Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-92-10

Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken

Jakob Mauss

Mai 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Philips, SEMA Group Systems, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken

Jakob Mauss

DFKI-D-92-10

Diese Arbeit wurde von Prof. Michael M. Richter und
Herrn Dipl. Inform Christoph Klauck betreut.

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-8902 C4).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1992

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken

Jakob Mauss

Mai 1992

Vorwort

Um eine enge Kopplung der CA*-Komponenten in einem CIM-System zu erreichen, sind vor allem geeignete Schnittstellen zwischen diesen Komponenten zu definieren. Ein Ansatz, der dazu derzeit untersucht wird, basiert auf sogenannten Feature-Sprachen und deren Repräsentation durch Graph-Grammatiken. In dieser Arbeit wird ein solcher Graph-Grammatik-Formalismus definiert, vor allem aber ein Graph-Parser dafür angegeben und dessen Anwendung auf das Feature-Erkennungsproblem in CIM-Systemen demonstriert. Der vorgestellte heuristisch steuerbare, chartbasierte bottom-up Parser ermöglicht die Analyse attributierter Graphen, die Werkstücke repräsentieren, mittels einer gegebenen Graph-Grammatik, die die Feature-Sprache (das Konzeptwissen) eines menschlichen Maschinenbau-Experten repräsentiert. Die Ausgabe des Parsers ist eine qualitative Beschreibung des Werkstücks in der jeweiligen Feature-Sprache in Gestalt eines sogenannten Feature-Baums.

Diese Arbeit wurde am DFKI in Kaiserslautern unter Benutzung der dortigen hervorragenden Rechnerausstattung durchgeführt, wofür ich dankbar bin. An dieser Stelle möchte ich mich auch bei Prof. Richter und den Mitarbeitern des ARC-TEC Projekts bedanken. Christoph Klauck hatte die Idee zu dieser Arbeit. Bei ihm möchte ich mich hier ganz besonders für das Korrekturlesen mehrerer Zwischenversionen und zahlreiche Verbesserungsvorschläge bedanken.

Kaiserslautern, Mai 1992 Jakob Mauss

Inhaltsverzeichnis

Kapitel 1

Motivation

Anonyme Standardprodukte werden zunehmend durch kundenspezifische Erzeugnisse verdrängt. Als Folge der damit verbundenen wachsenden Komplexität und Vielfalt der angebotenen Produkte müssen sich die meisten Fertigungsunternehmen auf schrumpfende Losgrößen, kürzer werdende Produktlebenszyklen und verringerte Entwicklungs- und Durchlaufzeiten einstellen.

Diese Entwicklung motiviert den Versuch, den Entwurfs- und Fertigungsprozeß zu optimieren, indem man die daran beteiligten CA*-Systeme[1] enger als bisher miteinander koppelt. Ziel dieser sogenannten CIM-Idee[2] ist die vollständig integrierte Computerunterstützung des gesamten Prozesses vom Entwurf bis zur Fertigung auf einer computergesteuerten Werkzeugmaschine.

Das CAD-System soll dabei eine vollständige Produktbeschreibung des zu fertigenden Werkstücks liefern. Vollständig heißt, daß die Beschreibung alle für die Fertigungsplanung relevanten Produktdaten enthält, also neben einer Beschreibung der geometrischen Form durch Punktkoordinaten, Kanten und Flächenelemente auch alle topologischen, funktionsbezogenen und technologischen Daten wie Oberflächenbeschaffenheit, Maßtoleranzen usw. Ein nachgeschaltetes KI-Planungsverfahren nutzt diese Informationen, um einen Arbeitsplan für die Fertigung des Produkts zu entwerfen. Dabei muß z.B. entschieden werden, welche Maschinen und Werkzeuge, Rohmaterialien und Einzelteile verwendet werden, wieviel Personal und Arbeitszeit zur Herstellung nötig ist usw.

Zur Realisierung der CIM-Idee sind vor allem geeignete Schnittstellen zwischen den einzelnen CA*-Systemen zu definieren. CAD-Systeme liefern z.B. heute meist nur eine geometrische Beschreibung eines Produkts, das ist im einfachsten Fall eine Liste mit Punktkoordinaten, Kanten und Oberflächenelementen. Eine Repräsentation des Produkts auf dieser rein geometrischen Ebene reicht aber zur Realisierung der CIM-Idee nicht aus, sie ist unvollständig. Zur Zeit wird von der ISO[3] an einem Datenformat - STEP[4] - gearbeitet, das 1993 weltweiter Standard werden soll. In diesem Format sollen alle während des Lebenszyklus eines Industrieprodukts auftretenden Daten abgebildet werden können. Durch die Standardisierung soll Kompatibilität zwischen den verschiedenen CA*-Systemen erreicht werden.

[1] Computer Aided (Design, Process Planing, Manufacturing ...)
[2] Computer Integrated Manufacturing
[3] International Standardisation Organisation
[4] Standard for the Exchange of Product Model Data

In einem Datenformat wie STEP oder einem anderen, dem CIM-Gedanken verpflichteten Format ist ein Produkt nicht nur auf der rein geometrischen Ebene repräsentiert, sondern zusätzlich auf anderen, mehr qualitativeren Ebenen, auf denen Begriffe wie Topologie, Baugruppe oder Funktion eine Rolle spielen. Auf den damit verfügbaren, universellen Beschreibungselementen lassen sich dann firmen- und domänenspezifische sogenannte *Feature* definieren.

In [BKL 91b] wird der Begriff Feature genauer untersucht und als ein Beschreibungselement definiert, das auf den geometrischen und technologischen Daten eines Produkts aufsetzt und das ein Experte in seiner Domäne mit bestimmten Informationen assoziiert.

Im Gegensatz zum universellen STEP-Format ist die Feature-Beschreibung eines Produkts firmenspezifisch, d.h. abhängig von der in der Firma benutzten Terminologie, den Produktionsmethoden, verfügbaren Werkzeugen usw.

Die Feature-Beschreibung eines Produkts ist außerdem domänenspezifisch, d.h. geschieht aus der subjektiven Sicht eines Experten: Ein Konstrukteur sieht ein Produkt mit anderen Augen als z.B. ein Experte für Dreh- oder Frästeilfertigung. Alle drei werden darin jeweils andere, für ihr jeweiliges Arbeitsgebiet relevante Feature-Strukturen entdecken, mit denen sie gewisse Erfahrungen, Arbeitskniffe usw. verbinden.

Die Feature-Beschreibung eines Produkts trägt also firmen- und domänenspezifische Informationen, die für eine kompetente Arbeitsplanung unentbehrlich sind. Die notwendigerweise fehlende Eindeutigkeit der Feature-Beschreibung wirft für die Arbeitsplanung in einem CIM-System folgendes Problem auf:

- Die Produktbeschreibung liegt in einer Form vor, die mindestens alle geometrisch-topologischen Daten enthält, z.B. im STEP-Format, ggf. zusätzlich in Form einer Beschreibung in der Feature-Sprache eines Konstrukteurs.
- Für die Arbeitsplanung wird aber eine Beschreibung unter fertigungsrelevanten Gesichtspunkten, d.h. in der Feature-Sprache eines Fertigungsexperten gebraucht.

Eine rein geometrisch-topologische Beschreibung des Produkts ist für Planungszwecke ungeeignet. Das Problem der Umsetzung dieser Beschreibung in eine gegebene Feature-Sprache ist das sogenannte Feature-Erkennungsproblem.

1.1 Werkstücke als Graphen

Für die Lösung dieses Problems wird seit den 80er Jahren ein Ansatz untersucht, der auf Graphen bzw. auf Graph-Grammatiken basiert. Die Feature-Erkennung setzt dabei auf einer geometrisch-topologischen Repräsentation des Werkstücks auf: Werkstücke werden als Graphen repräsentiert, wobei die Knoten meist für elementare Flächenstücke stehen (Rechteckflächen, Kreisringflächen, Zylindermantelflächen usw.) und die Kanten die Nachbarschaftsbeziehungen zwischen den Flächen kodieren [GaHe 90, FlBr 89]. Dabei kann die Art der Nachbarschaft, z.B. konkav oder konvex, durch entsprechende Kantensorten repräsentiert werden [JoCh 88]. In komplizierteren Graph-Repräsentationen werden Knotensorten eingeführt, die Knoten repräsentieren dann je nach Sorte Punkte,

Kanten, geschlossene Konturlinien oder Flächen [FiSa 90]. In [CH 91] kodieren die Knoten je nach Sorte gerade, konvexe oder konkave Flächenkanten oder Flächenstücke. Der zu einem Werkstück gehörende Werkstückgraph läßt sich vergleichsweise einfach aus der Ausgabe eines CAD-Systems berechnen, falls diese Ausgabe mindestens eine vollständige geometrisch-topologische Werkstückbeschreibung liefert. Nicht alle der oben zitierten Arbeiten setzen zur Analyse der resultierenden Werkstückgraphen Graph-Grammatiken ein.

