlprozeß steuern [Hovy, 1987]. Neben der konzeptuellen Ausgangsstruktur spielen hier semantische und syntaktische Kriterien sowie die Einbettung in bisher gemachte Äußerungen (vor allem für die Wahl der Deskriptoren) eine wichtige Rolle.

Die Grammatik wird benötigt, um die syntaktische Struktur zu konstruieren. Dabei muß entschieden werden, welche syntaktische Form am geeignetsten ist (z.B. aktiv oder passiv), und es sollte sichergestellt werden, daß nur grammatikalisch korrekte Formen erzeugt werden. Damit das lexikalische und grammatikalische Wissen erweitert und verändert werden kann, sollten beide Wissensquellen deklarativ formuliert sein.

1.2.2 Architekturprinzipien von Generierungssystemen

Von besonderem Interesse bei der Konzeption eines NGS ist die Frage, wie der WTS und HTS Teil miteinander in Beziehung stehen bzw. miteinander kommunizieren. Man kann die grundlegende Architektur von NGS wie folgt klassifizieren (vgl. [Kempen, 1986] und [**?**]):

1. Sequentielle Modelle
2. Modelle mit Rückwirkungen
3. Integrierte Modelle

Bei sequentiellen Modellen besteht nur eine unidirektionale Kommunikationsrichtung vom WTS zum HTS Teil (vgl. [McKeown, 1985], [McDonald, 1980]). Bei Modellen mit

Rückwirkungen (u.a. [Hovy, 1987], [?]) besteht eine bidirektionale Verbindung zwischen den Teilen. Damit ist es möglich, Wechselwirkungen zwischen dem inhaltsfestlegenden und –realisierenden Teil zu modellieren. Insbesondere eignen sich solche Modelle für einen inkrementellen Generierungsprozeß, da sehr früh erste Teilergebnisse dem HTS Teil zur Verbalisierung übergeben werden können.

Die bisher beschriebenen Systeme sind modulare Modelltypen, d.h. die einzelnen Komponenten haben untereinander keinen Zugriff auf internes Wissen bzw. Operationen der anderen Systemteile. Im Gegensatz dazu stehen integrierte Modelle. Im System KAMP [Appelt, 1985], dem bekanntesten Vertreter dieser Modelle, werden alle Entscheidungen als unterschiedliche Aspekte eines hierarchischen Planungsprozesses betrachtet.

1.3 POPEL: Die Generierungskomponente in XTRA

In diesem Abschnitt wird das Generierungssystem POPEL vorgestellt, in dessen Rahmen die vorliegende Arbeit enstanden ist. Die wesentlichen Merkmale von POPEL sind der *bidirektionale* Kommunikationsfluß zwischen der inhaltsfestlegenden und inhaltsrealisierenden Komponente (POPEL ist ein Modell mit *Rückwirkungen*) und der sukzessive Auswahl– und Realisierungsprozeß (POPEL ist ein Modell zur *inkrementellen* Generierung). Aus der Einbettung von POPEL in das Dialogsystem XTRA ergeben sich folgende Anforderungen an den Leistungsumfang des Generierungsprozesses [?]:

- Generierung dialogspezifischer Ausdrücke, z.B. Ellipsen,
- Erzeugung von Erklärungen,
- Verwendung von Wissensquellen des Gesamtsystems und
- Generierung von Zeigegesten.

Bevor genauer auf die Architektur von POPEL eingegangen wird, folgt eine kurze Beschreibung von XTRA.

1.3.1 Das natürlichsprachliche Dialogsystem XTRA

Das Ziel von XTRA [?] ist es, als natürlichsprachliche Schnittstelle einen Dialog zwischen einem Anwender eines Expertensystems und dem Expertensystem zu ermöglichen. Als Kommunikationsmedium für die Dialogpartner dient in erster Linie die natürliche Sprache (hier das Deutsche). Darüber hinaus erlaubt XTRA beiden Dialogpartnern die Integration von Zeigegesten in ihre Äußerungen [?]. Ein Anwender hat damit die Möglichkeit, seine natürlichsprachlich formulierten Anfragen an das Expertensystem durch Zeigen auf bestimmte Bereiche des graphisch aufbereiteten Bildschirms zu betonen. Umgekehrt kann bei der Verbalisierung in XTRA die natürlichsprachliche Äußerung durch gezieltes Markieren von Graphiken unterstützt werden [?].

Die interne Verarbeitung in XTRA läßt sich in drei Phasen aufteilen (s.a. Abb. 1.1):

1. Die Analyse einer natürlichsprachlichen Benutzereingabe.

Figure 1.1: Die Architektur von XTRA

2. Die Verarbeitung der konzeptuellen Darstellung der Eingabe des Benutzers in Zusammenhang mit dem Expertensystem.

3. Die Generierung natürlichsprachlicher Äußerungen.

Ziel in der Analysephase ist die Überführung einer natürlichsprachlichen Eingabe in einen Ausdruck der konzeptuellen Wissensquelle von XTRA. Die konzeptuelle Darstellung ist Ausgangspunkt für die Bestimmung der relevanten Daten für das Expertensystem. Um dieses Ziel zu erreichen, wird die Eingabe morphologisch, syntaktisch und satzsemantisch untersucht.

Die morphologische Analyse wird von MORPHIX [**?**] durchgeführt, einem Modul zur Flexion deutscher Wörter, das auch in der Generierungsphase eingesetzt wird (vgl. Abschnitt 6.6). In der Analysephase reduziert MORPHIX die flektierten Wortformen der Eingabe auf ihre entsprechenden Stämme und extrahiert die morphosyntaktische Information der flektierten Formen. Für diesen Prozeß verwendet MORPHIX das Lexikon XTRALEX.

Anschließend wird die so deflektierte Eingabe von der Parsingkomponente SB–PATR [?] syntaktisch analysiert. Dabei wird die unifikationsbasierte Grammatik XTRAGRAM [?] eingesetzt. Das Ergebnis der syntaktisch analysierten Eingabe wird von der satzsemantischen Bearbeitungsphase ausgewertet. Zunächst wird das Ergebnis des syntaktischen Prozesses auf eine Darstellung der funktionalsemantischen Struktur abgebildet. Im wesentlichen handelt es sich hierbei um die Transformation der syntaktischen Struktur auf Ausdrücke erweiterter Kasusrahmen.

Diese Darstellung ist Ausgangspunkt für den nächsten Schritt – die referenzsemantische Analyse. Hier wird die funktionalsemantische Repräsentation der Eingabe in eine entsprechende Struktur der konzeptuellen Wissensquelle überführt, wobei die Objekte identifiziert werden, auf die sich die sprachlichen Ausdrücke der Eingabe bezogen haben. Die Beschreibung der natürlichsprachlichen Eingabe auf der konzeptuellen Ebene ist eine Darstellung dessen, was XTRA von der Eingabe "verstanden" hat.

Diese Darstellung der Eingabe ist Ausgangspunkt für die nächste große Verarbeitungsphase in XTRA: Extraktion der relevanten Teile in eine Anfrage für das Expertensystem. Hat das Expertensystem die Anfrage bearbeitet, wird das Ergebnis in Ausdrücke der konzeptuellen Wissensquelle transformiert.

Die Aufgabe der Generierungskomponente POPEL — sie ist für den dritten großen Verarbeitungsschritt in XTRA zuständig — ist die Erzeugung kommunikativ adäquater Äußerungen aus den konzeptuellen Teilen.

1.3.2 Die Architektur und Funktionsweise von POPEL

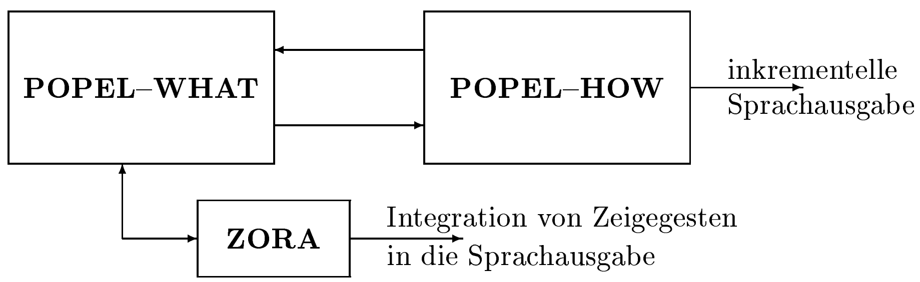


Figure 1.2: Grobarchitektur von POPEL

Abbildung 1.2 (S. 14) zeigt die Grobstruktur von POPEL (ohne die verwendeten Wissensquellen). Der inhaltsfestlegende Teil POPEL–WHAT wählt sukzessive Einheiten der konzeptuellen Wissensquelle aus, ordnet sie in der gewünschten Reihenfolge an, in der sie in der Dialogäußerung erscheinen sollen, und übergibt sie sofort der inhaltsrealisierenden Komponente POPEL–HOW. Die eintreffenden Segmente werden von POPEL–HOW (wenn möglich) über alle Darstellungsebenen transformiert: von der konzeptuellen in eine oberflächensemantische Ebene und von dort in die syntaktische Beschreibungsebene. Teilstrukturen bzw. Segmente dieser Ebene werden linearisiert, flektiert und schließlich auf einen Oberflächenstring abgebildet.

Die dritte Komponente ist der Zeigegestengenerator ZORA. Seine Aufgabe ist die Erzeugung von Zeigegesten und ihre Integration in die Äußerung. Auf ZORA wird im weiteren nicht mehr eingegangen.

Es wurde bereits erwähnt, daß POPEL ein Modell mit Rückwirkungen ist, d.h. POPEL–WHAT und POPEL–HOW können während des Generierungsprozesses bidirektional miteinander kommunizieren. Die notwendigen Rückwirkungen zwischen dem inhaltsfestlegenden und inhaltsrealisierenden Teil findet dann statt, wenn:

- POPEL–HOW für den Abbildungsprozeß über die verschiedenen Ebenen mehr konzeptuelle Ausgangsstrukturen benötigt, als von POPEL–WHAT zur Verfügung gestellt wurde;

- POPEL–HOW eine Nominalphrase generiert. Hierbei müssen z.B. Entscheidungen bezüglich folgender Fragen behandelt werden: Soll ein indefiniter nominaler Ausdruck erzeugt werden oder soll bei einer früheren Erwähnung nun eine definite oder pronominalisierte Phrase generiert werden?

- POPEL–WHAT von POPEL–HOW mitgeteilt wird, in welcher Abfolge es die syntaktischen Teilstrukturen linearisert hat. Dies hat einen Einfluß auf die kommunikative Dynamik oder Thema–Rhema Struktur der als nächstes zu verbalisierenden Teile.

Die bidirektionale Verknüpfung ist Grundlage für den inkrementellen Generierungsprozeß in POPEL: POPEL–WHAT muß nicht erst die gesamte konzeptuelle Struktur der Äußerung bestimmen, sondern übergibt bereits sehr früh erste konzeptuelle Teilstrukturen für die Verbalisierung. POPEL–HOW kann sofort mit der Inhaltsrealisierung dieser Teile beginnen, während POPEL–WHAT parallel dazu die nächsten Einheiten bestimmt. Da POPEL–HOW in der Lage ist, unvollständige Strukturen zu entdecken und die fehlende Information bei POPEL–WHAT anzufordern, beeinflußt POPEL–HOW in diesem Fall den Auswahlprozeß für weitere Teilstrukturen in POPEL–WHAT. Das bedeutet: Es können während der Inhaltsbestimmung frühzeitig Beschränkungen bei der sprachlichen Umsetzung des Inhalts berücksichtigt und der Inhalt den sprachlichen Restriktionen angepaßt werden.

1.3.3 Die Wissensquellen von POPEL

Ein weiteres wichtiges Merkmal von POPEL ist die Verwendung von Wissensquellen des Gesamtsystems XTRA. Wissensquellen, die gemeinsam von der Analyse und Generierung eingesetzt werden (sie werden daher auch als *bidirektionale* Wissensquellen bezeichnet) haben folgende Vorteile:

- Der Aufwand für XTRA reduziert sich. Da nur eine Wissensquelle für bestimmte Teilaufgaben verwendet wird, verringert sich neben dem Speicherbedarf vor allem der Verwaltungsaufwand für dieses Wissen.

- Wichtiger ist aber: Der eine Verarbeitungsabschnitt kann immer die Ergebnisse der "Gegenseite" in seinen Berechnungen einbeziehen. In einem Dialogsystem wie XTRA ist der Einsatz von gemeinsamen Wissensquellen Voraussetzung dafür, bei der Erzeugung einer Antwort auf eine Äußerung des Benutzers gezielt Bezug zu nehmen, z.B. durch Generierung von Ellipsen oder Anaphern.

Die Wissensquellen, die in der Analysephase und in POPEL eingesetzt werden sind die konzeptuelle und funktionalsemantische Wissensquelle, das Dialoggedächtnis und das Lexikon. Auf der syntaktischen Beschreibungsebene werden z.Z. zwei unterschiedliche Grammatiken eingesetzt. Gemeinsam ist beiden Grammatiken, daß sie auf dem gleichen Formalismus, nämlich PATR–II, basieren, so daß ein großer Teil gemeinsamer Entwicklungswerkzeuge für die Erstellung der Grammatiken eingesetzt werden kann. Allerdings sind die Prozesse, die auf den Grammatiken operieren derart verschieden, daß es nicht möglich war, die z.Z. der Entwicklung von POPEL–HOW existierende Version der Analysegrammatik XTRAGRAM in der Generierungsphase zu verwenden.

Es folgt eine kurze Beschreibung der wichtigsten Wissensquellen des Gesamtssytems, die in POPEL eingesetzt werden:

Die konzeptuelle Wissensquelle (Conceptuel Knowledge Base CKB)

Die CKB ist die zentrale Wissensquelle von XTRA. Sie repräsentiert Allgemein– und Sachwissen des Gesamtsystems. Der zugrundliegende Repräsentationsformalismus ist SB–ONE, eine Weiterentwicklung von KL–ONE [?]. Neben Konzepten und Relationen (und ihren Beziehungen zueinander) als Darstellungsmittel erlaubt SB–ONE eine Partitionierung der Vererbungshierarchie der Konzepte und der Individualisierungen in unterschiedliche Kontexte. Damit ist es in der CKB möglich, Wissen über die Pläne und Überzeugungen des Benutzers und des Systems zu repräsentieren.

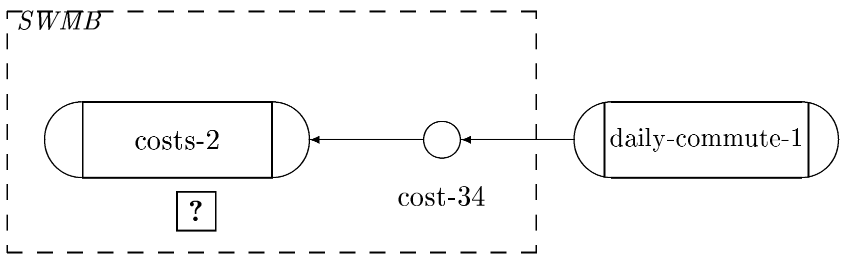


Figure 1.3: Ein Beispiel für den Inhalt im Kontext SWMB.

Die entsprechende Komponente in XTRA, die diese Einteilung der CKB vornimmt und verwaltet, ist das System BGP–MS (Belief, Goal and Plan – Maintenance System). Für POPEL ist der Kontext SWMB (System Wants Mutual Belief Exists) am wichtigsten. Dieser Kontext enthält dasjenige konzeptuelle Wissen von XTRA, von dem das System verlangt, daß es auch vom Benutzer "gewußt" wird. Daher wird dort das Ergebnis der

Beantwortung der Benutzeranfrage abgelegt. Der Inhalt im SWMB ist Ausgangsstruktur für den Generierungsprozeß, im Speziellen für POPEL–WHAT (s. Abb. 1.3, S. 16).

Die funktionalsemantische Wissensquelle (Functional Semantic Structure FSS)

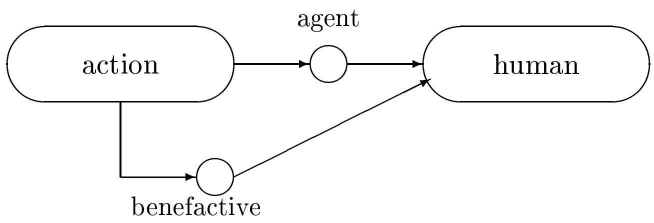


Figure 1.4: Ein kleiner Ausschnitt aus der FSS T–Box.

Die FSS ist ebenfalls in SB–ONE ausgedrückt. Sie definiert, wie Elemente von Prädikatsklassen mit Elementen anderer abstrakter Wortklassen zu semantisch korrekten komplexen Ausdrücken kombiniert werden dürfen. Die Beziehungen zwischen den Klassen werden mit Hilfe von Kasusrahmenbeschreibungen (dargestellt durch entsprechende Rollenangaben) ausgedrückt (s. Abb. 1.4, S. 17).

In diesem Sinne steht die FSS in der Tradition von Fillmores Kasusgrammatik, d.h. sie beschreibt die oberflächensemantische Wohlgeformtheit sprachlicher Ausdrücke[4].

Das Dialoggedächtnis (Linguistic Dialog Memory LDM)

Diese Wissensquelle repräsentiert den bisherigen Verlauf und die Struktur des Dialogs. Das LDM besteht aus drei Teilen:

1. Den referentiellen Objekten
2. Dem Dialogfolgegedächtnis
3. Dem Dialogkontextmodell

Für jedes im Dialog eingeführte Objekt der CKB wird ein referentielles Objekt erzeugt. Die interne Struktur solcher Objekte besteht aus Verweisen auf bestimmte Wissensquellen von XTRA. Mit Hilfe dieser Verweise wird festgehalten, welche konzeptuelle und funktionalsemantische Darstellung mit den eingeführten Objekten assoziiert ist und welche Position jedes Objekt im bisherigen Dialogverlauf einnimmt. Dies wird im Dialogfolgegedächtnis repräsentiert. Wichtig ist, daß dort die Sequenz der Dialogbeiträge von beiden Dialogpartnern (i.a. XTRA und ein Benutzer) festgehalten wird. Das Dialogfolgegedächtnis erlaubt

[4]Damit ist die FSS allerdings mit den gleichen Problemen konfrontiert, die sich aufgrund des theoretischen Status für die Kasustheorie ergeben (vgl. [?], [?]).

es – relativ zur Position – festzustellen, wann Objekte im Dialog von wem eingeführt wurden.

Im Dialogkontextmodell wird der Dialog nach thematischen und rhetorischen Kriterien strukturiert. Hierfür verfügt das Modell über mehrere Kontexträume, mit denen die verschiedenen Abschnitte des Dialogs modelliert werden. Die Kontexträume besitzen Verbindungen zu den Elementen des Dialogfolgegedächtnisses. Mit Hilfe einer Zustandsbeschreibung für einen Kontextraum kann u.a. festgestellt werden, welcher Dialograum gerade aktiv ist und welches im Dialog eingeführte Objekt ihn eröffnete.

Chapter 2

Überblick über die inhaltsrealisierende Komponente POPEL–HOW

In diesem Kapitel wird die Architektur und Funktionsweise der inhaltsrealisierenden Komponente POPEL–HOW vorgestellt. Im wesentlichen handelt es sich um diejenigen Teile der Komponente, die in [**?**] ausführlich beschrieben sind. Sie werden daher hier nur soweit erläutert, wie sie für das Verständis der Hauptthemen dieser Arbeit notwendig sind. Im letzten Abschnitt werden dann die besonderen Anforderungen an eine inkrementelle und parallele Konstruktion syntaktischer Strukturen in POPEL–HOW aufgeführt.

2.1 Die Architektur von POPEL–HOW

Die Abbildung 2.1 (S. 20) zeigt die Architektur von POPEL–HOW. POPEL–HOW besteht aus vier Teilsystemen. Die Teilsysteme – sie werden auch als *Prozeßebenen* bezeichnet – entsprechen den Beschreibungsebenen, die während des Verbalisierungsprozesses einer Äußerung durchlaufen werden.

Ausgangspunkt für die Realisierung ist die konzeptuelle Wissensquelle CKB. Individualisierte Strukturen dieser Repräsentationsebene werden mit Hilfe von Abbildungsregeln auf entsprechende Strukturen der funktionalsemantischen Ebene FSS transformiert. FSS Strukturen werden (ebenfalls regelbasiert) auf die syntaktische Beschreibungsebene abgebildet.

In POPEL–HOW wird die letztgenannte Beschreibungsebene durch zwei Prozeßebenen modelliert. Die zentrale Wissensquelle in diesen beiden Ebenen ist POPEL–GRAM, eine auf dem PATR–II Formalismus basierende *Unifikationsgrammatik*. Sie wird in dieser Arbeit (vgl. Abschnitt 4.1) ausführlich vorgestellt. Ein wichtiges Merkmal von POPEL–GRAM ist die Aufteilung der Regeln in solche, die die unmittelbare Konstituentenstruktur beschreiben und solche, die die lineare Abfolge spezifizieren. Dies erlaubt es, die Konstruktion syntaktischer Strukturen auf der DBS–Ebene (Dependence Based Structures) und die Bestimmung der linearen Struktur auf der ILS–Ebene (Inflected Linearized Structures) getrennt durchzuführen. Diese Trennung in zwei Teilebenen unterstützt den parallelen und inkrementellen Generierungsprozeß, da u.a. beim parallelen Konstruktionsprozeß der syntaktischen Struktur von der linearen Ordnung abstrahiert werden kann [**?**]. In der ILS–Ebene wird zudem noch die Flexion der Stämme mit Hilfe des Morphologiemoduls MORPHIX sowie die Ausgabe verbalisierter Strukturen durchgeführt.

Die ersten drei Ebenen besitzen als operationale Grundlage das gleiche *verteilte parallele Modell* (s.a. 2.2). Die Erzeugung von Strukturen auf diesen Ebenen wird durch *kooperierende Prozesse* geleistet. Jedem aktiven Prozeß wird ein *Segment* der entsprechenden Wissensquelle zugeordnet. In diesem Sinne beschreibt ein Prozeß die *Aktivierung* einer Teilstruktur. Das zentrale Problem hierbei ist die Frage, wie die entsprechenden Wissensquellen sinnvoll *segmentiert* werden können. Die Aufgabe eines Prozesses ist die

Figure 2.1: Die Architektur von POPEL–HOW

Abbildung seines Segmentes in die nächste Prozeßebene. Dabei wird ein neuer Prozeß kreiert, dem als Segment die Zielstruktur der Abbildung zugeordnet wird.

Die verteilte parallele Verarbeitung in POPEL–HOW unterstützt die inkrementelle Spezifikation der Eingabe: Teile der Eingabe können bereits frühzeitig in die nächste Ebene abgebildet werden, während laufend neue Eingabeteile in POPEL–HOW eintreffen. Da das Ergebnis des Abbildungsprozesses ein neuer Prozeß mit einem entsprechenden Segment ist, wird das inkrementelle Eingabeverhalten auf allen Prozeßebenen erreicht.

Ist die Abbildung eines Prozesses aufgrund eines unterspezifizierten Segments nicht möglich, kann der Prozeß durch Kommunikation mit Prozessen der übergeordneten Ebene die fehlende Information anfordern. Es ist allerdings möglich, daß ein aufgeforderter Prozeß diese Information nicht bereitsstellen kann, da ihm selbst wichtige Information fehlt. Um dennoch die an ihn gestellte Anfrage zu erfüllen, sendet er seinerseits eine Anforderung an Prozesse seiner übergeordneten Ebene. Auf diese Weise wird eventuell die Forderung nach Bereitsstellung fehlender Information bis auf die konzeptuelle Ebene weitergereicht.

Kann auch in der CKB–Ebene von POPEL–HOW die Anforderung nicht bearbeitet werden, so bedeutet dies, daß zum gegenwärtigen Zeitpunkt wichtige Segmente (und entsprechende Prozesse) nicht spezifiziert sind. Da sie jedoch für die Weiterführung des Verbalisierungsprozesses notwendig sind, wird von dieser Ebene aus POPEL–WHAT aufgefordert, die entsprechenden Segmente bereitzustellen. Falls POPEL–WHAT dies ursprünglich gar nicht oder zumindest zu diesem Zeitpunkt nicht vorgesehen hat, wird der Planungsprozeß entscheidend beeinflußt. Auf diese Weise hat POPEL–HOW einen direkten Einfluß auf die Festlegung in POPEL–WHAT.

2.2 Das zugrundeliegende parallele Prozeßmodell

Die Entwicklung des verteilten parallelen Modells basiert auf der Methode des *object-oriented concurrent programming* [?]. Modelle oder Systeme, die nach dieser Methode entwickelt werden, bestehen aus einer Kollektion von gleichzeitig ausgeführten Programmoduln – den *Objekten* –, die kooperativ an der Lösung einer Aufgabe arbeiten. Die Kommunikation zwischen den Objekten wird durch Versenden und Beantworten von Botschaften realisiert. Die Komplexität der Objekte hängt von der Art und Weise ab, wie das Problem in parallel verarbeitbare Segmente zerlegt werden kann und welche Operationen für die Bearbeitung der einzelnen Segmente benötigt werden.

In POPEL–HOW entsprechen die Objekte aktiven abgeschlossenen, d.h. nicht direkt von außen manipulierbaren Prozessen. Jedes Objekt führt zur Lösung seiner Teilaufgabe – Abbildung seines zugeordneten Segments der entsprechenden Wissensquelle in die nächste Ebene – sein eigenes sequentielles Programm aus. Dieses Programm bezeichnen wir als den *Lebenszyklus* eines Objektes. Die wesentlichen Schritte des Zyklus sind:

1. Bekanntmachung mit neuen Objekten
2. Wenn möglich, Aufbau von Kommunikationsverbindungen zu diesen Objekten
3. Versuch der Abbildung des zugeordneten Segments in die nächste Ebene

4. Versenden oder Bearbeiten von Anfragen bezüglich unterspezifizierter Segmente

5. Prüfung, ob neue Objekte eingetroffen sind

6. Weiter mit Schritt 1

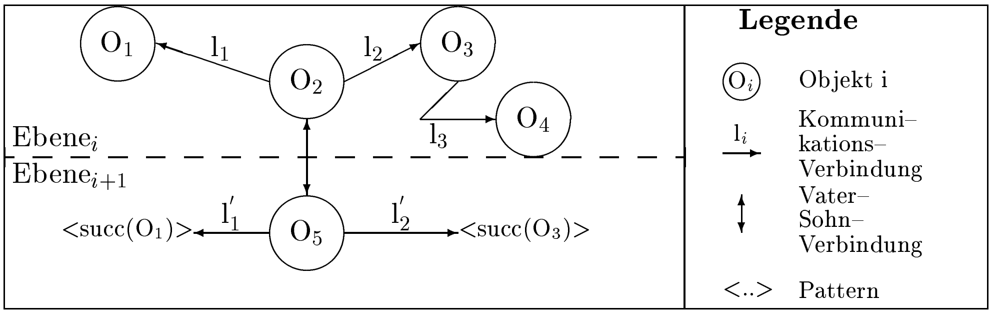


Figure 2.2: Objekte von zwei benachbarten Ebenen

Jedes Objekt prüft zunächst, ob es mit neuen Objekten eine Nachbarschaftsbeziehung etablieren kann. Eine derartige Verbindung kann dann hergestellt werden, wenn die Objekte in der zugrundeliegenden Wissensquelle in direkter Relation zueinander stehen. Ist dies der Fall, nimmt ein Objekt die Adresse eines neuen Objektes auf. Die Adresse dient dem Objekt als *Kommunikationsverbindung* (Communication Link *ComLi*). Sie läßt sich graphisch als gerichtete Kante darstellen (s. Abb. 2.2).

Die Menge aller Kommunikationsverbindungen konstituiert den *Kontext* eines Objektes. Die Entscheidung, welche der neuen Objekte als Nachbarn bzw. Kommunikationspartner zugelassen werden sollen, kann nur mit Hilfe des zugeordneten Segments getroffen werden. Beim Aufbau einer aktuellen Kommunikationsverbindung müssen die beteiligten Objekte synchronisiert werden.

Der Transformationsschritt wird mit lokalen Abbildungsregeln durchgeführt. Während des Abbildungsprozesses vergleicht das Objekt seinen aktuellen Kontext mit dem Bedingungsteil einer Regel. Je nachdem, wie komplex die Regeln sind, ist in diesem Schritt Kommmunikation zu mittelbar verknüpften Objekten notwendig. Ist der Abbildungsprozeß erfolgreich, wird in der nächsten Ebene eine neues Objekt erzeugt. Zwischen erzeugendem Objekt (Vater–Objekt) und erzeugtem Objekt (Sohn–Objekt) wird eine bidirektionale Verbindung aufgebaut. Über diese Verbindung werden Anfragen vom Sohn–Objekt an das Vater–Objekt übermittelt (s. Abb. 2.2).

Der Aktionsteil der angewendeten Regel spezifiziert den kontextuellen Rahmen für ein erzeugtes Objekt. Dieser *Rahmenkontext* legt fest, mit welchen Objekten es Komunikationsverbindungen etablieren kann. Die Elemente des kontextuellen Rahmens bezeichnen wir deshalb als potentielle Kommunikationsverbindungen oder *Kommunikationspattern* (Communication Pattern *ComPat*).

Jedes ComPat beschreibt das Verhältnis zwischen dem erwarteten Kommunikationspartner und seinem Vater–Objekt sowie zu seinem zugeordneten Segment. Scheitert die

Abbildung eines Objektes in die nächste Ebene, kann das Objekt mit Hilfe der Kommunikationspattern feststellen, welche Objekte in der Ebene, von denen wichtige Information benötigt wird, noch nicht erzeugt wurden; d.h. über die ComPat eines Objektes werden ebenfalls Anfragen an die übergeordnete Ebene gesendet.

Ein Objekt, das zu einem bestimmten Zeitpunkt seine Aufgabe gelöst hat, darf nicht terminieren. Die Gründe sind:

- Es sollte als kooperatives Objekt ständig in der Lage sein, Information an bestimmte Objekte zu versenden, um ihnen bei der Lösung ihrer Teilaufgabe zu helfen.

- Aufgrund der inkrementellen Eingabe ist es möglich, daß das Objekt mit neuen Objekten eine Verbindung aufbaut, so daß es nun ein spezielleres Segment repräsentiert. Wenn das Objekt über entsprechende Abbildungsregeln verfügt, muß es sein neues Segment in die nächste Ebene abbilden, damit dort ebenfalls die neue Information verarbeitet werden kann.

Daher ist die Terminierung aller Objekte in POPEL–HOW erst dann sinnvoll, wenn keine weitere Eingabe erfolgt. Da diese Entscheidung bei POPEL–WHAT liegt, erhält POPEL–HOW in diesem Fall eine spezielle Terminierungsbotschaft, die an alle Objekte übermittelt wird. Diejenigen Objekte, die bei Bekanntgabe der Terminierungsbotschaft noch kein Objekt in der nachfolgenden Ebene erzeugen konnten, wenden ausgezeichnete Regeln an. Mit Hilfe dieser Regeln ist ein Objekt in der Lage, sein unvollständiges Segment so weit zu erweitern, daß der Abbildungsprozeß auf jeden Fall gelingt. Diese ausgezeichneten Regeln werden daher auch als Defaultregeln bezeichnet[1]. Damit wird gewährleistet, daß POPEL–HOW auf jeden Fall mit einem – wenn auch unvollständigen – Ergebnis terminiert.

2.3 Segmentierung und Verteilung der SB–ONE basierten Wissensquellen

Um die deklarativ formulierten Wissensquellen in POPEL–HOW effizient mit den Objekten des verteilten parallelen Modells in Beziehung bringen zu können, werden sie in *elementare* Segmente zerlegt. Die Definition elementarer Segmente ist abhängig vom zugrundeliegenden Repräsentationsformalismus der Wissensquelle. Ich beschränke mich in diesem Abschnitt auf die SB–ONE basierten Ebenen, da die Segmentierung und Verteilung syntaktischer Strukturen in folgenden Kapiteln noch ausführlich erläutert wird.

Basiseinheiten der epistemologischen Ebene in SB–ONE (vgl. [**?**]) sind die Konzepte und ihre attributiven Beschreibungen der generellen und individuellen Beschreibungsebene. Eine attributive Beschreibung besteht im allgemeinen aus einer Rolle, Modalitäts– und Kardinalitätsangaben und der Werteinschränkung der Rolle. Wir definieren ein elementares Segment der SB–ONE basierten Wissensquellen daher als “ein individuelles Konzept plus seine attributiven Beschreibungen”[2].

In der Abbildung 2.3 sind die elementaren Segmente eines individualisierten SB–ONE Netzes durch die gestrichelten Linien dargestellt. Der rechte Teil der Abbildung zeigt,

[1]In [**?**] wird die Verarbeitung von Defaultregeln auf allen Prozeßebenen beschrieben.

[2]Zur Zeit werden in POPEL nur individuelle Konzepte verbalisiert (vgl. [**?**]).

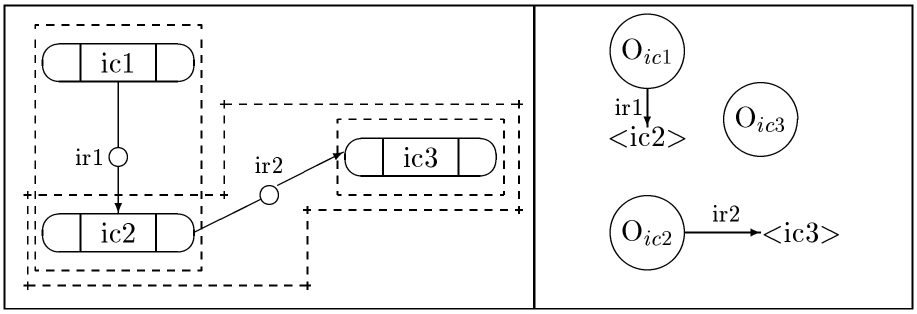


Figure 2.3: Segmentierung und Verteilung eines kleinen SB–ONE Netzes

wie die Segmente mit Objekten in Beziehung gebracht werden. Die Rollen der attributiven Beschreibung korrespondieren mit den erwarteten Kommunikationsverbindungen, und die Werteinschränkungen werden eingesetzt, um die Anzahl der möglichen Kommunikationspartner zu beschränken. Insgesamt beschreiben sie den kontextuellen Rahmen eines Objektes.

Die Zuordnung zwischen Segmenten der CKB und FSS und der Objektstruktur ist unmittelbar, d.h. ein Objekt repräsentiert *direkt* und *explizit* ein SB–ONE Segment. Diese direkte Beziehung erlaubt es auch, Objekte der CKB– und FSS–Ebene als *aktive individuelle Konzepte* zu betrachten. Die Gesamtheit miteinander kommunizierender Objekte repräsentiert ein *aktives individualisiertes SB–ONE Netz.*

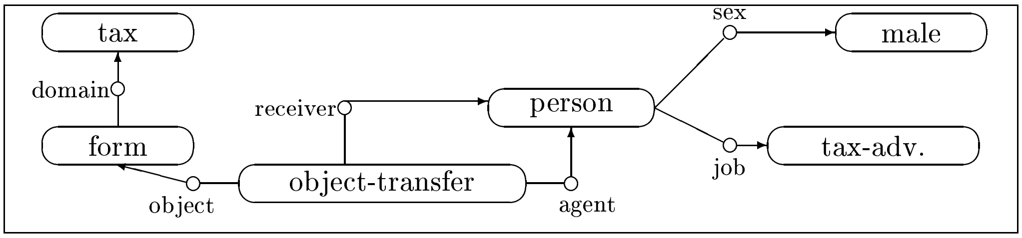
2.4 Verteilter Abbildungsprozeß zwischen den Ebenen

Die verschiedenen Wissensquellen müssen miteinander in Beziehung gebracht werden, damit überhaupt ein Abbildungsprozeß von konzeptuellen Strukturen in die Äußerungsebene möglich ist. In POPEL–HOW wird die Abbildung von Segmenten in die Nachfolgeebene durch Anwendung lokaler Abildungsregeln *verteilt* durchgeführt. Dies ist eine Voraussetzung dafür, daß die einzelnen Objekte so unabhängig wie möglich ihre Aufgabe lösen können.

Ausgangspunkt für die Formulierung von lokalen Übersetzungsregeln auf den SB–ONE basierten Ebenen CKB und FSS sind die generellen Konzepte. Jedem generellen Konzept wird eine Menge von lokalen Regeln zugeordnet. Die Regeln sind lokal in dem Sinne, daß sie sich auf den Typ und die attributiven Beschreibungen des Konzeptes beschränken. Der Bedingungsteil einer Regel beschreibt eine mögliche individuelle Ausprägung des generellen Konzeptes. Er schreibt vor, welche generellen Rollen individualisiert sein müssen, damit der Transformationsschritt durchgeführt werden kann. Der Aktionsteil spezifiziert, auf welches individualisierte Zielkonzept der nachfolgenden Ebene abgebildet werden soll[3]. Beispielsweise läßt sich für das generelle CKB–Konzept *object–transfer* des CKB–Netzes:

Beispiel 1

[3]In [?] wird gezeigt, wie komplexe Strukturen bzw. Teilnetze zusammenhängend auf ein Konzept der nächsten Ebene abgebildet und einzelne attributive Beschreibungen verarbeitet werden können.