1.2 Feature-Erkennung als Graph-Parsing-Problem

In zwei der oben zitierten Arbeiten werden die zu erkennenden Feature durch Produktionen einer Graph-Grammatik repräsentiert. [FiSa 90] verwenden kontextfreie Graph-Grammatik-Produktionen. Die Produktionen werden nicht miteinander verkettet, d.h. Feature nicht aus anderen Featuren zusammengesetzt. Feature sind also nicht hierarchisch strukturiert. Die rechten Seiten der Produktion sind daher ziemlich groß. Die Knoten der rechten Regelseite werden in einer sogenannten Compilierungsphase nach Restriktivität geordnet. Dadurch wird der Aufwand bei der Instantierung einer Produktion verringert (early pruning). In [CH 91] werden sogenannte Web-Grammatiken zur Feature-Definition verwendet. Beide Grammatik-Formalismen sind kontextfrei, d.h. die linke Seite einer Produktion besteht jeweils aus einem einzigen Knoten. Jede Produktion repräsentiert eine Feature-Definition und die ganze Grammatik repräsentiert die Feature-Sprache eines menschlichen Experten. Feature-Erkennung wird damit zu einem Parsing-Problem.

1.3 Problemstellung

Vorgeschichte

Am DFKI[5] in Kaiserslautern wird im Rahmen des Projekts ARC-TEC[6] eine Expertensystem-Shell für ein Teilgebiet der Fertigungstechnik entwickelt. Exemplarische Anwendungen dieser Shell sind z.B. Expertensysteme für die Arbeitsplanung von Dreh- bzw. von Frästeilen. Im Rahmen des Projekts wurde die Repräsentationssprache TEC-REP [BKL 91a] entwickelt, die auf dem in Abschnitt 1.1 und 1.2 beschriebenen Graph-Ansatz basiert: Werkstücke werden als attributierte Graphen repräsentiert, die Knoten stehen für elementare Oberflächenstücke, die Knotensorten kodieren den Flächentyp. Die Knoten haben Attribute, die die geometrischen und technologischen Detailinformationen tragen. Die Kanten des Graphen kodieren die Topologie des Werkstücks: Zwei Knoten sind benachbart, wenn die entsprechenden Flächenstücke einen gemeinsamen Rand haben; Berührung einzelner Eckpunkte reicht dazu nicht.

Rotationssymmetrische Drehteile werden in TEC-REP durch Graphen repräsentiert, deren Knoten eine lineare Kette bilden. Diese Graphen können als Strings aufgefaßt werden. In [BKS 91] wird eine Feature-Grammatik für die eingeschränkte Klasse rotationssymmetrischer Drehteile angegeben. Sie wurde im D-PATR-Formalismus [Kar 86] implementiert und an einigen Beispielen getestet. Die dabei gemachten Erfahrungen führten zu

[5] Deutsches Forschungszentrum für Künstliche Intelligenz

[6] Aquisition, Repräsentation und Compilation von TEChnischen Wissen

dem Entschluß, ein eigenes, speziell auf das Feature-Erkennungsproblem zugeschnittenes Graph-Parsing-Tool zu entwickeln. Damit sollte nicht nur die Behandlung des Spezialfalls der durch String-Grammatiken beschreibbaren Drehteile möglich sein, sondern auch der Fall der durch allgemeine Graph-Grammatiken beschreibbaren Frästeile.

Zielsetzung der Arbeit

Gegenstand dieser Arbeit ist die Entwicklung eines Graph-Parsers für die Feature-Erkennung, der auf TEC-REP als Repräsentationssprache für Werkstücke aufsetzt. Die Feature sind durch Produktionen einer Graph-Grammatik definiert. Bei der Entwicklung des Parsers werden die folgenden Feature-spezifischen Charakteristika berücksichtigt:

1. *Überlappung:* Feature können sich überlappen, d.h. ein Flächenstück kann zu mehreren Featuren gehören (siehe Abb.1.2).

2. *Dimensionsabhängigkeit:* Topologisch identische Objekte können in Abhängigkeit von ihren geometrischen Abmessungen als unterschiedliche Feature erkannt werden.

3. *Hierarchie:* Die vollständige Feature-Beschreibung eines Werkstücks bildet eine Hierarchie, den sogenannten Feature-Baum.

4. *Qualitative Beschreibung:* Die Feature-Beschreibung sollte sich von der geometrisch-topologischen Ebene lösen und eine kompaktere, qualitative Interpretation des Werkstücks liefern.

5. *Ambiguenz:* Feature lassen sich i. allg. auf sehr viele syntaktisch unterschiedliche, semantisch aber oft gleichwertige Weise herleiten, man ist aber meist nur an einer, möglichst guten, d.h. prägnanten Beschreibung interessiert.

6. *Ähnliche Feature:* Feature unterscheiden sich oft nur in Details, eine kompakte Formulierbarkeit und effiziente Behandlung von Varianten und Spezialfällen ist wünschenswert.

1.4 Gliederung der Arbeit und Überblick

Zur Lösung der im vorigen Abschnitt gestellten Aufgabe wird im zweiten Kapitel zunächst ein Graph-Grammatik-Formalismus ANLGG[7] definiert, in dem Feature und Feature-Sprachen mit den angegebenen Charakteristika formulierbar sind. Abb.1.1 zeigt ein einfaches Feature und seine Definition durch eine ANLGG Produktion.

Ein Winkel (Sorte ANGLE) besteht aus zwei miteinander verbundenen Rechteckflächen (Sorte RA wie RectAngular), die innerhalb des Winkel-Features big-face und small-face heißen. Diese graphische Definition wird durch weitere Bedingungen präzisiert: Die beiden Flächen müssen Material in einem konvexen Winkel einschließen und die Oberfläche von small-face darf nicht größer sein als die von big-face. Außerdem hat der Winkel ein Attribut area, das seine Gesamtoberfläche angibt.

[7] Attributed Node Labeled Graph Grammar

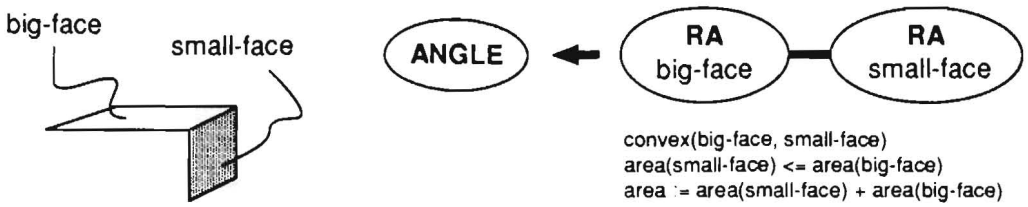


Abbildung 1.1: Eine einfache Feature-Definition

Abb.1.2 zeigt eine weitere Produktion einer ANLGG. Eine Ecke (Sorte CORNER) besteht aus zwei Winkeln, die einander in ihren größten Seitenflächen überlappen, d.h. diese Flächen sind identisch. Außerdem müssen die beiden kleineren Winkelflächen einander berühren, die Kante zwischen den beiden small-faces spezifiziert diese sogenannte tiefe Nachbarschaft. In ANLGG Produktionen werden drei Arten von Kanten (Relationen) verwendet, um die Verbindung oder Überlappung zweier Knoten oder Knotenkomponenten zu spezifizieren oder Verbindungen ausdrücklich zu verbieten.

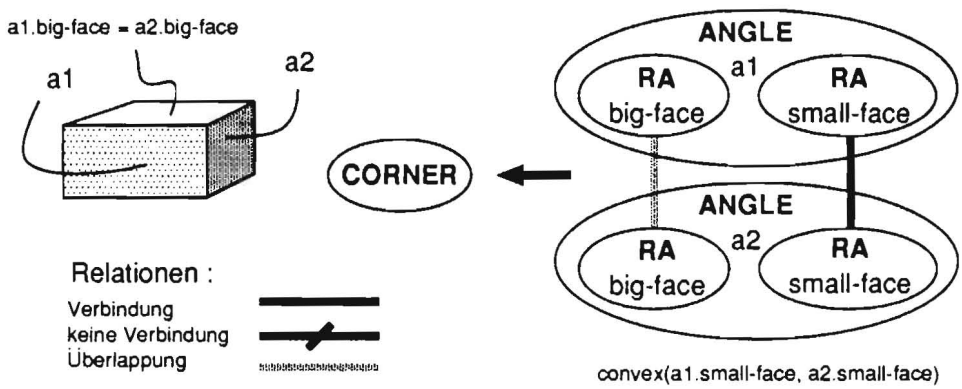


Abbildung 1.2: Eine Feature-Definition mit tiefer Nachbarschaft und Überlappung

Das zweite Kapitel endet mit einem Vergleich der ANLGGs mit anderen Graph-Grammatik-Formalismen.

Das dritte Kapitel bildet den Kern dieser Arbeit. Hier wird der heuristisch gesteuerte Chart-Parser für ANLGGs vorgestellt. Der Parser arbeitet bottom-up und basiert auf einem Parser für Flußdiagramme von R. Lutz [Lut 89]. Hauptunterschiede sind die Verwendung von sucheleitenden und suchraumbeschneidenden Heuristiken, die Möglichkeit der Behandlung von Überlappungen, die Möglichkeit, Nachbarschaft und Überlappung tief (durch Angabe von Pfaden) zu spezifizieren und der Einsatz von Sortenhierarchien.

Das Kapitel wird mit einer Diskussion der Eigenschaften des Parsers beschlossen. Dabei werden Termination, Vollständigkeit und Komplexität angesprochen und untersucht, ob der ANLGG-Formalismus und der zugehörige Parser die oben genannten Anforderungen erfüllt.

Der Parser wurde im Rahmen dieser Arbeit in Common Lisp implementiert und mit Feature-Grammatiken für Dreh- und für Frästeile getestet. Die dabei gesammelten Erfahrungen werden im vierten Kapitel besprochen.

Im fünften Kapitel werden die Resultate dieser Arbeit zusammengefaßt. Dabei werden einige offen gebliebene Fragen angesprochen. Insbesondere werden hier Strategien zur

Erreichung praktikabler Parserlaufzeiten skizziert, die aber im Rahmen dieser Arbeit nicht mehr praktisch erprobt wurden.

Der Parser wurde um Module für die Entwicklung und das Austesten von ANLGGs erweitert. Das resultierende Parsing-Tool *GraPaKL* wird im Anhang (A und B) dokumentiert. Außerdem findet sich hier (C) ein Papier, daß während der Anfertigung dieser Arbeit zusammen mit Christoph Klauck erarbeitet wurde.

Kapitel 2

Graph-Grammatiken und ANLGGs

In diesem Kapitel werden ANLGGs[1] formal definiert. Die eingeführten Begriffe werden dabei jeweils am Beispiel der Graphen aus Abb.2.1 (mit drei Knoten RA_1, RA_2 und RA_3) und Abb.2.2 und den beiden ANLGG-Produktionen p_1 und p_2 aus Abb.2.3 erläutert.

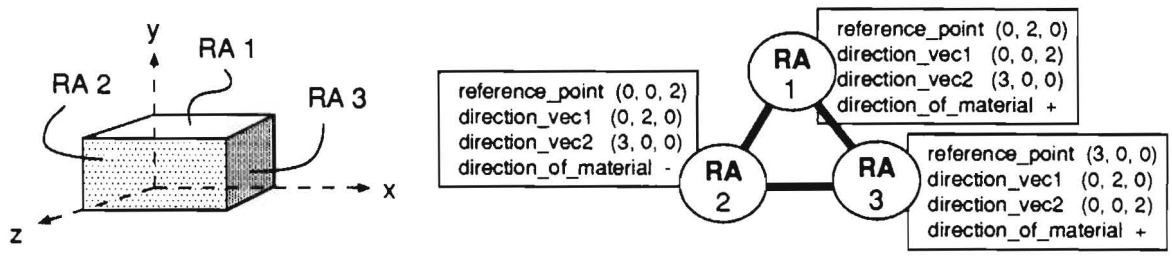


Abbildung 2.1: Ein attributierter, terminaler Graph mit drei Knoten

2.1 Definitionen

Ein Knoten ist in unserem Formalismus eine Merkmalstruktur (feature structure [2]) mit zwei Arten von Merkmalen: Attribut und Rolle.

Definition 2.1 *Ein* **Knoten** v *ist eine partielle Funktion* $v : \mathrm{dom}(v) \cap \mathcal{R} \rightarrow \mathcal{V}$ *und* $\mathrm{dom}(v) \cap \mathcal{A} \rightarrow \mathcal{W}$. *Dabei ist* $dom(v) \subset \mathcal{R} \cup \mathcal{A}$ *und* $\mathcal{V}, \mathcal{A}$ *und* $\mathcal{R}$ *sind die paarweise disjunkten, nichtleeren, abzählbar unendlichen Mengen aller Knoten, Attribute und Rollen.* $\mathcal{W}$ *ist eine beliebige, nicht leere Menge von Attributwerten. Die Elemente aus* $\mathbf{roles}(v) := \mathrm{dom}(v) \cap \mathcal{R}$ *sind die* **Rollen** *von* v, *die Elemente der Menge* $\mathbf{cast}(v) := \mathrm{val}(v\,|_{\mathrm{roles}(v)})$ *sind die* **Rollenwerte** *von* v. *Die Elemente aus* $\mathrm{dom}(v) \cap \mathcal{A}$ *sind die* **Attribute** *von* v[3].

[1] Attributed Node Labeled Graph Grammars

[2] Dabei bezieht sich feature nicht etwa auf den Begriff Feature wie er im vorigen Kapitel eingeführt wurde, sondern auf den gleichnamigen Begriff in der Computerlinguistik. Dort spielen Feature-Strukturen speziell im Kontext der Unifikations-Grammatiken eine ähnliche Rolle wie hier die Knoten [Sel 87, Kar 86].

[3] Dabei bezeichnen $\mathrm{dom}(v)$ und $\mathrm{val}(v)$ den Definitions- bzw. Wertebereich von v und $|$ die Einschränkung einer Funktion auf die angegebene Menge.

Der Attributwert $w := v(a) \in \mathcal{W}$ eines Attributs a in einem Knoten v ist ein beliebiges Objekt, also ein Stück Information, eine Zahl, ein Symbol, ein normierter Vektor usw. Beispielsweise ist der Knoten RA_1 für genau vier Attribute (reference_point, direction_of_material usw.) definiert (Sprechweise: „Er hat vier Attribute") und es gilt z.B. RA_1(reference_point) = (0, 2, 0).

Der Rollenwert $v^* := v(r) \in \mathcal{V}$ einer Rolle r in einem Knoten v ist selbst ein Knoten, falls die Rolle in v definiert ist; roles(v) liefert alle Rollen des Knotens und cast(v) die zugehörigen Rollenwerte.

Außerdem definieren wir cast* als den transitiven Abschluß der Funktion cast, d.h. cast*(v) liefert alle Rollenwerte von v und Rollenwerte von Rollenwerten von v usw., einschließlich v. Der Stern in cast* steht also für null oder beliebig viele rekursive Anwendungen von cast.

Definition 2.2 *Die Knoten in* cast*(v) *heißen* **Komponenten** *von v. Dabei ist*

$$\mathbf{cast}^*(v) := \begin{cases} \{v\} & : \ falls\ \mathrm{cast}(v) = \emptyset \\ \bigcup_{v^* \in \mathrm{cast}(v)} \mathrm{cast}^*(v^*) & : \ sonst \end{cases}$$

Manchmal verwenden wir Knotenbezeichnungen mit hochgestelltem Stern, wie in v^*, um anzudeuten, daß v^* eine Komponente von v ist. Man kann die Funktion cast auch als partielle Ordnung $<$ auffassen, d.h. $\forall\ v^* \in \mathrm{cast}(v) : \ v^* < v$, und cast* als deren transitiven Abschluß. Das Hasse-Diagramm[4] der Komponenten eines Knotens v bildet dann einen gerichteten, azyklischen Graphen (dag[5]) mit v als Wurzel. Die Komponenten eines Knotens werden also durch cast hierarchisch strukturiert (siehe Abb.2.4).