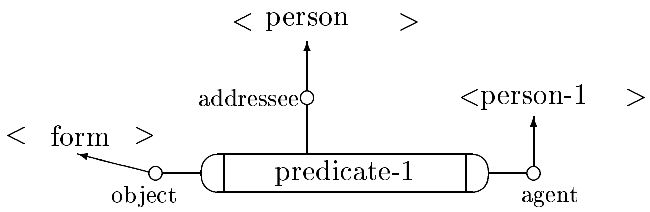


folgende Regelmenge spezifizieren[4]:

if agent & object & receiver
then create predicate
with
agent:=agent & addressee:=receiver & object:=object

if agent & object
then create predicate
with
agent:=agent & object:=object

Mit der ersten Regel läßt sich ein individualisiertes FSS–Konzept der folgenden Form erzeugen :



Die Notation < ... > ist ein Pattern für FSS Individualisierungen, die durch Auswertung von Regeln entsprechender genereller Konzepte erzeugt werden.

Objekte von SB–ONE basierten Ebenen können mit Hilfe dieser Regeln überprüfen, ob ihr zugeordnetes Segment ausreichend spezifiziert ist, um es erfolgreich in die nächste Ebene überführen zu können. Da die Segmente der Objekte individuelle Konzepte darstellen, ist ein direkter Zugriff auf die entsprechende Regelmenge möglich. Ein CKB–Objekt vergleicht seinen Kontext (d.h. die Menge seiner aktuellen ComLi) mit den Bedingungsteilen jeder Regel. Die Regel, die mit dem Kontext und mit dem individuellen Konzept übereinstimmt, wird für den Abbildungsprozeß herangezogen und ein entsprechendes FSS–Objekt erzeugt, für das im Aktionsteil der Regel das entsprechende Segment spezifiziert ist. Sind mehrere Regeln anwendbar, so wird je nach verfolgter Regelstrategie eine entsprechende Auswahl getroffen [**?**].

Der Übergang von der CKB– auf die FSS–Ebene ist ein *homogener* Transformationsschritt, da beide Wissensquellen in SB–ONE implementiert sind. Erzeugte FSS–Objekte repräsentieren *potentielle* individuelle Konzepte, wenn der Kontext noch nicht vollständig

[4]Ich verwende hier eine informelle Syntax. Die formale Spezifikation von Regeln ist in [**?**] beschrieben.

spezifiziert ist, d.h. noch nicht alle im Kontext angegebenen Kommunikationspartner existieren.

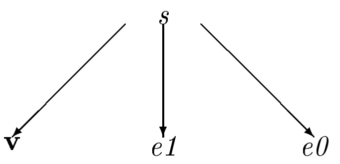
Der Übergang von der FSS– auf die DBS–Ebene wird ebenfalls verteilt durchgeführt. Jede Regel der Regelmenge eines generellen FSS–Konzeptes formuliert, auf welches syntaktische Segment ein entsprechendes individuelles FSS–Konzept transformiert werden kann.

Die Formulierung des syntaktischen Segments in einer Regel hängt wesentlich vom zugrundeliegenden Formalismus der syntaktischen Beschreibungsebene ab. In POPEL–HOW handelt es sich hierbei um die unifikationsbasierte Grammatik POPEL–GRAM. Damit ist aber die Beziehung zwischen FSS und POPEL–GRAM *heterogen*.

Die linguistische Basis von POPEL–GRAM ist die Dependenztheorie (vgl. 3.1.1). Dies hat einen starken Einfluß auf die Definition der syntaktischen Segmente und die Art und Weise, wie sie verteilt verarbeitet werden können. Syntaktische Segmente in POPEL–GRAM sind Phrasen. Jede Phrase hat eine zentrale Konstituente. Diese Konstituente definiert zum einen die wesentlichen grammatikalischen Eigenschaften der Phrase und zum anderen die Anzahl und Art der weiteren unmittelbaren Konstituenten.

In einer Satzphrase s (Bsp. 2) ist v die zentrale Konstituente. Die weiteren Konstituenten (im Beispiel $e0$ und $e1$) hängen von dieser Konstituente ab.

Beispiel 2



Ich werde v im weiteren als *head*–Element und $e0$ sowie $e1$ als dessen *Dependenten* bezeichnen. Die Dependenten bilden den Dependenzrahmen für das head–Element. Da die Phrase s durch ihr head–Element definiert wird, sprechen wir der Phrase ebenfalls diesen Dependenzrahmen zu. Ausgehend von diesen Begriffen wird festgelegt, daß ein individualisiertes FSS-Konzept auf eine entsprechende Phrase und deren Dependenzrahmen abgebildet wird.

Eine Abbildungsregel für das generelle FSS Konzept des individuellen Konzepts in obigen Beispiel ist:

Beispiel 3

if	agent & object
then create	vcomp
with	
	e0:=agent & e1:=object

2.5 Die Wortwahl in POPEL–HOW

Beim Wortwahlprozeß in POPEL–HOW[5] unterscheiden wir zwischen der Bestimmung von Inhaltswörtern und Funktionswörtern. Bei den *Inhaltswörtern* handelt es sich um Lexeme, die mit nominalen oder prädikativen konzeptuellen Strukturen in Beziehung stehen. Im allgemeinen sind dies gerade Wörter der offenen Wortklassen (Nomen, Verben, Adjektive). Als *Funktionswörter* bezeichnet man die sprachlichen Elemente, die in erster Linie grammatikalische Bedeutung tragen und vor allem syntaktischstrukturelle Funktionen in einer Äußerung einnehmen. Im allgemeinen gehören die Funktionswörter den geschlossenen Wortklassen an, d.h. sie sind in ihrem Umfang beschränkt (u.a. Artikel, Präpositionen, Konjunktionen, Partikel).

In POPEL–HOW gehen wir von einer *zweistufigen* Wortwahl aus. In der ersten Stufe werden die Inhaltswörter und in der zweiten Stufe die Funktionswörter bestimmt. Die Inhaltswörter werden in POPEL–HOW beim Übergang von der konzeptuellen Ebene (CKB) in die funktionalsemantische Ebene (FSS) bestimmt. Für jedes individuelle Konzept wird festgelegt, welches Lexem für die aktuelle Ausprägung geeignet ist. Das so bestimmte Lexem bezeichnen wir als *head–Lexem*. Daneben ist es auch möglich, zusätzlich Lexeme auszuwählen, die das head–Lexem modifizieren. Diese Lexeme heißen *modifier–Lexeme*. Dadurch kann anstatt eines semantisch komplexen Begriffes eine gleichbedeutende Lexemgruppe herangezogen werden. Damit ist es möglich, z.B. alternativ "Kleinwagen" oder "kleines Auto" zu verbalisieren.

Bei der Auswahl der Deskriptoren für ein nominales Konzept muß berücksichtigt werden, ob das Konzept erstmals im Dialog eingeführt oder bereits zu einem früheren Zeitpunkt im Dialoggedächtnis abgelegt wurde. Ist das Konzept im Dialog neu eingeführt, wird in der syntaktischen Ebene von POPEL–HOW z.B. eine indefinite Nominalphrase erzeugt. Im anderen Fall wird entweder eine definite Nominalphrase oder eine pronominalisierte Phrase erzeugt. Die Enscheidung hierfür wird durch Interaktion mit POPEL–WHAT getroffen.

Die Funktionswörter werden beim Übergang von der funktionalsemantischen Ebene in die syntaktische Ebene (DBS) bestimmt. Die Wahl einer Präposition wird z.B. in Abhängigkeit des aktuellen semantischen Kasus bestimmt, der als Relation zwischen einem Prädikat und nominalen Element besteht.

Zur Zeit wird der Wortwahlprozeß auf jeder Stufe während der Auswertung der Abbildungsregeln durchgeführt. Der Bedingungsteil einer anwendbaren Regel schränkt ein, welche Lexeme für diese Abbildung eines Objektes sinnvoll ist. Zusätzlich ist jeder Regel ein Diskriminationsnetz zugeordnet, mit dem das angemessene Lexem bestimmt wird. Beim Übergang wird mit Hilfe des Netzes die aktuelle Belegung der individualisierten Rollen des CKB Konzeptes bestimmt, wobei auch Rollen berücksichtig werden können, die im Bedingungsteil nicht spezifiziert sind. Je nach Belegung wird ein unterschiedliches Lexem gewählt.

Beispiel 4

[5] Bei dem im Prototypen entwickelten Wortwahlprozeß handelt es sich um eine nicht vollständig implementierte Version des hier beschriebenen Ansatzes (vgl. [?].

if agent & object & receiver
then create predicate
with
agent:=agent & addressee:=receiver & object:=object

choose geben

Im zweiten Schritt des Wortwahlprozesses, werden beim Übergang von der FSS– auf die DBS–Ebene die Funktionswörter bestimmt. Hier fungieren die Bedingungsteile von Regeln ebenfalls als Diskriminationsnetze.

2.6 Die Konstruktion syntaktischer Strukturen in POPEL–HOW

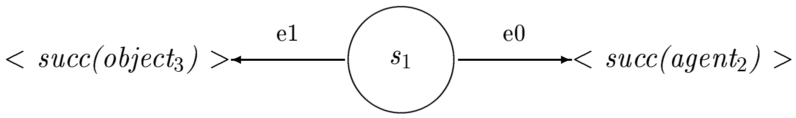
Die Aufgabe auf der syntaktischen Beschreibungsebene ist die Konstruktion der syntaktischen Struktur, ihre Flexion und Linearisierung. Der Konstruktionsprozeß soll entsprechend den Anforderungen an POPEL–HOW parallel und inkrementell verlaufen, sowie bidirektional mit der übergeordneten Ebene (der funktionalsemantischen Ebene) verknüpft sein. Die syntaktische Beschreibungsebene besteht in POPEL-HOW aus den beiden Prozeßebenen DBS und ILS. Auf der DBS–Ebene wird die syntaktische Struktur inkrementell und parallel konstruiert und auf der ILS–Ebene linearisiert sowie flektiert. Der ILS–Ebene liegt kein paralleles Operationsmodell zugrunde. Die Gründe hierfür sind in Kapitel 6.2 aufgeführt. Das Ergebnis der ILS–Ebene, der flektierte und linearisierte Oberflächenform, ist zugleich das Endergebnis des gesamten Generierungsprozesses.

2.6.1 Aktive Objekte als Ausgangsstruktur für den Konstruktionsprozeß

Die Objekte der DBS–Ebene werden durch den verteilten Abbildungsprozeß FSS $\longrightarrow$ DBS erzeugt. Sie sind die Ausgangsstrukturen für den Aufbau der syntaktischen Struktur.

Ein DBS–Objekt repräsentiert nach seiner Erzeugung eine Phrase. Der Kontext des Objektes wird durch den Dependenzrahmen der Phrase spezifiziert. Weiterhin verfügt das Objekt bereits über alle notwendigen Lexeme. Beispiel 5 zeigt ein DBS–Objekt, das durch Regelanwendung des FSS–Objektes aus Bsp. 1 erzeugt wurde. Die Bezeichner der Objekte zeigen an, welche Segmente mit den Objekten assoziiert sind. Die Indizes beziehen sich auf die Objekte.

Beispiel 5



Nach ihrer Erzeugung unterscheidet sich der Zustand der DBS–Objekte strukturell nicht von Objekten der CKB– oder FSS–Ebene. Im Unterschied zu diesen Objekten repräsentieren DBS–Objekte jedoch nicht unmittelbar das ihnen zugeordnete Segment. Im wesentlichen hängt das damit zusammen, daß die reine Konstituentenstruktur von Phrasen nur sehr *komprimiert* die entsprechende syntaktische Struktur wiedergibt (vgl. [?])[6]. DBS–Objekte beschreiben daher nach ihrer Erzeugung nur die externe Struktur einer Phrase. Sie müssen, damit ihre Segmente in der ILS–Ebene flektiert und linearisiert werden können, die explizite syntaktische Form ihrer Phrase selbst bestimmen.

2.6.2 Anforderungen an die Teilebenen und die Gestaltung der notwendigen Wissensquellen

Als besondere Anforderungen an Objekte der DBS–Ebene ergeben sich im wesentlichen folgende Punkte:

1. Die Lexeme und die Phrase eines DBS–Objektes müssen miteinander in Beziehung gebracht werden. Damit wird ein Gerüst für die explizite Phrasenstruktur geschaffen.

2. Wenn die Phrase mit anderen Phrasen syntaktisch in Beziehung stehen soll, müssen bekannte DBS–Objekte miteinander kooperieren, um relevante Information austauschen zu können.

3. Aufgrund mehrdeutiger Abbildungsmöglichkeiten befinden sich auf der DBS–Ebene eventuell konkurrierende Objekte.

4. Ein DBS–Objekt muß lokal für sich entscheiden können, ob seine explizite Phrasenstruktur vollständig genug beschrieben ist, damit sie in der ILS–Ebene weiterverarbeitet werden kann.

5. Wenn diese lokale Vollständigkeit nicht erfüllt werden kann, weil Objekte in der Ebene fehlen, die die relevante Information tragen, sollte ein DBS–Objekt in der Lage sein, die Erzeugung der fehlenden Objekte anzufordern.

Wenn alle diese Punkte berücksichtigt werden, ist es möglich, auch die Erzeugung der syntaktischen Struktur in der geforderten inkrementellen Art und Weise durchzuführen. Syntaktische Objekte besitzen dann die Fähigkeit, fehlende Information anzufordern, was dazu führen kann, daß die Bestimmung des Inhalts einer Äußerung durch syntaktische Restriktionen beeinflußt wird.

Bei der Konzeption der ILS–Ebene muß im wesentlichen darauf geachtet werden, daß

1. die DBS–Strukturen ungeordnet und zu unterschiedlichsten Zeitpunkten in der ILS–Ebene eintreffen,

2. die eintreffenden DBS–Strukturen syntaktisch korrekt linearisiert werden,

3. die durch POPEL–WHAT bestimmte Reihenfolge der konzeptuellen Einheiten jedoch möglichst beibehalten werden kann;

[6] In Abschnitt 5.1 gehe ich genauer auf das Verhältnis syntaktisches Objekt vs. Segment ein. Dies kann erst verständlich erklärt werden, wenn der zugrundeliegende Formalismus beschrieben ist.

4. bei der Lineariserung das bestehende Verhältnis zwischen der expliziten Phrasenstruktur und der linearen Ordnung berücksichtigt wird,

5. die Flexion korrekt durchgeführt wird und

6. schließlich, daß eine grammatikalisch korrekte Ausgabe produziert wird, was aufgrund der parallelen und inkrementellen Generierung nicht trivial ist.

Auf eine genauere Beschreibung der Probleme und ihrer Lösungen werde ich im Kapitel 6.2 eingehen.

Grundlage für den Generierungsprozeß auf beiden Ebenen ist eine syntaktische Grammatik, die folgende Eigenschaften erfüllen sollte:

1. Explizite Trennung zwischen Phrasenstruktur und linearer Abfolge,

2. explizite Darstellung der Phrasenstruktur,

3. lexikonbasierter Regelauswahlmechanismus,

4. im parallelen Modell verteilt einsetzbar,

5. mit Strukturen der funktionalsemantischen Beschreibungsebene direkt in Beziehung zu bringen,

6. deklarative Darstellung.

Die ersten vier Punkte sind eine Folge der Anforderungen, die an die Entwicklung der DBS– und ILS–Ebene gestellt sind. Die Beachtung von Punkt 5. ist wichtig, damit die syntaktischen Ebenen mühelos in die gesamte Verbalisierungskomponente integriert werden können. Die Berücksichtigung des letzten Punktes schließlich erlaubt es, das syntaktische Wissen zu erweitern oder zu ändern, möglichst ohne einen Einfluß auf das operationale Modell zu haben.

Chapter 3

Linguistische und formale Grundlagen der syntaktischen Beschreibungsebenen

3.1 Linguistische Grundlagen

Für die Einbindung linguistischen Wissens in POPEL–HOW (speziell von syntaktischem Wissen) ist man an einer Theorie interessiert, die die Sichtweise der parallelen und inkrementellen Generierung unterstützt. Wir haben uns für die Dependenztheorie als grundlegende linguistische Theorie entschieden. Die folgenden Punkte haben uns beeinflußt:

1. Die Theorie beschreibt die Beziehungen von sprachlichen Elementen derselben Segmentierungstufe. Dabei stehen die Beziehungen der lexikalischen Elemente im Vordergrund.

2. Die darauf aufbauenden mehrstelligen Relationen *Dependenz* und *Valenz* erlauben eine objektorientierte Sichtweise.

3. Dependenzbasierte Grammatiken machen eine strikte Trennung zwischen der strukturellen Beschreibung von Elementen und ihrer linearen Abfolge.

4. Gerade für ein Generierungssystem, das das Deutsche als Zielsprache hat, ist die Tatsache wichtig, daß sich die Dependenztheorie für die Beschreibung des Deutschen bewährt hat (vgl. z.B. [**?**]).

Aufgrund dieser Eigenschaften ist eine dependenzbasierte Grammatik für den parallelen und inkrementellen Generierungsprozeß in POPEL-HOW sehr gut geeignet. Es folgt nun eine Einführung in die Dependenztheorie sowie eine Charakterisierung der auf ihr basierenden Begriffe.

3.1.1 Die Dependenztheorie

Die Annahme, daß Sätze von natürlichen Sprachen eine innere Struktur besitzen, ist Grundlage für die strukturalistischen Syntaxtheorien, die seit Mitte dieses Jahrhunderts entwickelt werden. Das Erkennen der *satzimmanenten* Struktur ist wesentlich für das Verstehen von sprachlichen Ausdrücken, da daß Verstehen eines Satzes mehr voraussetzt als die bloße lineare Abfolge seiner Elemente, der Wörter [?]. Jedes Verstehen geht von einer Organisation des Satzes aus. Das unterschiedliche Zusammenfügen von Elementen eines Satzes korreliert mit einem differierenden Verstehen. Je nachdem, wie man die Einheiten in dem Satz "Maria sieht Peter mit dem Fernrohr auf dem Berg stehen." strukturell gliedert, kann man den Satz so interpretieren, daß entweder "Maria" oder "Peter" das Fernrohr benutzt.

Die Grammatikregeln einer Sprache – in unserem Fall des Deutschen – beschreiben, wie ein Satz strukturell organisiert werden kann[1].

Entscheidend für die Struktur eines Satzes ist die *Relation*, die die Strukturelemente ordnet. Grundlegende Relation in der Dependenztheorie, die erstmals von L. Tesnière 1959 als Methode zur formalen Beschreibung des Französischen entwickelt wurde [?], ist die *Konnexion*.

Die Konnexion beschreibt als syntaktische Relation die abstrakten Abhängigkeitsbeziehungen zwischen den Elementen eines Satzes, wobei diese Beschreibung unabhängig von der linearen Ordnung der Elemente ist. Anders als zum Beispiel die Syntaxtheorien des anglo–amerikanischen Strukturalismus, existiert in der Dependenztheorie seit jeher die Auffaßung einer strikten Trennung zwischen linearer Abfolge und satzimmanenter Struktur[2].

Ausgangspunkt für die Untersuchung des relationalen Zusammenhangs eines Satzes sind die Wörter des Satzes oder die lexikalischen Kategorien der Wörter. Für zwei Elemente eines Satzes legt die Konnexionsrelation fest [?],

1. ob das eine Elemente nur mit dem anderen auftreten darf,
2. beide Elemente stets gemeinsam auftreten müssen oder
3. sich gegenseitig ausschließen.

So können zum Beispiel die beiden Wörter "schneller" und "Vogel" gemeinsam in dem Ausdruck "ein schneller Vogel" vorkommen, aber nicht die Elemente "schnell" und "Vogel" in dem Ausdruck "*ein schnell Vogel". Die Konnexion beschreibt demnach das *geregelte Miteinandervorkommen* der Elemente eines Satzes. Die linguistischen Untersuchungsmethoden zur Bestimmung dieser Relation sind Eliminierungs – und Substitutionstests. Mit ihnen werden die syntagmatischen und paradigmatischen Verhältnisse der Elemente von Sätzen untersucht und bestimmt, welche Elemente welche anderen Elemente in ihrem Vorkommen restringieren[3].

Formal ist die Konnexion eine symmetrische und irreflexive Relation. Sie ist dann aber nur eine Verbindung von Elementen, die besagt, daß zwischen diesen Elementen eine

[1] Allerdings kann meiner Meinung nach die Darstellung der Satzstruktur und das Regelsystem in erster Linie nur ein Leistungsmodell darstellen. Syntaxmodelle sind stark von der subjektiven Einschätzung des Grammatikers geprägt und von dessen Interpretation der empirisch vorliegenden Fakten. Keine der existierenden und oft miteinander konkurriendes Theorien über die syntaktische Struktur sprachlicher Äußerungen darf als naturgegeben vorausgesetzt werden.

[2] Erst mit der Entwicklung von ID/LP–Grammatiken (vgl. Abschnitt 3.2.3) im Rahmen der GPSG [?] wird diese Trennung auch von anglo–amerikanischen Linguisten in ihren Syntaxtheorien berücksichtigt.

[3] Zum Status dieser syntaktisch–operationellen Tests vgl. [?].

Abhängigkeitsbeziehung besteht. Legt man zusätzlich fest, welches Element von welchem anderen Element abhängig ist, wird aus der symmetrischen Relation eine asymmetrische Relation. Diese Relation bezeichnet man als *Dependenz.*

Die Dependenzrelation läßt sich graphisch als gerichtete Kante repräsentieren. Die Knoten, die durch diese Kanten verbunden sind, charakterisieren die in Relation stehenden Elemente des Satzes [**?**]. Zum Beispiel läßt sich das Abhängigkeitsverhältnis der Elemente “bellt” und “Hund” in dem Ausdruck “Der Hund bellt.” graphisch wie folgt notieren:

$$\begin{tikzcd} \text{“bellt”} \arrow[d] \\ \text{“Hund”} \end{tikzcd}$$

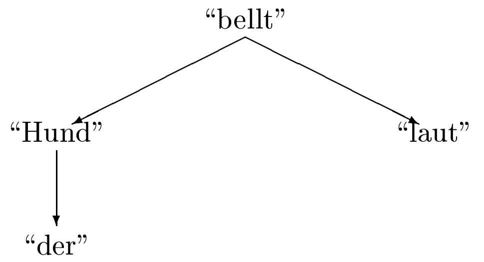
Man sagt dann auch, daß “Hund” von “bellt” *abhängt* oder, daß “bellt” das Element “Hund” *regiert* (*regiert* ist die inverse Relation zu *abhängt*). “bellt” wird als *Regent* von “Hund” bezeichnet und “Hund” als *Dependent* von “bellt”. Von “Hund” wird gesagt, daß es im *Regensbereich* von “bellt” liegt.

An dieser Stelle muß betont werden, daß die Ausrichtung der Dependenzrelation nicht durch “die Sprache” vorgegeben ist. Nach [**?**] ist es prinzipiell willkürlich, ob in der Nominalphrase “ein Ast” das Nomen den Artikel oder der Artikel das Nomen regieren soll. Es sind Kriterien der Zweckmäßigkeit, die ein Grammatiker bei der Festlegung der Abhängigkeitsverhältnisse hervorhebt. Allerdings muß er die Entscheidung im Rahmen seiner Theorie begründen können.

Das Ergebnis der Untersuchung eines Satzes nach dependentiellen Kriterien ist eine hierarchische Struktur der Abhängigkeitsbeziehungen der Elemente eines Satzes. Als formales Beschreibungsmittel dienen *Stemmata.* Ein Stemma ist eine baumartige Darstellung der Elemente eines Satzes, in der die Knoten gemäß der Dependenzrelation geordnet sind. Aufgrund der formalen Eigenschaften der Dependenzrelation DR gelten für ein Stemma folgende Eigenschaften:

1. Jedes Stemma ist ein gerichteter Graph, da DR asymmetrisch ist.
2. Jedes Stemma ist azyklisch, da DR irreflexiv ist.
3. Jedes Stemma ist verzweigend, weil jeder Knoten höchstens von einem Element regiert werden darf, selbst aber mehrere abhängige Elemente haben kann. (DR ist rechtseindeutig, da gilt: x_1, x_2, x_3 $\in$ Satzelement und $(x_1,x_2) \in DR$ und $(x_1,x_3) \in DR \Rightarrow x_2 = x_3$.)
4. Jedes Stemma ist zusammenhängend, d.h. für beliebige Elemente x1, x2 $\in$ Stemma und x1 $\neq$ x2 gibt es einen Weg, der x1 und x2 verbindet.
5. Wegen 1. – 4. ist ein Stemma insbesondere ein gerichteter Baum.

Der Satz "Der Hund bellt laut" läßt sich z.B. durch folgendes Stemma repräsentieren:



Der Wurzelknoten des Stemmas für einen Satz ist das alle anderen regierende Element des Satzes. Es ist somit das *strukturelle Zentrum*. In den Dependenzgrammatiken für das Deutsche wird das Hauptverb eines Satzes als das zentrale Element betrachtet[4]. Als zentrales Element legt es die Art und die Anzahl der übrigen Elemente fest, d.h. das Verb eröffnet den strukturellen Rahmen eines Satzes.

3.1.2 Der Valenzbegriff

Die Fähigkeit des Verbes, Leerstellen zu eröffnen wird innerhalb der Dependenztheorie als *Valenz* bezeichnet. Bei der Untersuchung der Valenz der Verben unterscheidet man zwischen verschiedenen Ebenen der Valenz, nämlich: logische, semantische und syntaktische Valenz [?].

Die syntaktische Valenz[5] eines Verbes legt fest, wieviele Elemente vom Verb direkt abhängen und welche syntaktischen Funktionen diese Elemente in der Äußerung einnehmen. Bezüglich der Modalität der Leerstellen eines Verbes unterscheidet man zwischen *notwendigen* (valenzgebundenen) und *nicht notwendigen* (valenzungebundenen) Leerstellen. Die notwendigen Leerstellen werden als *Ergänzungen*, die nicht notwendigen als *Angaben* bezeichnet.

Charakteristisch für Ergänzungen ist ihre Eigenschaft, nur bei bestimmten Valenzträgern aufzutreten, d.h. sie sind subklassenspezifisch in ihrem Vorkommen restringiert. Es werden daher die Verben als Valenzträger aufgrund der Art der Ergänzungen in syntaktischer Hinsicht subkategorisiert. So gibt es einwertige Verben, die als Belegung für ihre Leerstelle Ergänzungen im Nominativ benötigen (z.B. "fiebern") oder dreiwertige Verben, deren Leerstellen jeweils mit Ergänzungen im Nominativ, Akkusativ und Genitiv belegt werden müssen (wie z.B. "beschuldigen") (s.a. Abschnitt 3.1.3).

Die Angaben sind im Unterschied zu den Ergänzungen nicht subklassenspezifisch restringiert. Sie können syntaktisch bei allen Verben vorkommen[6]. Das heißt aber auch, daß Angaben die Leerstellen eines Valenzträgers nicht besetzen. Da sie sich nicht auf die Valenz von Verben beziehen, können sie syntaktisch beliebig hinzugefügt oder weggelassen

[4] Dies gilt übrigens auch für Grammatiken anderer Syntaxtheorien, wie z.B. GPSG oder LFG. Viele Linguisten lassen auch Nomen und Adjektive als strukturellen Mittelpunkt zu [?].

[5] Im weiteren wird die syntaktische Valenz auch kurz als Valenz bezeichnet, falls es kontextuell eindeutig ist.

[6] Vorausgesetzt, daß das Verb und die Angaben semantisch widerspruchsfrei sind [?].

werden. Zu den Angaben zählt man u.a. kausale, konditionale und lokale Ausdrücke (s.a. Abschnitt 3.1.3).

Für das, was mit einer Äußerung gemeint ist, spielen sie eine genauso große Rolle wie die Ergänzungen [?]. Oft sind sie die kommunikativ wichtigsten Teile, da mit ihnen über den eigentlichen Sachverhalt einer Äußerung hinaus das Verhältnis des Sprechers zu seiner Äußerung festgestellt werden kann.

Ergänzungen werden zusätzlich unterschieden in *obligatorische* und *fakultative* Ergänzungen. Obligatorische Ergänzungen sind notwendig, um grammatikalische Wohlgeformtheit eines Satzes zu garantieren. Fakultative Ergänzungen sind dagegen weglaßbar, ohne daß syntaktisch unkorrekte Formen entstünden. Das dreiwertige Verb "beschuldigen" muß mit einer Ergänzung im Nominativ (dem Subjekt) und Genitiv notwendig belegt werden. Die Ergänzung mit Kasus Akkusativ darf dagegen weggelassen werden. Die Äußerungen "Ich beschuldige ihn der Tat." und "Ich beschuldige ihn." sind daher beide syntaktisch korrekt, nicht aber "Ich beschuldige.".

Diese Charakterisierung der Ergänzungen gilt für Sätze, die isoliert von einem Text oder Dialog untersucht werden. Innerhalb eines Dialoges ist es möglich, daß selbst obligatorische Ergänzungen wegfallen dürfen. Es muß aber für den Addressaten solcher elliptischen Äußerungen möglich sein, aufgrund des Kontextes (oder seines Weltwissens) die fehlenden Teile zu ergänzen, um das, was mit der Äußerung gemeint ist, zu rekonstruieren. Der Ausdruck "fährt nach Hause" ist syntaktisch nicht korrekt, da das Subjekt nicht spezifiziert ist. Ist dieser Ausdruck aber als Antwort auf die Frage "Was macht Peter?" geäußert worden, so kann "Peter" als das fehlende Subjekt bestimmt werden.

3.1.3 Die Beschreibung komplexer Strukturen

Für die Untersuchung komplexer Strukturen ist es hilfreich, nicht immer die gesamte Struktur mit den Wortklassenelementen zu beschreiben, wie es oben getan wurde, sondern Teilstrukturen als Einheiten zu betrachten.

Will man z.B. bei einer ersten Analyse des Stemmas von "Der Hund bellt laut." (s.S. 34) nur die unmittelbaren Dependenten von "bellt" untersuchen, so wird der Teilbaum mit Wurzel "Hund" als Einheit betrachtet. Es ist dann möglich, dieser Teilstruktur in ihrer Gesamtheit bestimmte grammatikalische Merkmale zuzuweisen und zu untersuchen, welche Eigenschaften für sie aufgrund ihres Abhängigkeitsverhältnisses zum regierenden Element bestehen. Andererseits ist es möglich, die Teilstruktur von der Gesamtstruktur isoliert zu betrachten und zu untersuchen, welche grammatikalischen Eigenschaften diese Teilstruktur unabhängig von konkreten Abhängigkeitsbeziehungen besitzt. Dadurch wird es möglich zu bestimmen, wie sich die Eigenschaften dieser Struktur ändern, wenn sie als Dependent unterschiedlichen Elementen untergeordnet wird.

Engel[7] definiert Strukturen von Sätzen, die unabhängig von konkreten Abhängigkeitsbeziehungen betrachtet werden, als *Phrasen.* Das Wurzelelement bezeichnet Engel als den *Nukleus* der Phrase, von dem alle anderen Elemente der Phrase unmittelbar oder mittelbar abhängen. Die lexikalische Kategorie des Wurzelelements definiert, um welche Phrase es sich handelt. Da das Wurzelelement die wesentlichen Eigenschaften einer Phrase

[7]In diesem Abschnitt beziehe ich mich bei der Erklärung und Definition von Termini im wesentlichen auf [?].

festlegt, kann die gesamte Phrase auch als *Projektion* des Wurzelelements bezeichnet werden (vgl.a. Abschnitt 4.1.2). Die Phrase "der Hund" ist eine Nominalphrase NP, da "Hund" als lexikalische Kategorie Nomen besitzt. Das Nomen ist das zentrale Element der NP, die NP daher eine Projektion des Nomens. Der Ausdruck "auf der Lauer" ist eine Präpositionalphrase, da das regierende Element des Ausdrucks die Präposition "auf" ist.

Das regierende Element einer Phrase eröffnet einen Valenzrahmen. Dies bedeutet insbesondere, daß z.B. auch Nomen oder Adjektive, wenn sie als Nukleus in Phrasen auftreten, abhängige Elemente besitzen, die als Ergänzungen oder Angaben charakterisiert werden können. Das zentrale Element des Satzes ist weiterhin das Verb. Da das gesamte Stemma eines Satzes ebenfalls als Phrase aufgefaßt werden kann, wird der Satz als *Verbalphrase* definiert. Als zentrales Element definiert das Verb die wesentlichen Eigenschaften eines Satzes. Ein Satz ist danach die Projektion seines Verbes.

Phrasen, die als Dependenten im Regensbereich eines Elementes auftreten, definiert Engel als *Glieder*. Ihre Funktion innerhalb der Gesamtstruktur wird vom Regenten festgelegt. Die NP "Die Studenten" hat im Satz "Die Studenten sind strebsam." Subjektfunktion und in "Der Professor bedauert die Studenten." Objektfunktion.

Die Eigenschaft von Wörtern, andere Elemente als Glieder zu regieren, nennt Engel *Rektion*. Er beschränkt dabei den Begriff nicht nur auf die Eigenschaft regierender Elemente, den Kasus der Dependenten festzulegen, wie dies üblicherweise in der Linguistik getan wird (vgl. z.B. [**?**]). Damit spricht Engel jeder Wortklasse die Eigenschaft der Rektion zu. Je nach Wortklasse werden die abhängigen Elemente unterschiedlich bezeichnet. Glieder von Nomen heißen *Attribute*; Glieder von Verben werden von Engel *Satzglieder* genannt.

Ergänzungen von Verben, also die sublassenspezifischen Satzglieder bzw. Satzergänzungen, spielen gerade für die Beschreibung des Deutschen eine zentrale Rolle. Engel unterteilt die möglichen Satzergänzungen in zehn Unterklassen und bezeichnet sie mit e_0, e_1, e_2 usw. e_0 kennzeichnet z.B. die *Nominativergänzung*. Sie entspricht dem traditionellen Subjekt. e_1, e_2 und e_3 entsprechen den Objekten im Akkusativ, Genitiv und Dativ. e_5 kennzeichnet Adverbialobjekte und e_6 Richtungsobjekte[8].

Die Angaben von Verben und Satzangaben werden von Engel nach semantischen Kriterien in sechs Klassen unterteilt: *Situativa* (u.a. kausale, temporale und lokale Ausdrücke), *Existimatora* (spiegeln die persönliche Ansicht des Sprechers zum verbalisierten Ausdruck wider, wie z.B. "allerdings" , "immerhin" oder "tatsächlich"), *Valuativa* (enthalten wertende Angaben wie "bald" und "fast"), *Modificativa*, *Negationsangaben* und *adjungierte Adverbia* (z.B. "auch", "eben", "aber").

Die Verben lassen sich aufgrund der Anzahl ihrer Leerstellen unterteilen in *null–*, *ein–*, *zwei–*, *drei–*, und *vierwertige* Subklassen (diese Unterteilung ist einzelsprachlich abhängig). Berücksichtigt man zusätzlich noch die Qualität der Leerstellen, d.h. die verschiedenen Kombinationsmöglichkeiten der Satzergänzungen, lassen sich die Subklassen weiter unterteilen in Klassen von Verben mit gleicher Anzahl und Qualität der Leerstellen. Diese Klassen nennt Engel *Satzmuster*. Beispiele für zweiwertige Satzmuster sind $v(e_0, e_1)$ ("Maria kauft das Haus.") oder $v(e_0, e_6)$ ("Wir fahren nach Wien.").

In den Satzmustern wird nicht berücksichtigt, ob die Satzergänzungen obligatorisch oder fakultativ sind. Unterscheidet man die fakultativen Satzergänzungen z.B. durch

[8]Für eine ausführliche Beschreibung der Ergänzungen vgl. [?], Abschn. 5.4.

Klammerung von den obligatorischen, so erhält man speziellere Satzmuster. Engel nennt diese Satzmuster *Satzbaupläne.* Satzbaupläne sind Abstraktionen von der Valenz einzelner Verben. Dies bedeutet einerseits, daß mit jedem Verb ein ganz bestimmter Satzbauplan assoziiert ist (und mehrere, falls das Verb mehrere Valenzrahmen besitzt), andererseits sind Satzbaupläne abstrakte nichtlineare Strukturmodelle für Sätze, die u.a. keine Aussagen machen über: Satzgliedfolge, Satzart, Erweiterung durch Angaben und Tempus.

3.1.4 Die Beschreibung der linearen Abfolge

In dependenzbasierten Grammatiken wurde bereits sehr früh eine strikte Trennung zwischen satzimmanenter Struktur und linearer Reihenfolge vollzogen. Dabei wird davon ausgegangen, daß eine regelmäßige Beziehung zwischen der Dependenzstruktur und der linearen Struktur besteht, die oft mit Hilfe der Stemmata formal erschlossen wird (vgl. z.B. [?] und Abschnitt 4.1.4.1).

Engel beschreibt recht umfangreich die linearen Reihenfolgebeziehungen der strukturellen Elemente im Deutschen. Im Vordergrund stehen zwar die Folgebeziehungen der Satzglieder, es werden aber auch Abfolgerelationen anderer Phrasen, wie z.B. der NP, aufgeführt und erläutert.

Ausgangspunkt für die Untersuchung der linearen Struktur deutscher Sätze ist wiederum das Verb und dessen Eigenschaft, den strukturellen Rahmen von Sätzen zu bestimmen. Die verbalen Elemente eines Satzes (also finiter und infiniter Teil) bilden den Satzrahmen bzw. die Satzklammern. Der Satzrahmen teilt den Satz in *drei* Felder auf: in *Vorfeld*, *Mittelfeld* und *Nachfeld.* In dem Satz "Peter ist mit Maria nach Hause gefahren", bildet "Peter" das Vorfeld, der finite Teil "ist" des Verbalkomplexes "ist gefahren" die linke Satzklammer, "mit Maria nach Hause" das Mittelfeld und der infinite Teil des Verbalkomplexes "gefahren" die rechte Satzklammer. Das Nachfeld, das üblicherweise auf den infiniten Teil der verbalen Elemente folgt, ist in diesem Beispielsatz unbesetzt. Die zentralen Fragen für die Beschreibung der Felder sind:

- Welche Elemente dürfen in bestimmten Feldern auftreten und welche nicht?
- Welche Grundfolge der Elemente eines Feldes gibt es?
- Welche Abweichungen von der Grundfolge sind erlaubt und was sind die Kriterien dafür?

Im Vorfeld kann genau ein Satzglied auftreten. Dies gilt aber nur für *Aussagesätze* und *Ergänzungsfragen*, in denen ein Element im Satz erfragt wird:

> "Peter gehört ins Bett."
> "Wer gehört ins Bett?"

Unbesetzt bleibt das Vorfeld in *Imperativsätzen* und *alternativen* Fragesätze, auf die mit "ja" oder "nein" geantwortet wird und in eingeleiteten Nebensätzen.

In der Regel befindet sich das Subjekt des Satzes im Vorfeld. Es existieren aber für die Belegung des Vorfeldes kaum Beschränkungen, so daß fast alle Ergänzungen als vorfeldfähig charakterisiert werden können.

Wie bereits im Beispiel gezeigt, darf das Nachfeld unbesetzt bleiben, ohne daß ungrammatikalische Ausdrücke die Folge wären. Auf jeden Fall im Nachfeld müssen mit Konjunktionen verbundene Neben– oder Hauptsätze stehen [**?**]:

"Peter hat derart gelogen, daß sich die Balken bogen."

Im Mittelfeld können alle Satzglieder bis auf die eben genannten Nebensatzformen stehen. Insbesondere können hier auch Relativsätze von Satzgliedern aufgenommen werden, da sie nicht unmittelbar vom Verb regiert werden. Im Unterschied zum Vor– und Nachfeld kann das Mittelfeld aus mehreren Elementen bestehen. Charakteristisch für das Deutsche ist seine relative Freiheit bezüglich der Wortstellung. Daher gibt es nur sehr wenige syntaktische Einschränkungen für die Anordnung der Mittelfeldelemente. Die wichtigste ist, daß Pronomen sehr nahe am finiten Verbteil stehen müssen. Als Kriterium für die Reihenfolge der Elemente im Mittelfeld dient primär die Thema–Rhema–Gliederung oder die kommunikative Dynamik. Als Thema bezeichnet man das aus dem vorangegangen sprachlichen Kontext Bekannte, das Rhema bezeichnet dagegen die vorher noch nicht erwähnte neue Information. In POPEL wird die kommunikative Dynamik von der inhaltsfestlegenden Komponente POPEL–WHAT bestimmt. Die Aufgabe von POPEL–HOW beschränkt sich darauf, die syntaktischen Restriktionen zu beachten.

3.2 Die formale Repräsentation linguistischen Wissens

Um linguistische Grammatiktheorien und das damit verknüpfte Wissen effektiv in einem natürlichsprachlichen System einsetzen zu können, werden Formalismen benötigt, die – quasi als formale Schnittstelle – für die Umsetzung des Wissens eingesetzt werden können. Ähnlich wie mit Hilfe von Programmiersprachen kann der Linguist mit einem solchen Formalismus von der internen Darstellung im Computer abstrahieren und sich vollständig auf die Erstellung seiner Grammatik konzentrieren.

Die Kriterien für eine formale Sprache zur Darstellung grammatikalischen Wissens sind vergleichbar mit denen für Programmiersprachen [**?**]: *einfach zu verstehen, deklarativ, mathematisch fundiert* und *hinreichend ausdrucksstark*. Zusätzlich sollte diese Sprache auch *neutral* sein bezüglich konkreter Applikationen, damit das mit ihm codierte Wissen für die Analyse und Generierung natürlicher Sprache verwendet werden kann.

3.2.1 Der PATR–II–Formalismus

Eine solche "Programmiersprache für Linguisten" stellt der am SRI entwickelte PATR–II–Formalismus dar [**?**][9].

Ausgangspunkt für die Formalisierung in PATR ist die Darstellung sprachlicher Strukturen der syntaktischen und lexikalischen Beschreibungsebene als *gerichtete azyklische Graphen* (Directed Acyclic Graphs DAGs). DAGs repräsentieren grammatikalische Merkmale (wie z.B. *tempus*) und deren Werte (z.B. *präsens*) und die Beziehungen von Merkmalen zueinander. DAGs lassen sich wie folgt definieren:

[9]Im weiteren wird für die Bezeichnung PATR–II auch PATR verwendet.

Definition 1
Sei ATT eine endliche Menge von Merkmalen. Ein DAG ist entweder

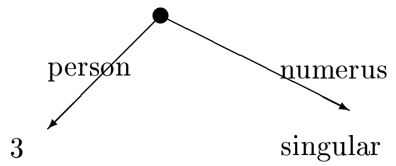
- ein atomares Merkmal $l \in ATT$, oder
- eine – möglicherweise leere – Menge S von Paaren $< l, v >$, wobei $l \in ATT$, v (der Wert eines Merkmales) ein DAG ist und es gilt, daß S sich nicht *überdecken* darf.

(*Überdeckung* läßt sich rekursiv wie folgt definieren (vgl. [**?**]):
$\forall < l, v > \in S$: S überdeckt v und S überdeckt alles, was von v überdeckt wird. l überdeckt sich selber.)

DAGs, die nur aus einem einzigen Wert bestehen, heißen *atomar*, andernfalls *komplex*. Der DAG, der keinen Wert hat, wird als leerer DAG bezeichnet. Komplexe DAGs beschreiben partielle Funktionen mit den Merkmalen als Definitionsbereich und ihren Werten als Bildbereich. Für den Bildbereich eines Merkmals l schreibt man auch $dom(l)$. Der Bildbereich eines komplexen DAGs D wird ebenfalls als $dom(D)$ notiert, wobei der Wert die Menge aller seiner (unmittelbaren) Merkmale ist, also wieder ein DAG.

Mögliche Darstellungsweisen für DAGs sind Graphen oder Mengen von Merkmal/Werte Paaren. Beide Darstellungsweisen sind äquivalent. Zum Beispiel läßt sich der DAG mit den Merkmalen *numerus* mit Wert *singular* und *person* mit Wert *3* als Menge und als Graph wie folgt darstellen[10]:

$$\begin{bmatrix} person: & 3 \\ numerus: & singular \end{bmatrix}$$



Dieser DAG kann selbst in einem anderen DAG eingebettet sein. Er wird dann als *Teil*–DAG bezeichnet:

$$\begin{bmatrix} cat: & np \\ & \\ kongruenz: & \begin{bmatrix} person: & 3 \\ numerus: & singular \\ kasus: & [\,] \end{bmatrix} \end{bmatrix}$$

Um in diesem komplexen DAG den Wert des Merkmals *numerus* zu erhalten, muß ein *Pfad* spezifiziert werden, in dem angegeben ist, über welche Merkmale der Wert von *numerus* erreichbar ist; z.B. bedeutet der Pfad <*kongruenz numerus*>, daß zuerst der Wert des Merkmals *kongruenz* bestimmt werden muß und dann von diesem Wert der Wert

[10] In den nächsten Beispielen werde ich bei der Darstellung von DAGs nur noch die Mengenschreibweise verwenden.

des Merkmals *numerus*. Dies läßt sich auch notieren als $dom(< kongruenz\ numerus >)$. Der Wert [ ] des Merkmals *kasus* bezeichnet den leeren DAG. Er fungiert als Variable und kann als Wert einen beliebigen DAG aufnehmen.

Es ist möglich, daß sich bestimmte Merkmale einen gemeinsamen Wert teilen (dies wird auch als *structure sharing* bezeichnet):

$$\begin{bmatrix} 1: & \begin{bmatrix} cat: & np \\ kongruenz: & \begin{bmatrix} person: & 3 \\ numerus: & singular \end{bmatrix} \end{bmatrix} \\ 2: & \begin{bmatrix} cat: & VP \\ kongruenz: & < 1\ kongruenz > \end{bmatrix} \end{bmatrix}$$

In diesem DAG haben die Teil–DAGs von 1 und 2 jeweils ein Merkmal *kongruenz*, das denselben DAG als Wert hat. Dargestellt ist dies durch Angabe eines Pfades unter dem Merkmal *kongruenz* im DAG 2. Der Pfad repräsentiert eine gerichtete Kante auf den entsprechenden Werte–DAG.

DAGs werden in PATR verwendet, um die explizite morphosyntaktische Struktur von Phrasen und Lexemen zu repräsentieren. Wie sich welche Teilstrukturen unter welchen Bedingungen zu größeren Strukturen zusammenfassen lassen, ist in *Regeln* formuliert. In ihrer Gesamtheit beschreiben die Regeln, wie die Sätze einer Sprache strukturiert sind. Dabei sollte es im Prinzip für das Aufstellen der Regeln unerheblich sein, ob die Satzstruktur analysiert oder generiert wird. Regeln im PATR–Formalismus bestehen aus zwei Teilen:

1. einem kontextfreien Teil

2. einer Menge von Spezifikationen

Der kontextfreie Teil beschreibt die Konstituentenstruktur einer Phrase. Er drückt aus, wie Konstituenten zu neuen, größeren Konstituenten aufgebaut werden können bzw. wie Konstituenten in kleinere Konstituenten zerlegt werden können. Zum Beispiel beschreibt der kontextfreie Teil $s \longrightarrow np1\ vp\ np2$ der Regel

$$\begin{array}{l} s \longrightarrow np1\ vp\ np2 \\ < np1\ person > = < vp\ person > \\ < np1\ person > =\ 3 \\ < np1\ kasus > =\ nom \\ < np2\ kasus > =\ akk \end{array}$$

daß die Nominalphrasen $np1$ und $np2$ zusammen mit einer Verbalphrase vp einen Satz s konstituieren. Der kontextfreie Teil legt auch eine lineare Ordnung der Teilkonstituenten, nämlich $np1 < vp < np2$.

Der Spezifikationsteil bezieht sich auf die DAGs, die mit den Konstituenten assoziiert sind. Mit ihm wird der Bildbereich bestimmter Merkmale eingeschränkt. Allerdings wird nicht verlangt, daß die DAGs diese Merkmale auch tatsächlich haben müssen.

Eine einzelne Spezifikation ist einerseits als Gleichung zwischen Merkmalen von zwei Konstituenten aufzufassen. Eine Gleichung schreibt vor, welche Merkmale der Konstituenten identische Werte haben müssen. Gleichungen im Spezifikationsteil einer Regel sind also *Wert–vergleichend*. Beispielsweise besagt die Spezifikation

$$< np1\ person > = < vp\ person >$$

in obiger Regel, daß der Wert von *person* für $np1$ und vp identisch sein muß, unabhängig davon, wie der aktuelle Wert lautet und wo er definiert wird. Wert–vergleichende Spezifikationen werden u.a. dann formuliert, wenn *Kongruenzphänomene* beschrieben werden sollen. Es ist hierbei möglich, in den Gleichungen durch Angabe von Pfaden anzugeben, über welche anderen Merkmale die zu überprüfenden Merkmale zu erreichen sein sollen.

Mit einer Spezifikation kann aber auch ausgedrückt werden, daß ein Merkmal einer Konstituenten einen bestimmten Wert haben muß. Solche Spezifikationen sind dann *Wert–identifizierend*. In der Beispielregel ist die Spezifikation

$$< np1\ kasus > = \ nom$$

Wert–identifizierend. Damit wird für die Anwendbarkeit der Regel festgelegt, daß die entsprechende Nominalphrase $np1$ den Kasus im Nominativ haben muß.

Die Basisoperation zur Überprüfung der Korrektheit der Beziehungen zwischen Konstituenten und zur Komposition von Konstituenten in PATR ist die *Unifikation*:

Definition 2 Seien d_1, d_2 DAGs. Für die Unifikation d gilt:

- $d = d_1$, wenn $d_1 = d_2$,
- $d = d_1$, wenn d_1 atomar und $d_2 = [\,]$,
- $d = d_2$, wenn d_2 atomar und $d_1 = [\,]$,
- wenn weder d_1 noch d_2 atomar, dann gilt:

$$\forall\ l : < l, v_1 > \in d_1 \text{ und } < l, v_2 > \in d_2 \Longrightarrow$$
$$< l, unifikation(v_1, v_2) > \in d,$$
$$\forall\ l : < l, v > \in (d_1 \cup d_2) - (d_1 \cap d_2) \Longrightarrow$$
$$< l, v > \in d$$

Zwei DAGs d_1 und d_2 unifizieren miteinander, wenn es möglich ist, beide Graphen von den Wurzelknoten ausgehend "zur Deckung" zu bringen. Es ist nach diesem Prozeß nicht mehr möglich, einen der DAGs zu identifizieren. Die Unifikation wird daher auch als *destruktive* Operation bezeichnet. Das Ergebnis der Unifikation ist ein neuer DAG d, der durch Vereinigung von d_1 und d_2 entsteht. d umfaßt auch die Merkmale, die z.B. nur in d_1 vorkamen. Beispielsweise ergibt die Unifikation der DAGs

$$\begin{bmatrix} subjekt: & < kongruenz > \\ kongruenz: & \begin{bmatrix} person: & 3 \end{bmatrix} \end{bmatrix}$$

und

$$\left[\ subjekt: \ \left[\ numerus: \ \ singular\ \right]\ \right]$$

den DAG

$$\left[\begin{array}{ll} subjekt: & < kongruenz > \\ & \\ kongruenz: & \left[\begin{array}{ll} person: & 3 \\ numerus: & singular \end{array}\right] \end{array}\right]$$

Da das Ergebnis der Unifikation von DAGs durch deren Vereinigung entsteht, ist es möglich, Teilstrukturen zu größeren Strukturen zusammenzufassen (wie dies für den Ergebnis–DAG im obigen Beispiel der Fall ist). Diese *monotone* Eigenschaft ist charakteristisch für die Unifikation auf Graphen. Sie ist z.B. in der Termunifikation – die in vielen Gebieten der KI und Informatik eingesetzt wird – nicht gegeben, da für Terme gilt, daß sie vollständig sein müssen [**?**]. Es ist mit der Termunifikation von zwei Termen t_1 und t_2 nicht möglich, einen Term zu erzeugen, der neben der gemeinsamen Information von t_1 und t_2 auch die Information besitzt, die z.B. nur t_1 trägt.

Die Unifikation der Spezifikationen einer Regel erlaubt es, während der Untersuchung eines Satzes Information über Merkmale im Ableitungsbaum fließen zu lassen. Allerdings ist die Unifikation neutral bezüglich der Flußrichtung, d.h. es ist im Ergebnis–DAG nicht mehr nachvollziehbar, woher die Information kam. Ist es erforderlich, die Richtung des Flusses von grammatikalischer Information zu bestimmen[11], muß man für einen Teil der Attribute eine besondere Semantik definieren, mit der dies möglich wird (in POPEL–GRAM, der Grammatik von POPEL–HOW (s. Abschnitt 4.1), wird dies z.B. durch die spezielle Interpretation des Attributes *synchronize* geleistet).

Für den Anwendungsbereich einer Grammatik ist die Tatsache wichtig, daß die Unifikationsoperation *assoziativ* und *kommunikativ* ist. Aussagen in unifikationsbasierten Grammatikformalismen sind aufgrund dieser Tatsache *ordnungsinvariant* sowie *bidirektional* [**?**]. Daher ist es prinzipiell möglich mit PATR *bidirektionale Grammatiken* zu notieren, also Grammatiken, die für die Analyse und Generierung in einem NSS einsetzbar sind (vgl. [**?**]). Wenn man aber Operationen in Abhängigkeit von der konkreten Applikation hinzufügt oder Merkmale einführt, die ein *nichtmonotones* Verhalten verursachen [**?**], dann ist es möglich, daß die Bidirektionalität der Unifikation keine Rolle mehr spielt und damit auch die Grammatik diese Eigenschaft verliert.

3.2.2 Eine Entwicklungsumgebung für PATR–basierte Grammatiken

Basierend auf dem PATR–Formalismus wurde eine Reihe von Entwicklungsumgebungen (*Tools*) für unifikationsbasierte Grammatiken entwickelt (u.a. Z–PATR [**?**], D–PATR [**?**]).

Diese Tools erlauben es, in einer dem Linguisten vertrauten Form Regeln und Lexikonelemente zu schreiben, die von entsprechenden Routinen in die DAG–Strukturen

[11] Vgl. hierzu [**?**], der die Untersuchung des Flusses von grammatikalischer Information als aktuellen Forschungsgegenstand bezeichnet.

von PATR übersetzt werden. Weiterhin bieten sie eine Fülle von Editier– und Trace–Möglichkeiten für die Überprüfung der Korrektheit der formulierten Regeln und lexikalischen Elemente.

Gemeinsam ist allen Tools, daß sie dem Linguisten für die Grammatiken einen Parser zur Verfügung stellen. Dieser Parser kann in einem NLS zur syntaktischen Analyse verwendet werden oder als Verifikationsmittel dienen, um die Korrektheit der zu entwickelnden Grammatik zu testen. Stellvertretend für diese Tools wird das System SB–PATR [?], eine Reimplementation von D–PATR, vorgestellt. Mit diesem Tool ist die Analysegrammatik von XTRA entwickelt worden. Der in SB–PATR integrierte unifikationsbasierte Chart–Parser führt die syntaktische Analyse durch. SB–PATR wird ebenfalls im Generierungssystem POPEL–HOW eingesetzt. Allerdings sind anstelle des Parsers für die Erzeugung syntaktischer Strukturen entsprechende Operationen im Rahmen dieser Arbeit entwickelt worden (s. Kapitel 5). Alle anderen Module (u.a. das Unifikationsmodul, der Regelübersetzer) werden in beiden Richtungen gleichermaßen eingesetzt.

3.2.2.1 Regeln in SB–PATR

Regeln in SB–PATR werden als Listenausdrücke wie folgt definiert:

Definition 3
Eine Regel in SB–PATR ist eine Liste bestehend aus einem kontextfreien Teil kfT und einem Spezifikationsteil Spt:

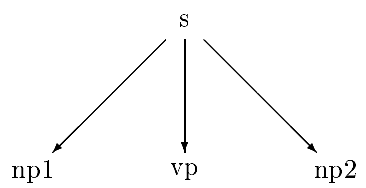
(kfT Spt)

Definition 4
Der kontextfreie Teil kfT beteht aus einer Folge von n Konstituenten mit n > 1. Der Folge der Konstituenten wird implizit von links nachs rechts beginnend mit 0 ein eindeutiger numerischer Index zugewiesen.

Der kontextfreie Teil einer Regel läßt sich graphisch als gerichteter n–ärer Baum mit Tiefe 1 repräsentieren. Das Element mit Index 0 bildet die Wurzel. Alle anderen Elemente des kfT sind Töchter dieses Elementes. Zum Beispiel hat in der Regel

(s $np1$ vp $np2$)

die Vaterkonstituente s den Index 0, die Tochterkonstituente $np1$ den Index 1, vp den Index 2 und $np2$ Index 3. Die graphische Darstellung als n-ären Baum ist:



Der Spezifikationsteil *Spt* besteht aus einer Folge von *Spezifikationen*. Eine Spezifikation ist eine 2–elementige Liste. *Spt* wird formal wie folgt definiert:

Definition 5

Spt ::= { spec }*
spec::=({ attr | path} {path | value})

attr ist ein atomares Merkmal, *path* eine Liste und *value* entweder ein atomares Merkmal, eine Liste von Spezifikationen oder eine Abkürzung für solch eine Liste (der abkürzende Name wird mit @ präfigiert).

Es ergeben sich vier verschiedene Arten von Spezifikationen: (*attr value*), (*path value*), (*attr path*) und (*path path*). Die ersten beiden Ausprägungen sind *Wert–identifizierend*, die letzten beiden *Wert–vergleichend*.

Da sich die Spezifikationen auf die Konstituenten des kontextfreien Teils beziehen sollen, wird in jeder Spezifikation Gebrauch von der impliziten Indizierung der Konstituenten gemacht. Jeder Pfad *path* erhält als erstes Element seiner Liste den Index der Konstituente, auf deren DAG sich die Pfadangabe bezieht. Analog erhält jedes *attr* einer *spec* als Wert einen Index. Ist das atomare Symbol eines *val* numerisch, so ist damit kein Index einer Konstituente assoziiert.

Der Spezifikationsteil *Spt* läßt sich graphisch als DAG repräsentieren. Der DAG des Spezifikationsteils der Regel

```
(s  np1 vp np2
    (0 (2))
    ((2 agree person) (1 agree person))
    ((2 agree person) third))
    ((2 case) nom))
    ((3 case) acc))
```

sieht z.B. so aus:

$$\begin{bmatrix} 0: & <2> \\ & \\ 1: & \begin{bmatrix} agree: & \begin{bmatrix} person: & <2\ agree\ person> \end{bmatrix} \end{bmatrix} \\ & \\ 2: & \begin{bmatrix} case: & nom \\ agree: & \begin{bmatrix} person: & third \end{bmatrix} \end{bmatrix} \\ 3: & \begin{bmatrix} case: & acc \end{bmatrix} \end{bmatrix}$$

Da sich die Spezifikationen einer SB–PATR Regel mit einem eindeutigen Index auf die Elemente von kfT beziehen, läßt sich jede Regel ebenfalls als DAG repräsentieren. Jede Konstituente des kfT wird im Regel–DAG durch seinen Index repräsentiert. Der Name der Konstituente wird als Wert dem reservierten Attribut *cat* zugewiesen. Der DAG für obige Regel sieht dann wie folgt aus:

$$\begin{bmatrix}
0: & \begin{bmatrix} \mathit{cat}: & s \\ \mathit{fset}: & < 2\ \mathit{fset} > \end{bmatrix} \\
1: & \begin{bmatrix} \mathit{cat}: & \mathit{np}1 \\ \mathit{fset}: & \begin{bmatrix} \mathit{case}: & \mathit{nom} \\ \mathit{agree}: & \begin{bmatrix} \mathit{person}: & < 2\ \mathit{fset}\ \mathit{agree}\ \mathit{person} > \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
2: & \begin{bmatrix} \mathit{cat}: & \mathit{vp} \\ \mathit{fset}: & \begin{bmatrix} \mathit{agree}: & \begin{bmatrix} \mathit{person}: & \mathit{third} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
3: & \begin{bmatrix} \mathit{cat}: & \mathit{np}2 \\ \mathit{fset}: & \begin{bmatrix} \mathit{case}: & \mathit{acc} \end{bmatrix} \end{bmatrix}
\end{bmatrix}$$

Der Teil–DAG unter dem Merkmal 0 repräsentiert den DAG der Vaterkonstituenten. Das Merkmal *cat* nennt den Namen dieses Teil–DAGs, der DAG unter dem Merkmal $fset$ den Spezifikationsteil. Der DAG der Vaterkonstituenten wird im folgenden auch als *Vater–DAG* bezeichnet. Analoges gilt für die Teil–DAGs unter den Merkmalen 1 und 2, die die Tochterkonstituenten repräsentieren. Sie werden im folgenden auch als die *Tochter–DAGs* einer Regel bezeichnet.

In SB–PATR wird jede Regel in einen entsprechenden DAG übersetzt. Das Merkmal $fset$ wird dabei automatisch erzeugt und braucht daher bei der Definition der Regel nicht aufgeführt zu werden. Das Merkmal ist notwendig, damit bei der Unifikation von Regel–DAGs die reservierten (wie z.B. *cat*) von den anwenderdefinierten Merkmalen unterschieden werden können.

3.2.2.2 Lexikoneinträge und Templates

Die lexikalischen Elemente werden ebenfalls mit Hilfe einer Listennotation vom Grammatikentwickler formuliert und von SB–PATR in DAGs übersetzt.

Ein Lexikoneintrag in SB–PATR hat folgende Listenstruktur:

Lexikoneintrag::= (Lexem $\{$Teileintrag$\}^{+}$)
Teileintrag ::=(Kategorie $\{$Spezifikation$\}^{+}$)

Jeder Teileintrag repräsentiert eine morphologische Kategorie des Lexems. Hat das Lexikonelement mehrere Teileinträge, so ist es bzgl. der morphologischen Kategorie ambig.

Eine Spezifikation ist entweder eine 2–elementige Liste der Form, wie sie in Def. 5 (s. S. 44) angegeben ist oder ein Templatebezeichner. Ein Template ist eine Liste von Spezifikationen. Der Templatebezeichner dient daher als Abkürzung für solch eine Liste. Die Einträge für “Mann” und “fahr” sehen zum Beispiel wie folgt aus:

(Mann (n (genus mas)))
(fahr (v vmain))

Der Teileintrag von “Mann” kennzeichnet das Element als Nomen, wobei durch die Spezifikation

(genus mas)

festgelegt ist, daß das Merkmal *genus* von “Mann” den Wert *mas* besitzt. Im Teileintrag von “fahr” wird von der Möglichkeit Gebrauch gemacht, morphosyntaktische Information durch Templatebezeicher abzukürzen. Neben der Kategorieangabe für “fahr” enthält der Teileintrag den Bezeichner *vmain*. *vmain* ist die Abkürzung für folgendes Template:

(vmain (Predicate (invertible false))) ,

wobei *Predicate* selber wieder ein Templatebezeicher ist. Das erste Element des Templates von *vmain* ist der Bezeichner selber.

Allgemein werden Templates in SB–PATR wie folgt definiert:

$$\begin{array}{ll} \text{Template} & ::= (\ \text{Templatebezeichner}\ \ \{\text{Spec–or–Temp}\}^{+}) \\ \text{Spec–or–Temp} ::= & \qquad (\text{spec} \mid \{\text{Template}\}^{+})\ , \end{array}$$

wobei *spec* wieder eine 2–elementige Form ist, wie sie in Def. 5, S. 44 eingeführt ist.

Der Templatemechanismus erlaubt die Eliminierung von redundanter Information in Lexikoneinträgen. Information, die z.B. einer Klasse von Verben gemeinsam ist, kann separat mit Hilfe eines Templates repräsentiert werden. Die spezifischen Einträge einzelner Verben benötigen dann nur noch den entsprechenden Templatebezeichner. Da die Templates rekursiv definiert sind, verfügt man mit dem Templatemechansismus über die Möglichkeit, lexikalische Information und damit das Lexikon hierarchisch und modular zu strukturieren [**?**].

Die Listennotation der Lexikonelemente und der Templates sind vor allem für den Grammatikentwickler die formale Grundlage. Das Repräsentationsmittel für diese Elemente in SB–PATR sind DAGs. Jeder Lexikoneintrag und die damit verknüpften Templates werden von SB–PATR in entprechende DAGs übersetzt, wobei während der Übersetzung die Templatebezeichner in einem speziellen Expansionsschritt durch ihre explizite Darstellung ersetzt werden.

Ähnlich wie bei der Übersetzung der grammatikalischen Regeln in DAGs (s. Abschnitt 3.2.2.1), wird die Kategoriebezeichnung des Lexems als Wert dem reservierten Lexem *cat* zugewiesen. Zwei weitere spezielle Merkmale (*lex* und *sense*) werden ebenfalls beim Übersetzungsschritt automatisch hinzugefügt. Beide bekommen als identischen Wert das aktuelle Lexem. Alle übrigen Merkmale werden dem Merkmal $fset$ untergeordnet. Dies gilt insbesondere für die DAGs der Tepmplates. Der DAG des Lexikoneintrag von “fahr” hat als DAG folgende Gestalt:

$$\begin{bmatrix} cat: & v \\ lex: & fahr \\ sense: & < lex > \\ & \\ fset: & \begin{bmatrix} invertible: & false \\ body: & \begin{bmatrix} trans: & \begin{bmatrix} pred: & < sense > \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Der DAG mit Wurzel *body* ist das Ergebnis der Expansion des Templatebezeichners $Predicate$.

3.2.2.3 Relevanz des PATR–II Formalismus für die Generierung

Bisher ist der hier vorgestellte Formalismus und die auf ihn basierenden Tools nur in der Analyserichtung innerhalb von NSS eingesetzt worden. Im nächsten Kapitel zeige ich, wie PATR auch in der Generierungsrichtung eingesetzt werden kann. Die Vorteile, die sich für das Formulieren der syntaktischen Wissensquelle in POPEL–HOW daraus ergeben, sind:

1. Syntaktische Strukturen können aufgrund der formalen Eigenschaften der DAGs *systematisch* strukturiert werden.

2. Als grundlegende Operation für den Aufbau der syntaktischen Struktur, für deren Verifikation und den Transport von Merkmalen steht die *Unifikation* zur Verfügung.

3. Die syntaktischen Regeln können *deklarativ* formuliert werden. Dies ermöglicht es weitgehend unabhängig von den Anforderungen an die Verfahren zum Aufbau der syntaktischen Strukturen in POPEL–HOW (*parallel*, *inkrementell*, *bidirektional*) die Regeln zu erweitern bzw. zu ändern.

4. Für die Strukturierung der Lexikoninformation kann der Templatemechanismus verwendet werden.

5. Es können die gleichen Werkzeuge für die Übersetzung der Regeln und Lexikonelemente in DAGs eingesetzt werden, wie in der Analyse.

Es hat sich im Rahmen der Entwicklung von POPEL–HOW gezeigt, daß der Formalismus und die Basisfunktionen für die Generierung übernommen werden können. In diesem Sinne kann PATR auch als bidirektional bezeichnet werden. Allerdings mußten die Verarbeitungsalgorithmen (im wesentlichen der Chart–Parser) aufgrund der besonderen Anforderungen der Generierung ersetzt werden.

3.2.3 ID/LP–Grammatiken

In Abschnitt 2.6.1 wurde erläutert, daß in POPEL–HOW die Trennung von der Phrasenstruktur und der linearen Ordnung erforderlich ist und formal unterschieden werden muß.

Grammatiken, die die Eigenschaft der linearen Folge formal unterscheiden, werden als ID/LP–Grammatiken (Immediate Dominance/Linear Precedence) bezeichnet. Sie wurden in den achtziger Jahren im Rahmen der GPSG–Theorie [?] entwickelt.

ID/LP–Grammatiken (kurz ID/LP–G) lassen sich folgendermaßen motivieren: In einer kontextfreien Regel

$$vp \rightarrow np\ v\ pp$$

ist nicht nur explizit die Relation der *Dominanz* ausgedrückt, sondern auch implizit die Relation der *linearen Abfolge*. Im Beipiel heißt dies, daß vp die Konstituenten v, np und pp dominiert und, daß np in der linearen Kette vor v und pp stehen muß. Die Nominalphrase "Kevin" , die Präpositionalphrase "mit seinem roten Flitzer" und die Verbalphrase "fährt" können demnach nur zur Verbalphrase "Kevin fährt mit seinem roten Flitzer" zusammengefügt werden. Die im Deutschen gültige Phrase "mit seinem roten Flitzer fährt Kevin" wäre mit dieser Regel nicht konstruierbar, obwohl sich sich nur in der linearen Ordnung von der ersten Phrase syntaktisch unterscheidet.

In einer ID/LP–G werden diese zwei Relationen *explizit* hervorgehoben und durch getrennte Regelmengen formal unterschieden. Die Regel

$vp \to np,\ v,\ pp$

in einer ID/LP–G besagt jetzt lediglich, daß vp die Konstituenten np, v und pp *unmittelbar* dominiert – daher werden solche Regeln als *ID–Regeln* (*Immediate Dominance*) bezeichnet. In dieser Regel ist keine Aussage mehr über die lineare Abfolge der Konstituenten von vp ausgedrückt. Daher werden die Tochterkonstituenten einer ID–Regel auch als Menge charakterisiert. Formal wird dies in einer Regel dadurch ausgedrückt, daß die Töchter durch Kommata getrennt sind.

Die Darstellung der linearen Abfolge der Tochterkonstituenten wird jetzt explizit in den *LP–Regeln* (*Linear Precedence*) vorgenommen. Mit der LP–Regel

$np < v$
$pp < v$

wird durch das Symbol $<$ ausgesagt, daß np oder pp vor v stehen muß.

Unabängig von bestimmten ID–Regeln hat die Ordnungsrelation der LP–Regel für die beteiligten Konstituenten globale Gültigkeit, d.h. np steht *immer* vor v. Fehlt die Regel, so dürfen np und v beliebig permutiert werden.

Im Unterschied zu kontextfreien Grammatiken existieren in ID/LP–G also zwei getrennte Regelmengen. Shieber konnte 1983 zeigen, daß sich jede ID/LP–G in eine stark äquivalente kontextfreie Grammatik überführen läßt und umgekehrt jede kontextfreie Grammatik in eine schwach äquivalente ID/LP–G [?]. Jede ID/LP–G beschreibt daher eine kontextfreie Sprache. Allerdings erlaubt die explizite Darstellung der beiden Relationen, Verallgemeinerungen über die partielle Ordnung der Konstituenten auszudrücken. Ein Vorteil ist die Reduktion von explizit aufzuführenden Regeln. So drücken die folgenden Regeln

ID–Regel: $s, \to a,\ b,\ c,\ d$
LP–Regel: $b\ < d$

das gleiche aus, wie die zwölf aus ihnen herleitbaren äquivalenten kontextfreien Regeln.

Die *linguistische* Motivation für die Entwicklung von ID/LP–Grammatiken liegt gerade in der Möglichkeit, mit der Einführung von LP–Regeln auf einer abstrakteren Beschreibungsebene Aussagen über die lineare Abfolge von Wörtern oder Satzteilen zu machen, als dies bisher in Phrasenstrukturgrammatiken der Fall gewesen ist.

Damit kann das Phänomen der *freien Wortstellung*, was gerade für die deutsche Sprache charakteristisch ist, auf Ebene der LP–Regeln betrachtet werden und muß nicht zusätzlich auf der Ebene der satzstrukturbildenden Regeln [?] formuliert werden.

3.2.4 Die Formulierung von ID/LP–Grammatiken im PATR–Formalismus

Die Formulierung einer Grammatik, die als Komponente eine ID/LP–Grammatik enthalten soll, ist in PATR problematisch, da der Formalismus in seiner Definition nur über genau eine Regelmenge verfügt. Der Formalismus erlaubt es nicht, in *direkter* Weise

ID/LP–Grammatiken zu formulieren. [**?**] sind der Meinung, daß hierfür eine Erweiterung oder Modifikation des bisherigen Formalismus erforderlich ist, um dann die Phänomene der variablen Wortstellung adäquat behandeln zu können.

Eine dort vorgestellte mögliche Erweiterung zur Behandlung des Phänomens "freie Wortstellung" sieht vor, den Typbereich ganz bestimmter Attribute zu erweitern. Am Beispiel der Subkategorisierung von Verben zeigen sie, daß es sinnvoll ist, als Typ des Attributes *syncat* neben atomaren Symbolen auch *Listen* und *Mengen* zuzulassen. Für die Unifikation solcher Attribute ist es dann notwendig, eine geeignete Operation zu definieren. Dies bedeutet aber, daß die Definition der Unifikation selbst erweitert werden muß.

Diese Vorgehensweise hat ebenso den Nachteil, daß weiterhin keine explizite Trennung zwischen ID–Regeln und LP–Regeln vorgenommen wird. Die Linearisierungsregeln sind den ID–Regeln untergeordnet, da sie als Merkmal an diese annotiert werden. Linearisierungsphänomene können dann nicht unabhängig von der strukturellen Satzbeschreibung untersucht werden, was aber gerade durch die Formulierung von ID/LP–Grammatiken geleistet werden sollte. Ich zeige im nächsten Kapitel, wie durch eine andere Interpretation der Regelsyntax dennoch ein direktes Formulieren von ID–/LP–Regeln möglich ist.

Chapter 4

Die Repräsentation syntaktischen Wissens in einem parallelen, inkrementellen Modell

Nachdem nun mit der Dependenztheorie die linguistischen und mit dem PATR–II Formalismus die formalen Grundlagen vorgestellt sind, zeige ich in diesem Kapitel, wie beides zur Formulierung einer parallel und inkrementell einsetzbaren Unifikationsgrammatik miteinander in Beziehung gebracht werden kann.