Definition 2.3 *Ein* **Pfad** *in eine Rolle r_0 ist eine nicht leere, endliche Sequenz $\pi = (r_0, \ldots, r_n)$ von Rollen*

Mit Pfaden werden Attribut- und Rollenwerte in Knoten bezeichnet: $v(\pi)$ bezeichnet die Komponente v_n, so daß gilt: $v_0 = v(r_0), v_1 = v_0(r_1), \ldots, v_n = v_{n-1}(r_n)$, falls $v_0, v_1, \ldots, v_{n-1}$ definiert sind. Das letzte Element in π kann auch ein Attribut sein, $\pi = (r_0, \ldots, r_{n-1}, a_n)$. In diesem Fall bezeichnet $v(\pi)$ den Attributwert $v_{n-1}(a_n)$. Beispielsweise ist π = (small-face reference_point) ein Pfad in die Rolle small-face und bezeichnet im Knoten $ANGLE_4$ (siehe Abb.2.2) den Attributwert $ANGLE_4(\pi)$ = (0, 0, 2).

Noch eine abkürzende Schreibweise: Sei $V \subseteq \mathcal{V}$ eine Menge von Knoten. Dann bezeichnet V^* alle Knoten und Komponenten von Knoten in V, also $V^* = \bigcup_{v \in V} \mathrm{cast}^*(v)$.

Ein Graph besteht hier aus den bereits eingeführten attributierten Knoten. Die Knoten haben Sorten und sind durch ungerichtete Kanten verbunden. Die Knotensorten werden in ähnlichen Formalismen, z.B. NLCGGs[6] Node Label genannt, daher die Bezeichnung Attributed Node Labeled Graph Grammar.

[4]Durch ein Hasse-Diagramm läßt sich eine Partialordnung über einer nicht zu großen Menge A übersichtlich darstellen. Dabei werden die Elemente von A durch Punkte dargestellt und zwei verschiedene Elemente x,y ∈ A durch eine Linie verbunden, falls x ≤ y und kein z ≠ x,y existiert mit x ≤ z ≤ y. Üblicherweise wird in diesem Fall x unterhalb von y gezeichnet.

[5]directed acyclic graph

[6]Node Label Controlled Graph Grammars, [EnRo 90]

Definition 2.4 *Ein* **Graph** *ist ein 4-Tupel* $g := (V, E, S, \varphi)$, *mit*

$V \subset \mathcal{V}$ *ist eine endliche Menge von Knoten,* $|V|$ *ist die Anzahl der Knoten in* g.

$E \subseteq V^* \times V^*$ *ist eine Menge von ungerichteten Kanten.*

S *ist ein endliches Alphabet von Sorten.*

φ *ist eine Sortenfunktion* $\varphi : V^* \rightarrow S$.

Für einen Knoten v liefert $\varphi(v)$ die **Sorte** des Knotens v. Man beachte, daß E und φ nicht nur für die Knoten aus V, sondern auch für deren Komponenten definiert sind. Die Komponenten eines Knotens spannen eine (ggf. leeren) Graphen auf, Knoten haben also selber Graphstruktur. Beispielsweise läßt sich der Knoten ANGLE_4 in Abb.2.2 auch als Graph mit zwei Knoten und einer Kante beschreiben.

Definition 2.5 *Ein* **terminaler Graph** *ist ein Graph* $g_0 = (V_0, E_0, S, \varphi_0)$ *mit*

$$\forall\ v \in V_0 : \text{cast}(v) = \emptyset \quad \textit{und}\ \text{metasort}(\varphi(v)) = \textit{terminal}$$

Dabei ist metasort eine Funktion, die durch die jeweilige Graph-Grammatik definiert wird und jeder Sorte aus S eine sogenannte Metasorte zuordnet. Wir nehmen hier in unserem Beispiel an:

$$\text{metasort} = (\underset{\textit{terminal}'}{\text{RA}}, \underset{\textit{nonterminal}'}{\text{R-ANGLE}}, \underset{\textit{nonterminal}'}{\text{ANGLE}}, \underset{\textit{goal}}{\text{CORNER}})$$

Damit ist der Graph aus Abb.2.1 ein terminaler Graph. Er stellt sich mit der obigen Definition für Graphen wie folgt dar:

$g_1 = (V_0, E_0, S, \varphi_0)$

$V_0 = \{\text{RA}_1, \text{RA}_2, \text{RA}_3\}$
$E_0 = \{e_1, e_2, e_3\}$
$e1 = (\text{RA}_1, \text{RA}_2)$
$e2 = (\text{RA}_1, \text{RA}_3)$
$e3 = (\text{RA}_2, \text{RA}_3)$
$S = \{\text{RA, R-ANGLE, ANGLE, CORNER}\}$
$\varphi_0 = (\underset{\text{RA}}{\text{RA}_1}, \underset{\text{RA}}{\text{RA}_2}, \underset{\text{RA}}{\text{RA}_3})$

Wir setzen voraus, daß die zu parsenden Eingabegraphen terminale Graphen sind und das Graphen und Graph-Grammatik kompatibel sind, d.h. mit demselben Sortenalphabet S arbeiten.

Zum Vergleich: Der Graph aus Abb.2.2 stellt sich mit der obigen Definition für Graphen so dar:

$$g_1 = (V_1, E_1, S, \varphi_1)$$

$$\begin{array}{ll} V_1 & = \{\mathrm{RA}_3, \mathrm{ANGLE}_4\} \\ E_1 & = \{e_1, e_2, e_3, e_4\} \\ & e1 = (\mathrm{RA}_1, \mathrm{RA}_2) \\ & e2 = (\mathrm{RA}_1, \mathrm{RA}_3) \\ & e3 = (\mathrm{RA}_2, \mathrm{RA}_3) \\ & e4 = (\mathrm{RA}_3, \mathrm{ANGLE}_4) \\ S & = \{\mathrm{RA}, \text{R-ANGLE}, \mathrm{ANGLE}, \mathrm{CORNER}\} \\ \varphi_1 & = \left(\begin{smallmatrix} \mathrm{RA}_1 & \mathrm{RA}_2 & \mathrm{RA}_3 & \mathrm{ANGLE}_4 \\ \mathrm{RA} & \mathrm{RA} & \mathrm{RA} & \mathrm{ANGLE} \end{smallmatrix}\right) \end{array}$$

Dieser Graph ist kein terminaler Graph, da er mit ANGLE_4 einen Knoten enthält, für den cast Rollenwerte liefert, d.h. $\mathrm{cast}(\mathrm{ANGLE}_4) = \{\mathrm{RA}_1, \mathrm{RA}_2\}$. Beachte, daß der Graph nur zwei (nicht vier) Knoten enthält und daß die Kanten des Graphen nicht nur zwischen seinen Knoten, sondern auch zwischen den Komponenten der Knoten definiert sind. Der Graph ist übrigens durch Anwendung der Produktion $p1$ (siehe Abb.2.3) auf die beiden Knoten RA_1 und RA_2 aus dem terminalen Graphen in Abb.2.1 entstanden.

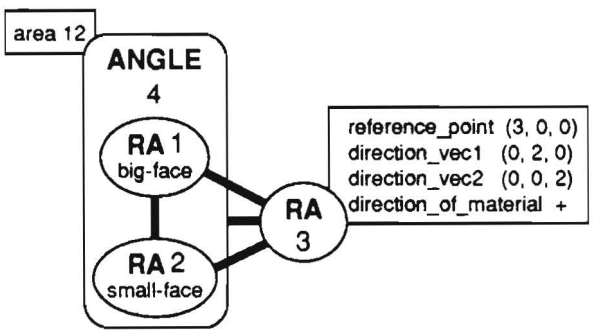


Abbildung 2.2: Ein Graph mit zwei Knoten

Das Sortenalphabet S ist durch eine Partialordnung $<_S$ strukturiert, die Sortenhierarchie, eine Subsumptionshierarchie.