4.1 POPEL–GRAM: Die Grammatik von POPEL–HOW

Die zentrale Wissensquelle auf der DBS– und ILS–Ebene in POPEL–HOW ist die Unifikationsgrammatik POPEL–GRAM. Diese Grammatik erfüllt die in Abschnitt 2.6.2 gestellten Anforderungen. Eine wesentliche Eigenschaft von POPEL–GRAM ist die Formulierung der Regeln nach dependenztheoretischen Gesichtspunkten. POPEL–GRAM ist im PATR–II Formalismus bzw. mit dem Tool SB–PATR implementiert worden. Durch den Einsatz von SB–PATR können die Regeln deklarativ formuliert werden. Daher steht im folgenden mehr die Strukturierung der Regeln und weniger ihre inhaltliche Erläuterung im Vordergrund.

4.1.1 Zur Notation der Regeln in POPEL–GRAM

In POPEL–GRAM soll die Trennung der Konstituentenstruktur von der linearen Abfolge, d.h. die Darstellung von ID– und LP–Regeln, *explizit* formuliert werden. Da dies mit dem bisher vorgestellten PATR–II Formalismus nicht unmittelbar möglich ist, muß die Definition der Regeln modifiziert werden.

Ausgangspunkt für die Struktur der Regeln in POPEL–GRAM ist die gleiche Listennotation, wie sie bereits für Regeln in SB–PATR eingeführt wurde (s. Abschnitt 3.2.2.1). Allerdings wird in POPEL–GRAM der Teil einer Regel, der bisher die kontextfreie Konstituentenstruktur beschrieb, neu interpretiert. Damit ist es möglich, zwei Regelmengen zu definieren:

- In der ID–Regelmenge wird der (bisherige) kontextfreie Teil als *unmittelbare Dominanz* interpretiert und
- in der LP–Regelmenge als *lineare Abfolge*.

Implementationshinweis:

Die ID–Regeln sind in POPEL–HOW an die gloable Variable *popel–gram–id–rules* gebunden, die LP–Regeln an die Variable *popel–gram–linearization–rules*.

4.1.2 Die Formulierung von ID–Regeln in POPEL–GRAM

Konstituentenstruktur

Eine Regel der Form (der Spezifikationsteil sei leer)[1]

$(s\ e0\ v\ e1)$

als ID–Regel interpretiert bedeutet, daß s die drei Tochterkonstituenten $e0, e1$ und v unmittelbar dominiert. Für diese Konstituenten ist keine lineare Abfolge festgelegt, daher ist obige Regel äquivalent mit den bisherigen Regeln

$(s\ v\ e0\ e1)$
$(s\ v\ e1\ e0)$
$(s\ e0\ e1\ v)$
$(s\ e1\ e0\ v)$
$(s\ e1\ v\ e0)$

In gleicher Weise wird auch in der Regel

$(np\ det\ ap\ n)$

keine lineare Abfolge zwischen den Tochterkonstituenten det, ap und n vorgeschrieben, so daß diese Regel äquivalent wie folgt beschrieben werden kann:

$(np\ n\ det\ ap)$

Dependenz

Der zentrale Gesichtspunkt bei der Formulierung der ID–Regeln von POPEL–GRAM ist es, zusätzlich zur Dominanzrelation die *Dependenzrelation* auszudrücken. Dann können bei der Konstruktion von ID–Regeln dependenztheoretische Prinzipien herangezogen werden.

Ausgangspunkt für die Beschreibung der Dependenzrelation sind die Tochterkonstituenten einer ID–Regel. Mit Hilfe des speziellen Merkmals *head* wird eine der

[1] Bei der Kategoriebezeichnung, die ich in diesem und in den nächsten Beispielen verwende, beziehe ich mich auf [?].

Tochterkonstituenten ausgezeichnet. Es wird festgelegt, daß die so ausgezeichnete Konstituente dasjenige Element t_r ist, das alle seine Geschwister *regiert*, und daß alle Geschwister im Sinne der Dependenztheorie von t_r *abhängen*. Die Menge der abhängigen Geschwisterkonstituenten wird auch als der *Dependenzrahmen* einer ID–Regel bezeichnet.

t_r definiert die unmittelbaren Abhängigkeitsverhältnisse in einer Phrase. Es ist somit das *zentrale* Element und wird daher auch als *head–Element* bezeichnet (der damit assoziierte DAG als *head–DAG*)[2]. Das head–Element charakterisiert die strukturell wesentliche Einheit einer ID–Regel in POPEL–GRAM. Damit definiert es aber auch die zentralen Merkmale der Vaterkonstituente. Die Vaterkonstituente ist somit eine *Projektion* des head–Elementes.

Im Spezifikationsteil genügt eine Gleichung, um eine Tochterkonstituente als head–Element auszuzeichnen und gleichzeitig die *Projektionsbeziehung* zwischen Vaterkonstituente und head–Element zu formulieren.

Beispiel 6
In der Regel

(s v e0 e1

((0 head) (1 head)))

wird durch den rechten Teil

(1 head)

der Gleichung

((0 head) (1 head))

die Konstituente mit Index 1 als head–Element gekennzeichnet und durch die gesamte Gleichung die Projektionsbeziehung mit der Vaterkonstituente (diese hat stets Index 0) beschrieben. Damit ist s eine Projektion von v. Bezeichnet s das Satzsymbol der Grammatik und v die lexikalische Kategorie von Verben, so wird durch diese Regel das Verb als zentrales Element des Satzes gekennzeichnet und ein Satz als Projektion seines Verbes definiert.

Beispiel 7
In der Regel

(np dp ap n

((0 head) (3 head)))

ist np eine Projektion von n.

[2]Ähnlich wird auch das head–Element einer Phrase in der $\overline{X}$–Theorie definiert (vgl. [?]).

Valenz

Unter dem Merkmal *head* eines head–Elementes wird ebenfalls festgelegt, welche Konstituenten des Dependenzrahmens als Ergänzungen oder Angaben fungieren. Die Ergänzungen konstituieren den Valenzrahmen des head–Elementes. Er wird unter dem Merkmal *valence* des regierenden Elementes aufgeführt.

Beispiel 8
In der Regel

```
(s v e0 e6 an

        ((1 head valence value) 2)
        ((1 head valence 1 form) e0)
        ((1 head valence 2 form) e6)
        ((0 head) (1 head)))
```

sind die Kategorien *e*0 und *e*6 als Ergänzungen charakterisiert, da sie unter dem Merkmal *valence* aufgeführt werden. Da die Kategorie *an* nicht im Valenzrahmen angegeben ist, wird sie als Angabe gekennzeichnet[3], die in einer aktuellen Äußerung nicht belegt werden muß. Daher umfaßt obige Regel auch die Regel

```
(s v e0 e6

        ((1 head valence value) 2)
        ((1 head valence 1 form) e0)
        ((1 head valence 2 form) e6)
        ((0 head) (1 head)))
```

Beispiel 9
Die Regel

```
(np n dp ap

        ((0 head) (1 head)))
```

besitzt zum Beispiel nur die Angaben *dp* (Determinativphrasen) und *ap* (Adjektivphrasen), aber keine Ergänzungen. Damit müssen aber die entsprechenden syntaktischen Strukturen während des Generierungsprozesses nicht aufgebaut werden. D.h. diese Regel kann gleichermaßen zum Aufbau der Nominalphrase "der kleine Peter" verwendet werden, wie auch zur Konstruktion von "Peter".

Die Modalität der Ergänzungen, d.h. die Information, ob sie weggelassen werden dürfen oder nicht, wird von den lexikalischen Elementen bestimmt, die als Belegung für das head–Element herangezogen werden.

[3] Angaben werden in POPEL–GRAM rekursiv definiert, d.h. daß hiermit beliebig viele Angaben "gemeint" sind.

Beispiel 10
Das Verb "geben" besitzt folgenden Valenzrahmen (vgl. auch Abschnitt 4.1.5):

```
(value 3)
((1 form) e0)
((1 mod) obli)
((2 form) e1)
((2 mod) obli)
((3 form) e3))
((3 mod) obli)
```

Er charakterisiert "geben" als dreiwertiges Verb, wobei die Leerstellen mit den Satzergänzungen e0, e1 und e3 obligatorisch belegt werden müssen[4]. Von den folgenden Satzregeln

```
(s v e0 e6 an

        ((0 head valence value) 2)
        ((0 head valence 1 form) e0)
        ((0 head valence 2 form) e6)
        ((0 head) (1 head))

(s v e0 e1 e3 an

        ((0 head valence value) 3)
        ((0 head valence 1 form) e0)
        ((0 head valence 2 form) e1)
        ((0 head valence 3 form) e3)
        ((0 head) (1 head)))
```

stimmt nur die letzte mit dem Valenzrahmen des Verbs überein. Gleichzeitig wird von allen Ergänzungen gefordert, daß sie in einer aktuellen Äußerung belegt sein müssen.

Die Beispiele verdeutlichen, daß die lexikalische Information des head–Elements den Regelauswahlmechanismus entscheidend beeinflußt. In diesem Sinne ist der Konstruktionsprozeß in POPEL–HOW *lexikongesteuert.*

Merkmalstransport

Syntaktische und morphosyntaktische Information wird unter dem speziellen Merkmal *syntax* gesammelt. Diese Merkmalsbündelung hat zum einen den Vorteil, daß die entsprechende Information beim Merkmalstransport als Einheit betrachtet werden kann. Zum anderen erlaubt diese Zusammenfassung eine Typisierung von gleichartigen Merkmalen. So haben die Merkmale *person* und *numerus*, die unter dem Merkmal *syntax* auftreten, den Typ *syntax*. Das Merkmal *head* faßt z.B. alle Merkmale zusammen, die im

[4]Der Valenzrahmen kann auch mit Hilfe des Templatemechanismus abgekürzt werden, wodurch der Lexikonaufwand reduziert wird. In diesem Beispiel verwende ich allerdings der Klarheit halber die explizite Form.

wesentlichen vom head–Element bestimmt werden. Diese Typisierung erlaubt es, in einfacher Weise Merkmalsbündel unterschiedlicher Konstituenten in Beziehung zu bringen. Weiter Typen, die in POPEL–GRAM definiert sind, sind *surface* und *synchronize*. Sie werden weiter unten eingeführt.

Beispiel 11
In der Regel

(s v e0 e1

((0 head) (1 head))
((0 syntax) (1 syntax))
((1 syntax person) (2 syntax person))
((1 syntax numerus) (2 syntax numerus)))

wird durch die eine Gleichung

((0 syntax) (1 syntax))

die gesamte syntaktische Information des Verbs an die Satzphrase weitergereicht. Dieses Beispiel verdeutlicht nebenbei, wie einfach und elegant mit Hilfe der Gleichungen

((1 syntax person) (2 syntax person))
((1 syntax numerus) (2 syntax numerus)))

Kongruenzphänomene – hier zwischen dem Verb und dem nominalen Satzglied $e0$ (dem Subjekt) – in PATR ausgedrückt werden können.

Beispiel 12
Noch deutlicher wird dies in der Regel

(np n dp ap

((0 syntax) (1 syntax))
((1 syntax) (2 syntax))
((2 syntax) (3 syntax))) ,

in der die letzten zwei Gleichungen ausreichen, um die Kongruenz zwischen Nomen, Artikelphrase *dp* und attributiver Adjektivphrase *ap* zu formulieren.

Parallele und unabhängige Verbalisierung der Lexeme

Die Lexeme einer Phrase werden unter dem speziellen Merkmal *surface* gesammelt. Dies gilt insbesondere auch für die Tochterkonstituenten.In den Regeln für eine Grammatik, die in einem sequentiellen Modell eingesetzt werden kann, müssen entsprechende Gleichungen zwischen dem *surface* Merkmal der Vaterkonstituenten und aller Tochterkonstituenten formuliert werden, um alle Lexeme einer Phrase zu erhalten.

Beispiel 13
In der Regel

```
(s v e0 e1
        ((0 head) (1 head))
        ((0 surface s v) (1 surface))
        ((0 surface s e0) (2 surface))
        ((0 surface s e1) (3 surface)))
```

werden alle Lexeme, die den Satz konstituieren, unter dem $surface$ Merkmal von s gesammelt.

In einem parallelen Modell, in dem die syntaktische Struktur, wie dies in POPEL–HOW der Fall ist, gleichzeitig von mehreren parallel arbeitenden Objekten aufgebaut wird, dürfen nicht alle Lexeme der Phrase zugeordnet werden. Es muß garantiert werden, daß jedes Objekt seine Lexeme "selbst" verbalisiert.

Wenn z.B. angenommen wird, daß die Konstituenten v, $e1$ und $e0$ jeweils von unabhängigen Objekten aufgebaut werden, muß die enstprechende Satzregel, wie folgt lauten:

Beispiel 14

```
(s v e0 e1
        ((0 head) (1 head))
        ((0 surface s v) (1 surface)))
```

Damit wird gewährleistet, daß für s nur die verbalen Lexeme des head–Elementes v in der ILS–Ebene weiterverarbeitet werden.

Generell müssen unter dem Merkmal $surface$ einer Phrase (bzw. Vaterkonstituenten) auf jeden Fall die Lexeme des head–Elementes gesammelt werden (wegen der Projektivitätsbeziehung). Ob darüber hinaus noch Lexeme der übrigen unmittelbaren Konstituenten der Phrase herangezogen werden ist davon abhängig, ob diese Konstituenten auf ein oder mehrere aktive Objekte verteilt werden.

Richtung des Merkmalstransports

Der parallele Konstruktionsprozeß verlangt, in besonderen Situationen nachvollziehen zu können, welche *Richtung* der Wert bestimmter Merkmale genommen hat (vgl. Kapitel 5). In Abschnitt 3.2.1 wurde darauf hingewiesen, daß dies in den DAGs jedoch nicht ohne weiteres möglich ist. Wir führen deshalb das spezielle Merkmal *synchronize* ein. Mit Hilfe dieses Merkmals ist es möglich festzustellen, welche Richtung die Werte bestimmter Merkmale genommen haben. Hierbei unterscheiden wir zwei Fälle:

1. Die Information fließt in Teilstrukturen einer Phrase (die Information fließt nach unten (*down*)).
2. Die Information kommt von einer übergeordneten Phrase (die Information kommt von oben (*up*)).

Beide Eigenschaften werden ebenfalls durch spezielle Merkmale ausgedrückt. Das Merkmal *synchronize* hat damit folgende schematische Merkmalsstruktur:

$$\left[\ synchronize: \begin{bmatrix} down: & [...] \\ up: & [...] \end{bmatrix}\ \right]$$

Einige Beispiele sollen die Wirkungsweise des Merkmals *synchronize* verdeutlichen. Dabei ist es nicht notwendig, daß die Merkmale *down* und *up* stets gemeinsam unter *synchronize* vorkommen müssen.

$$\begin{bmatrix} syntax: & \begin{bmatrix} voice: & passive \end{bmatrix} \\ synchronize: & \begin{bmatrix} down: & < syntax\ voice > \end{bmatrix} \end{bmatrix}$$

Für eine Phrase bedeutet diese Information, daß mit der Definition der Phrase das Merkmal *voice* den Wert *passive* erhält und daß dieser Wert in Teilstrukturen fließt und vererbt wird. Für *voice* ist also eine Flußrichtung nach unten spezifiziert.

Hat eine Phrase den DAG:

$$\begin{bmatrix} syntax: & \begin{bmatrix} voice: & [\,] \end{bmatrix} \\ synchronize: & \begin{bmatrix} up: & < syntax\ voice > \end{bmatrix} \end{bmatrix}$$

ist festgelegt, daß sie den Wert für *voice* von einer übergeordneten Phrase erhalten wird (und wenn *voice* einen Wert erhalten hat, kann man nachvollziehen, daß der Wert von oben gekommen ist).

Im DAG

$$\begin{bmatrix} syntax: & \begin{bmatrix} voice: & [\,] \end{bmatrix} \\ synchronize: & \begin{bmatrix} down: & < syntax\ voice > \\ up: & < syntax\ voice > \end{bmatrix} \end{bmatrix}$$

ist der Wert von *voice* ebenfalls unbekannt. Es ist aber durch das gemeinsame Auftreten der Merkmale *up* und *down* angegeben, daß er in einer übergeordneten Phrase spezifiziert und an Teilstrukturen weitergereicht wird. Die Phrase, die obigen DAG besitzt, ist somit Empfänger und Sender des Wertes vom Merkmal *voice*.

Ein Beispiel

Die bisherigen Erläuterungen machen deutlich, daß in den ID–Regeln die Merkmale systematisch strukturiert werden. Dadurch können Merkmale mit gemeinsamen Funktionen gebündelt und als Einheiten betrachtet werden. Noch klarer wird die Merkmalsstruktur, wenn eine ID–Regel als DAG dargestellt wird.

Beispiel 15
So läßt sich die ID–Regel

$(s\ \ v\ e0\ e1$

((0 syntax) (1 syntax))
((1 syntax voice) active)
((1 syntax voice) (2 syntax voice))
((1 syntax voice) (3 syntax voice))
((1 syntax number) (2 syntax number))
((1 syntax person) (2 syntax person))
((0 synchronize down voice) (1 syntax voice))
((1 head valence value) 2)
((1 head valence 1 form) e0)
((1 head valence 2 form) e1)
((0 head) (1 head))
((0 surface s v) (1 surface)))

als DAG wie folgt darstellen:

$$
\begin{bmatrix}
0: & \begin{bmatrix} cat: & s \\ fset: & \begin{bmatrix} syntax: & <1\ fset\ syntax> \\ head: & <1\ fset\ head> \\ synchronize: & \begin{bmatrix} down: & \begin{bmatrix} voice: & <1\ fset\ syntax\ voice> \end{bmatrix} \end{bmatrix} \\ surface: & \begin{bmatrix} s: & \begin{bmatrix} v: & <1\ fset\ surface> \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
1: & \begin{bmatrix} cat: & v \\ fset: & \begin{bmatrix} syntax: & \begin{bmatrix} voice: & active \\ number: & <2\ fset\ syntax\ number> \\ person: & <2\ fset\ syntax\ person> \end{bmatrix} \\ head: & \begin{bmatrix} valence: & \begin{bmatrix} value: & 2 \\ 1: & \begin{bmatrix} form: & e0 \end{bmatrix} \\ 2: & \begin{bmatrix} form: & e1 \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ surface: & [\,] \end{bmatrix} \end{bmatrix} \\
2: & \begin{bmatrix} cat: & e0 \\ fset: & \begin{bmatrix} syntax: & \begin{bmatrix} voice: & <1\ fset\ syntax\ voice> \\ number: & [\,] \\ person: & [\,] \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
3: & \begin{bmatrix} cat: & e1 \\ fset: & \begin{bmatrix} syntax: & \begin{bmatrix} voice: & <1\ fset\ syntax\ voice> \end{bmatrix} \end{bmatrix} \end{bmatrix}
\end{bmatrix}
$$

4.1.3 Exkurs: Endozentrische Konstruktionen

Die Definition der Phrase im ID–Teil ähnelt den endozentrischen Konstruktionen (sie werden auch head–modifier Konstruktionen genannt), die im Rahmen der Konstituentenstrukturgrammatiken definiert wurden und auch in heutigen Syntaxmodellen eine bedeutende Rolle spielen (vgl. GPSG oder Government Binding Theorie [?]). *Endozentrische*

Konstruktionen zeichnen sich dadurch aus, daß solche syntaktischen Einheiten zur selben Formklasse gehören wie eine ihrer Konstituenten. Es ist möglich, die gesamte Einheit durch diese ausgezeichnete Konstituente zu ersetzen. Die Nominalphrase

"der kleine Peter"

läßt sich zum Beispiel in einer syntaktischen Konstruktion durch das Nomen

"Peter"

austauschen, ohne daß dadurch die gesamte Äußerung ungrammatisch wird. Die ausgezeichnete Konstituente wird als das *head*–Element der Phrase bezeichnet und alle anderen Konstituenten als dessen *modifier*–Elemente. Konstruktionen, die nicht durch eine ihrer Konstituenten substituierbar sind, heißen *exozentrisch*.

Endozentrische Konstruktionen ließen sich auch in einer Dependenzgrammatik definieren. So könnte der Nukleus einer Phrase als head und dessen abhängige Elemente als modifier bezeichnet werden. Allerdings muß dabei berücksichtigt werden, ob die abhängigen Elemente als obligatorische oder fakultative Ergänzungen charakterisiert werden. Eine endozentrische Konstruktion, wie sie oben definiert ist, wäre eine Phrase, deren Nukleus nur fakultative Ergänzungen hätte. Alle Phrasen, die obligatorische Ergänzungen hätten, wären demnach exozentrisch.

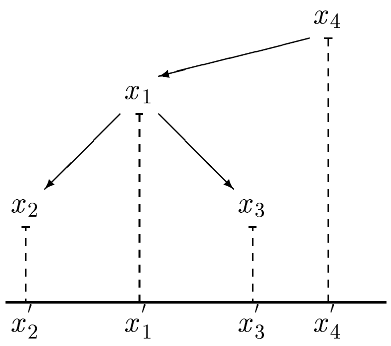
4.1.4 Die Linearisierungsregeln von POPEL–GRAM

Grundlage für die Formulierung der Linearisierungsregeln in POPEL–GRAM ist die Einteilung des Satzes in Felder (vgl. Abschnitt 3.1.4). Dabei wird eine unmittelbare Darstellung der in dem Abschnitt erläuterten Beschreibung der linearen Abfolge im Deutschen angestrebt. Es wird sich zeigen, daß hierfür eine sehr viel komprimiertere Darstellung der Abfolgephänomene erforderlich ist, als dies z.B. mit den im Rahmen der ID/LP–Grammatiken (s. Abschnitt 3.2.3) definierten binären LP–Regeln möglich ist.

Bevor ich in Abschnitt 4.1.4.3 auf die Formulierung der LP–Regeln eingehe, werden ich kurz das Verhältnis zwischen syntaktischer Struktur und linearer Abfolge charakterisieren. Dieses Verhältnis hat einen starken Einfluß auf die Definition der Linearisierungsregeln.

4.1.4.1 Dependenz und Position

Ausgangspunkt für die Beschreibung des Verhältnisses zwischen Dependenz und Position ist die *Projektion* der Elemente eines Dependenzbaumes auf entsprechende Punkte einer waagerechten Geraden, die die lineare Abfolge repräsentiert. Dies läßt sich graphisch wie folgt darstellen:



Wenn es möglich ist, alle Projektionen durch senkrechte Linien darzustellen, heißt der entsprechende Dependenzbaum *projektiv*. Genauer: Ein Dependenzbaum ist projektiv, wenn für jeden Knoten x sein Bild x' und die Bilder aller von ihm abhängigen Knoten sämtlich auf einem zusammenhängenden Stück der Projektionsgeraden liegen [**?**].

Kann ein Dependenzbaum nicht sofort projektiv dargestellt werden, ist es eventuell möglich, ihn durch Umordnung der Knoten projektiv zu arrangieren. Allerdings gibt es Dependenzbäume, die sich nicht projektiv mit einer gültigen linearen Abfolge in Beziehung bringen lassen. In diesen Fällen existiert zwischen abhängigen Elementen keine lokale Nachbarschaftsbeziehung in der linearen Abfolge.

In diesem Zusammenhang muß aber betont werden, daß die Bestimmung der Projektivität neben der linearen Abfolge auch von der angenommenen dependentiellen Struktur abhängt. Es ist durchaus der Fall, daß eine Veränderung der dependentiellen Verhältnisse die Gültigkeit der Projektivität verändert (vgl. hierzu auch Abschnitt 18).

4.1.4.2 Projektive ID–Strukturen in POPEL–GRAM

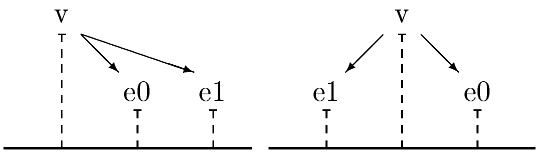
Die Projektivitätsbedingung läßt sich auch auf die Konstituentenstruktur von POPEL–GRAM anwenden. Jede ID–Regel läßt sich graphisch als n–ärer Baum der Tiefe 1 darstellen. Da zwischen den Tochterkonstituenten die Dependenz definiert ist, lassen sich die Blattknoten als Knoten eines Dependenzbaums darstellen. Die Knoten des Dependenzbaums können dann auf eine Gerade projeziert werden, die die Position der Elemente widerspiegelt. Daher bezeichne ich den aus einer ID–Regel extrahierten Dependenzbaum als *Positionsbaum*. Die Positionsbäume sind Ausgangspunkt für die Bestimmung der korrekten Abfolge in POPEL–HOW.

Beispiel 16
Die ID–Regel

```
(s v e0 e1
        ((0 head) (1 head)))
```

besitzt folgenden ID–Baum und Positionsbaum

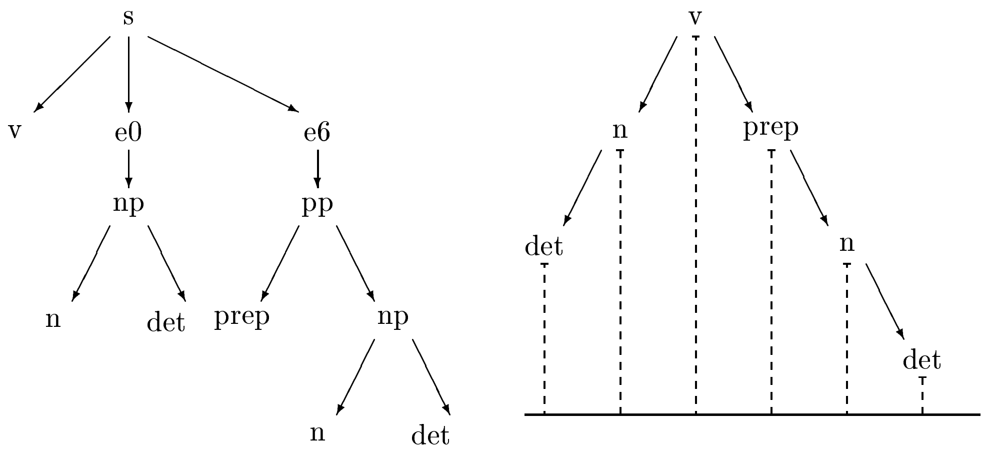
Im allgemeinen gibt es für den Baum einer ID–Regel mehrere mögliche Positionsbäume. So sind für obige Regel u.a. folgende Bäume ebenfalls konstruierbar:



Der linke Positionsbaum zeigt z.B. die lineare Abfolge, wie sie für Entscheidungsfragen gültig ist.

Positionsbäume lassen sich auch für komplexe Konstituentenstrukturen ableiten. Dabei wird die Tatsache berücksichtigt, daß die Abhängigkeitsverhältnisse transitiv sind. Es werden wiederum nur die Blattknoten im entsprechenden Positionsbaum dargestellt (die regierenden Elemente im ID–Baum sind der Einfachheit halber alle ganz links im entsprechenden Teilbaum angeordnet).

Beispiel 17



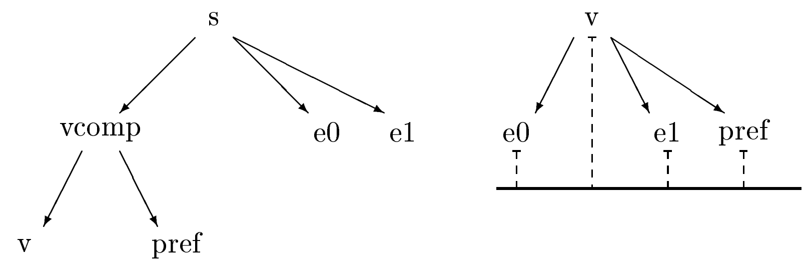
In diesem Zusammenhang möchte ich darauf hinweisen, daß bisher in den Satzregeln von POPEL–GRAM die Verbalkonstituente durch ein Element – das Hauptverb – beschrieben ist. Das folgende Beispiel zeigt, wie komplexe Verbalteile in POPEL–GRAM dargestellt und auf einen entsprechenden Positionsbaum projeziert werden können[5]. Dabei sei noch einmal daran erinnert, daß für die dependentielle Anordnung die Gesichtspunkte der Zweckmäßigkeit ausschlaggebend sind. Unter diesem Gesichtspunkt legen wir fest, daß die dependentielle Struktur so gewählt werden sollte, daß die Projektivitätsbedingung nicht verletzt wird.

Beispiel 18
Für Verben mit abtrennbarem Präfix (pref) gelte folgende ID–Regel:

(vcomp v pref

((0 head) (1 head)))

Ein komplexer Konstituentenstruktur– bzw. Positionsbaum für einen entsprechenden Satz läßt sich wie folgt darstellen:



Analog können z.B. auch Partizipialformen mit der Regel

(vcomp v aux

((0 head) (1 head))
((1 syntax form) partizip))

oder Modalformen mit

(vcomp v modalv

((0 head) (1 head))
((1 syntax form) infinitiv))

[5]Zur Zeit werden in POPEL–GRAM keine komplexen Verbalteile in den ID–Regeln dargestellt, da aufgrund der vollständig beschriebenen morphosyntaktischen Struktur des Hauptverbes die entsprechenden finiten und infiniten Verbteile während der Flexionsphase von MORPHIX berechnet werden (s. Abschnitt 4.1.4.3).

dargestellt werden.

In allen Fällen sind die Positionsbäume projektiv. Daher beschreiben solche Konstruktionen in diesem Sinne keine "Long Distance" Phänomene.

Problematisch sind dagegen rekursiv formulierte Konstruktionen, wie z.B. Angaben. Durch Anwendung folgender Regel

(an ein–an an)

– ein–an subsumiert die möglichen syntaktischen Ausdrucksformen (Nominal–, Adjektiv–, Präpositionalphrase etc., vgl. [?]) – werden alle Angaben in einer komplexen Konstituentenstruktur unter einem gemeinsamen Teilbaum angeordnet. Damit können sie aber nur "am Stück" umgeordnet werden. Eine Äußerung wie

"Peter fliegt morgen früh mit Maria ins Ausland." ("morgen", "früh" und "mit Maria" seien die Angaben),

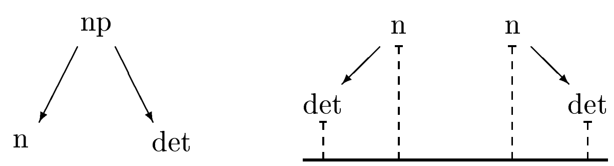
ist projektiv, aber nicht

"Morgen früh fliegt Peter mit Maria ins Ausland."

Da die Projektivitätsbedingung Grundlage für die Bestimmung der korrekten linearen Abfolge ist, werden durch rekursive Regeln mögliche Abfolgen nicht erlaubt. Das Problem läßt sich aber dadurch lösen, daß während der Konstruktion eines entsprechenden Positionsbaum das rekursive Element nicht berücksichtigt wird, d.h. im Positionsbaum alle Angaben nebeneinander angebracht werden. Die hierfür notwendigen Mechanismen sind allerdings z.Z. in POPEL–HOW noch nicht implementiert.

4.1.4.3 Aufgabe und Notation der Linearisierungsregeln

Es ist möglich, Positionsbäume zu konstruieren, die unkorrekte lineare Abfolgen beschreiben. Zum Beispiel beschreibt in der folgenden Abbildung der rechte Positionsbaum eine ungültige lineare Abfolge:



Die Aufgabe der Linearisierungsregeln in POPEL–GRAM ist es, unkorrekte Positionsbäume bzw. lineare Abfolgen zu vermeiden. Da in POPEL die Komponente POPEL–WHAT durch das inkrementelle Eingabeverhalten im Prinzip die Reihenfolge bereits angibt, fungieren die Linearisierungsregeln demnach als negativer Filter. Prinzipiell beziehen sich die Abfolgeaussagen auf die präterminalen Elemente von komplexen Konstituentenstrukturbäumen des ID–Teils. Aufgrund der Beziehung zwischen syntaktischer Struktur und linearer Abfolge ist es möglich, diese Elemente auch in den LP–Regeln abstrakt zusammenzufassen. Die LP–Regeln in POPEL–GRAM beschreiben dann in ihrer komprimierten Form implizit die Menge der erlaubten Positionsbäume. Während der Linearisierungsphase beschränken die LP–Regeln damit die Menge der möglichen Positionsbäume.

Grundlage für die Notation der Regeln ist wie bei den ID–Regeln die in Abschnitt 3.2.2.1 vorgestellte Listennotation der Regeln in SB–PATR. In einer LP–Regel wird der kontextfreie Teil interpretiert als *vorschreibende Abfolgerelation*.

Eine Regel der Form:

(x y_1 y_2 y_3)

besagt dann: Enthält die zu linearisierende Struktur x die Elemente y_1, y_2 und y_3, so müssen sie in der Reihenfolge:

$$y_1 < y_2 < y_3$$

angeordnet werden. Es ist aber keine Existenz der Elemente gefordert, d.h. wenn nur y_1 und y_3 in der Struktur vorkommen, ist die korrekte Abfolge:

$$y_1 < y_3$$

In diesem Sinne ist der rechte Teil obiger LP–Regel eine Abkürzung für die binären Lineariserungsaussagen:

$$\begin{array}{l} y_1 < y2 \\ y_1 < y3 \\ y_2 < y3 \end{array} ,$$

wie sie explizit in ID/LP–Grammatiken spezifiziert würden.

Das linke Element (im Beispiel x) kennzeichnet die Menge der Positionsbäume, für die diese LP–Regel anwendbar ist. Dabei unterscheiden wir zwei Fälle:

- Das Element bezieht sich auf die Konstituente, dessen head–Element Wurzelknoten im Positionsbaum ist (z.B. LP–Regeln für np, pp oder s).
- Das Element bezieht sich auf eine ganze Klasse von Konstituenten (z.B. Vorfeld oder Mittelfeld)[6].

Der letzte Fall erlaubt es z.B. Regeln der folgenden Art zu schreiben:

(vorfeld any)

Die Regel besagt: An der Stelle, in der das Vorfeld besetzt werden soll, darf ein beliebiger Positionsbaum angeordnet werden. Es handelt sich bei dieser Regel im Prinzip um keine "echte" Linearisierungsregel, da sie auf der linken Seite ein Element hat. Sie ist nur im Zusammenhang mit anderen Regeln anwendbar, erlaubt aber die komprimierte Schreibweise.

In den LP–Regeln können auch Spezifikationen angegeben werden. Ihre Verwendung erlaubt zweierlei:

[6]Inwiefern die LP–Regeln dadurch als "bedingt" global oder lokal zu charakterisieren sind, ist z.Z. noch nicht untersucht.

1. Bei der Anwendung der Linearisierungsregeln kann das aktuelle Verhältnis zwischen syntaktischer Struktur und linearer Abfolge berücksichtigt werden.
2. Ähnlich wie mit Hilfe der Merkmale *surface* und *synchronize* im ID–Teil von POPEL–GRAM, können in den LP–Regeln besondere Merkmale spezifiziert werden, die den parallelen Generierungsprozeß unterstützen.

Im ersten Fall beziehen sich die Spezifikationen auf die Belegung bestimmter Merkmale des ID–DAGs. Dadurch können im Lineariserungsprozeß syntaktische Besonderheiten berücksichtigt werden. Der zweite Fall ist Voraussetzung dafür, daß die Art und Weise der Verteilung der syntaktsichen Strukturen berücksichtigt werden kann (s. Kapitel 6.2).

Am Beispiel der Abfolgerelation im Aussagesatz will ich dies verdeutlichen[7]. Die lineare Abfolge in einem Aussagesatz ist nach [**?**]:

Vorfeld – finiter Verbteil – Mittelfeld – infiniter Verbteil – Nachfeld .

Dies läßt sich unmittelbar durch folgende LP–Regel ausdrücken:

Beispiel 19

```
(s ff finv mf infinv rf
        ((0 syntax sentence–form) declarative)

        ((1 dependent) +)
        ((3 dependent) +)
        ((5 dependent) +)) ,
```

wobei s das Satzsymbol von POPEL–GRAM bezeichnet und das Vorfeld durch ff (frontfield), der finite Verbteil durch $finv$, das Mittelfeld durch mf, der infinite Verbteil durch $infinv$ und das Nachfeld durch rf (restfield) gekennzeichnet ist.

Mit der wertidentifizierenden Gleichung:

((0 syntax sentence–form) declarative)

ist festgelegt, daß die Linearisierungsregel nur für solche Satz–DAGs angewendet werden darf, deren Merkmal *sentence–form* den Wert *declarative* hat, also nur für Aussagesätze.

Die Regel berücksichtigt die Tatsache , daß in der Flexionsphase für das Hauptverb entsprechend der morphosyntaktischen Information der finite und infinite Teil berechnet wird.

Bei der Linearisierung der Dependenten eines Satz–DAGs muß berücksichtigt werden, daß aufgrund des parallelen Konstruktionsprozesses Teil–DAGs von s als eigenständige Objekte existieren. Mit Hilfe der Spezifikation

(($<$ index $>$ dependent) +) ,

mit index $\in N$, wird dies in einer LP–Regel markiert. So bedeutet die Gleichung

[7]Im Anhang B befinden sich alle zur Zeit formulierten LP-Regeln.

((1 dependent) +)

in Bsp. 19, daß das Vorfeld ff durch den DAG eines anderes Objektes belegt werden wird. Die entsprechende Regel für das Vorfeld ist folgende:

(ff any)

Das Symbol *any* deutet an, daß als Belegung für das Vorfeld genau ein beliebiges vom Hauptverb abhängiges Element auftreten darf. Damit ist diese Regel für alle Ergänzungen und Angaben des Verbs gültig.

4.1.5 Die Lexikonstruktur in POPEL–HOW

Die lexikalischen Elemente werden in POPEL–HOW ebenfalls als DAGs repräsentiert. Ein lexikalischer DAG hat folgende Struktur:

$$\begin{bmatrix} cat: & ... \\ fset: & \begin{bmatrix} syntax: & [...] \\ head: & \begin{bmatrix} stem: & ... \\ ... & \end{bmatrix} \\ surface: & \begin{bmatrix} stem: & < fset\ head\ stem > \\ syntax: & < 1\ fset\ syntax > \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Grundlage für den Aufbau der DAG–Struktur ist die Listennotation für lexikalische Elemente, die in Abschnitt 3.2.2.2 beschrieben ist. Ausgehend vom Ergebnis des Wortwahlprozesses und spezifischer Information aus XTRALEX wird für jedes Lexem eine Listenstruktur aufgebaut und in einen entsprechenden DAG übersetzt.

Beispiel 20
Für das Nomen "haus" liefert das Wortwahlergebnis neben dem Lexem, die Kategorie *nomen*, den Numerus *sg* sowie Information darüber, daß "haus" nicht elidiert werden soll. Diese Werte und die Numerusinformation werden während der Deskriptorenwahl durch Interaktion mit POPEL–WHAT bestimmt. Aus dem Stammlexikon von XTRALEX wird der Genus *ntr* extrahiert. Damit ergibt sich folgender Listeneintrag:

```
("haus"

        (nomen
                ((syntax numerus) sg)
                ((syntax gender) ntr)
                ((synatx elision) –)
                SurfaceLex))
```

SurfaceLex ist ein Template für:

((surface head) (head stem))
((surface syntax) (syntax))

Der Listeneintrag von "Haus" kann dann in folgenden DAG übersetzt werden:

$$
\begin{bmatrix}
cat: & nomen \\
fset: & \begin{bmatrix}
syntax: & \begin{bmatrix} gender: & ntr \\ elision: & - \\ numerus: & sg \end{bmatrix} \\
head: & \begin{bmatrix} stem: & \text{“}Haus\text{''} \end{bmatrix} \\
surface: & \begin{bmatrix} stem: & < fset\ head\ stem > \\ syntax: & < 1\ fset\ syntax > \end{bmatrix}
\end{bmatrix}
\end{bmatrix}
$$

Implementationshinweis:

Die Templates sind in der Hash–Tafel *popel–templates* abgelegt. Die Templatebezeichner fungieren als Schlüssel.

In analoger Weise werden alle lexikalischen Elemente in entsprechende DAGs übersetzt. Bei Verben ist zu beachten, daß neben der Information von Tempus, Modus und Satzform (diese Information wird durch Interaktion mit POPEL–WHAT erhalten) aus XTRALEX der Valenzrahmen extrahiert wird.

Implementationshinweis:

form–stem–dag ist die Funktion, die die Übersetzung eines Lexems des Wortwahlergebnisses in einen lexikalischen DAG durchführt. Aktueller Parameter ist ein Teileintrag des Wortwahlergebnisses.

4.2 Die Basisoperation zur Generierung syntaktischer Strukturen in POPEL–HOW

Im Gegensatz zum Analyseprozeß, in dem es das Ziel ist, die syntaktische Struktur einer natürlichsprachlichen Äußerung zu *rekonstruieren*, ist es in der Generierung das Ziel, sie zu *konstruieren*.

Für den Aufbau sind Operationen notwendig, die die *konstruktive* Sichtweise des Generierungsprozesses widerspiegeln. Die Definition der Operationen wird sehr stark von der formalen und linguistischen Grundlage der Grammatik, sowie vom zugrundeliegenden Architekturprinzip und Kontrollfluß des Generierungssystems beeinflußt.

In POPEL–HOW wird die syntaktische Struktur inkrementell von parallel arbeitenden, kooperierenden Objekten aufgebaut[8]. Jedes Objekt bestimmt sich aus seinen Ausgangsstrukturen (der Phrasenbezeichnung und den Lexemen) den Teil der syntaktischen Struktur, den es möglichst unabhängig verbalisieren soll. Zur Vervollständigung und zur Überprüfung der Grammatikalität der Teilstrukturen, müssen miteinander verbundene Objekte ihre Strukturen in Beziehung bringen können.

Die zugrundeliegende Repräsentationsstruktur der syntaktischen Information ist der DAG. Zur Lösung der oben aufgeführten Aufgaben benötigen die Objekte Operationen, mit denen sie ihre DAGs Aufbauen und den notwendigen Merkmaltransport durchführen können. Als zentrale Wissensquelle hierfür dient POPEL–GRAM.

Die prinzipielle Vorgehensweise der Operationen ist ähnlich: Je zwei DAGs werden miteinander *unifiziert*, wobei zwischen den DAGs eine Teil/Ganzes–Relation besteht. Es war naheliegend, hierfür ein Verfahren zu entwickeln, das als Basisoperation im parallelen und inkrementellen Konstruktionsprozeß eingesetzt werden kann.

4.2.1 Ein Suchverfahren als Basisoperation

Die Ausgangsstrukturen des Verfahrens sind zwei *nicht* zusammenhängende Graphen d_A und d_I. d_A (A steht für "äußerer DAG") ist der unvollständige DAG einer Phrase und d_I (I steht für "innerer DAG") möglicher Teil–DAG von d_A. Das Ziel des Verfahrens – es wird im weiteren *dag–search* genannt – ist es, d_I in d_A so zu integrieren, daß der auf diese Weise modifizierte DAG $d_A^{'}$ nach der Operation mehr Information codiert als vorher.

Beispiel 21
Zum Beispiel läßt sich in der Nominalphrase:

[8] Im nächsten Kapitel werde ich diesen Konstruktionsprozeß im Detail erläutern.

$$\begin{bmatrix} 0: & \begin{bmatrix} cat: & np \\ fset: & \begin{bmatrix} syntax: & <1\ fset\ syntax> \\ head: & <1\ fset\ head> \\ surface: & \begin{bmatrix} np: & \begin{bmatrix} det: & <2\ fset\ surface> \\ n: & <1\ fset\ surface> \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ 1: & \begin{bmatrix} cat: & n \\ fset: & \begin{bmatrix} syntax: & [\,] \\ head: & [\,] \\ surface: & [\,] \end{bmatrix} \end{bmatrix} \\ 2: & \begin{bmatrix} cat: & det \\ fset: & \begin{bmatrix} syntax: & <1\ fset\ syntax> \\ surface: & [\,] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

der nominale DAG:

$$\begin{bmatrix} cat: & n \\ fset: & \begin{bmatrix} syntax: & \begin{bmatrix} gender: & ntr \\ elision: & - \\ numerus: & sg \end{bmatrix} \\ head: & \begin{bmatrix} stem: & "Haus" \end{bmatrix} \\ surface: & \begin{bmatrix} stem: & <fset\ head\ stem> \\ syntax: & <1\ fset\ syntax> \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

unter dem Teil–DAG mit Wurzel 1 der Nominalphrase integrieren. Das Ergebnis ist der modifizierte NP–DAG:

$$\begin{bmatrix}
0: & \begin{bmatrix} cat: & np \\ fset: & \begin{bmatrix} syntax: & <1\ fset\ syntax> \\ head: & <1\ fset\ head> \\ surface: & \begin{bmatrix} np: & \begin{bmatrix} det: & <2\ fset\ surface> \\ n: & <1\ fset\ surface> \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
1: & \begin{bmatrix} cat: & n \\ fset: & \begin{bmatrix} syntax: & \begin{bmatrix} gender: & ntr \\ elision: & - \\ numerus: & sg \end{bmatrix} \\ head: & \begin{bmatrix} stem: & "Haus" \end{bmatrix} \\ surface: & \begin{bmatrix} stem: & <fset\ head\ stem> \\ syntax: & <1\ fset\ syntax> \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
2: & \begin{bmatrix} cat: & det \\ fset: & \begin{bmatrix} syntax: & <1\ fset\ syntax> \\ surface: & [\,] \end{bmatrix} \end{bmatrix}
\end{bmatrix}$$

Der DAG d'_A (im Beispiel der modifizerte NP–DAG) ist eine spezifischere Ausprägung von d_A oder anders ausgedrückt: d_A *subsumiert* d'_A. Die Subsumptionsrelation läßt sich formal wie folgt charakterisieren:

Definition 6
Ein komplexer DAG d subsumiert (als Zeichen: $\sqsubseteq$) einen komplexen DAG d' gdw. $\forall$ Merkmale $l \in dom(d)$: $d(l) \sqsubseteq d'(l)$ und $\forall$ Pfade p,q: $d(p) = d(q) \Rightarrow d'(p) = d'(q)$.

Ein atomarer DAG subsumiert keinen DAG und kann von keinem DAG subsumiert werden. Der leere DAG [ ] subsumiert jeden DAG, da er keine Information enthält und somit die allgemeinste Beschreibung darstellt.

Die Subsumption ist ein spezieller Fall der Kompatibilität:

Definition 7
Zwei DAGs d und d' sind kompatibel, wenn es einen DAG d'' gibt, so daß $d \sqsubseteq d''$ und d' $\sqsubseteq d''$.

Sind zwei DAGs kompatibel, können sie miteinander unifiziert werden (vgl. S. 41, Def. 2).

Mit Hilfe dieser Begriffe ist es möglich, die zu lösende Aufgabe von *dag–search* auch als *Suchproblem* zu formulieren:

- Suche einen relevanten Teil–DAG d_T in d_A, der mit d_I kompatibel ist.
- Kann d_T gefunden werden, so unifiziere d_T mit d_I. Ersetze d_T durch den Ergebnis–DAG d'_T der Unifikationsoperation.

- Da d_T' nun Teil–DAG von d_A ist, gilt für den Ergebnis–DAG d_A' der Suchoperation: $d_A \sqsubseteq d_A'$.

Es ist möglich, daß d_I einen DAG beschreibt, der mit einem Teil–DAG von d_A kompatibel ist, der in d_A *implizit* beschrieben ist. Dies ist zum Beispiel dann der Fall, wenn es sich bei d_I um einen DAG handelt, der nur mit einer *mittelbaren* Konstituenten von d_A unifizieren würde. Um diesen implizit formulierten Teil–DAG zu erhalten, muß d_A während des Suchprozesses entsprechend *expandiert* werden. Der Expansionsschritt in *dag–search* wird mit Hilfe von POPEL–GRAM durchgeführt.

Der Algorithmus von *dag–search* (vgl. Abb. 4.1, S. 73) ist eine Variante von *graphsearch*, einem elementaren Suchverfahren in der Künstlichen Intelligenz [?]. Viele andere wichtige Suchverfahren, wie z.B. A, A^* oder Alpha–Beta–Verfahren basieren auf diesem Verfahren. Bevor in Abschnitt 4.2.2 die wesentlichen Schritte des Algorithmus von *dag–search* im Detail beschrieben werden, folgt daher eine kurze Charakterisierung von *graphsearch*.

Ausgangspunkt für die systematische Suche mit *graphsearch* ist die Darstellung der möglichen Lösungszustände der Ausgangsstruktur in einem gerichteten *Zustandssgraphen*. Dieser Zustandsgraph wird auch als *Suchraum* bezeichnet. Jeder Knoten des Graphen repräsentiert einen möglichen Lösungszustand. Das Ziel von *graphsearch* ist es, denjenigen Knoten zu finden, der a.) eine Lösung repräsentiert und b.) dessen Pfad zum Wurzelknoten bzgl. der Kosten einer Kante minimal ist. Zu diesem Zweck ist jeder Knoten mit seinem Vorgängerknoten über einen Zeiger verbunden.

Da der Suchraum je nach Problemstellung enorm groß sein kann, liegt der Zustandsgraph in den meisten Fällen nur implizit vor, d.h. seine Knoten und Kanten werden erst während des Suchprozesses erzeugt. Nach dem Expansionsschritt wird für jeden neuen Knoten der minimale Pfad zur Wurzel neu berechnet. Die Auswahl des als nächsten zu expandierenden Knotens im Zustandsgraph hängt von der Suchstrategie ab. Wird der Knoten mit geringster Tiefe ausgewählt, so verfolgt *graphsearch* eine *Breitensuche* und bei Auswahl des Knotens mit größter Tiefe *Tiefensuche*. Will man den "aussichtsreichsten" Knoten expandieren, so benötigt *graphsearch* heuristische Information.

4.2.2 Der Algorithmus von dag–search

In Abschnitt 4.2.1 sind die Ausgangsstrukturen von *dag–search* – d_A und d_I – bereits eingeführt worden. Zusätzlich wird mit e externes Wissen spezifiziert, das während des Suchprozesses eingesetzt wird, um die Größe des Suchraums zu beschränken. Die mit e verbundene Information bezieht sich auf die Kategorie des Teil–DAGs von d_A (bzw. auf den Wert des Merkmals *cat*), unter dem sich der DAG d_T befindet, mit dem d_I auf Kompatibilität überprüft werden soll. Dabei ist es möglich, daß e bereits die Kategorie des gesuchten Teil–DAGs spezifiziert (Dies ist z.B. der Fall, wenn in obigem Beispiel e den Wert n besitzt.). Das entsprechende Wissen und der Wert für e wird von den Operationen zur Verfügung gestellt, die *dag–search* als Basisoperation einsetzen (s. Abschnitt 5.2).

Ausgangspunkt für den Suchprozeß ist also d_A. Dort wird ein Teil–DAG gesucht, der mit d_I kompatibel ist, wobei durch den Wert von e der Suchraum eingeschränkt wird. Ist der Suchprozeß erfolgreich, liefert *dag–search* d_A inklusive dessen ensprechend modifizierten Teil–DAG. Die Abbildung 4.2.2 (S.73) zeigt den Aufbau des Algorithmus *dag–search*.

dag–search(d_A, d_I, e):

begin

1. Erzeuge einen Suchbaum T und eine Schlange Q über den Blättern von T;

2. **loop: if** Q ist leer **then exit from loop with** FAIL **else**

    3. Entferne das erste Element aus Q. Nenne das Element q;

    4. **if** $d(q)$ und d_I sind kompatibel **then**
    unifiziere $d(q)$ mit d_I;
    exit from loop with Zielknoten q **fi**

    5. Expandiere q. Füge die neuen Elemente als Blätter in T ein und als neue Elemente in Q;

    6. Sortiere Q mit Hilfe von e um **fi**

7. **go loop**

8. **if** q ist Zielknoten **then**
propagiere Veränderungen von $d(q)$ (s. 4.) durch
fortschreitende Unifikation in Richtung Wurzelknoten von T
fi

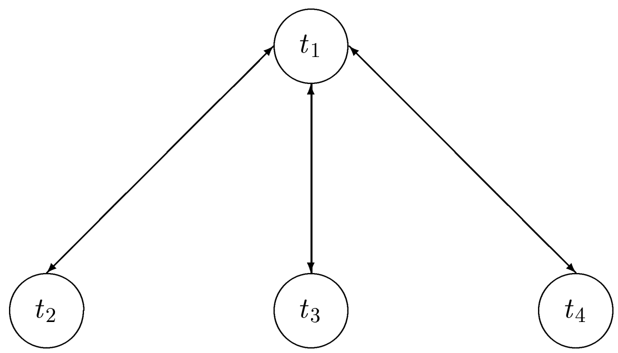
end

Figure 4.1: Grobstruktur des Algorithmus von *dag–search*

In der ersten Phase des Verfahrens (Schritt 1.) werden der Suchbaum T und die Schlange Q erzeugt und initialisiert. Für die Initialisierung von T wird d_A herangezogen. Abb. 4.2 zeigt den initialen Suchbaum T für den DAG $d(t_1)$[9]:

[9]Die DAGs in diesem Beispiel und in den nächsten werden ohne ihre Merkmalsbeschreibung gezeigt, da es für das Verständnis nicht auf ihre vollständige Struktur ankommt.

$$d(t_1) = \begin{bmatrix} 0: & \begin{bmatrix} cat: & vcomp \end{bmatrix} \\ 1: & \begin{bmatrix} cat: & v \end{bmatrix} \\ 2: & \begin{bmatrix} cat: & e0 \end{bmatrix} \\ 3: & \begin{bmatrix} cat: & e1 \end{bmatrix} \end{bmatrix}$$



$num(t_2) = 1$ $\qquad$ $num(t_3) = 2$ $\qquad$ $num(t_4) = 3$

$d(t_2) = \begin{bmatrix} cat: & v \end{bmatrix}$ $\qquad$ $d(t_3) = \begin{bmatrix} cat: & e0 \end{bmatrix}$ $\qquad$ $d(t_4) = \begin{bmatrix} cat: & e1 \end{bmatrix}$

Figure 4.2: Beipiel für einen Suchbaum.

Jeder Knoten t_i von T (außer dem Wurzelknoten) enthält einen Tochter–DAG des DAGs vom Vorgängerknoten. Zum Beispiel enthält t_3 den Tochter–DAG

$$\begin{bmatrix} cat: & e1 \end{bmatrix}$$

vom Knoten t_1. Aus Zweckmäßigkeitsgründen ist das numerische Attribut des Tochter–DAGs und der entsprechende Wert getrennt abgelegt. So bezeichnet $num(t_i)$ das numerische Attribut eines Tochter–DAGs und $d(t_i)$ dessen Wert.

Je zwei Knoten t_i und t_{i+1}, wobei t_{i+1} Nachfolgerknoten von t_i ist, sind *bidirektional* miteinander verbunden. Damit wird es möglich, in Schritt 8. des Algorithmus (s.u.) die relevante Information des Zielknotens in Richtung des Wurzelknotens zu propagieren.

Während der Initialisierung von T wird bereits das durch e zur Verfügung gestellte Wissen eingesetzt: Es werden nur für die Tochter–DAGs von d_A Knoten erzeugt, für die gilt:

$$\text{wert}(cat) = \text{wert}(e).$$

Hat e z.B. den Wert $e0$, so besitzt die Wurzel des initialen Suchbaums in Abbildung 4.2 nur den Tochterknoten t_3. Damit kann e bereits im Initialisierungsschritt den Suchraum erheblich einschränken.

Jeder Blattknoten von T ist Kandidat für einen Zielknoten (s.u.). Die Blätter werden daher in eine Schlange Q eingefügt, wodurch eine Abarbeitungsreihenfolge festgelegt wird. Für den Suchbaum im obigen Beispiel wird Q wie folgt initialisiert:

$$\mathrm{Q} ::= (t_2\ t_3\ t_4)$$

Ist die Initialisierungsphase beendet, kann mit dem eigentlichen Suchprozeß begonnen werden (Schritt 2.). Die Elemente von Q werden der Reihe nach überprüft. Ein Element q ist Zielknoten, wenn der DAG von q mit d_I kompatibel ist (Schritt 4.).

Ist q nicht Zielknoten, so ist das nächste Element von Q Kandidat für einen Zielknoten. Bevor dieses Element aus der Schlange entfernt und überprüft wird, muß zuvor q expandiert werden und die neuen Elemente aus dem Expansionsschritt (s.u.) in den Suchbaum T sowie in die Schlange Q eingefügt werden.

Grundlage für den Expansionsschritt (Schritt 5.) sind die ID–Regeln von POPEL–GRAM, die bereits in entsprechende DAGs (s. Abschnitt 4.1.2) übersetzt sind. Sie werden im weiteren *ID–DAGs* genannt. Alle ID–DAGs, die mit $d(q)$ kompatibel sind, werden mit diesem DAG unifiziert. Da $d(q)$ der DAG einer Teilkonstituente des Vorgängerknotens ist (s. o.), beschreiben die einzelnen ID–DAGs alternative Ausprägungen der Konstituenten. Gibt es nur einen ID–DAG, so war q bereits sehr genau spezifiziert oder es existiert nur eine Beschreibung für die entsprechende Konstituente in der Grammatik. Der Aufwand des Suchprozesses und die Größe des Suchraums hängt demnach von der Spezifikation des DAGs des Zielknotenkandidaten und von der Anzahl der ID–Regeln ab.

Die mit $d(q)$ unifizierten ID–DAGs D_{ID} sind Ausgangspunkt für den Expansionsschritt. Ähnlich, wie in der Initialisierungsphase von T (s. o.), wird für jeden $d_{ID} \in D_{ID}$ ein Teilbaum der Tiefe 1 erzeugt. Jeder dieser Teilbäume stellt die Expansion von q dar. Er wird daher in den bisherigen Suchbaum eingefügt. Die Wurzeln der neuen Teilbäume von T sind Schwesterknoten von q[10].

Der Expansionsschritt macht deutlich, warum in *dag–search* kein Suchgraph sondern ein Suchbaum erzeugt wird: Grundlage für die Struktur des Suchbaums ist die Konstituentenstruktur von POPEL–GRAM.

Die Expansion des Knotens t_3 (s. Abb. 4.2) liefert z.B. den Teilbaum[11] (vgl. Abb. 4.3): Die Struktur dieses Teilbaumes spiegelt die Konstituentenstruktur der Regel

(e0 np)

wider, die für die Expansion des Knotens angewendet wurde.

Abbildung 4.4 zeigt den Suchbaum T aus Abb. 4.2 nach dem Einfügen des neuen Teilbaumes.

Wird in einem nächsten Expansionsschritt der Blattknoten *np* expandiert, so zeigt der entsprechend modifizierte Suchbaum (s. Abb. 4.5) alle zu diesem Zeitpunkt möglichen Konstituentenstrukturen der Ausgangsstruktur d_A (wobei angenommen wird, daß drei *np*–Regeln "feuerten").

[10] q kann nun aus T entfernt werden. Dies hat aber keinen Einfluß auf die Größe des Suchraums, da q bereits aus der Schlange Q entfernt ist und nur deren Elemente den Suchraum aufspannen.

[11] Um die Darstellung der folgenden Teilbäume übersichtlicher zu gestalten, sind die Knoten mit den Kategoriebezeichnungen der entsprechenden DAGs gekennzeichnet.

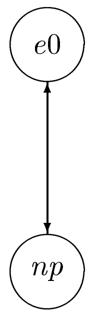


Figure 4.3: Teilbaum für die Regel (e0 np)

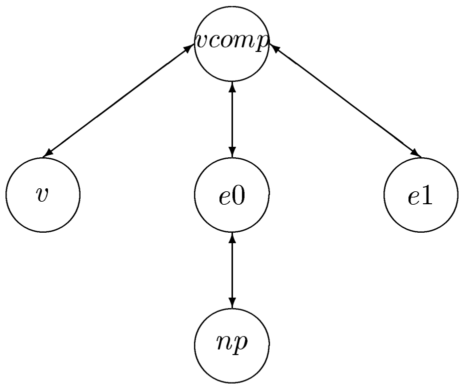


Figure 4.4: Expansion des Teilbaumes aus Abb. 4.2

Die Blätter der neuen Teilbäume von T sind Kandidaten für den Zielknoten. Sie werden daher in die Schlange Q aufgenommen.

Wenn die Elemente der Schlange nicht umsortiert werden, ist das nächste zu untersuchende Element der Schlange entweder ein Schwesterknoten von q oder ein Blattknoten der nächsten Stufe. Aufgrund der Abarbeitungsreihenfolge einer Schlange (*First In First Out*) wird also immer ein Knoten mit geringster Tiefe ausgewählt; *dag–search* realisiert in diesem Fall *Breitensuche*. Obwohl dadurch garantiert werden kann, daß ein Zielknoten (falls es einen gibt) immer über den kürzesten Pfad erreicht wird, ist keine Möglichkeit gegeben, den Suchraum vernünftig einzuschränken.

Aus diesem Grunde werden die Elemente der Schlange nach dem Expansionsschritt neu sortiert[12]. Für diesen Sortiervorgang wird das durch e extern spezifizierte Wissen

[12]Wenn die Elemente umsortiert wurden, verliert Q die Eigenschaft einer Schlange. Ich werde daher im weiteren Q als Liste bezeichnen. Es gilt aber weiterhin, daß das erste Element als nächstes ausgewählt wird.

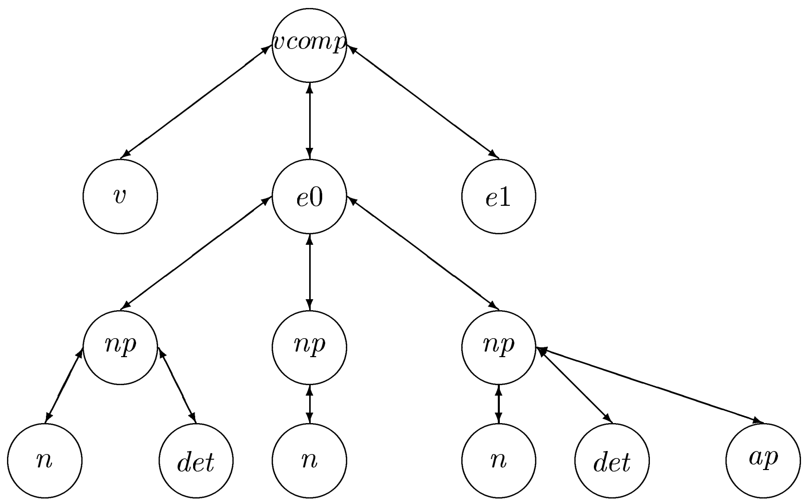


Figure 4.5: Expansion des Teilbaumes aus Abb. 4.4

herangezogen: Alle Elemente von Q, deren DAGs die Information in e erfüllen, werden an den Anfang der Liste gesetzt. Damit wird festgelegt, daß die Knoten, die am wahrscheinlichsten zum Ziel führen, auch als erste untersucht werden.

Bei diesem Sortiervorgang ist zu beachten, daß sowohl die Knoten, deren DAGs mit dem Wert von e übereinstimmen, wie auch die übrigen Knoten relativ zueinander in der Reihenfolge stehen bleiben, wie sie in die Liste eingefügt wurden. Dies erlaubt es, die Breitensuchestrategie partiell zu erhalten.

Soll z.B. die Liste

$$\mathrm{Q} ::= (t_2\ t_3\ t_4\ t_5\ t_6\ t_7),$$

so sortiert werden, daß t_6 und t_7 als die wahrscheinlichsten Kandidaten am Anfang stehen, wobei der Index der Elemente die zeitliche Reihenfolge des Einfügens widerspiegelt, so hat die Liste nach dem Sortiervorgang folgende Struktur:

$$\mathrm{Q} ::= (t_6\ t_7\ t_2\ t_3\ t_4\ t_5),$$

Das Sortieren der Liste und damit die Aufgabe der Breitensuche über alle Elemente der Schlange hat allerdings einen großen Nachteil, der vor allem bei rekursiven Strukturen auftritt: Es ist möglich, daß der Algorithmus nicht terminiert, obwohl ein Zielknoten existiert.

Befinden sich zum Beispiel in einer Liste zwei Knoten t_i und t_{i+1} und sei $d(t_i)$ durch die Regel:

$$\text{(ap adj ap)}$$

spezifiziert worden und $d(t_{i+1})$ durch:

$$\text{(np n ap)} \ .$$

Wenn sich die Information in e auf die Kategorie der ap–Regel bezieht (d.h. wert$(e)=ap$), so werden solange neue Knoten (mit entsprechenden ap–DAGs) in den Suchbaum eingehängt und in die Liste an den Anfang gesetzt, bis ein kompatibler DAG unter diesem Pfad gefunden wird. Es ist aber möglich, daß solch ein DAG nicht durch Expansion dieser Regel gefunden werden kann, sondern durch Expansion von t_{i+1}. Da sich die Information von e nicht ändert, wird dieser Knoten niemals expandiert werden.

Die Information von e führt in diesem Fall (im Gegensatz zu ihrem Zweck) dazu, daß der Suchraum nicht beschränkt werden kann. Um solche "Irrwege" zu vermeiden, wird in *dag–search* eine untere Schranke angegeben, mit der die maximale Tiefe des Suchbaums begrenzt wird. Wird diese Schranke erreicht, ohne das ein Zielknoten bestimmt wird, wird der Expansionsschritt abgebrochen und, ohne neue Elemente in die Schlange einzufügen, mit Schritt 2. weitergemacht.

Der Wert dieser Schranke hängt von der Struktur der Grammatik ab. Bei einer Grammatik, die flache Strukturen aufbaut (wie z.B. POPEL–GRAM), ist der Wert geringer als bei einer Grammatik, die sehr tief geschachtelte Strukturen erzeugt.

Implementationshinweis:

In POPEL–HOW wird die Schranke durch die globale Variable *max–depth–bound* repräsentiert. Zur Zeit ist die Variable mit dem Wert 10 vorbesetzt. Er kann aber vom Benutzer neu definiert werden.

Findet *dag–search* einen Zielknoten q, so erhält $d(q)$ durch Unifikation mit d_I neue Information (vgl. Schritt 4. in Abb. 4.1). In Schritt 8. werden die Veränderungen von $d(q)$ entlang des Pfades:

$$q \Rightarrow t_1$$

in Richtung des Wurzelknotens t_1 *propagiert.* Grundlage für diesen Schritt ist die *bidirektionale* Verbindung der Knoten des Suchbaumes, die im Initialisierungsschritt und während der Expansion von Knoten aufgebaut wird. Schritt 8. läßt sich dann im Detail wie folgt beschreiben:

loop

8.1. **if** Wurzel t_1 erreicht **then**
  if d_T existiert **then**
  return $d(t_1)$ und d_T **fi**
 else return FAIL fi

8.2. **if** not(d_T) & cat($d(q)$) $= e$ **then**
 $d_T ::= d(q)$ **fi**

8.3. bestimme den Teil–DAG d_{sub} vom Vorgänger von q,
 dessen Wurzel gleich ist mit $num(q)$

 $d(q)$ ::= unifiziere d_{sub} mit $d(q)$
 q::= Vorgänger von q

go loop

Im wesentlichen wird die neue Information durch sukzessive Unifikation in Schritt 8.3. in Richtung Wurzel transportiert. $d(t_1)$ (vgl. 8.1.) entspricht dem modifizierten DAG d'_A der Ausgangsstruktur d_A, mit der der Suchbaum T initialisiert wurde (vgl. Abschnitt 4.2.1). Der DAG d_T repräsentiert die Konstituente der Phrase von d_A, unter der die Veränderung stattgefunden hat. Diese wurde ja mit e extern bestimmt.

Es wäre möglich, nicht den gesamten Pfad bis zur Wurzel zu traversieren, sondern an dem Knoten den Propagierungsschritt zu beenden, in dessen DAG keine Veränderungen mehr stattfinden. Dafür wird eine Relation benötigt, mit der festgestellt werden kann, daß ein DAG von einem anderen DAG *echt subsumiert* wird. Mit Hilfe dieser Relation kann dann festgestellt werden, ob ein DAG durch Unifikation verändert wurde.

Chapter 5

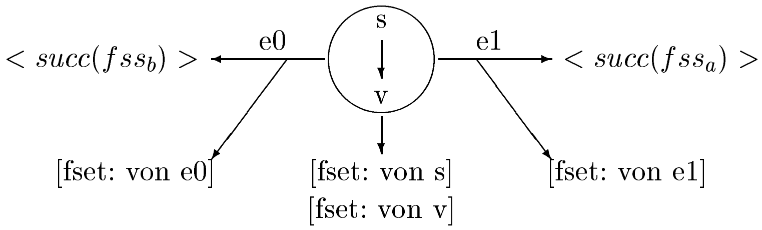
Die dependenzbasierte Strukturebene DBS

In diesem Kapitel wird der parallele und inkrementelle Aufbau der syntaktischen Struktur auf der DBS–Ebene beschrieben. Ausgangsstruktur sind die durch den verteilten Abbildungsprozeß $FSS \longrightarrow DBS$ erzeugten syntaktischen Objekte. Jedes Objekt repräsentiert die externe Struktur einer Phrase (vgl. Abschnitt 2.6.1). Um sie erfolgreich in die nächste Ebene abbilden zu können, muß die explizite syntaktische Struktur bestimmt werden. Nachdem nun mit POPEL–GRAM der zugrundeliegende Formalismus für die DBS–Ebene bekannt ist, zeige ich zunächst die Verteilung des syntaktischen Wissens auf die Objektstruktur der DBS–Ebene. Damit wird es möglich, die externe Darstellung eines DBS–Objektes, die in Abschnitt 2.4 nur angedeutet wurde, genauer zu spezifizieren.

5.1 Die Verteilung syntaktischer Strukturen im parallelen Modell

Die ID–Regeln bzw. ID–DAGs beschreiben die unmittelbare Konstituentenstruktur einer Phrase und formulieren explizit die syntaktischen Beziehungen der Konstituenten untereinander. Es ist naheliegend, jeden ID–DAG als mögliches syntaktisches Segment zu erklären und es mit einem Objekt des parallelen Modells in Beziehung zu bringen. Allerdings kann die Beziehung zwischen der Objektstruktur und der Struktur des ID–DAGs nicht so direkt sein, wie z.B. die Beziehung zwischen FSS–Objekt und FSS–Individualisierung (s. Abschnitt 2.3), da ID–DAGs im allgemeinen durch komplexe Graphen beschrieben werden. Der Rahmenkontext eines DBS–Objektes kann somit nicht allein für die Repräsentation des syntaktischen Segments herangezogen werden. Ein Objekt erhält daher einen Zeiger auf seinen ihm zugewiesenen ID–DAG. Besitzt POPEL–GRAM mehrere ID–DAGs für eine Phrase, werden alle Alternativen in einer Menge gesammelt und dem Objekt zugewiesen.

Die Beziehung zwischen Objekt und DAG läßt sich wie folgt darstellen (wobei angenommen ist, daß keine alternativen ID–Regeln für die Phrase existieren):



In dieser Darstellung ist der dem Objekt zugewiesene ID–DAG in seine Konstituenten und zugehörige Merkmalsbeschreibungen zerlegt und auf die Objektstruktur verteilt worden. Die Beziehung zwischen Objekt und ID–DAG läßt sich dann wie folgt beschreiben: Ein DBS–Objekt repräsentiert die Vaterkonstituente bzw. den Vater–DAG und – da die Vaterkonstituente die Projektion des head–Elementes ist – das head–Element bzw. den head–DAG einer Phrase. Der kontextuelle Rahmen des Objekts repräsentiert den Dependenzrahmen des ID–DAGs. Den Konstituenten sind ihre jeweiligen Merkmalsbeschreibungen zugewiesen.

Diese Darstellung verdeutlicht, daß die Konstituentenstruktur einer ID–Regel als Verbindung zwischen Objekt und Merkmalsstruktur fungiert[1]. Da der kontextuelle Rahmen eines DBS–Objektes den Dependenzrahmen einer ID–Regel widerspiegelt, stellt das Objekt quasi einen *lokalen Dependenzbaum* dar. Objekte, die über den Rahmenkontext eine Kommunikationsverbindung aufbauen, werden daher auch als *Dependenten* bezeichnet und das Objekt selbst als *Regent*. Die Gesamtheit miteinander kommunizierender DBS–Objekte stellen einen *aktiven Dependenzbaum* dar. Bezieht man noch die Merkmalsstruktur ein, so charakterisieren miteinander verknüpfte DBS–Objekte eine aktive komplexe syntaktische Struktur.

DBS–Objekte werden von FSS–Objekten durch Regelanwendung erzeugt. Die Abbildungsregeln $FSS \longrightarrow DBS$ konstituieren die Objektstruktur (der Wortwahlprozeß liefert das head–Element). Damit die ID–Regeln (im speziellen die Merkmalsbeschreibungen) sinnvoll mit den Objekten in Beziehung gebracht werden können, müssen die Regeln mit der Konstituentenstruktur von POPEL–GRAM widerspruchsfrei in Beziehung stehen, d.h. der *kontextuelle* Rahmen eines Objektes muß mit dem *dependentiellen* Rahmen einer ID–Regel übereinstimmen und das Objekt selbst mit der Vaterkonstituenten und dem head–Element.

Damit legen aber die Abbildungsregeln fest, wie feinkörnig die syntaktische Struktur zu segmentieren und zu verteilen ist. Zur Zeit werden durch die Abbildungsregeln $FSS \longrightarrow DBS$ in POPEL–HOW Objekte auf der DBS–Ebene erzeugt, die (isoliert betrachtet) Satz–, Nominal– und Adjektivphrasen repräsentieren. Das bedeutet z.B., daß Determinativphrasen nicht durch eigenständige Objekte vertreten sind[2]. Dies ist beim Formulieren der ID– und LP–Regeln zu beachten (s.a. Abschnitt 4.1.2 und Abschnitt 4.1.4).

[1] Allerdings kann diese Darstellung in der Implementation nicht nachgebildet werden, da die DAGs der Phrasen in SB–PATR immer als Einheit betrachtet werden müssen. Aus diesem Grund sind die ID–DAGs über einen Zeiger mit dem Objekt verknüpft.

[2] Obwohl dies natürlich in POPEL–HOW möglich ist. Es ist allein eine Folge der Formulierung der Abbildungsregeln.

5.2 Der Lebenszyklus der DBS–Objekte

Grundlage für den parallelen und inkrementellen Konstruktionsprozeß ist das in Abschnitt 2.2 vorgestellte parallele Basismodell. Die Objekte der DBS–Ebene verfügen über die gleiche Funktionalität innerhalb des gesamten Systems, wie die Objekte der CKB– und FSS–Ebene. Die DBS–Ebene kann daher homogen in POPEL–HOW integriert werden.