Definition 2.6 *Sei S eine Menge von Sorten und $<_S$ eine Partialordnung von S. Dann ist $<_S$ eine* **Sortenhierarchie** *für S und für zwei Sorten $s_1, s_2 \in S$ ist s_1 eine* **Subsorte** *von s_2 (s_2 subsummiert s_1), falls $s_1 <_S s_2$.*

Der Grammatik-Formalismus behandelt jeden Knoten der Sorte s_1 wie einen Knoten der Sorte s_2. Sortenhierarchien werden hier eingeführt, um die Anzahl der Produktionen zu verringern, die man braucht, um eine gegebene Domäne zu beschreiben. Zum Beispiel ist R-ANGLE (R wie Rectangular) hier eine Subsorte von ANGLE und steht für den rechtwinkligen Spezialfall eines Winkels. Produktionen, wie z.B. p_2 in Abb.2.3, die ANGLE in ihren rechten Seiten verwenden, brauchen dann nur einmal niedergeschrieben zu werden, (und nicht etwa noch in weiteren Spezialversionen für R-ANGLE). Der ANLGG-Formalismus macht keine Aussagen oder Annahmen über die Mengenbeziehungen zwischen den Attributen und Rollen (der Knoten) einer Sorte und (der Knoten) ihrer

Subsorten. Die Entscheidung, ob Sortenhierarchien mit strikter Vererbung[7] der Rollen und/oder Attribute verstanden werden, wird hier bewußt dem Grammatik-Entwickler überlassen. Überhaupt ist ein Leitgedanke bei der Definition der ANLGGs, so wenig wie möglich zu definieren, d.h. nur das festzulegen, was aus Sicht des ANLGG-Parsers unbedingt festgelegt werden muß.

Der ANLGG-Formalismus versteht sich so gesehen nicht als graph-grammatische Theorie, sondern als Rahmenwerk zur Implementierung einer Klasse von Graph-Grammatiken, ähnlich wie der D-PATR-Formalismus [Kar 86] keine linguistische Theorie darstellt, sondern ein Instrument zur Implementierung solcher Theorien (HPSG, GPSG, LFG etc.).

Eine ANLGG[8], im folgenden kurz Graph-Grammatik oder GG, besteht im wesentlichen aus einer Menge von Graph-Grammatik-Produktionen (oder Regeln).

Definition 2.7 *Eine* **Graph-Grammatik**
ist ein 6-Tupel $gg := (S, <_S, P, R, \text{sort}, \text{metasort})$ *mit*

S *ist dasselbe Sortenalphabet wie in Def.2.4*

$<_S$ *ist eine Partialordnung von* S*, die Sortenhierarchie*

P *ist eine endliche Menge von Produktionen*

R *ist die endliche Menge aller Rollenspezifikationen der Produktionen in* P

$\text{sort} : P \cup R \rightarrow S$ *und*

$\text{metasort} : S \rightarrow \{terminal, nonterminal, goal\}$ *sind totale Funktionen.*

Sind $\{v_1, \ldots, v_n\}$ geeignete Komponenten in einem Graphen g, so wird die Produktion p angewendet, indem man die v_i aus g entfernt und durch einen Knoten v, eine sogenannte Instanz von p, ersetzt. Das wird weiter unten noch präzisiert. Offen gelassen ist bis hier z.B. noch die Frage, ob und wie der hinzugefügte Knoten v mit anderen Knoten des Graphen g verbunden wird. Dieses Problem taucht in allen Graph-Grammatik-Formalismen auf; die sogenannte Einbettung von v in g wird durch eine Einbettungsregel kontrolliert[9].

Eine andere Frage ist, auf welche Weise p die Eignung der Knoten $\{v_1, \ldots, v_n\}$ spezifiziert. In Graph-Grammatiken wird das üblicherweise über den Begriff der Graph-Isomorphie[10]

[7]Sortenhierarchien lassen sich zur Repräsentation von taxonomischem Wissen verwenden, siehe z.B. [BS 85]. Eine Idee ist dabei die Verfeinerung von Konzepten zu spezielleren Subkonzepten, wobei die Subkonzepte Informationen ihrer jeweiligen Superkonzepte übernehmen oder „erben". In solchen Sortenhierarchien gibt es meist ein Top-Element, also ein allgemeinstes Konzept, (z.B. THING in KL-ONE). Dem könnte in ANLGGs eine Top-Sorte ANY entsprechen. Auf diese Weise lassen sich dann sehr allgemein fragmentierte Feature definieren, d.h. Feature, die an einer bestimmten Stelle durch einen beliebig füllbaren Zwischenraum (ein Feature der Sorte ANY) „fragmentiert" werden.

[8]Attributed Node Labeled Graph Grammar

[9]Bei klassischen String-Grammatiken ergibt sich eine entsprechendes Einbettungsproblem zunächst nicht. In neueren linguistischen Theorien [Sel 87] wird aber im Sinne einer Modularisierung der Theorie manchmal zwischen der ungeordneten Menge der Konstituenten einer Phrase oder eines Satzes und ihrer linearen Ordnung unterschieden (ID/LP, Immediate Dominace/Linear Precedence in GPSG und HPSG), die Spezifikation der Wortordnung in solchen ID/LP-Grammatiken läßt sich mit der Festlegung einer Einbettungsregel in Graph-Grammatiken vergleichen.

[10]Zwei Graphen g_1 und g_2 sind isomorph, falls es eine Bijektion $V_1 \rightarrow V_2$ zwischen ihren Knoten gibt, die sorten- und nachbarschaftserhaltend ist.

definiert: Die (rechte Seite einer) Regel wird als Graph g_{rhs} aufgefaßt. Wenn ein zu reduzierender Graph einen zu g_{rhs} isomorphen Teilgraphen h enthält, gelten die Knoten von h als durch die Regel spezifiziert und dürfen durch Anwendung der Regel ersetzt werden. Im Fall der ANLGGs ist aber die Definition eines passenden Graph-Isomorphiebegriffs technisch schwierig, erstens wegen der Verwendung von Überlappungen und zweitens wegen der Verwendung von tiefen Nachbarschafts- und Überlappungsbeziehungen, also der Spezifikation von Beziehungen nicht nur zwischen den Knoten eines Graphen, sondern auch zwischen deren Komponenten. Graph-Isomorphie ist also hier kein passender Begriff. Außerdem ist uns hier mit der Einführung eines deklarativen Isomorphiebegriffs wenig gedient. Aus Sicht des ANLGG-Parsers wird eine prozedurale Definition gebraucht (wann matcht eine Produktion einen Graphen, bzw. wie testet man das ?). Anstatt also hier eine Isomorphie zu definieren, die dann anschließend in eine äquivalente prozedurale Definition umgesetzt werden muß, wählen wir hier gleich die benötigte Darstellung in Gestalt sogenannter Rollenspezifikationen und überlassen es dem Graph-Grammatik-Theoretiker, einen entsprechenden Isomorphiebegriff zu formulieren.

Definition 2.8 *Eine* **Produktion** *ist eine nicht leere, endliche Menge* $p = \{r\text{-}spec_1, \ldots, r\text{-}spec_n\}$ *von zusammenhängenden Rollenspezifikationen.*

Jede Rollenspezifikation (unter-)spezifiziert dabei einen Knoten, so daß alle Rollenspezifikationen einer Produktion p zusammen eine Klasse H_p von Graphen spezifizieren, in denen dieselben relationalen Constraints zwischen Knoten und deren Komponenten und dieselben funktionalen Constraints zwischen den Attributwerten der Knoten und ihrer Komponenten erfüllt sind. Eine Produktion p ist dann in einem Graphen g anwendbar, wenn es in g einen Subgraphen $h \in H_p$ gibt. Die Beziehung zwischen p und H_p ist hier, wie weiter oben schon bemerkt, um einiges komplizierter als klassische Graph-Isomorphie, also als die Beziehung eines Graphen g_{rhs} zur Klasse $H_{g_{rhs}}$ der zu ihm isomorphen Graphen.

Das Zusammenhängen der Rollenspezifikationen einer Produktion p garantiert, daß die Graphen in H_p zusammenhängende Graphen sind. Das wird weiter unten noch präzisiert.

Definition 2.9 *Eine* **Rollenspezifikation** *für eine Rolle* r
ist ein 5-Tupel $r\text{-}spec = (r, C^{\approx}, C^{\not\approx}, C^{=}, C^{func})$ *mit*

r *ist die spezifizierte Rolle*

$C^{\approx}, C^{\not\approx}, C^{=}$ *sind endliche Mengen von relationalen Pfadconstraints für* r. *Sie dienen der Spezifikation von Verbindungen* ($\approx$), *verbotenen Verbindungen* ($\not\approx$) *und Überlappungen* ($=$) *zwischen je zwei Komponenten.*

C^{func} *ist eine endliche Menge von funktionalen Constraints, Sortenconstraints und Prädikaten für* r.