Implementationshinweis:

Ein DBS–Objekt ist eine Instanz des Flavors **ckb–object**. Dieses Flavor hat als Mixin das Flavor **process** (s. [?]). Ein DBS–Objekt hat als zusätzliche Instanzvariablen:

1. **dag-category**
2. **dags**
3. **dag–changed–p**
4. **max–index**
5. **outer–object**
6. **word–choice**
7. **rule**

Diese Variablen werden zur Bearbeitung der DAGs eines Objektes benötigt.

Um die mit ihnen assoziierten DAGs verwalten und bearbeiten zu können, benötigt ein DBS–Objekt zusätzliche Operationen. Die Operationen müssen im wesentlichen folgendes leisten:

1. Aufbau der objekteigenen DAG–Struktur
2. Transport von Merkmalsspezifikationen
3. Verknüpfung von Teilstrukturen zu größeren Strukturen.

Unter Einbeziehung dieser zusätzlichen Operationen läßt sich der Lebenszyklus für ein DBS–Objekt wie folgt beschreiben:

begin
1. Initialisierung, dabei Aufbau der DAG–Struktur
   do forever:
   2. Bekanntmachung mit neuen Objekten
   3. Wenn möglich, Aufbau von Kommunikationsverbindungen zu diesen Objekten, dabei Austausch von Merkmalsinformation
   4. Wenn die DAG–Struktur lokal vollständig ist, Abbildung in die ILS–Ebene
   5. Versenden oder Bearbeiten von Anfragen bzgl. unterspezifizierter DAG–Struktur
   6. Überprüfung, ob neue Objekte eingetroffen sind

   od

end

Während des Initialisierungschrittes konstruiert ein DBS–Objekt mit Hilfe der Phrasenbezeichnung, den Lexemen und der relevanten ID–Regel von POPEL–GRAM seinen DAG. Sind mehrere ID–Regeln verfügbar, werden entsprechend viele initiale DAGs aufgebaut. Da dies der allgemeine Fall ist, gehe ich im weiteren davon aus, daß ein DBS–Objekt eine Liste von DAGs besitzt. Die initialen DAGs beschreiben möglicherweise noch nicht die vollständige syntaktische Struktur. Jedes DBS–Objekt benötigt ein *lokales Vollständigkeitskriterium*, um dieses für sich festzustellen. Die fehlende Information für unterspezifizierte Merkmale kann ein DBS–Objekt von seinen unmittelbaren Dependenten bekommen, mit denen es in gleicher Weise die Kommunikationsverbindung etabliert, wie die Objekte der anderen Ebenen. Der notwendige Merkmalstransport wird durch Unifikation der beteiligten DAGs realisiert.

Fehlt ein Dependent und mit ihm der DAG im Kontext, muß das DBS–Objekt dieses fehlende Objekt anfordern. Ist der DAG eines Objektes lokal vollständig, kann es seine zentrale Aufgabe – die Abbildung – lösen. Da das gesamte syntaktische Wissen eines DBS–Objektes in seinen DAGs formuliert ist, sind sie Ausgangspunkt für den Abbildungsprozeß.

Es ist möglich, daß auf der DBS–Ebene Objekte aktiv sind, die vom selben FSS–Objekt erzeugt wurden. Die Objekte unterscheiden sich dadurch, daß sie zu unterschiedlichen Zeitpunkten mit unterschiedlichen Regeln aufgebaut wurden. Ursache hierfür ist im wesentlichen das inkrementelle Eingabeverhalten und die nicht notwendigerweise eindeutigen Regeln in POPEL–HOW.

Das durch erneute Regelanwendung eines FSS–Objektes erzeugte DBS–Objekt ist Konkurrent zu dem bereits existierenden DBS–Objekt. Es repräsentiert entweder eine speziellere Beschreibung oder stellt eine syntaktische Umformulierung dar. DBS–Objekte, die solche Objekte in ihrem Kontext aufnehmen, müssen daher in der Lage sein, *konkurriende* DAG–Beschreibungen bearbeiten zu können.

Die einzelnen Phasen des Lebenszyklus werden nun in den nächsten Abschnitten im Detail erläutert.

5.3 Aufbau der initialen DAG–Struktur von DBS–Objekten

Ausgangspunkt für den Aufbau der initialen DAG–Struktur eines DBS–Objektes ist dessen Wortwahlergebnis und die durch die Abbildungsregel angegebene externe Phrasenstruktur. Die wesentlichen Schritte in dieser Phase sind

- Übersetzung des Wortwahlergebnisses in lexikalische DAGs,
- Bestimmung der relevanten ID–DAGs,
- Integration der lexikalischen DAGs mit den ID–DAGs und
- Identifizierung alternativer ID–DAGs.

Die Übersetzung des Wortwahlergebnisses in entsprechende lexikalische DAGs ist bereits in Abschnitt 4.1.5 erläutert. Für die Bestimmung der relevanten ID–DAGs wird mit Hilfe des Kategoriebezeichners der Vaterkonstituente ein *Such–DAG* erzeugt. Alle ID–DAGs von POPEL–GRAM, die mit dem Such–DAG kompatibel sind, werden als mögliche Kanditaten in der Menge D_{init} gesammelt.

$$\begin{bmatrix} \mathit{cat}: & \mathit{np} \end{bmatrix}$$

ist z.B. der Such–DAG, mit dem alle Nominalphrasen und deren ID–DAGs aus POPEL–GRAM extrahiert werden. In der Regel werden in diesem Schritt mehrere DAGs gleicher Kategorie als mögliche Kandidaten bestimmt, da es für eine Phrase mehrere Strukturierungsalternativen gibt, die aber bis zu diesem Zeitpunkt noch nicht näher spezifiziert werden können. Es handelt sich also in erster Linie um eine Vorauswahl der in Frage kommenden syntaktischen Segmente.

Im nächsten Schritt werden die lexikalischen DAGs in die ID–DAGs aus D_{init} integriert. Als Basisoperation wird in diesem Schritt *dag–search* eingesetzt. Zuerst werden alle $d_{init} \in D_{init}$ mit dem lexikalischen DAG des head–Lexems d_{head} in Beziehung gebracht. Der entsprechende Aufruf von *dag–search* lautet

(dag–search d_{init} d_{head} cat_{head})

Da das head–Lexem als zentrales Element der Phrase deren wesentliche grammatikalische Information determiniert, wird die Menge D_{init} durch die Unifikation mit d_{head} eingeschränkt. Ob in diesem Schritt die Anzahl der möglichen Kandidaten auf genau einen reduziert werden kann, was eine eindeutige syntaktische Struktur bedeuten würde, hängt von der Genauigkeit der aktuellen Merkmalsbeschreibung von d_{head} ab. Wenn zum Beispiel für ein verbales head–Lexem zu diesem Zeitpunkt der Wert des Merkmals *voice* noch nicht spezifiziert worden ist, muß die Aktiv– oder Passivkonstruktion der entsprechenden Satzphrase gleichberechtigt weiterbehandelt werden. In diesem Sinne steuert die lexikalische Information den Auswahlprozeß der relevanten DAGs eines DBS–Objektes.

Die durch das head–Lexem modifizierten DAGs D'_{init} werden anschließend mit den modifier–Lexemen (vgl. Abschnitt 2.5) strukturell in Beziehung gebracht (durch wiederholte Anwendung der Operation *dag–search*). Nach der Integration der modifier–Lexeme bzw. der entsprechenden DAGs ist im Prinzip die Initialisierungsphase für ein DBS–Objekt beendet. Es ist jedoch möglich, daß das DBS–Objekt zu diesem Zeitpunkt noch über mehrere initiale DAGs verfügt (s.o.). Um diese im weiteren Verlauf des Konstruktionsprozesses individuell unterscheiden zu können, vergibt das DBS–Objekt an jeden initialen DAG einen eindeutigen *Index*. Das Objekt erzeugt für jeden DAG einen *Index–DAG* der Form

$$\begin{bmatrix} \mathit{index}: & \mathit{number} \end{bmatrix}$$

mit *number* $\in$ N und fügt ihn durch Unifikation dem entsprechenden DAG hinzu. Dabei muß darauf geachtet werden, daß für beliebige initiale DAGs des Objektes d_i und d_j mit i, j $\in$ N die Werte von *index* paarweise verschieden sind.

Das Merkmal *index* ist ein technisches Merkmal, d.h. es wird intern zur Verwaltung der DAGs eines DBS–Objektes benötigt[3]. Es sollte daher von den eigentlichen grammatikalischen Merkmalen unterschieden werden. In POPEL–HOW wird aus diesen Gründen jedem DAG, der einem DBS–Objekt zugewiesen wird, das spezielle Merkmal *process* hinzugefügt. Unter diesem Merkmal wird dann das Merkmal *index* mit entsprechendem Wert abgelegt. Hierdurch wird es strukturell im DAG von Merkmalen wie *head* oder *surface* klar getrennt.

[3]Technische Attribute, wie z.B. *label* oder *arity*, werden auch in D–PATR verwendet, um das Parsen eines Satzes zu kontrollieren (vgl. [?]).

Implementationshinweis:

Der Initialisierungsschritt für ein DBS–Objekt ist als Methode **:create–initial–dag** für Prozeßobjekte vom Typ **dbs–object** definiert. Diese Methode benötigt keine Argumente, da ein DBS–Objekt auf alle notwendigen Daten selbst zugreifen kann.

5.4 Der Merkmalstransport zwischen kommunizierenden DBS–Objekten

5.4.1 Die Ursachen für unspezifizierte Merkmalsbeschreibungen

Aufgrund der existierenden Kongruenzbeziehungen zwischen verschiedenen Teilstrukturen, die im Spezifikationsteil der ID–Regeln von POPEL–GRAM angegeben sind, bestehen zwischen Objekten, die solche Teilstrukturen zugewiesen bekamen, Abhängigkeiten hinsichtlich syntaktischer Merkmale. Geht man davon aus, daß während der Initialisierungsphase die Werte von kongruenten Merkmalen nur einem Objekt bzw. seinem syntaktischen Segment zugewiesen werden, so verfügt das mit diesem Objekt kongruente Objekt nach der Initialisierung entsprechende unspezifizierte Merkmale. Zum Beispiel existiert zwischen dem verbalen Element einer Satzphrase und dem Subjekt Kongruenz zwischen den Merkmalen *numerus* und *person*. Die Belegung dieser Merkmale wird von der Phrase, die die Subjektfunktion übernimmt, durchgeführt. Damit hat aber ein Satz–Objekt nach seiner Initialisierungsphase unspezifizierte Merkmale.

Ein anderes Beispiel ist die Kasuszuweisung bei Nominalphrasen. Es ist nicht sinnvoll, einer Nominalphrase, die isoliert betrachtet wird, einen Kasus zuzuweisen. Dieser ist abhängig von der Funktion, die die Phrase in einer komplexen Struktur (z.B. einem Satz) einnimmt (s.a. Abschnitt 3.1.3). Da das regierende Element die Funktion seiner abhängigen Elemente festlegt, spezifiziert es auch den jeweiligen Kasus. In diesem Sinne existiert Kasuskongruenz zwischen dem regierenden Element und seinen abhängigen Elementen. Für ein DBS–Objekt, das eine Nominalphrase repräsentiert, hat dies zur Folge, daß das Merkmal *case* nach der Initialisierungsphase unspezifiziert ist.

In beiden Fällen werden die Werte für die unspezifizierten Merkmale zur weiteren Verarbeitung (u.a. Flexion) benötigt. Im Prinzip kann dieser Transport von Merkmalen durch Unifikation der betroffenen DAGs realisiert werden. In POPEL–HOW sind aber die Träger von kongruenten Strukturen parallel und unabhängig arbeitende Objekte. Daher ist der Merkmalstransport nicht durch "einfaches" unifizieren zu realisieren.

5.4.2 Der Merkmaltransport in einem parallelen Modell

Der Merkmaltransport kann nur zwischen miteinander kommunizierenden Objekten stattfinden. Für zwei miteinander in Beziehung stehende Objekte gilt das Verhältnis der Dependenz, d.h. ein Objekt O_r *regiert* das andere Objekt O_d (s. Abschnitt 5.1). Da der kontextuelle Rahmen von O_r den Dependenzrahmen des syntaktischen Segmentes repräsentiert, wird durch die kontextuelle Bindung von O_d dessen syntaktische Funktion bestimmt.

Die miteinander in Verbindung stehenden Objekte sind nun in der Lage, den Merkmalstransport durchzuführen. Um den Austausch von syntaktischer Information zu ermöglichen, werden die DAGs von O_r und O_d miteinander *unifiziert*. Aufgrund bestehender Gleichungen kann dann Information von einem DAG zum anderen fließen. Für die Objekte bedeutet die Unifikation ihrer DAGs die Vervollständigung des syntaktischen Segmentes und die Übermittelung syntaktischer Information an den beteiligten Partner. Weiterhin gilt, daß die DAGs beider Objekte gemeinsam ein komplexeres syntaktisches Segment repräsentieren.

Als Basisoperation für diesen Schritt wird wiederum *dag–search* verwendet. Da jeweils O_r und O_d über alternative DAGs verfügen können, reicht ein einfacher Aufruf von *dag–search* nicht aus, sondern diese Operation muß über alle DAG–Kombinationen ausgeführt werden. Die Operation, sie wird im weiteren *splice–dags* genannt, läßt sich folgendermaßen beschreiben:

Für zwei kommunizierende Objekte O_r und O_d der DBS–Ebene
für die gilt O_r *regiert* O_d:

```
begin
        X := Y := ∅;
        N := DAGs(O_r);
        M := DAGs(O_d);
        e := Bezeichner von Kontextelement;
        forall n ∈ N do
                forall m ∈ M do
                        X := X ∪ (dag–search(n,m,e))
                    od;
                Y := Y ∪ min–depth(X);
                X := ∅
            od;
    return Y;
end
```

Implementationshinweis:
Die Operation *splice–dags* ist als Methode **:splice–dags** für Objekte vom Typ **dbs–object** definiert. Sie hat drei Parameter:

1. **outer–dags**, die DAGs des Objektes O_r
2. **providing–object**, das abhängige Element O_d
3. **entry–point**, der Einstiegspunkt für *dag–search*

Bevor die zu unifizierenden DAGs n und m der Operation *dag–search* als aktuelle Parameter übergeben werden, müssen die DAGs *kopiert* werden. Sonst kann wegen des destruktiven Verhaltens der Unifikation nicht gewährleistet werden, daß für zwei DAGs von M, m_i und m_{i+1}, n jeweils identisch ist (analog für DAGs n_i und n_{i+1}).

Ist die Operation *dag–search* für einen DAG n erfolgreich, so liefert *dag–search* für n maximal $\mid M \mid$ Paare X = (n_{neu}, m_{neu}) von DAGs. Es ist möglich, daß für zwei Paare (n_{neu_1}, m_{neu_2}) und (n_{neu_3}, m_{neu_4}) die Anzahl der notwendigen Expansionsschritte l unterschiedlich ist. Aufgrund der Dependenzrelation, die zwischen den Elementen der Paare bestehen muß, wird mit Hilfe der Operation *min–depth* dasjenige Paar von X gewählt, dessen l minimal ist[4].

Für jedes n wird also nur ein Paar aus X gewählt. Daher liefert die Operation *splice–dags* maximal $\mid N \mid$ Paare Y von DAGs (n_{neu}, m_{neu}).

Für jedes Element aus Y wird das Ergebnis der Unifikationsoperation protokolliert. Festgehalten wird:

- o_r, der Name des regierenden Objekts,

- i, der Index vom DAG n,

- l, die Anzahl der Expansionsschritte,

- *true*, *dag–search* war erfogreich.

Dieses 4–Tupel, es wird im weiteren auch *splice–info* genannt, wird dem abhängigen Objekt O_d zugewiesen. Das Objekt hat damit die Möglichkeit, zu späteren Zeitpunkten nachzuvollziehen, mit welchen DAGs von welchen regierenden Objekten es erfolgreich *splice–dags* durchführte (s. Abschnitt 6.4). Analog wird auch protokolliert, daß *splice–dags* für ein Paar (n, m) scheiterte. Das entsprechende 4–Tupel lautet dann:

$(o_r, i, \text{nil}, false)$

Nach erfolgreicher Durchführung von *splice–dags* wird für O_r eine Menge N_{neu} und für O_d eine Menge M_{neu} angelegt, in denen die modifizierten DAGs n_{neu} und m_{neu} aufgenommen werden. Dabei ist es wiederum wichtig, die DAGs vor ihrem Einfügen zu *kopieren.* Für die Kardinalität dieser Mengen gilt:

$\mid N_{neu} \mid = \mid M_{neu} \mid, \mid N_{neu} \mid \leq \mid N \mid.$

Weiterhin gilt:

$\forall\, n \in N_{neu}$: $\text{cat}(n_i) = \text{cat}(n_j)$, mit i $\neq$ j
$\forall\, m \in M_{neu}$: $\text{cat}(m_i) = \text{cat}(m_j)$, mit i $\neq$ j

Der letzte Punkt besagt, daß nach *splice–dags* sich in N_{neu} und M_{neu} nur DAGs mit gleicher Kategorie befinden.

Ist die Operation *splice–dags* für kein Paar (n, m) erfolgreich, behält jedes Objekt seine ursprünglichen DAGs. O_r und O_d konnten zwar miteinander eine Kommunikationsverbindung etablieren, es scheiterte aber der Versuch, sie syntaktisch in Beziehung zu bringen. Da es möglich ist, daß zu O_r oder O_d konkurriende Objekte in die Ebene eintreffen, werden die ursprünglichen DAGs weiterhin benötigt.

Da die DAGs während der Durchführung von *splic–dags* kopiert werden müssen, bleiben die Veränderungen lokal auf die Objekte beschränkt. Das bedeutet, daß die DAGs

[4] Bei Paaren mit gleichem l wird willkürlich das erste Paar gewählt.

der Objekte, die von O_d regiert werden und die mit O_d zu einem früheren Zeitpunkt erfolgreich ihre DAGs austauschten, die neue Information nicht erhalten können.

O_d muß also die Änderungen an seine Dependenten weiterreichen. Dabei betrachtet O_d nur die Dependenten, mit denen es über seinen Kontext in Verbindung steht und mit denen es erfolgreich die Operation *splice–dags* durchführen konnte, was für die jeweiligen Dependenten von O_d protokolliert wurde.

Die Weitergabe der neuen Information an die DAGs entsprechender Dependenten wird mit Hilfe der *Unifikation* realisiert. Da es möglich ist, daß die DAGs der Dependenten zwischenzeitlich verändert wurden, wird die Unifikation nur durchgeführt, wenn die DAGs *kompatibel* sind.

Da die Dependenten von O_d evtl. die neu erhaltene Information ebenfalls an ihre Dependenten weiterreichen, *fließt* die Modifikation der syntaktischen Struktur, die durch die lokal ausgeführte Operation *splice–dags* zwischen O_r und O_d ausgelöst wurde, dennoch über das gesamte Netz der aktiven und miteinander verbundenen DBS–Objekte an die relevanten Stellen.

Implementationshinweis:

Die Operation, die die Veränderungen eines DBS–Objekts an mittelbar betroffene Objekte weitergibt, ist als Methode **:update–dependencies** definiert. Sie hat keine formalen Parameter.

Dieses *dynamische* Verhalten der Objekte bei der Weitergabe von Änderungen in der syntaktischen Struktur ist m.M.n. gleichzusetzen mit der *destruktiven* Veränderung der syntaktischen Struktur durch die Unifikation. In diesem Sinne *simulieren* die Objekte das destruktive Verhalten.

5.5 Abbildung der DAGs von DBS–Objekten in die ILS–Ebene

5.5.1 Ein Vollständigkeitskriterium für syntaktische Strukturen

Damit sich aktive DBS–Objekte durch Abbildung ihrer DAGs in die ILS–Ebene fortpflanzen können, verfügt jedes Objekt über ein Kriterium, mit dem es entscheiden kann, ob seine DAGs lokal genügend Information codieren. Ausgangspunkt für die Definition eines *lokalen Vollständigkeitskriteriums* ist der DAG des head–Elementes d_{head} einer Phrase. Als zentrales Element ist d_{head} Grundlage für die Definition der Phrase und für die Abhängigkeitsverhältnisse der übrigen unmittelbaren Konstituenten (vgl. Abschnitt 4.1.2). Solange d_{head} unspezifizierte Merkmale besitzt, fehlt der gesamten Phrase relevante Information. Daher definiere ich das lokale Vollständigkeitskriterium wie folgt:

Definition 8
Eine Phrase ist *lokal vollständig*, wenn für alle Merkmale des DAGs des head–Elementes der Wert nicht der leere DAG [ ] ist.

Implementationshinweis:

Die Funktion **contains-null-dag-p** überprüft, ob ein DAG, der als Argument übergeben wird, ein Merkmal mit Wert [ ] besitzt (die Funktion hat keine weiteren Argumente). Ist dies der Fall, liefert die Funktion den boolschen Wert T andernfalls NIL.

Merkmale wie z.B. $surface$, die nur redundante Information beinhalten, können in der Liste *suppress–arcs–for–complete–p* aufgenommen werden. Die Elemente dieser Liste werden beim Test ignoriert, wodurch der Test insgesamt weniger aufwendig wird.

5.5.2 Die Abbildung lokal vollständiger DAGs in die ILS–Ebene

Ein DBS–Objekt muß den Test auf lokale Vollständigkeit während seiner aktiven Zeit für jeden DAG sehr oft durchführen. Um diesen Test effizient zu gestalten, besitzt jeder DAG das Attribut *complete–p*. Dieses Attribut signalisiert dem DBS–Objekt, ob der Test auf lokale Vollständigkeit bereits erfolgreich durchgeführt wurde oder ob er evtl. erneut durchgeführt werden muß. Da es sich bei *complete–p* um ein Attribut handelt, mit Hilfe dessen ein DBS–Objekt seine DAGs bzgl. des Tests verwalten kann, befindet sich das Attribut unter dem Merkmal *prozeß*.

Ist der Wert von *complete–p* '—', so wird der DAG auf lokale Vollständigkeit hin untersucht. Wenn der Test erfolgreich ist, kann der DAG in die ILS–Ebene abgebildet werden. Vorher wird der Wert des Attributes auf '+' gesetzt. Damit wird verhindert, daß bei Veränderung des Kontextes und anschließendem *splice–dags* der bereits lokal vollständige DAG erneut abgebildet wird. Dies würde in der ILS–Ebene zu unnötigen Redundanzen führen, für deren Verarbeitung entsprechende Mechansimen bereitgestellt werden müßten.

Der Abbildungsschritt von DAGs in die ILS–Eben ist recht einfach: Da sich unter dem Vater–DAG die gesamte relevante Information der Phrase befindet (einschließlich der Lexeme) reicht es aus, den Vater–DAG einer Phrase in die ILS–Ebene abzubilden. Mit der Merkmalsbeschreibung dieses DAGs können die Lexeme korrekt flektiert werden. Damit auch die Linearisierung in der ILS–Ebene korrekt durchgeführt werden kann (vgl. Abschnitt 6.2), erzeugt ein DBS–Objekt für einen lokal vollständigen DAG, den es in die ILS–Ebene verschickt, ein 5–Tupel ILS–E der Form

$$(name\ order\ regenten\ dag\ string),\ \text{mit}$$

$name$: Der Name des DBS–Objektes, zu dem der DAG gehört

$order$: Die gewünschte Position in der gesamten Äußerung

$regenten$: Eine Liste, die alle 4–Tupel *splice–info* enthält

dag: Der Vater–DAG

$string$: Ein Platzhalter für den in der ILS–Ebene zu erzeugenden Oberflächenstring

5.6 Das Anfordern fehlender syntaktischer Information

Nachdem ein DBS–Objekt O mit allen seinen bekannten Objekten die Operation *splice–dags* ausgeführt hat, ist es weiterhin möglich, daß keiner der DAGs von O lokal vollständig ist. Da O bestrebt ist, sich möglichst rasch und unabhängig von anderen Objekten der DBS–Ebene fortzupflanzen, fordert O bestimmte Objekte der übergeordneten FSS–Ebene auf, die fehlende Information bereitzustellen. Im Unterschied zur FSS–Ebene, auf der der Ausgangspunkt für das Anfordern fehlender Information der Zustand des Kontextes eines FSS–Objektes ist, bilden die DAGs von O den Ausgangspunkt für das Anfordern fehlender syntaktischer Information.

5.6.1 Ursachen für lokal unvollständige DAGs

Aufgrund der Definition des *lokalen Vollständigkeitskriteriums* sind die DAGs von O unvollständig, weil der DAG des head–Elementes d_{head} Merkmale besitzt, die zur Zeit noch keine Werte haben. Um die Fortpflanzung von O zu ermöglichen muß O wissen, wo die Werte solcher Merkmale definiert werden. Für das Lokalisieren relevanter unspezifizierter Merkmale in einem DAG d von O sind folgende Fälle zu unterscheiden:

1. d_{head} besitzt undefinierte Merkmale.
2. Die head–Elemente der DAGs, die die unvollständigen DAGs von O syntaktisch regieren, enthalten undefinierte Merkmale.
3. Die Elemente des Dependenzrahmens von d (die von d_{head} regierten unmittelbaren Konstituenten) haben unspezifizierte Merkmale.

Für das Anfordern fehlender Information ist allein der dritte Fall von Interesse. In den anderen Fällen ist es nicht möglich bzw. sinnvoll, entsprechende Objekte der übergeordneten Ebene aufzufordern, die fehlende Information bereitzustellen. Dies hat folgende Erklärung: Fehlt der Wert für ein Merkmal, das in d_{head} definiert wird (Fall 1.), so liegt die Ursache für die Unvollständigkeit von d darin, daß der lexikalische DAG des head–Elementes während der Initialisierungsphase von O undefinierte Werte enthielt. Damit hängt die Unvollständigkeit von d mit einem lückenhaften Ergebnis des Wortwahlprozesses bzw. unvollständiger Lexikoninformation zusammen. Da POPEL–HOW korrekte Daten voraussetzt, ist diese Art von Unvollständigkeit nicht bearbeitbar.

Wichtig für das Verständis von 2. ist die Tatsache, daß ein aktives Objekt in der DBS–Ebene über unidirektionale Verbindungen mit seinen bekannten Objekten in Beziehung steht. Im Kontext von O befinden sich demnach nur solche Objekte, die von O regiert werden. O und die von ihm abhängigen Elemente konstituieren einen Baum. Befände sich ein unspezifiziertes Merkmal im DAG des head–Elementes eines DAGs, in dem d abhängige Konstituente ist (und über Gleichungen mit dem head–Element in Beziehung steht), und wäre dieser DAG noch nicht aufgebaut, weil das entsprechende regierende Objekt von O noch nicht erzeugt wurde, hätte O keinen Bezugspunkt zu Objekten, von denen es die fehlende Information anfordern könnte. Es müßte quasi an die gesamte übergeordnete Ebene eine Anfrage bezüglich der fehlenden Information versenden. Da dies aber zu aufwendig ist, wartet O bis das entsprechende Objekt erzeugt wird und sich mit O bekannt macht.

5.6.2 Die Durchführung der Anforderungsoperation

Da bereits ein vollständiger DAG ausreicht, um den Abbildungsprozeß in die ILS–Ebene für O erfolgreich durchzuführen, wird von den DAGs von O ein DAG d_{kand} ausgewählt[5].

Im ersten Schritt der Anforderungsoperation, im folgenden *dag–request* genannt, werden alle DAGs des Dependenzrahmens von d_{kand} bestimmt, die unspezifizierte Merkmale besitzen. Dabei sind nur die DAGs von Interesse, die mit dem head–DAG über Gleichungen in Beziehung stehen. Um diese zu bestimmen, werden für jeden DAG des Dependenzrahmens folgende Operationen durchgeführt:

1. Bestimme alle Merkmale, die den leeren DAG als Wert haben; nenne diese Menge M_{undef}.

2. Entferne alle Merkmale aus M_{undef}, die nicht mit dem DAG des head–Elementes über Gleichungen in Beziehung stehen, oder die direkt bzw. indirekt mit dem Pfad *<0 fset synchronize down>* verknüpft sind. In diesem Fall handelt es sich um Merkmale, deren Werte von DAGs bestimmt werden, die d_{kand} unmittelbar oder mittelbar regieren (vgl. Abschnitt 4.1.2).

Nach Evaluierung dieser Operationen kennt O alle abhängigen DAGs d_d von d_{kand}, die mit dem head–DAG in Beziehung stehen und unspezifizierte Merkmale besitzen. Diese DAGs sind Ursache für die lokale Unvollständigkeit von d_{kand}.

Aufgrund der Semantik der Abbildungsregeln $FSS \rightarrow DBS$ besteht eine Beziehung zwischen den Elementen des Dependenzrahmens und dem Kontext von O. Die Elemente des Dependenzrahmens, die mit einem Kontextelement in Beziehung stehen, sind demnach durch eigenständige Objekte vertreten. Ob dies für alle Elemente gilt, hängt vom Grad der Verteilung ab (s.a. Abschnit 5.1).

Für jeden DAG d_d wird im nächsten Schritt von *dag–request* das entsprechende Kontextelement überprüft. Dabei sind folgende Fälle zu unterscheiden:

1. Das Kontextelement existiert nicht,

2. das Kontextelement existiert, aber es ist noch nicht mit einem aktiven DBS–Objekt besetzt,

3. das Kontextelement existiert und ist bereits mit einem aktiven DBS–Objekt besetzt.

Im ersten Fall umfaßt der Dependenzrahmen mehr Elemente, als im Kontext von O enthalten sind. Beim Überprüfen der lokalen Vollständigkeit zeigt sich aber, daß der Kontext von O unterspezifiziert ist[6].

Tritt dieser Fall ein, so fordert O dasjenige FSS–Objekt, welches O erzeugte, auf, ein Konkurrenzobjekt O' zu erzeugen. Der Kontext von O' umfaßt neben dem Kontext von O auch das fehlende Kontextelement. O' baut die gleiche initiale DAG–Struktur auf und regiert die gleichen DBS–Objekte wie O. Für das “neue” Kontextelement existiert

[5] Zur Zeit gibt es in POPEL–HOW noch keine Kriterien, mit denen bestimmt werden kann, welcher der alternativen DAGs der geeignetste ist. Daher wird von alternativen DAGs der erste gewählt und mit d_{kand} bezeichnet.

[6] Dies gilt aber nur, wenn das fehlende Element auch tatsächlich als Objekt vertreten sein sollte. Ob dies der Fall ist, kann aber leicht mit Hilfe des Merkmals $surface$ überprüft werden (vgl. Abschnitt 4.1.2).

noch keine Belegung in Form eines aktiven DBS–Objektes. Ist dies Ursache für die lokale Unvollständigkeit der DAGs von O', liegt für dieses Kontextelement Fall 2. vor.

Wenn für ein Kontextelement K der zweite Fall zutrifft, ist zu diesem Zeitpunkt das entsprechende DBS–Objekt O_K, das K belegen soll, noch nicht erzeugt worden. Da O nicht wissen kann, ob O_K jemals erzeugt wird, es aber die Existenz von O_K benötigt, damit es mit diesem Objekt den notwendigen Merkmalstransport durchführen kann, fordert es in der FSS–Ebene ein entsprechendes Objekt auf, O_K zu erzeugen. Das entsprechende FSS–Objekt wird mit Hilfe des Musters bestimmt, das im Kontextelement Platzhalter für das zu erzeugende DBS–Objekt ist (vgl. auch [**?**]).

Trifft für das Kontextelement K Fall 3. zu, kennen sich die Objekte O und O_K bereits und haben zum Zeitpunkt t ihrer Bekanntmachung die Operation *splice–dags* ausgeführt. Damit hängt aber die lokale Unvollständigkeit von d_{kand} mit der lokalen Unvollständigkeit der DAGs von O_K zum Zeitpunkt t zusammen.

d_{kand} kann also nur vollständig werden, wenn die DAGs von O_K vollständig sind. Das regierende Objekt O fordert daher sein abhängiges Objekt O_K auf, sich lokal zu vervollständigen. Im Prinzip übergibt O seinem abhängigen Element die Bearbeitung der eigenen lokalen Unvollständigkeit. DBS–Objekte sind hiermit im Unterschied zu Objekten der anderen Ebenen in der Lage, Objekte der eigenen Ebene aufzufordern, die fehlende Information bereitzustellen.

Implementationshinweis:

Mit Hilfe der Information *splice–info*, mit dem jeder Dependent von O den Merkmalstransport mit O protokolliert hat, kann dasjenige Objekt bestimmt werden, das die Anfrage bearbeiten soll. Die entsprechende Operation ist als Methode **:object–to–send–request** definiert.

Ist das abhängige Objekt bestimmt, sendet O diesem Objekt eine Anfrage **request**. **request** ist ein 3–Tupel mit folgenden Elementen:

contents: Die unspezifizierten Merkmale, die ihre Werte mit entsprechenden Merkmalen der DAGs des abhängigen Objektes teilen.

type: Gibt an, das es sich um einen **dag–request** handelt.

requestor: Der Name des anfragenden Objekts O.

Wenn O_K die Anfrage zum Zeitpunkt t', mit $t < t'$, bearbeiten will, ist es möglich, daß sich die DAGs von O_K bereits geändert haben[7], d.h. der Zustand eines DAGs d zum Zeitpunkt t ist anders als zum späteren Zeitpunkt t'.

Da der DAG d sich während des Zeitintervalls $(t' - t)$ verändert haben muß, wenn O_K die Anfrage von O sinnvoll bearbeiten will, gilt für den Zustand des modifizierten DAGs d' von d zum Zeitpunkt t':

- $d \sqsubseteq d'$ oder
- $d \neq d'$

[7]In Abschnitt 5.4 ist darauf hingewiesen, daß Veränderungen der DAG–Struktur eines Objektes O nur in Richtung seiner Dependenten transportiert wird. Das regierende Objekt von O erhält die neue Information nicht.

Im ersten Fall verfügt der modifizierte DAG über neue Information. Daher kann die Anfrage des regierenden Objektes O nun dadurch bearbeitet werden, daß der unvollständige DAG von O mit dem modifizierten DAG $d^{'}$ unifiziert wird (wobei die Operation *dag–search* eingesetzt wird).

Implementationshinweis:

Im allgemeinen erhält ein Objekt mehrere **request** Anfragen . Es puffert alle Anfragen in der Warteschlange **requestee** zwischen. Diese Warteschlange wird vom Objekt asynchron abgearbeitet. Der DAG $d^{'}$ wird mit Hilfe der Merkmale im Element **contents** von **request** bestimmt. Es wird der DAG ausgewählt, der die Merkmale und eine aktuelle Belegung dieser Merkmale besitzt.

Wenn sich die DAGs d und $d^{'}$ nicht subsumieren, so sind die DAGs nicht kompatibel. D.h., die Modifizierung des DAGs d zum Zeitpunkt $t^{'}$ hatte eine Veränderung von Werten von bestimmten Merkmalen zur Folge. Diese Art von Veränderung wird in POPEL–HOW durch das Auftreten von konkurrierenden Objekten auf der DBS–Ebene ausgelöst (vgl. Abschnitt 5.2). Im nächsten Abschnitt wird die Verarbeitung von DAGs konkurriender Objekte eingehend erläutert.

5.7 Die Bearbeitung von DAGs konkurrierender DBS–Objekte

5.7.1 Monotones vs. nicht–monotones Verhalten

Die in [**?**] geschilderte Erzeugung konkurrierender Objekte ergibt sich auch für Objekte auf der DBS–Ebene (vgl. auch Abschnitt 5.2). Für das Verhältnis zwischen den DAGs von zwei konkurrierenden Objekte O_1 und O_2 gilt:

1. Die DAGs von O_1 *subsumieren* die DAGs von O_2, oder

2. die DAGs der Objekte sind ungleich.

Werden die DAGs von O_1 durch die DAGs von O_2 ersetzt, so impliziert 1. ein *monotones* Wachstum der syntaktische Gesamtstruktur miteinander in Beziehung stehender Objekte und 2. ein *nicht–monotones* Wachstum.

Ein Beispiel soll dies verdeutlichen. Die Ausgangssituation sei folgende: Durch erfolgten Kontextaustausch ist O_1 in den Regensbereich von O_3 gekommen. Die Objekte haben bereits die Operation *splice–dags* durchgeführt und so relevante Information ausgetauscht. Tritt O_2 als Konkurrent zu O_1 in die DBS–Ebene ein, übernimmt es dessen Platz im Kontext von O_3[8].

Dies erfordert nun, die DAGs von O_3 entsprechend zu modifizieren bzw. zu aktualisieren. Subsumieren die DAGs von O_1 die DAGs von O_2, kann die Modifikation der DAGs von O_3 mit Hilfe der Operation *splice–dags* durchgeführt werden (s. Abschnitt 5.4).

[8]In diesem und im nächsten Beispiel gehe ich davon aus, daß die momentane Regelauswahlstrategie in POPEL–HOW vom Typ *maximal* ist, d.h. ein neu eintreffendes Objekt ist bezüglich seines Kontextes auf jeden Fall spezieller als das ältere (vgl. [**?**]). Dies hat keinen Einfluß auf das Ergebnis, erleichtert aber die Erklärung.