Eine Rolle r kann innerhalb einer ANLGG mehrmals spezifiziert werden, innerhalb einer Produktion aber nicht mehr als einmal. Die Pfadconstraints spezifizieren relationale Constraints zwischen Knoten und Komponenten der Graphen in H_p. Sie entsprechen den ungerichteten Kanten in der grafischen Darstellung einer ANLGG-Produktion (siehe Abb.2.3). Schwarze Kanten spezifizieren dabei Verbindungen, durchgestrichene schwarze

Kanten explizit verbotene Verbindungen und graue Kanten Überlappungen[11]. Weil in ANLGGs diese Constraints nicht nur zwischen Knoten, sondern auch zwischen deren Komponenten spezifiziert werden können, werden zu ihrer Formulierung Pfade benutzt:

Definition 2.10 *Die Funktion* **roles**(p) *liefert die Menge* $\{r_1, \ldots, r_n\}$ *der durch* p *spezifizierten Rollen. Ein* **Pfadconstraint** *für* r_i *in* p *ist ein 2-Tupel* (π_i, π_j) *von Pfaden. Dabei ist* π_i *ein Pfad in* r_i *und* π_j *ein Pfad für irgendeine andere Rolle* $r_j \neq r_i$ *aus* roles(p). *Pfadconstraints treten in einer Produktion immer paarweise auf, d.h. wenn* (π_i, π_j) *ein Pfadconstraint für* r_i *in* p *ist, dann ist* (π_j, π_i) *ein Pfadconstraint gleichen Typs für* r_j *in* p.

Zum Beispiel spezifiziert der Pfadconstraint ((a1 big-face) (a2 big-face)) für die Rolle a1 in der Produktion p_2 (graue Kante in Abb.2.3) die Überlappung der entsprechenden Komponenten. Die Rollenspezifikationen einer Produktion p sind zusammenhängend, d.h. jede Rollenspezifikation ist mit mindestens einer anderen Rollenspezifikation aus p verbunden. Dadurch wird garantiert, daß die durch eine Produktion p definierte Graphklasse H_p nur zusammenhängende Graphen enthält.

Definition 2.11 *Zwei Rollenspezifikationen* $r\text{-}spec_i$ *und* $r\text{-}spec_j$ *einer Produktion* p *sind durch den Pfadconstraint* (π_i, π_j) **verbunden**, *falls* $(\pi_i, \pi_j) \in r\text{-}spec_i.C^{\approx} \cup r\text{-}spec_i.C^{=}$ *und* $(\pi_j, \pi_i) \in r\text{-}spec_j.C^{\approx} \cup r\text{-}spec_j.C^{=}$.

Die Spezifikationen $r\text{-}spec_i$ und $r\text{-}spec_j$ sind also miteinander verbunden, falls der Pfadconstraint eine Verbindung ($\approx$) oder Überlappung ($=$) spezifiziert, Spezifikation einer verbotenen Verbindung ($\not\approx$) reicht dazu nicht.

Die funktionalen Constraints in C^{func} dienen der Spezifikation beliebiger funktionaler Constraints zwischen den Rollen- und Attributwerten der Knoten und Komponenten der Graphen in H_p. Dabei wird im Fall der Sortenconstraints die Sorte $\varphi(v)$ ebenfalls als Attributwert aufgefaßt. Damit können z.B. Constraints der Art „Wenn die Attribute der Komponenten eines Knotens die Eigenschaft A haben, dann muß die Sorte des Knotens B sein“ formuliert werden.

Definition 2.12 *Ein* **funktionaler Constraint**
für r *in* p *ist ein Ausdruck der Form* $a = \mathrm{f}(\pi_1, \ldots, \pi_n)$
wobei a *ein Attribut ist,* f *eine beliebige n-stellige Funktion und mindestens einer der Pfade* π_i *ein Pfad in* r *ist.*

Ein **Sortenconstraint** *für* r *in* p *ist ein Ausdruck der Form* $\varphi = \mathrm{f}(\pi_1, \ldots, \pi_n)$
wobei f *eine n-stellige Funktion ist, die eine Sorte* $s \leq_S \mathrm{sort}(p)$ *liefert und mindestens einer der Pfade* π_i *ein Pfad in* r *ist.*

Ein **Prädikat** *für* r *in* p *ist ein Ausdruck der Form* $\mathrm{p}(\pi_1, \ldots, \pi_n)$
wobei p *irgendein n-stelliges Prädikat ist und mindestens einer der Pfade* π_i *ein Pfad in* r *ist.*

[11] Auf $C^{\not\approx}$ Constraints zur Spezifikation verbotener Überlappungen wird hier verzichtet. Alle Überlappung, die nicht explizit spezifiziert werden, gelten als verboten. Eine nicht spezifizierte Nachbarschaft gilt dagegen erst als verboten, wenn sie explizit durch einen $C^{\not\approx}$ Constraint verboten wird. Diese Festlegung ist ziemlich willkürlich, führt aber in der hier anvisierten Anwendung zu kompakten Feature-Definitionen.

Ein n-stelliger Constraint tritt in den Rollenspezifikationen einer Produktion insgesamt n mal auf.

In Abb.2.3 ist z.B. *area := area(small-face) + area(big-face)* ein zweistelliger funktionaler Constraint[12] für die Spezifikation der beiden Rollen small-face und big-face der Produktion p_1. Zusätzlich wäre hier ein Sortenconstraint denkbar, der die Subsorte R-ANGLE liefert, falls small-face und big-face orthogonal aufeinander stehen, und ANGLE sonst.

Wir kommen jetzt zur Definition der Beziehung zwischen einer Produktion p und der Klasse der durch sie spezifizierten Graphen H_p, d.h. zu dem Teil der Graph-Grammatik-Definition, der üblicherweise über Graph-Isomorphie definiert wird:

Definition 2.13 *Sei $g = (V, E, \varphi, S)$ ein Graph. Ein Knoten $v \in V$ ist eine* **Instanz** *einer Produktion $p = \{r\text{-}spec_1, \ldots, r\text{-}spec_n\}$, falls*

1. $\varphi(v) \leq_S \text{sort}(p)$ *und*
2. $cast(v) = \{v_1, \ldots, v_n\} \in V^*$, *so daß jeder Rollenwert $v_i = v(r_i)$ in $cast(v)$ genau eine Rollenspezifikation $r\text{-}spec_i$ für r_i erfüllt.*

Die v_i spannen einen zusammenhängenden Graphen $h \in H_p$ auf, h ist zusammenhängend, weil die $r\text{-}spec_i$ zusammenhängen. Weil die v_i genau die Rollenwerte der Instanz v sind und die Kanten zwischen den v_i auch nach Anwendung der Produktion noch definiert sind, hat v also selbst Graphstruktur. Man kann die Anwendung einer Produktion p hier auch als Zuweisung $v := h \in H_p$ auffassen. Beachte, daß die Sorte der Instanz v eine Subsorte der Sorte der Produktion p sein darf, andernfalls hätte die Verwendung der Sortenconstraints keinen Sinn. Durch die Sortenconstraints wird ein funktionaler Zusammenhang zwischen der Sorte der Instanz v und ihren Komponenten hergestellt.

Jetzt muß noch geklärt werden, wie die Constraints der Rollenspezifikation im einzelnen zu deuten sind, d.h. was sie spezifizieren.

Definition 2.14 *Ein Knoten v_i* **erfüllt** *im Knoten v eine Rollenspezifikation $r\text{-}spec_i$ für eine Rolle r_i einer Produktion p, gdw. die folgenden Bedingungen erfüllt sind:*

1. $v_i = v(r_i)$
2. $\varphi(v_i) \leq_S \text{sort}(r\text{-}spec_i)$
3. *Jeder Pfadconstraint (π_i, π_j) in $r\text{-}spec_i$ referenziert zwei Knoten $v_i^* = v(\pi_i)$ und $v_j^* = v(\pi_j)$ (Dabei ist nach Definition der Pfadconstraints v_i^* zugleich Komponente von v und von v_i.) Die Knoten v_i^* und v_j^* sind, falls $(\pi_i, \pi_j) \in \ldots$*

   $\ldots C^{\approx}$: *miteinander verbunden, d.h. $(v_i^*, v_j^*) \in E$*

   $\ldots C^{\not\approx}$: *nicht miteinander verbunden, d.h. $(v_i^*, v_j^*) \notin E$*

   $\ldots C^{=}$: *identisch, d.h. $v_i^* = v_j^*$*

[12] In Abb.2.3 ist eine leichter lesbare Schreibweise gewählt, der Constraint läßt sich aber auch in der Form *area* = f(π_1, π_2) darstellen.

4. *Für jeden funktionalen Constraint*
$$a = \mathrm{f}(\pi_1, \ldots, \pi_n) \text{ in } C^{func} \text{ gilt:} \quad v(a) = \mathrm{f}(v(\pi_1), \ldots, v(\pi_n))$$
für jeden Sortenconstraint
$$\varphi = \mathrm{f}(\pi_1, \ldots, \pi_n) \text{ in } C^{func} \text{ gilt:} \quad \varphi(v) = \mathrm{f}(v(\pi_1), \ldots, v(\pi_n)) \leq_S \mathrm{sort}(p)$$
für jedes Prädikat
$$\mathrm{p}(\pi_1, \ldots, \pi_n) \text{ in } C^{func} \text{ gilt:} \quad \mathrm{p}(v(\pi_1), \ldots, v(\pi_n)) = TRUE$$

Die zuletzt eingeführten Begriffe sollen am Beispiel der Produktion p_2 (siehe Abb.2.3) verdeutlicht werden: Die Produktion hat zwei Rollen a1 und a2. Die Rolle a1 ist dabei innerhalb der Produktion p_2 durch die folgende Rollenspezifikation definiert:

$r\text{-}spec = \{r, C^{\approx}, C^{\not\approx}, C^{=}, C^{func}\}$

r = a1
$C^{\approx}$ = {((a1 small-face), (a2 small-face))}
$C^{\not\approx}$ = $\emptyset$
$C^{=}$ = {((a1 big-face), (a2 big-face))}
C^{func} = {convex ((a1 small-face), (a2 small-face))}

Durch die Produktion p_2 wird eine Klasse H_{p_2} von Graphen definiert, deren Repräsentanten alle die folgenden Eigenschaften gemeinsam haben: Sie bestehen jeweils aus zwei Knoten der Sorte ANGLE oder einer der Subsorten von ANGLE, z.B. R-ANGLE. In beiden Knoten sind Rollenwerte für die Rollen big-face und small-face definiert, im Falle von big-face sind diese Rollenwerte identisch (Überlappung). Die beiden Rollenwerte für small-face sind Knoten, die durch eine Kante verbunden sind und das Prädikat convex ist für diese beiden Knoten erfüllt, d.h. hier: Sie repräsentieren Flächen, die in einem konvexen Winkel Material einschließen.

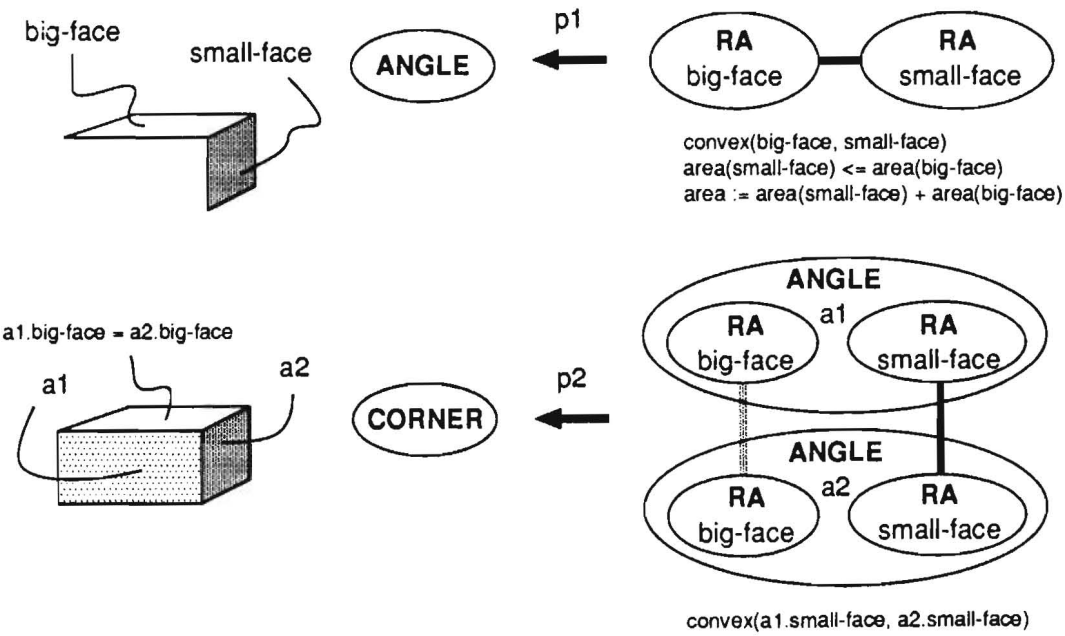


Abbildung 2.3: Eine ANLGG mit zwei Produktionen

Wir haben jetzt alle Definitionen zusammen, um die Begriffe *Anwendung einer Produktion*, *Ableitung* und *Graphsprache* definieren zu können.

Wie weiter oben schon erwähnt wurde, muß die Definition der Anwendung einer Produktion u.a. erklären, wie ein neu hinzugefügter Knoten v in den resultierenden Graphen eingebettet wird. Diese Einbettungsregel ist hier durch die intendierte Verwendung des Formalismus (Graphen als geometrische Objekte) vorgegeben: Zwei Objekte berühren einander, wenn sich mindestens zwei ihrer Komponenten berühren.

Auf Graphen übertragen heißt das: Der hinzugefügte Knoten v ist mit einem Knoten v_j^* verbunden, wenn v_j^* keine Komponente von v ist und mit mindestens einer Komponente v^* von v verbunden ist. Diese Einbettungsregel führt zu folgender Definition der Anwendung einer Produktion auf einen gegebenen Graphen:

Definition 2.15 *Eine* **Anwendung** α *einer Produktion* p *auf einen Graph* g_n *ist ein 3-Tupel* $\alpha = (g_n, p, g_{n+1})$. *Dabei ist* $g_{n+1} = (V_{n+1}, E_{n+1}, S, \varphi_{n+1})$ *der resultierende Graph mit:*

$V_{n+1} = \{v\} \cup V_n \setminus \mathrm{cast}(v)$, *wobei* v *eine Instanz von* p *ist.*

$E_{n+1} = E_n \cup \{(v, v_j^*) \mid \exists (v^*, v_j^*) \in E_n : \quad v^* \in \mathrm{cast}^*(v)$ *und* $v_j^* \in V_n^* \setminus \mathrm{cast}^*(v)\}$

$\varphi_{n+1} = \varphi_n$ *erweitert um* v *so, daß* $\varphi_{n+1}(v) \leq_S \mathrm{sort}(p)$

Beispielsweise ist das 3-Tupel (g_0, p_1, g_1) eine Anwendung der Produktion p_1, wobei g_0 der Graph aus Abb.2.1 und g_1 der Graph aus Abb.2.2 ist.

Die Anwendung einer Produktion ist hier anders als in den meisten Graph-Grammatik-Formalismen „von rechts nach links" definiert, d.h. gibt an, wie ein Teilgraph durch einen einzelnen Knoten ersetzt werden kann. Der ziemlich unspezifischen Einbettungsregel wegen ist die Anwendung einer ANLGG-Produktion nur in dieser einen Richtung eindeutig definiert, d.h. eine Anwendung ist hier nicht eindeutig umkehrbar.

Auf einen gegebenen Graphen g_n können i. allg. mehrere Produktionen anwendbar sein und eine einzige Produktion kann an mehreren Stellen des Graphen anwendbar sein. Die Anwendung einer Produktion kann dann die Anwendung derselben oder anderer Produktionen unmöglich machen, d.h. ANLGGs sind nicht notwendigerweise konfluent[13].

Definition 2.16 *Eine* **Ableitung** δ *eines terminalen Graphen* g_0 *in einer ANLGG* $gg = (S, <_S, P, R, \mathrm{sort}, \mathrm{metasort})$ *ist eine endliche Sequenz* $\delta = (\alpha_0, \ldots, \alpha_n), \quad n \geq 1$ *von Anwendungen mit:*

$\alpha_i = (g_i, p_i \in P, g_{i+1}), \quad 0 \leq i < n$

$g_n = (V_n, E_n, S, \varphi_n)$ *mit* $V_n = \{v_g\}$ *und* $metasort(\varphi(v_g)) = goal$.