Für die DAGs von O_3 zum Zeitpunkt t vor Durchführung der Operation und dem Zeitpunkt t' nach dieser Operation (mit $t < t'$) gilt das Verhältnis der Subsumption. Damit ist aber die syntaktische Gesamtstruktur, die durch die Objekte gebildet wird, *monoton* gewachsen.

Liegt für das Verhältnis der DAGs von O_1 und O_2 Fall 2. vor, so ist O_2 durch eine Regel eines FSS–Objektes erzeugt worden, die eine unterschiedliche syntaktische Struktur beschreibt als die Regel, die O_1 erzeugte.

Dies ist z.B. dann der Fall, wenn sich der Kontext des FSS–Objektes derart ändert, daß das semantische Wissen auf eine andere syntaktische Phrase abgebildet werden muß. Wenn es sich bei dem FSS-Objekt um ein Handlungsprädikat (z.B. "laufen") handelt, das in den Regensbereich eines Modalprädikates (z.B. "wollen") aufgenommen werden soll, kann es für die neue Situation sinnvoll sein, das Handlungsprädikat auf eine Nominalphrase mit Head "der Lauf" abzubilden, anstatt, wie vor der kontextuellen Veränderung, auf eine Satzphrase.

Eine andere Möglichkeit ist gegeben, wenn bei der erneuten Regelanwendung eines FSS–Objektes eine andere Lexemwahl stattfindet, z.B. wenn es aufgrund des veränderten Kontextes sinnvoller ist "das Vieh" anstatt "die Kühe" als head–Element für das FSS–Objekt auszuwählen.

In beiden Fällen haben die entsprechenden DAGs der DBS–Objekte O_1 und O_2 unterschiedliche Merkmale (im ersten Fall unterscheiden sich die Werte der Kategorie und im zweiten Fall die Lexeme).

Wenn O_2 das Objekt O_1 im Kontext von O_3 ersetzt, müssen auch die DAGs von O_3 aktualisiert werden. Aufgrund unterschiedlicher Merkmale zwischen den DAGs von O_1 und O_2 ist dies aber nicht ohne weiteres mit *splice–dags* durchführbar, da diese Operation versucht, die entsprechenden DAGs zu unifizieren, was auf jedenfall scheitern wird.

Kann die Modifikation mit Hilfe einer anderen Operation durchgeführt werden, so bedeutet diese Art von Veränderung für die Gesamtstruktur, daß sie zu diesem Zeitpunkt *nicht–monoton* anwächst. An dieser Stelle im Generierungsprozeß wird das sukzessive Wachsen der Gesamtstruktur unterbrochen.

5.7.2 Eine notwendige nicht–monotone Operation

Die erforderliche Operation muß in der Lage sein, einen bestimmten Teil–DAG d_1 eines DAGs durch einen neuen Teil–DAG d_2 lokal zu ersetzen. Da im Prinzip d_1 *überschrieben* wird, kann hierfür nicht die Unifikation direkt eingesetzt werden. Der Teil–DAG d_2 muß allerdings mit den Merkmalen von d_1 kompatibel sein, deren Werte von außen, d.h. von übergeordneten Strukturen definiert werden, damit insgesamt keine Inkonsistenzen auftreten. Bei den Merkmalen handelt sich es gerade um diejenigen, die im DAG d_1 unter dem Merkmal $<$ *synchronize up* $>$ spezifiziert sind (vgl. Abschnitt 4.1.2).

Als Basisoperation für die nicht–monotone Operation könnte prinzipiell die Operation *dag–search* eingesetzt werden. Allerdings benötig das Objekt O, dessen DAG d teilweise überschrieben werden soll, einen *Historiemechanismus*, der für jeden DAG von O chronologisch die Veränderungen festhält. Mit Hilfe von bekannten Backtrackingverfahren wäre es dann möglich, d so zu rekonstruieren, wie er vor Einbau des zu überschreibenden Teil–DAGs spezifiziert war. Danach wäre es relativ trivial, den neuen Teil–DAG d_2 (s.o.) mit Hilfe der Operation *dag–search* in d zu integrieren.

Zwischen den Zeitpunkten t_n und t_m, mit $t_n < t_m$, ist es jedoch möglich, daß d durch die DAGs anderer abhängiger Objekte von O verändert wurde. Wenn also d zum Zeitpunkt t_m rekonstruiert werden muß, darf die Information der Teil–DAGs, die nach t_n aber vor t_m integriert wurden, nicht verloren gehen. Der Vorteil, das nicht–monotone Verhalten von DAGs mit der monotonen Operation *dag–search* zu simulieren, verlangt demnach einen immensen Verwaltungsaufwand der DAGs eines DBS–Objektes.

Eine Operation, die direkt zum Zeitpunkt t_m die Ersetzung durchführt, die also *backtrackfrei* operieren kann, ist daher dringend erforderlich. Ich habe ein Verfahren entwickelt, im folgenden *change–dags* genannt, mit dem in einem gegebenen DAG d gezielt und unmittelbar ein Teil–DAG d_1 durch einen neuen Teil–DAG d_2 ersetzt werden kann. Die Abbildung 5.1 (S. 96) zeigt den Aufbau des Algorithmus *change–dags*.

change–dags(d, d_1, d_2):

begin
1. **if** es existiert eine aktuelle Beziehung zwischen d und d_1 **then**

    2. **if** d_1 in d lokalisiert und d_2 ist konsistent mit d_1 **then**
        3. integriere d_2
        4. entsprechende Einträge *splice–info* löschen und einfügen

        else *splice–info* auf den neuen Stand bringen

        fi

    else

    5. integriere d_2 mit *dag–search*
    6. trage neue *splice–info* ein

    fi

end

Figure 5.1: Grobstruktur des Algorithmus von *change–dags*

Ob eine aktuelle Beziehung zwischen den DAGs d und d_1 existiert, kann mit Hilfe von *splice–info* festgestellt werden. Der Eintrag ist bei erfolgreicher Durchführung der Operation *splice–dags* der entsprechenden Objekte erzeugt worden (s. Abschnitt 5.4). Ist dies der Fall, wird im nächsten Schritt von *change–dags* d_1 mit Hilfe der Operation *dag–search* in d lokalisiert. Wenn d_1 und d_2 konsistent sind, d.h. die Teil–DAGs unter dem Pfad $<$ *synchronize up* $>$ sind kompatibel (s.o.), kann d_1 durch d_2 ersetzt werden. Die *Integration* von d_2 in d wird durch *Überschreibung* von d_1 mit d_2 geleistet. Diese Überschreibungsoperation unterscheidet sich von der Standardunifikation (s. Abschnitt 3.2.1, Def. 2) in folgenden Punkten:

1. Falls zwei Spezifikationen unterschiedliche Werte haben, wird der Wert der letzteren Spezifikation genommen.

2. Falls der DAG d_1, der überschrieben werden soll, Merkmale besitzt, die der andere DAG d_2 nicht hat, werden sie ignoriert.

Die Überschreibungsoperation[9] ist nicht kommutativ. Die mit der Operation verbundene Vereinigung kann daher auch als "einseitige Vereinigung" charakterisiert werden. Wenn Merkmale von d_1 ignoriert werden, weil sie nicht in d_2 auftreten, muß beachtet werden, daß alle Merkmale, die über Gleichungen mit den zu entfernenden Merkmalen von d_1 den selben Wert teilen, gelöscht werden, d.h. den leeren DAG erhalten.

Beispiel 22
Im DAG

[9] Eine vergleichbare Operation – *overwrite–unify* – wird auch in D–PATR bzw. SB–PATR beim Übersetzen der Listennotation eines Lexikoneintrages in einen lexikalischen DAG eingesetzt. Allerdings unterscheidet sich diese Operation von der Standardunifiaktion nur im ersten Punkt.

$$\begin{bmatrix} x: & \begin{bmatrix} z: & < y \; b > \end{bmatrix} \\ y: & \begin{bmatrix} a:1 \\ b:2 \end{bmatrix} \end{bmatrix}$$

wird der Teil–DAG

$$\begin{bmatrix} a:1 \\ b:2 \end{bmatrix}$$

ersetzt durch den DAG

$$\begin{bmatrix} a:3 \\ c:4 \end{bmatrix}$$

Der gesamte DAG hat dann folgende Gestalt:

$$\begin{bmatrix} x: & \begin{bmatrix} z: & [\,] \end{bmatrix} \\ y: & \begin{bmatrix} a:3 \\ c:4 \end{bmatrix} \end{bmatrix}$$

Chapter 6

Die Linearisierung und Flexion in POPEL–HOW

6.1 Aufgabenstellung

Die ILS–Ebene ist die letzte Verarbeitungsstufe in POPEL–HOW. Ausgangsstrukturen auf dieser Ebene sind die ILS–Einträge ILS–E, die von DBS–Objekten übermittelt werden. Jeder ILS–E besteht aus fünf Elementen (s. Abschnitt 5.5.2, S. 89). Das zentrale Element eines ILS–E ist der Vater–DAG eines lokal vollständigen DAGs des DBS–Objektes, das den ILS–E erzeugte und auf die Ebene abbildete (vgl. Abschnitt 5.5.2). Die wesentlichen Aufgaben, die auf dieser Ebene bearbeitet werden, sind

- Flexion der Lexeme eintreffender DAGs,
- Aufbau einer syntaktisch korrekten linearen Abfolge und
- Ausgabe der flektierten Äußerungsteile.

Um auch auf dieser Ebene das inkrementelle Eingabeverhalten in POPEL–HOW aufrechtzuerhalten, müssen neu eintreffende DAGs in die bisher aufgebaute lineare Abfolge integriert und möglichst sofort geäußert werden. Die zentrale Wissensquelle, die für die Bestimmung der linearen Abfolge eingesetzt wird, sind die LP–Regeln von POPEL–GRAM. Die LP–Regeln beschreiben syntaktisch korrekte Positionsbäume. Damit kann der inkrementelle Aufbau der linearen Abfolge auch charakterisiert werden als sukzessive Konstruktion von Positionsbäumen.

6.2 Rekonstruktion der dependenziellen Struktur der DBS–Objekte

Die korrekten Positionsbäume können nur dann bestimmt werden, wenn die Dependenzstruktur der DAGs in der ILS–Ebene rekonstruiert wird. Ausgangspunkt hierfür sind die ILS–E. Mit jedem eintreffenden ILS–E kann nachvollzogen werden, von welchem DBS–Objekt dasjenige Objekt regiert wird, das den ILS–E abbildete. Damit ist es möglich, verschiedene ILS–E dependentiell in Beziehung zu bringen. Im speziellen gilt dies auch für die DAGs der Einträge. Grundlage für die Rekonstruktion der Dependenzstruktur ist die Information in dem Element *regenten* (vgl. Abschnitt 5.5.2) eines ILS–E. Mit *splice–info* ist dort protokolliert worden, mit welchem DAG eines regierenden Objektes erfolgreich ein Merkmalstransport durchgeführt wurde. Die nachfolgende Abbildung zeigt die dependentielle Struktur von vier miteinander in Beziehung stehenden ILS Einträgen (zur Erläuterung der Elemente siehe Abschnitt 5.5.2, Seite 89).

$$\begin{tikzcd}
& (O_2, 2, d_2, (...), string_2) \arrow[dl] \arrow[dr] & \\
(O_1, 2, d_1, (O_2, d_2, ...), string_1) \arrow[d] & & (O_4, 3, d_4, (O_2, d_2, ...), string_4) \\
(O_3, 5, d_3, (O_1, d_1, ...), string_3) & &
\end{tikzcd}$$

Zum Beispiel gilt für den DAG d_2 vom ILS–E des Objektes O_2: d_2 regiert d_1 von Objekt O_1 sowie d_4 von Objekt O_4. Insgesamt ergibt sich für die miteinander in Beziehung stehenden ILS–E Einträge eine Baumstruktur. Einen solchen Baum werde ich im weiteren als *Verbalisierungsbaum* VB bezeichnen.

Es ist möglich, daß ein abhängiger DAG d_d von unterschiedlichen DAGs D_r regiert wird. Dies ist der Fall, wenn ein regierendes Objekt über mehrere alternative DAGs verfügt, die mit d_d eines abhängigen Objektes erfolgreich einen Merkmalstransport bzw. die Operation *dag–search* ausführen konnten (vgl. Abschnitt 5.4), oder wenn ein regierendes Objekt Konkurrenten hat, die ebenfalls mit d_d erfolgreich einen Merkmalstransport durchführten (vgl. Abschnitt 5.1). In beiden Fällen befinden sich in dem Element *regenten* des ILS–E von d_d mehrere Einträge *splice–info*. Hiermit kann auf der ILS–Ebene nachvollzogen werden, daß der DAG d_d als Teil–DAG in unterschiedlichen DAGs bzw. syntaktischen Lesarten auftritt. Beispielsweise gilt für den DAG d_5 des ILS–Eintrages

$$(O_5, 1, d_5, ((O_1, d_1, ...)\ (O_1^{'}, d_3^{'}, ...)), string_5)\ ,$$

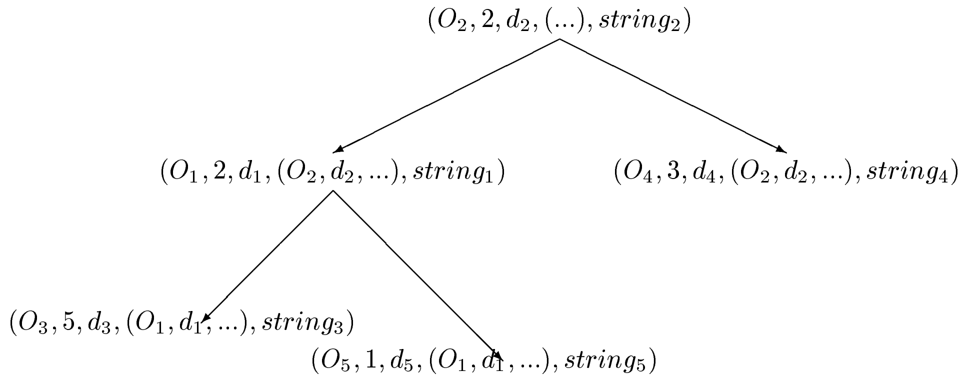
daß er jeweils von den DAGs d_1 und $d_3^{'}$ der konkurrierenden Objekte O_1 und $O_1^{'}$ regiert wird. Um solche "mehrdeutigen" ILS–Einträge mit den bereits existierenden ILS–Einträgen eindeutig in Beziehung bringen zu können, werden entsprechend der Anzahl der Elemente im Element *regenten* "regenten" viele ILS–Einträge mit jeweils einem Element *splice–info* erzeugt. Für obigen Eintrag ergeben sich damit folgende zwei sich nur in dem Element *regenten* unterscheidende Einträge:

$$(O_5, 1, d_5, (O_1, d_1, ...), string_5)$$
$$(O_5, 1, d_5, (O_1^{'}, d_3^{'}, ...), string_5)$$

Es wird versucht, jeden einzelnen dieser ILS–Einträge wird versucht, in die bestehende Baumstruktur der bereits existierenden ILS–Einträge zu integrieren. Ist obiger Verbalisierungsbaum die Ausgangsstruktur, so kann dort nur der Eintrag

$$(O_5, 1, d_5, (O_1, d_1, ...), string_5)$$

mit den bisherigen Einträgen in Beziehung gebracht werden. Der entsprechende VB sieht wie folgt aus:



Obwohl der zweite Eintrag nicht in den bisherigen VB integriert werden kann, wird er als alternativer VB weiterbetrachtet. Dieser VB wird auch in Zukunft mit dem anderen VB über keinen ILS–E in Verbindung treten. Er ist somit Teilbaum einer Paraphrase zum ersten VB. Auf die entsprechenden Ausgabeprobleme gehe ich in Abschnitt 6.5 ein.

Der inkrementelle und parallele Konstruktionsprozeß auf der DBS–Ebene hat zur Folge, daß die DAGs bzw. die ILS–E ungeordnet und zu unterschiedlichen Zeitpunkten in der ILS–Ebene eintreffen. Bei der Linearisierung und Ausgabe müssen folgende Punkte beachtet werden:

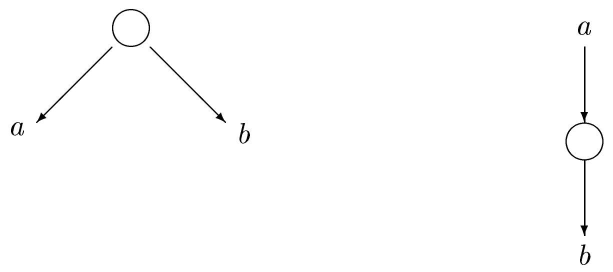
1. Es ist möglich, daß ein eintreffender ILS–E noch nicht in den bisher aufgebauten Verbalisierungsbaum[1] integriert werden kann.

2. Die zeitliche Aktivierung der Objekte auf der CKB–Ebene, die in den DBS–Objekten festgehalten ist, soll – wenn syntaktisch möglich – auf der ILS–Ebene nachvollzogen werden.

Kann ein eintreffender ILS–E und damit sein DAG nicht in den bisherigen Verbalisierungsbaum VB integriert werden, existiert im VB entweder kein Knoten, der den neuen ILS–E regiert, oder der neue ILS–E ist nicht regierendes Element des VB. In beiden Fällen fehlt zu diesem Zeitpunkt ein ILS–Eintrag, der beide Strukturen verbinden könnte. Graphisch läßt sich dieser Sachverhalt wie folgt darstellen (a sei das neue Element, b Wurzel des VBs und der Kreis symbolisiere das fehlende Element)[2]:

[1] Im weiteren gehe ich davon, daß auf der ILS–Ebene keine alternativen Verbalisierungsbäume erzeugt werden, d.h. ein beliebiger ILS–E hat nur ein Element *splice–info*.

[2] Die beiden Fälle lassen sich mit den in [Kempen, 1987] erläuterten Expansionsschritten von Bäumen, nämlich *upward* und *insertion* vergleichen. Auf diese Punkte gehe ich in Abschnitt 6.5 noch ein.



Da der Verbalisierungsbaum Ausgangspunkt für die Bestimmung der korrekten linearen Abfolge ist, kann, solange das verbindende Element fehlt, keine gemeinsame lineare Folge bestimmt werden. Ein sofortiges Äußern der neuen Teile würde unter Umständen ungrammatische und unzusammenhängende Satzteile produzieren, die vom Benutzer nicht verstanden würden.

Beispiel 23
Angenommen, es sei bereits ein VB aufgebaut, der als Wurzel einen ILS–E mit einer Ergänzung im Nominativ (e0) hat. Der bereits produzierte Äußerungsteil sei "der fleißige Steuerberater". Tritt als nächstes Element ein ILS–E mit einer Präpositionalergänzung (e6) ein, für den der Äußerungsteil "nach Hamburg" produziert werden könnte, würde die bisherige Gesamtäußerung "der fleißige Steuerberater nach Hamburg" ausgegeben werden. Geht man davon aus, daß zwischen diesen beiden Ergänzungen im normalen Fall ein finites Verb stehen muß (z.B. "fahren"), würde ein sofortiges Äußern von "nach Hamburg" eine nicht korrekte Form zur Folge haben.

Solange das verbindende Element fehlt, verhalten sich die beiden Teilstrukturen wie alternative Verbalisierungsbäume. Im Unterschied zu VBs, die Paraphrasen repräsentieren, können aber beide Strukturen zu einem späteren Zeitpunkt in Verbindung gebracht werden.

6.3 Verarbeitungsreihenfolge in der ILS–Ebene

Der ILS–Ebene liegt kein paralleles Modell zugrunde, sondern ein Prozeß übernimmt die Bearbeitung aller Aufgaben. Im wesentlichen hängt das mit der zentralen Aufgabe zusammen, eine gegebene Struktur zu sequentialisieren. Da nur ein Prozeß die Aufgabe bearbeitet, bedeutet der Übergang in die ILS–Ebene eine Transformation aktiver Strukturen in passive.

Es treffen nur solche DAGs in der ILS–Ebene ein, die lokal vollständig sind. Diese lokale Vollständigkeit bezieht sich auch auf die Information, die zur Flexion der Lexeme benötigt wird. Da zur Zeit angenommen wird, daß die LP–Regeln keine Auswirkungen auf die ID–Struktur haben, hat insgesamt die ILS–Ebene keine Verbindung zur DBS–Ebene, d.h. von der ILS–Ebene wird kein Feedback gestartet.

Die zentrale Datenstruktur für den ILS–Prozeß ist eine Schlange ILS–Q. Sie enthält alle neu eingetroffenen ILS–E. Wenn die Schlange keine Elemente hat, wird der ILS–Prozeß "schlafen" gelegt. Der zugrundeliegende Algorithmus für den ILS–Prozeß kann wie folgt beschrieben werden:

if ILS–Q **is empty then wait**
else
1. ILS–E ::= **pop**(ILS–Q);
2. führe Flexion für ILS–E durch;
3. erzeuge Lesartenviele Eintrage ILS–E_i;
4. füge ILS–E_i in bisher rekonstruierte Dependenzstruktur ein;
5. führe Lineariserung über Dependenzstruktur durch;
6. gib relevante Teile aus;

fi

Für das erste Element der Schlange werden alle präterminalen Elemente des DAGs vom entsprechenden ILS–E bestimmt und flektiert. Die DAGs der präterminalen Elemente befinden sich unter dem Merkmal $surface$. Jeder präterminale DAG ist dort repräsentiert durch seine Kategorie, sein Lexem und die morphosyntaktische Information, die zur Flexion benötigt wird.

So ergibt sich für eine Präpositionalphrase mit folgendem Teil–DAG unter dem Merkmal $surface$:

$$
\left[\begin{array}{ll}
pp: & \left[\begin{array}{ll}
prep: & \left[\begin{array}{ll} stem: & auf \\ syntax: & \left[\begin{array}{ll} case: & acc \end{array}\right] \end{array}\right] \\
np: & \left[\begin{array}{ll}
np-eng: & \left[\begin{array}{ll}
det: & \left[\begin{array}{ll} stem: & indef \\ syntax: & \left[\begin{array}{ll} number: & sg \\ gender: & ntr \\ case: & acc \\ article: & indef \end{array}\right] \end{array}\right] \\
n: & \left[\begin{array}{ll} stem: & haus \\ syntax: & \left[\begin{array}{ll} number: & sg \\ gender: & ntr \\ case: & acc \\ article: & indef \\ elision: & - \\ anapher: & - \end{array}\right] \end{array}\right]
\end{array}\right] \\
ap: & \left[\begin{array}{ll}
adj: & \left[\begin{array}{ll} stem: & klein \\ syntax: & \left[\begin{array}{ll} number: & sg \\ gender: & ntr \\ case: & acc \\ articel: & indef \\ comparation: & pos \end{array}\right] \end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

folgende Liste mit präterminalen DAGs:

(prep–DAG n–DAG det–DAG adj–DAG)[3]

Diese Liste ist Ausgangspunkt für den Flexionsschritt mit dem Morphologiemodul MORPHIX (s.a. Abschnitt 6.6). Für jeden präterminalen DAG wird ein entsprechender Aufruf für MORPHIX erzeugt. Dies gilt aber nicht, wenn die präterminale Kategorie eine nichtflektierbare Wortklasse bezeichnet (wie z.B. Präpositionen) – in diesem Fall erübrigt sich eine Flexion – und wenn der Wert des Merkmals $elision$ (falls es spezifiziert

[3]Die Namen sind Bezeichner für die präterminalen DAGs, z.B. bezeichnet prep–DAG den DAG für die Präposition "auf".

wurde) gleich '+' ist – in diesem Fall wird die Generierung der entsprechenden Wortform unterdrückt.

Implementationshinweis:

Es ist möglich, daß sich die Bezeichnungen für die morphosyntaktischen Merkmale in der Grammatik und MORPHIX unterscheiden. Deshalb müssen vor der Aktivierung von MORPHIX die Bezeichner entsprechender Merkmale übersetzt werden. Die Abbildung wird in POPEL–HOW durch das Hasharray *feature–mapping* realisiert, wobei die Bezeichner von POPEL–GRAM als Schlüssel fungieren.

Die so flektierten Wortformen werden zusammen mit der präterminalen Kategorie in einer Liste gesammelt. Das Ergebnis für das Beispiel lautet z.B.

((prep "auf") (n "mauer") (det "einer") (adj "kleinen"))

Es wird im entsprechenden ILS–E dem Element *string* zugewiesen. Der ILS–E besitzt damit neben dem DAG nun auch die (unsortierte) präterminale Kette.

In Abschnitt 4.1.4.3 wurde bereits darauf hingewiesen, daß bei der Flexion von Verben, der finite und infinite Teil von MORPHIX bestimmt wird. Da in den LP–Regeln die Trennung eines flektierten Verbs in diese Teile explizit verlangt ist, wird die Trennung in der Ergebnisliste nachvollzogen. Beispielsweise liefert die Flexion des verb–DAGs

$$\begin{bmatrix} v & \begin{bmatrix} stem: & \text{“}fahr\text{''} \\ syntax: & \begin{bmatrix} tense: & perfect \\ mood: & indicative \\ voice: & passive \\ person: & 3 \\ numerus: & sg \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

die dreielementige Wortform

"ist gefahren worden".

Die entsprechende Ergebnisliste lautet dann

((vfin "ist") (vinfin "gefahren worden"))

Eine ähnliche Subklassifizierung wird auch für die präterminale Kategorie perspron (Personalpronomen) durchgeführt (nach [**?**]).

Nach dem Flexionsschritt werden "regenten"viele ILS–Einträge ils–e vom ILS–E erzeugt. Diese Einträge unterscheiden sich nur durch ihre unterschiedliche *splice–info*. Für jeden bereits existierenden Verbalisierungsbaum VB wird überprüft, ob ils–e den Baum regiert oder von einem Element des Baums regiert wird. Wenn ils–e den VB regiert, wird er als neue Wurzel in den Baum eingefügt. Insgesamt wächst der durch ils–e modifizierte VB nach oben (In [Kempen, 1987] wird diese Art der Erweiterung einer syntaktischen Struktur als *upward expansion* bezeichnet.). Ist ils–e abhängiges Element, wird er als neuer Blattknoten in den aktuellen VB eingefügt. VB wächst in diesem Fall nach unten ([Kempen, 1987] bezeichnet dies als *downward expansion*). Ils–e kann nicht zwischen zwei bereits verbundenen Knoten eingefügt werden ([Kempen, 1987] nennt dies *insertion*), weil die DAGs von unmittelbaren Nachfolgeknoten eines ils–e unmittelbare Konstituenten repräsentieren.

Wenn ils–e nicht in den bisher aufgebauten VB integriert werden kann, weil im VB kein Knoten existiert, mit dem ils–e dependentiell in Beziehung steht, wird der aktuelle ils–e als neuer Verbalisierungsbaum aufgenommen.

Implementationshinweis:

Die Tests sind als Methoden **:check–if–dominates** und **:check–if–being–dominated** für Objekte vom Typ **dbs–object** formuliert. Sie haben als aktuellen Parameter den einzufügenden Knoten ils–e.

Wenn keine Paraphrasen angenommen werden (d.h. es gibt keine "regenten"viele Einträge), existiert nach diesem Schritt nur ein einizger modifizierter (oder neuer) Verbalisierungsbaum VB. Dieser VB ist Ausgangspunkt für den Linearisierungsschritt, der im nächsten Abschnitt erläutert wird.

6.4 Die Anwendung der Linearisierungsregeln

Ziel in dieser Phase ist es, ausgehend vom modifizierten Verbalisierungsbaum die korrekte lineare Abfolge zu bestimmen. Die Verarbeitungsrichtung des zugrundeliegenden Algorithmus verläuft top–down, depth–first. Abbildung 6.1 (S. 106) zeigt den Algorithmus. Begonnen wird mit der Wurzel des Verbalisierungsbaums VB. LPR ist die LP–Regel, die als linkes Element mit der Kategorie übereinstimmt, die direkt unter dem Merkmal *surface* des Vater–DAGs des aktuellen ils–e zu finden ist. Für den DAG auf Seite 102 ist z.B. nur die LP–Regel relevant, die als linkes Element *pp* als Wert besitzt. Gibt es im LP–Regelteil mehrere alternative LP–Regeln, wird nur diejenige gewählt, deren syntaktische Merkmalsbeschreibung mit der des DAGs des aktuellen Knotens übereinstimmt .

Wenn für den aktuellen Knoten keine LP–Regel existiert, gibt es für den entsprechenden DAG keine vorgeschriebene lineare Abfolge. Damit ist aber keine Reihenfolge der Elemente des *string* (s.o.) vorgegebenen. Für das Element *string*

((n "haus") (det "das") (adj "kleine))

ergäbe der Oberflächenstring

"haus das kleine".

Existiert eine Regel, so ist sie Positionierungsgrundlage. Der rechte Teil (rhs) einer Regel LPR bezieht sich auf die Reihenfolge der unmittelbaren Konstituenten (und der head–Elemente) der entprechenden Phrase (vgl. Abschnitt 4.1.4.3). Daher wird der rechte Teil der LP–Regel von links nach rechts abgearbeitet und überprüft, welcher DAG die entsprechende Position des aktuellen rechten Elements r_i einnehmen kann. Bei der Bestimmung der korrekten Belegung für die entprechende Position muß unterschieden werden, von welchem Typ die Kategorie von r_i ist. Folgende Fälle werden unterschieden:

– r_i ist ein nichtterminales Element (non–terminal–cat). Eine nichtterminale Kategorie bezieht sich auf den DAG des aktuellen Knotens. Sie besagen nur, daß zur Bestimmung der linearen Abfolge weitere LP–Regeln ausgewählt werden müssen. r_i wird dadurch quasi expandiert. Für das Element *np* der LP–Regel:

(pp prep np)

existiert eine eigene LP–Regel der Form:

(np det adj ap n)

Die gesamte Linearisierungsregel für *pp* expandiert demnach zu folgender LP–Regel:

(pp prep det adj ap n)

– r_i ist ein Element, das sich auf die DAGs von abhängigen Objekten bezieht. Solche Elemente sind mit dem Merkmal *dependent* mit Wert '+' versehen (s. Abschnitt 4.1.4.3). Auf der ILS–Ebene bedeutet dies, daß der aktuelle Knoten entsprechende Söhne hat. Existieren zum gegenwärtigen Zeitpunkt Söhne, wird der Algorithmus rekursiv für den ersten Sohn ausgeführt, wobei für den Knoten nur die LP–Regel mit linkem Element gleich r_i bestimmt wird. Dabei ist es möglich, daß ein Knoten gewählt wird, dessen *order* einen größeren Wert besitzt als seine Geschwisterknoten. In diesem

```
verbalize–tree(LPR, node):
    begin
     1.   if keine LPR then return string(node) else
           2.     for i=1..n with r_i ∈ rhs(LPR) do:
           3.     case type(r_i):
                   4.  non–terminal–cat: do
                       verbalize–tree(LPR(r_i), node);
                       od
                   5.  object–cat: do
                            for first(SONS) with cat=r_i do
                            verbalize–tree(LPR(r_i), first(SONS)) od
                       od
                   6.  any: do
                            sort(SONS(node));
                            verbalize–tree(LPR(cat(first(SONS))), first(SONS))
                       od
                   7.  multiple–any: do
                            sort(SONS(node));
                            for every son ∈ SONS(node) do
                            verbalize–tree(LPR(cat(son)), son) od
                       od
                   8.  else do
                            choose–string(string(node), r_i);
                            return choosen–string
                       od
           esac
     od
     fi
    end
```

Figure 6.1: Algorithmus zur Bestimmung der linearen Abfolge

Fall wird die gewünschte Reihenfolge von POPEL–WHAT durch die vorgegebene Reihenfolge in den LP–Regeln unterdrückt.

– Hat das aktuelle rechte Element den Wert *any* oder *mutiple–any*, handelt es sich um eine abstrakte Kategorie. Beide Kategorien beschreiben Mengen und beziehen sich mit ihren Elementen auf die DAGs von abhängigen Knoten. Bei *any* wird als Belegung für das rechte Element ein beliebiger Sohnknoten bestimmt. Dabei sollte es sich um den Knoten mit kleinsten Wert *order* handelt, da dadurch die intendierte Reihenfolge nachvollzogen wird. Zu diesem Zweck müssen aber die Söhne vorher nach *order* sortiert werden.

– Im letzten Fall, der unterschieden wird, handelt es sich beim rechten Element um eine präterminale Kategorie. Hiermit bezieht sich das Element aber auf die lexikalischen Elemente im Eintrag *string* des aktuellen Knotens. Es wird dasjenige lexikalische Element bestimmt, dessen Kategorie gleich r_i ist. Hat r_i z.B. den Wert *det* und *string* den Wert

((n "haus") (det "das")) ,

wird die Position des aktuellen rechten Elementes mit dem Wert "das" belegt.

6.5 Die Ausgabe in POPEL–HOW

Mit dem oben beschriebenen Mechansimus läßt sich für einen modifizierten Verbalisierungsbaum eine korrekte lineare Folge bestimmen. Für die Ausgabe sind aber im Prinzip nur die modifizierten Teile relevant. Je nachdem, wo diese modifizierten Teile im Verbalisierungsbaum "sitzen", ist die Ausgabe mehr oder weniger direkt durchzuführen.

Eine unmittelbare Ausgabe ist dann problematisch, wenn die neuen Äußerungsteile gemäß den Linearisierungsregeln nicht am rechten Ende der bereits produzierten Äußerung angehängt werden können. Folgende Fälle müssen unterschieden werden:

- Der modifizierte VB steht mit dem VB, von dem aus die bisherige Äußerung erzeugt wurde, nicht in Beziehung.
- Der neue Äußerungsteil muß aus syntaktischen Gründen vor der bisherigen Äußerung stehen.
- Der neue Äußerungsteil ist eine Verbalisierungsalternative zu einem Teil der bisherigen Äußerung.

Liegt der erste Fall vor, muß die Ausgabe des modifizierten VBs so lange unterdrückt werden, bis ein Verbalisierungsbaum existiert, der die nicht zusammenhängenden VBs als Teilbäume aufnimmt. Sei zum Beispiel die Nominativergänzung "der kleine Peter" bereits geäußert worden. Wenn als nächstes die Ergänzung " mit dem Auto" verbalisiert werden soll, muß ihr Äußern solange unterdrückt werden, bis ein Verb geäußert werden kann, das beide Ergänzungen an sich bindet und sie somit in Beziehung bringt. Erst dann kann eine syntaktisch korrekte Ausgabe erzeugt werden.

Soll ein neuer Äußerungsteil verbalisiert werden, der aus syntaktischen Gründen vor der bisherigen Äußerung positioniert werden muß, wird der entsprechende rechte Teil der alten Äußerung wiederholt. Ist "fährt nach Hause" bereits ausgegeben worden und "Peter" der neue Äußerungsteil, wird "Peter fährt nach Hause" verbalisiert.

Der dritte Fall ähnelt dem zweiten. Auch hier muß in eine bereits gemachte Äußerung ein neuer Teil integriert werden. Allerdings besteht hierbei die Möglichkeit, daß der neue Teil zwischen zwei bereits existierenden Äußerungen eingefügt werden muß. In diesem Fall werden zur Zeit in POPEL–HOW der gesamte Äußerungsteil links und rechts von der neuen Äußerung wiederholt. Nach "der Peter" soll zum Beispiel "kleine" geäußert werden. Gemäß den Linearisierungsregeln wird "der kleine Peter" verbalisiert.

6.6 Die Flexion lexikalischer Elemente

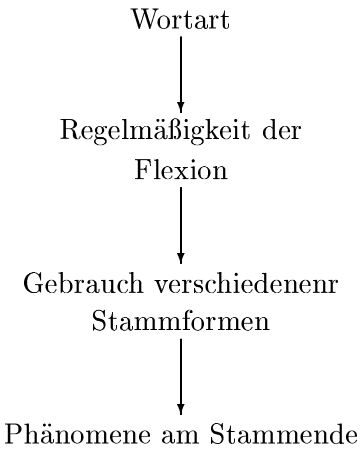
Es wurde bereits mehrfach erwähnt, daß die Flexion der Lexeme mit dem Morphologiemodul MORPHIX durchgeführt wird. Die zum Zeitpunkt der Entwicklung von POPEL–HOW vorliegende Version von MORPHIX [**?**] war allerdings nur in der Lage, flektierte Wortformen zu *analysieren*. Im Rahmen dieser Arbeit haben wir die Version jedoch so weit modifiziert, daß es mit dem erweiterten MORPHIX auch möglich ist, Wortformen zu *generieren*. Damit steht dem Gesamtsystem XTRA eine Morphologiekomponente zur Verfügung, die in der Analyse– und Generierungsrichtung gleichermaßen zum Einsatz kommt [**?**].

6.6.1 Das Klassifikationssytem

Im wesentlichen war die Modifikation von MORPHIX möglich, weil in der neuen Version systematisch Gebrauch von einem Klassifikationssystem gemacht wurde. Das Klassifikationssytem ist Grundlage für die zwei zentralen Wissensquellen von MORPHIX:

1. dem Lexikon XTRALEX und
2. dem Allomorphlexikon der Endungen (IAL Inflectional Allomorph Lexicon).

Die nachfolgende Abbildung zeigt die schematische Darstellung des in MORPHIX verwendeten Klassifikationssytems:



Das Schema verdeutlicht, daß es sich um ein hierarchisches Klassifikationssytem handelt, in dem neben morphosyntaktische auch phonomorphologische Merkmale berücksichtigt werden.

Am Beispiel der Verben zeige ich, wie dieses System verwendet wird. Zuerst betrachten wir die *morphosyntaktischen* Merkmale. Ein charakteristisches Merkmal von Verben ist die Verwendung unterschiedlicher Segmente bei der Konjugation je nach Regelmäßigkeit des Verbs. Zum Beispiel lassen sich alle Konjugationsformen des Lexems "fahr" mit den Segmenten "fahr", "fuhr", "fähr" und "fahr" bilden.

Alle möglichen Verbsegmente lassen sich nach [**?**] in vier *Segmenttypen* einteilen:

1. Das VGFA–Segment ist mit dem lexikalischen Stamm (z.B. "fahr") identisch.
2. Das VGFB–Segment ist das größte gemeinsame Anfangsstück derjenigen Formen, die mit ihm gebildet werden (1.Pers. Sing., 2./3. Pres. Sing. sowie Imperativ Sing., z.B. "fähr")
3. Das VGFC–Segment ist mit der 1. Person Singular Imperativ identisch (z.B. "fuhr").
4. Das VGFD–Segment ist gleich der 1. Person Singular im Konjunktiv II (z.B. "führ").

Die Segmenttypen verweisen im Lexikon auf die möglichen Grundformen eines Verbes. Mit Hilfe dieser Segmenttypen lassen sich alle Verben des Deutschen in elf Verbtypen einteilen. Der Verbtyp 1 ist z.B. dadurch gekennzeichnet, daß entsprechende Verben, die zu diesem Typ gehören, für alle Formen ein und dasselbe Segment benutzen. Daher bezeichnet man diese Verben auch als regelmäßig. Um so grösser die Zahl eines Verbttyps wird, um so unregelmäßiger werden die entsprechenden Verbformen gebildet. Die Klasse 11 enthält z.B. das völlig unregelmäßige Verb "sein". Das Verb "fahr" gehört zum Verbtyp 6.

Die Segmenttypen beschreiben nur morphosyntaktische Merkmale. Es ist jedoch möglich, die einzelnen Typen durch *morphophonologische* Merkmalen zu verfeinert. Diese Merkmale machen sich durch Veränderungen am Stammende oder am Anfang der Flexionsendung bemerkbar. Im Deutschen wird dies meist durch Einfügen oder Löschen von Teilen beim Übergang von Stamm und Endung realisiert. Beispielsweise erhalten alle Verben, deren Stamm mit "t" oder "d" enden, bei der Konjugation der regelmäßigen Formen ein "e". Daher ist die 2. Person Singular Form "arbeitest" korrekt, aber nicht "arbeitst". Für den Segmenttyp VGFA haben wir auf diese Art z.B. sieben Unterklassen bestimmt. In ähnlicher Weise werden auch die anderen flektierenden Wortarten (u.a. Nomen, Adjektive) klassifiziert.

Für jede Wortart läßt sich die gesamte Information über ihre Klassen durch einen mehr oder weniger komplexen n–ären Baum repräsentieren. Die Anzahl der Verzweigungen und die Tiefe der Bäume hängt vom Feinheitsgrad der klassifizierten Information ab.

Jedem Lexikoneintrag wird ein entsprechender Ausschnitt des n–ären Baums seiner Wortart zugewiesen. Ist das Lexem bezüglich der Wortart homograph, hat es entsprechend viele Teilbäume. Im Prinzip ist damit für jedes Lexem seine potientielle Flektierbarkeit festgelegt. Die folgende Tabelle zeigt Beispiele für Lexikoneinträge:

geh : ((WORTART VERB) (VTYP 3)
: (VGFA A1) (VGFC C1 "ging"))

ra : ((WORTART NOMEN) (GENUS MAS) (SG 0))
ras : ((WORTART VERB) (VTYP 1) (VGFA A4))
rast : ((WORTART VERB) (VTYP 1) (VGFA A2))
rast : ((WORTART NOMEN) (GENUS FEM) (SG 0) (PL 4))
raste : ((WORTART NOMEN) (GENUS FEM) (SG 0) (PL 3))

alt : ((WORTART ADJEKTIV) (KOMP EST)
: (UML NEC) (MAIN A1))
alter : ((WORTART NOMEN) (GENUS NTR)
: (SG 2) (PL 6))

haus : ((WORTART NOMEN) (GENUS NTR)
: (SG 3) (PL 9) (UML T))

Um zu entscheiden, ob eine aktuelle Wortform eine korrekte Flexionsendung enthält, wird ebenfalls das Klassifikationssystem herangezogen, so daß die möglichen Flexionsendungen des Deutschen nach den gleichen Kriterien strukturiert werden können. Im IAL ist jeder Endung ihre gesamte mögliche Kombination morphosyntaktischer Information zugewiesen. Damit die möglichen Alternativen voneinander unterschieden werden können, wird jeder unterschiedlichen morphosyntaktischen Information ein Aussschnitt des n–ären Baums der entsprechenden Wortart zugewiesen. Jedem Blattknoten des Teilbaumes wird die entsprechende morphosyntaktische Information zugewiesen. Damit ist ausgehend von der Endung ein eindeutiger Pfad zu spezifischer Information gegeben. Die Anzahl der Blattknoten spiegelt die Mehrdeutigkeit der Endung wider. Die folgende Abbildung zeigt einen Ausschnitt der Klassifikation der hochgradig mehrdeutigen Endung "en":

en

6.6.2 Der Algorithmus für die Generierung flektierter Wortformen

Ausgangspunkt für die Flexion lexikalischer Elemente ist der kanonische Stamm und spezifizierte grammatikalische Information. Diese Information wird in POPEL–HOW durch den Aufbau der DAGs bestimmt. Die wesentliche Aufgabe ist die Transformation des kanonischen Stamms auf eine Wortform gemäß der spezifizierten grammatikalischen Merkmale. Der Algorithmus für die morphologische Generierung läßt sich wie folgt beschreiben:

begin
1. Lexikonzugriff für das Lexem
2. **if not** erfolgreich **then return**
   keine Flexion möglich **else**
3. Bestimmung des Oberflächenstamms;
4. Konstruktion eines Suchpfades;
5. Bestimmung der Endung durch Traversierung relevanter
   n–ärer Bäume mit dem Suchpfad;
6. **if not** erfolgreich **then return**
   keine Flexion möglich **else**
7. Ausgabe der flektierten Wortform

end

Der Algorithmus besteht im Prinzip aus den zwei wesentlichen Teilaufgaben:

- Berechnung des Oberflächenstamms (Schritt 3)
- Bestimmung der korrekten Flexionsendung (Schritt 5)

Die Bestimmung der korrekten Wortform kann mit der angegebenen grammatikalischen und der aus dem Lexikon extrahierten Information eindeutig durchgeführt werden. Der Generierungsalgorithmus ist lexikonbasiert: Eine Form kann nur erzeugt werden, wenn das angegebene Lexem im Lexikon gefunden werden kann.

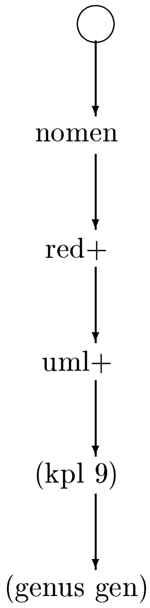
Bei der Berechnung des Oberflächenstamms müssen folgende Veränderungen am Stamm beachtet werden:

- Umlautung
- Konsonantenverdopplung
- Stammvokaländerungen
- Suppletivstämme

Umlautung findet z.B. bei der Pluralbildung von Nomen ("Haus" → "Häuser") oder bei der Steigerung von Adjektiven statt ("schwarz" → "schwärzer"). Endet ein Stamm auf <Konsonat>"in" oder "nis", findet eine Verdopplung des letzten Buchstabens bei der Pluralbildung von Nomen statt ("Freundin" → "Freundinnen"). Stammvokaländerungen finden sich vor allem bei Verben. Sie sind durch die Segmenttypen beschrieben. Als Suppletivstämme bezeichnet man solche Oberflächenstämme, deren Herleitung durch keine nachvollziehbare Regelmäßigkeit erklärt werden kann. Ein Beispiel für eine solche unregelmäßige Form ist das Verb "sein" mit seinen Konjugationsformen "bin", "bist", "ist", "sind" und "seid".

In MORPHIX wird z.Z. nur die Umlautung algorithmisch bestimmt. Die Konsonantenverdopplung wird durch die Aufnahme entsprechender Endungen im IAL geleistet. Die Phänomene Stammvokalveränderung und Suppletivstämme werden durch Aufnahme entsprechender unregelmäßiger Stämme bearbeitet. In diesen Fällen läßt sich die Oberflächenstammberechnung als Suche nach dem korrekten unregelmäßigen Stamm interpretieren.

Nachdem der Oberflächenstamm berechnet ist, muß als nächste Aufgabe die korrekte Endung bestimmt werden. Alle Endungen befinden sich im IAL. Um die korrekte Endung zu bestimmen, wird ein Suchpfad aufgebaut, der als Blatt die spezifizierte morphosyntaktische Information für die gewünschte Endung enthält. Diese Information wird aus der angegebenen grammatikalischen Information, aus Information, die bei der Stammberechnung bestimmt wurde, und der Klassifikationsinformation aus dem Lexikon hergeleitet. Die Wurzel des Pfades wird durch ein Platzhalterelement belegt. Bei erfolgreicher Endungssuche wird dieses Element durch die gefundene Endung ersetzt. Für das Nomen "Haus" mit spezifizierter Information: Numerus = pl und Genus = Genitiv wird zum Beispiel folgender Suchpfad erzeugt (red+ deutet an, daß ein Umlaut reduziert wurde, uml+, daß das Nomen im Plural einen Umlaut erhalten muß und kpl bezeichnet die Pluralklasse):



Dieser Suchpfad wird mit allen relevanten n–ären Bäumen der Endungen im IAL verglichen. Die Suche ist erfolgreich, wenn das Platzhalterelement durch eine Endung ersetzt werden kann. Für das Beispiel ist dies die Endung "ern". Die flektierte Wortform ist demnach "Häusern".

Im Prinzip wäre es auch möglich, die Elemente des IAL für die Generierungsrichtung so umzustrukturieren, daß die Wurzel durch die morphosyntaktische Information und die Blattknoten durch die Endungen belegt werden. Dies würde den Suchaufwand erheblich reduzieren. Zur Zeit wird diese Übersetzung des IAL nicht durchgeführt. Um den Suchaufwand zu reduzieren, werden die Endungen lediglich nach ihrer Wortklassenzugehörigkeit vorsortiert.

Chapter 7

Abschließende Bemerkungen

7.1 Zusammenfassung und Wertung

POPEL–HOW ist die erste inhaltsrealisierende Komponente, die einen über mehrere Ebenen bezogenen parallelen, inkrementellen und bidirektionalen Verbalisierungsprozeß realisiert.

Im Gegensatz zu inhaltsrealisierenden Komponenten, die eine vollständige Beschreibung der Ausgangsstruktur erhalten (vgl. u.a. [McKeown, 1985], [Busemann, 1987]), kann POPEL–HOW den Verbalisierungsprozeß ausgehend von unterspezifizierten Teilstrukturen durchführen.

Es ist in der Lage, fehlende Information zu entdecken und bei der übergeordneten Komponente POPEL–WHAT anzufordern. In diesem Sinne ist POPEL–HOW eine sehr selbständige Komponente, die es der inhaltsfestlegenden Komponente erlaubt, sich auf die wesentlichen Aufgaben beim inhaltlichen Planungsprozeß zu konzentrieren.

POPEL–HOW als inhaltsrealisierender Teil des Gesamtgenerierungssystems POPEL erhält damit einen sehr viel höheren Stellenwert und eine größere Eigenverantwortung während des Generierungsprozeßes als es bisher in Generierungssystemen der Fall ist (vgl. u.a. [McKeown, 1985] und [?]).

Zur Realisierung des Verbalisierungsprozesses ist ein sehr komplexer Informationsfluß notwendig, der hohe Anforderungen an die zugrundeliegenden Wissensquellen und Operationen setzt. Um die Anforderungen bezüglich der Operationen zu reduzieren, werden die einzelnen Teilstrukturen auf miteinander kooperierende Prozesse verteilt. Die Gesamtaufgabe wird dann von einer Gemeinschaft parallel arbeitender Prozesse gelöst, wobei jeder Prozeß nur für die Lösung einer isolierten Teilaufgabe zuständig ist. Allerdings bedingt dies die Möglichkeit der Dekomposition und Verteilung der zugrundeliegenden Wissensquellen.

Ein zentraler Gesichtspunkt bei der Entwicklung von POPEL–HOW ist die strikte Trennung der Wissensquellen und Operationen zur Konstruktion der spezifischen Strukturen auf jeder Ebene. Diese Vorgehensweise bietet den Vorteil, daß die zugrundeliegenden Wissensquellen deklarativ formuliert werden können oder bei der Entwicklung der Operationen von konkreten Inhalten weitgehend abstrahiert werden kann. Diese modulare Modellkonzeption hat sich gerade bei der Entwicklung der syntaktischen Wissensquelle bewährt, indem im Prinzip auf das vorhandene parallele Basismodell aufgesetzt werden konnte. Jedoch hat sich beim Entwurf der Grammatik POPEL–GRAM gezeigt, daß eine vollständige Trennung von Wissenseinheiten und Operationen im allgemeinen nicht möglich ist. Die explizite Formulierung der inhaltlichen Strukturen wird meiner Meinung nach davon beeinflußt, ob der zugrundeliegende Kontrollfluß z.B. sequentiell oder parallel verläuft. Dies ist bei einer effizienten Gestaltung der Wissensquellen zu beachten.

Im Vordergrund der Realisierung von POPEL–HOW standen zunächst die Enwicklung des parallelen Basismodells und der ebenenspezifischen Operationen sowie der Versuch, für die Entwicklung der einzelnen Wissensquellen vorhandene Werkzeuge einzusetzen. Beim Entwurf der Wissensquellen wurde ebenfalls versucht werden, vorhandenes Wissen, das bereits in der Analysephase verwendet wird, für den Generierungsprozeß zu benutzen. Es ist im Rahmen der Entwicklung von POPEL–HOW gelungen, folgende Wissensquellen und die damit verknüpften Werkzeuge des Gesamtsystems XTRA bidirektional zu nutzen:

1. die konzeptuelle Wissensquelle CKB
2. die funktionalsemantische Wissensquelle FSS
3. das Lexikon XTRALEX

Insgesamt bedeutet der bidirektionale Einsatz dieser Wissensquellen die Bestätigung des grundelegenenden Entwurfs von XTRA.

Die Analysegrammatik XTRA konnte in POPEL–HOW nicht eingesetzt werden, da sie wichtige Eigenschaften (wie z.B. Trennung von Konstituenz und linearer Abfolge) nicht erfüllt. Für den parallelen und inkrementellen Konstruktionsprozeß mußte daher eine eigene Grammatik entwickelt werden. Dies ist auch deshalb notwendig gewesen, weil bisher in keinem implementierten Generierungsystem eine derartige Grammatik eingesetzt wird und demnach zur Verfügung gestanden hätte.

Allerdings ist es möglich gewesen, die von mir entwickelte Grammatik POPEL–GRAM ebenfalls im PATR–II Formalismus zu beschreiben. Es können bei der Erweiterung oder Änderung von POPEL–GRAM die relevanten Module der Entwicklungsumgebung SB–PATR verwendet werden. Damit ist mit POPEL–GRAM erstmals eine auf dem PATR–II Formalismus basierende Generierungsgrammatik in einem Generierungsystem eingesetzt.

Bei der Formulierung der Regeln in POPEL–GRAM geht es in erster Linie darum, grammatikalisches Wissen, das bereits vorhanden ist, effizient in einem System wie POPEL–HOW verfügbar zu machen. Es ist das Ziel, bestehendes linguistisches Know–how um– und einzusetzen. Daher ist (zumindest im gegenwärtigen Stadium der Entwicklung von POPEL–HOW) POPEL–GRAM nicht als linguistischer Formalismus zu verstehen, sondern als Mittel zur Beschreibung des grammatikalischen Wissens in POPEL–HOW. In diesem Sinne ist POPEL–GRAM von solchen Generierungssystemen abzugrenzen, die explizit überprüfen, inwiefern bestimmte linguistische Theorien und ihre zugrundeliegenden Formalismen für den Generierungsprozeß verwendet werden können (vgl. z.B. [Busemann, 1987])[1].

Während der Entwicklung der syntaktischen Wissensquelle POPEL–GRAM hat sich gezeigt, daß eine Grammatik, die in einem System wie POPEL–HOW eingesetzt wird, über folgende Eigenschaften verfügen sollte:

- Deklarativ formuliert

[1]Dies schließt natürlich prinzipiell ihren Einsatz bei nachgewiesener Eignung nicht aus.

- Lexikonbasierte Regelauswahl
- Explizite Trennung von Konstituentenstruktur und linearer Abfolge
- Kontrollierbarer Merkmalsfluß
- Parallel und inkrementell einsetzbar
- Unifikationsbasiert

Es handelt sich hierbei im wesentlichen um solche Eigenschaften, die auch Kempen für eine inkrementell einsetzbare Grammatik fordert [Kempen, 1987], wobei bei POPEL–GRAM noch der besondere Aspekt der Parallelität zu berücksichtigen ist. Es hat sich gezeigt, daß die explizite Beachtung dependenztheoretischer Gesichtspunkte, wie sie in POPEL–GRAM realisiert sind, gerade für den parallelen Konstruktionsprozeß sehr gut geeignet ist.

Es wurde bereits weiter oben betont, daß bei der Realisierung von POPEL–HOW im Rahmen dieser Arbeit die Entwicklung der Algorithmen im Vordergrund gestanden hat. Es wird im Rahmen dieser Arbeit nicht angestrebt, eine detaillierte Spezifikation linguistischen Wissens anzugeben. Daher ist der bisherige Sprachumfang bezüglich der Grammatik beschränkt. Zur Zeit werden auf der syntaktischen Ebene folgende Phänomene behandelt:

- Erzeugung von Haupt–, Befehls– und Fragesätzen
- Beachtung der freien Wortstellung im Deutschen
- Beliebige Passiv–, Tempus–, Modalformen
- Anaphorisierung mit Pronomen
- Elliptifizierung
- Einfache NP–Generierung

Bezüglich der Flexion gibt es aufgrund des Einsatzes von MORPHIX kaum Einschränkungen. Es werden alle wichtigen Wortklassen bearbeitet und bei den einzelnen Wortklassen die wichtigsten auftretenden Phänomene behandelt (z.B. alle Steigerungsformen bei Adjektiven, Erzeugung aller Formen auch der unregelmäßigen Verben, Generierung unregelmäßiger Pluralformen bei Nomen etc.).

7.2 Ausblick

POPEL–HOW ist in seiner jetzigen Form als ein Experimentiersystem zu verstehen. Es sind im Rahmen von POPEL–HOW das notwendige operationale Basismodell und die auf jeder Ebene spezifischen Operationen entwickelt worden. Für die Formulierung der Wissensquellen sind existierende Werkzeuge (teilweise modifiziert) zur Verfügung gestellt. Daher ist das Hauptaugenmerk für die nächsten Arbeiten an POPEL–HOW auf die Erweiterung der Wissensquellen und die Verfeinerung der zur Zeit nur in vereinfachter Form implementierten speziellen Teilaufgaben gerichtet. Bezogen auf die syntaktischen Ebenen in POPEL–HOW bedeutet dies:

- Erweiterung des Regelapparates von POPEL–GRAM (u.a. zur Behandlung von Nebensätzen, komplexen Nominalphrasen). Als Grundlage kann hierbei ebenfalls [?] herangezogen werden.

- Oft unterscheiden sich die alternativen Regeln für eine Phrase nur durch wenige Merkmale voneinander. Der Aufwand der Regelformulierung ließe sich durch Angabe von Regeln, die die Phrasen auf einer abstrakteren Stufe beschreiben, reduzieren. Allerdings existieren hierfür bisher noch keine Mechanismen.

- Erweiterung der Templates. Dabei sollte darauf geachtet werden, möglichst viele Templates von der Analysegrammatik zu verwenden. Dies würde zur Verringerung der Redundanz im Gesamtsystem beisteuern.

- Ausbau der Wortwahl und der NP–Generierung. Beide Aufgabenbereiche sind in POPEL–HOW zur Zeit nur rudimentär implementiert. Es sollte gerade bei der NP–Generierung verstärkt durch Interaktion mit POPEL–WHAT Dialog – und Partnerwissen eingesetzt werden.

- Zur Zeit findet bei Verbalisierungsalternativen kein befriedigender Auswahlprozeß statt. So werden z.B. Aktiv– und Passivtransformationen gleichberechtigt behandelt unabhängig davon, ob die Aktivform aufgrund der Reihenfolge ihrer Satzglieder ambig ist. In solchen Fällen, wo das Akkusativobjekt an erster Stelle der linearen Kette steht, sollte die Wahl z.B. des Aktivsatz "Maria fährt Peter" vermieden und anstelle dessen der Passivsatz "Maria wird von Peter gefahren" geäußert werden. Es wäre prinzipiell möglich, durch Anwendung von Antizipationsrückkopplung die synaktisch eindeutige Form zu wählen.

Dem Anspruch entsprechend ist es möglich, diese hier angesprochenen Erweiterungen ohne Modifikation der Konzeption von POPEL–HOW in die bestehende Version zu integrieren. Im Rahmen des Projektes POPEL wird POPEL–HOW in den genannten Punkten weiterentwickelt.

Appendix A

Die ID–Regeln von POPEL–GRAM

(vcomp v e0 e6 an

((0 syntax voice) active)
((0 syntax) (1 syntax))
((1 syntax voice) (2 syntax voice))
((1 syntax number) (2 syntax number))
((1 syntax person) (2 syntax person))
((0 synchronize down voice) (0 syntax voice))
((0 head valence value) 2)
((0 head valence 1 form) e0)
((0 head valence 2 form) e6)
((0 head valence 1 mod) obli)
((0 head) (1 head))
((0 surface vcomp v) (1 surface)))

(vcomp v e0 e1 e6 an

((0 syntax voice) active)
((0 syntax) (1 syntax))
((1 syntax voice) (2 syntax voice))
((1 syntax voice) (3 syntax voice))
((1 syntax number) (2 syntax number))
((1 syntax person) (2 syntax person))
((0 synchronize down voice) (0 syntax voice))
((0 head valence value) 3)
((0 head valence 1 form) e0)
((0 head valence 2 form) e1)
((0 head valence 3 form) e6)
((0 head valence 1 mod) obli)
((0 head) (1 head))
((0 surface vcomp v) (1 surface)))

```
(vcomp v e0 e1 e6 an

        ((0 syntax voice) passive)
        ((0 syntax) (1 syntax))
        ((0 syntax voice) (2 syntax voice))
        ((0 syntax voice) (3 syntax voice))
        ((3 syntax number) (1 syntax number))
        ((3 syntax person) (1 syntax person))
        ((0 synchronize down voice) (0 syntax voice))
        ((0 head valence value) 3)
        ((0 head valence 1 form) e0)
        ((0 head valence 2 form) e1)
        ((0 head valence 3 form) e6)
        ((0 head valence 1 mod) obli)
        ((0 head) (1 head))
        ((0 surface vcomp v) (1 surface)))

(vcomp v e0 e1 e3 an

        ((0 syntax voice) active)
        ((0 syntax) (1 syntax))
        ((1 syntax voice) (2 syntax voice))
        ((1 syntax voice) (3 syntax voice))
        ((1 syntax number) (2 syntax number))
        ((1 syntax person) (2 syntax person))
        ((0 synchronize down voice) (0 syntax voice))
        ((0 head valence value) 3)
        ((0 head valence 1 form) e0)
        ((0 head valence 2 form) e1)
        ((0 head valence 3 form) e3)
        ((0 head valence 1 mod) obli)
        ((0 head) (1 head))
        ((0 surface vcomp v) (1 surface)))

(vcomp v e0 e1 e3 an

        ((0 syntax voice) passive)
        ((0 syntax) (1 syntax))
        ((0 syntax voice) (2 syntax voice))
        ((0 syntax voice) (3 syntax voice))
        ((3 syntax number) (1 syntax number))
        ((3 syntax person) (1 syntax person))
        ((0 synchronize down voice) (0 syntax voice))
        ((0 head valence value) 3)
        ((0 head valence 1 form) e0)
        ((0 head valence 2 form) e1)
        ((0 head valence 3 form) e3)
```

```
    ((0 head valence 1 mod) obli)
    ((0 head) (1 head))
    ((0 surface vcomp v) (1 surface)))

(vcomp v e0 e1 an

    ((0 syntax voice) active)
    ((0 syntax) (1 syntax))
    ((1 syntax voice) (2 syntax voice))
    ((1 syntax voice) (3 syntax voice))
    ((1 syntax number) (2 syntax number))
    ((1 syntax person) (2 syntax person))
    ((0 synchronize down voice) (0 syntax voice))
    ((0 head valence value) 2)
    ((0 head valence 1 form) e0)
    ((0 head valence 2 form) e1)
    ((0 head valence 1 mod) obli)
    ((0 head) (1 head))
    ((0 surface vcomp v) (1 surface)))

(vcomp v e0 e1 an

    ((0 syntax voice) passive)
    ((0 syntax) (1 syntax))
    ((0 syntax voice) (2 syntax voice))
    ((0 syntax voice) (3 syntax voice))
    ((3 syntax number) (1 syntax number))
    ((3 syntax person) (1 syntax person))
    ((0 synchronize down voice) (0 syntax voice))
    ((0 head valence value) 2)
    ((0 head valence 1 form) e0)
    ((0 head valence 2 form) e1)
    ((0 head valence 1 mod) obli)
    ((0 head) (1 head))
    ((0 surface vcomp v) (1 surface)))

(vcomp v e0 e4 an

    ((0 syntax voice) active)
    ((0 syntax) (1 syntax))
    ((1 syntax voice) (2 syntax voice))
    ((2 syntax number) (1 syntax number))
    ((2 syntax person) (1 syntax person))
    ((0 synchronize down voice) (0 syntax voice))
    ((0 head valence value) 2)
    ((0 head valence 1 form) e0)
    ((0 head valence 2 form) e4)
```

```
    ((0 head valence 1 mod) obli)
    ((0 head) (1 head))
    ((0 surface vcomp v) (1 surface)))

(e1 np

    ((0 anapher) (1 anapher))
    ((0 syntax voice) active)
    ((0 syntax case) akk)
    ((1 syntax) (0 syntax))
    ((0 synchronize down case) (0 syntax case))
    ((0 synchronize up voice) (0 syntax voice))
    ((0 head) (1 head))
    ((0 surface np) (1 surface)))

(e1 np

    ((0 anapher) (1 anapher))
    ((0 syntax voice) passive)
    ((0 syntax case) nom)
    ((0 syntax) (1 syntax))
    ((0 synchronize down case) (0 syntax case))
    ((0 synchronize up voice) (0 syntax voice))
    ((0 head) (1 head))
    ((0 surface np) (1 surface)))

(e0 np

    ((0 anapher) (1 anapher))
    ((0 syntax voice) active)
    ((0 syntax case) nom)
    ((1 syntax) (0 syntax))
    ((0 synchronize down case) (0 syntax case))
    ((0 synchronize up voice) (0 syntax voice))
    ((0 head) (1 head))
    ((0 surface np) (1 surface)))

(e0 pp

    ((0 anapher) (1 anapher))
    ((0 syntax voice) passive)
    ((0 syntax case) dat)
    ((1 syntax) (0 syntax))
    ((0 synchronize down case) (0 syntax case))
    ((0 synchronize up voice) (0 syntax voice))
    ((0 head) (1 head))
    ((1 head stem) "von")
    ((0 surface pp) (1 surface)))
```

```
(e0 pp
        ((0 anapher) (1 anapher))
        ((0 anapher) -)
        ((0 syntax voice) passive)
        ((0 syntax case) akk)
        ((1 syntax) (0 syntax))
        ((0 synchronize down case) (0 syntax case))
        ((0 synchronize up voice) (0 syntax voice))
        ((0 head) (1 head))
        ((1 head stem) "durch")
        ((0 surface pp) (1 surface)))

(e6 pp
        ((0 anapher) (1 anapher))
        ((0 syntax) (1 syntax))
        ((0 synchronize down case) (0 syntax case))
        ((0 head) (1 head))
        ((0 surface pp) (1 surface)))

(e4 pp
        ((0 anapher) (1 anapher))
        ((1 syntax) (0 syntax))
        ((0 synchronize down case) (0 syntax case))
        ((0 head) (1 head stem))
        ((0 surface pp) (1 surface)))

(e3 np
        ((0 anapher) (1 anapher))
        ((0 syntax case) dat)
        ((1 syntax) (0 syntax))
        ((0 synchronize down case) (0 syntax case))
        ((0 head) (1 head))
        ((0 surface np) (1 surface)))

(an pp
        ((0 anapher) (1 anapher))
        ((0 head) (1 head))
        ((1 syntax) (0 syntax))
        ((0 synchronize down case) (0 syntax case))
        ((0 surface pp) (1 surface)))

(an np
```

```
    ((0 anapher) (1 anapher))
    ((0 head) (1 head))
    ((1 syntax) (0 syntax))
    ((0 syntax case) akk)
    ((0 synchronize down case) (0 syntax case))
    ((0 surface np) (1 surface)))

(pp prep np

    ((0 anapher) (2 anapher))
    ((1 syntax) (0 syntax))
    ((2 syntax case) (1 syntax case))
    ((0 head) (1 head))
    ((0 surface prep stem) (1 head stem))
    ((0 surface prep syntax) (1 syntax))
    ((0 surface np) (2 surface)))

(np np-eng

    ((0 anapher) (1 anapher))
    ((0 syntax) (1 syntax))
    ((0 synchronize down number) (0 syntax number))
    ((0 synchronize down person) (0 syntax person))
    ((0 synchronize down case) (0 syntax case))
    ((0 synchronize down article) (0 syntax article))
    ((0 synchronize down gender) (0 syntax gender))
    ((0 head) (1 head))
    ((0 surface np-eng) (1 surface)))

(np np-eng ne

    ((0 anapher) (1 anapher))
    ((0 syntax) (1 syntax))
    ((0 synchronize down number) (0 syntax number))
    ((0 synchronize down person) (0 syntax person))
    ((0 synchronize down case) (0 syntax case))
    ((0 synchronize down article) (0 syntax article))
    ((0 synchronize down gender) (0 syntax gender))
    ((2 syntax number) (1 syntax number))
    ((2 syntax person) (1 syntax person))
    ((0 head) (1 head))
    ((0 surface np-eng) (1 surface)))

(ne relpron vcomp

    ((0 syntax) (2 syntax))
    ((0 syntax number) (1 syntax number))
    ((0 syntax gender) (1 syntax gender))
```

```
        ((0 syntax type) relative)
        ((0 synchronize down number) (0 syntax number))
        ((0 synchronize down person) (0 syntax person))
        ((0 synchronize up) (0 synchronize down))
        ((0 surface relpron) (1 surface))

(ne np

        ((0 anapher) (1 anapher))
        ((0 anapher) -)
        ((0 syntax case) gen)
        ((1 syntax) (0 syntax))
        ((0 synchronize down case) (0 syntax case))
        ((0 head) (1 head))
        ((0 surface np) (1 surface)))

(np np-eng ap

        ((0 anapher) (1 anapher))
        ((0 syntax) (1 syntax))
        ((1 syntax) (2 syntax))
        ((0 head) (1 head))
        ((0 synchronize down number) (0 syntax number))
        ((0 synchronize down person) (0 syntax person))
        ((0 synchronize down case) (0 syntax case))
        ((0 synchronize down article) (0 syntax article))
        ((0 synchronize down gender) (0 syntax gender))
        ((0 surface np-eng) (1 surface))
        ((0 surface ap) (2 surface)))

(np-eng n det

        ((0 syntax article) def)
        ((0 anapher) -)
        ((0 anapher) (1 anapher))
        ((0 syntax) (1 syntax))
        ((1 syntax) (2 syntax))
        ((0 head) (1 head))
        ((0 surface n) (1 surface))
        ((0 surface det) (2 surface)))

(np-eng n det

        ((0 syntax article) indef)
        ((0 anapher) -)
        ((0 anapher) (1 anapher))
        ((0 syntax) (1 syntax))
        ((1 syntax) (2 syntax))
```

```
        ((0 head) (1 head))
        ((0 surface n) (1 surface))
        ((0 surface det) (2 surface)))

(np-eng n

        ((0 anapher) -)
        ((0 syntax article) without)
        ((0 anapher) (1 anapher))
        ((0 syntax) (1 syntax))
        ((0 head) (1 head))
        ((0 surface n) (1 surface)))

(np-eng perspron

        ((0 syntax) (1 syntax))
        ((0 anapher) +)
        ((0 syntax article) without)
        ((0 anapher) (1 anapher))
        ((0 head) (1 head))
        ((0 surface perspron) (1 surface)))

(ap adj ap

        ((0 syntax) (1 syntax))
        ((0 synchronize up number) (0 syntax number))
        ((0 synchronize up person) (0 syntax person))
        ((0 synchronize up case) (0 syntax case))
        ((0 synchronize up article) (0 syntax article))
        ((0 synchronize up gender) (0 syntax gender))
        ((0 synchronize down) (0 synchronize up))
        ((1 syntax gender) (0 syntax gender))
        ((1 syntax case) (0 syntax case))
        ((1 syntax) (2 syntax))
        ((0 head) (1 head))
        ((0 head rest) (2 head))
        ((0 surface adj) (1 surface))
        ((0 surface rest) (2 surface)))

(ap adj

        ((0 head) (1 head))
        ((0 synchronize up number) (0 syntax number))
        ((0 synchronize up person) (0 syntax person))
        ((0 synchronize up case) (0 syntax case))
        ((0 synchronize up article) (0 syntax article))
        ((0 synchronize up gender) (0 syntax gender))
        ((0 synchronize down) (0 synchronize up))
```

```
((1 syntax) (0 syntax))
((1 syntax gender) (0 syntax gender))
((1 syntax case) (0 syntax case))
((0 surface adj) (1 surface)))
```

Appendix B

Die Linearisierungsregeln von POPEL–GRAM

```
(vcomp frontfield v-fin mainfield v-infin restfield

        ((0 syntax sentence-form) declarative)
        ((1 dependent) +)
        ((3 dependent) +)
        ((5 dependent) +))

(vcomp v-fin mainfield v-infin restfield

        ((0 syntax sentence-form) decision-question)
        ((2 dependent) +)
        ((4 dependent) +))

(vcomp q-frontfield v-fin mainfield v-infin restfield

        ((0 syntax sentence-form) completion-question)
        ((1 dependent) +)
        ((3 dependent) +)
        ((5 dependent) +))

(q-frontfield interrogativpronomen)

(frontfield any)

(mainfield e1-0 e1-1 e1-3 multiple-any)

(restfield any)

(np det adj ap n ne

        ((0 anapher) -)
        ((3 dependent) +)
        ((5 dependent) +))
```

```
(np-eng det n)

(np e1-0
        ((0 anapher) +))

(np e1-1
        ((0 anapher) +))

(np e1-3
        ((0 anapher) +))

(pp prep np
        ((2 dependent) -))
```

Bibliography

[Appelt, 1985] D. Appelt. *Planning English Sentences.* Cambridge University Press, Cambridge, 1985.

[Busemann, 1987] S. Busemann. Generierung mit gpsg. In K. Morik, editor, *Proceedings der 11. GWAI*, Berlin, 1987. Springer.

[Hovy, 1987] E. Hovy. *Generating Natural Language Under Pragmatic Constraints.* PhD thesis, Yale University, 1987.

[Kaplan, 1983] J.S. Kaplan. Cooperative responses from a portable natural language data base query system. In M. Brady and R. C. Berwick, editors, *Computational Models of Discourse Processing*, pages 167–208. MIT-Press, Cambridge, 1983.

[Kempen, 1986] G. Kempen. Language generation systems. In I. Batori, W. Lenders, and W. Putschke, editors, *Computational Linguistics. An International Handbook on Computer Language Research and Applications.* Walter de Gruyter, Berlin/New York, 1986.

[Kempen, 1987] G. Kempen. A framework for incremental syntactic tree formation. In *Tenth IJCAI*, pages 655–660, Mailand, 1987.

[Mann, 1988] W. C. Mann. Rhetorical structure theory and implicit communication. In *4th IWG*, Santa Catalina Island, 1988. Abstract, 3 Pages.

[McDonald, 1980] D. D. McDonald. *Natural Language Production as a Process of Decision–making Under Constraints.* PhD thesis, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1980.

[McDonald, 1986] D. D. McDonald. Natural language generation: Complexities and techniques. In S. Nirenburg, editor, *Theoretical and Methodological Issues in Machine Translation.* Cambridge University Press, London, 1986.

[McKeown, 1985] K. R. McKeown. *Text generation.* Cambridge University Press, Cambridge, 1985.