Der Knoten v_g *heißt* **Parse** *oder* **Feature-Baum** *von* g_0.

[13]Ein Termersetzungssystem heißt ***konfluent***, wenn für alle Terme s, t_1, t_2 mit $s \xrightarrow{*} t_1$ und $s \xrightarrow{*} t_2$ ein Term t existiert, so daß $t_1 \xrightarrow{*} t$ und $t_2 \xrightarrow{*} t$ (zitiert aus [BlBü 87], Seite 119). Dabei steht $\xrightarrow{*}$ für null oder mehr Termersetzungsschritte.

Wie eingangs schon erwähnt, kann man die Funktion cast auch als partielle Ordnung der Komponenten eines Knotens auffassen und cast* als deren transitiven Abschluß. Das Hasse-Diagramm der Komponenten des Knotens v_g bildet dann einen gerichteten, azyklischen Graphen (dag) mit v als Wurzel, den sogenannten Feature-Baum. Abb.2.5 zeigt ein Beispiel für eine mögliche Ableitung des terminalen Graphen aus Abb.2.1. Attribute sind teilweise weggelassen. Der Knoten CORNER_6 ist ein Parse. Bei der zweiten Anwendung der Produktion p_1 werden RA_1 und RA_3 zu ANGLE_5 reduziert. Beachte, daß $\text{RA}_1 \notin V_1$ sondern $\in V_1^*$. RA_1 ist also kein Knoten, sondern Komponente eines Knotens.

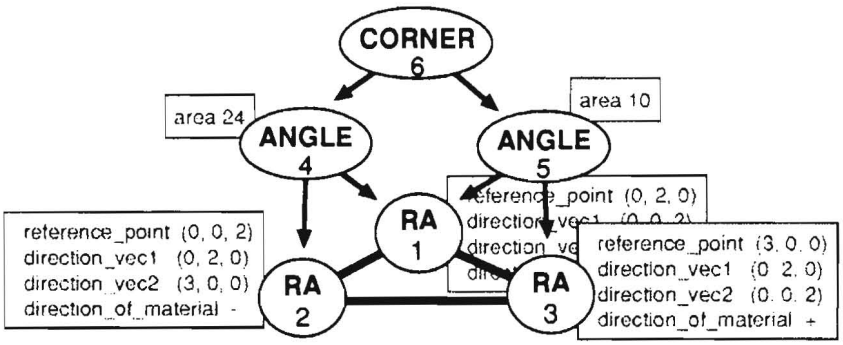
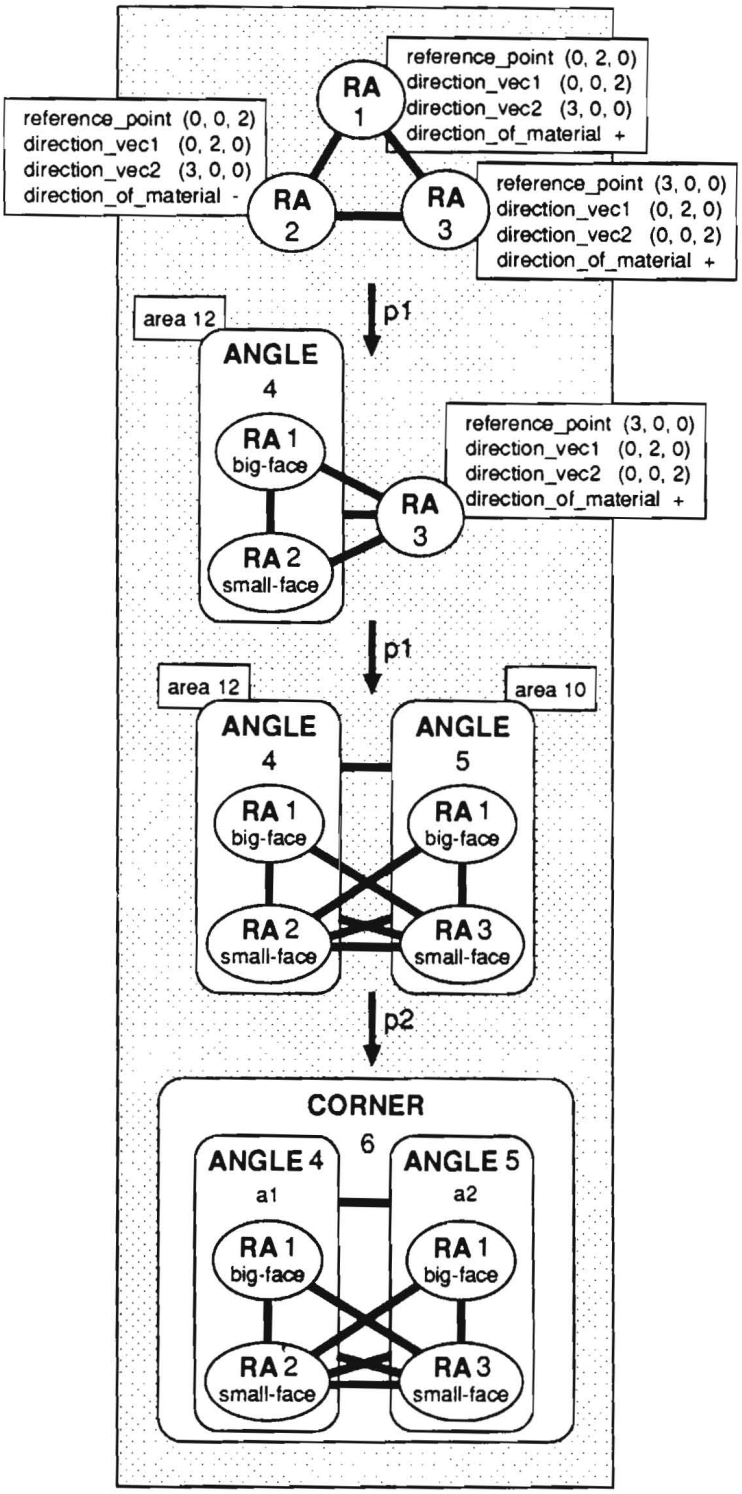


Abbildung 2.4: Ein Parse oder Feature-Baum

Abb.2.4 zeigt das Hasse-Diagramm seiner Komponenten. Die gerichteten Kanten des Diagramms muß man sich als mit den entsprechenden Rollen (a1, a2, small-face, usw.) gekennzeichnet vorstellen. Diese Kantenlabel sind in Abb.2.4 der Übersichtlichkeit halber weggelassen.

Definition 2.17 *Die von einer ANLGG gg erzeugte* **Sprache** $L(gg)$ *ist die Menge aller terminalen Graphen, für die es in gg mindestens eine Ableitung gibt.*



$V_0 = \{ 1, 2, 3\}$
$E_0 = \{ (1, 2)\ (1, 3)\ (2, 3) \}$

$V_1 = \{ 3, 4 \}$
$E_1 = E_0 \cup \{ (3, 4) \}$
$\varphi_1(4) = \text{ANGLE}$

$V_2 = \{ 4, 5 \}$
$E_2 = E_1 \cup \{ (2, 5)\ (4, 5) \}$
$\varphi_2(5) = \text{ANGLE}$

$V_3 = \{ 6 \}$
$E_3 = E_2$
$\varphi_3(6) = \text{CORNER}$

Abbildung 2.5: Eine Ableitung im ANLGG-Formalismus

2.2 ANLGGs und andere Grammatik-Formalismen

In diesem Abschnitt werden einige Beziehungen der ANLGGs zu anderen Formalismen und Begriffen hergestellt, angesprochen werden unifikationsbasierte Formalismen, Hyperedge-Replacement, Plex-, NLC- und Präzedenz-GGs und Klauselgraph Resolution.

2.2.1 Charakterisierung der ANLGGs

Graph-grammatisch lassen sich die im vorigen Abschnitt definierten ANLGGs folgendermaßen charakterisieren:

- *Kontextfreie Regeln:* Die linke Seite spezifiziert einen einzelnen Knoten, die rechte einen zusammenhängenden Graph. Statt des Begriffs der Graph-Isomorphie wird im ANLGG-Formalismus der Begriff der Spezifikation verwendet, um zu definieren, wann eine rechte Regelseite einen Graphen matcht: ähnlich, wie ein Graph g_{rhs} die Klasse H_{rhs} der zu ihm isomorphen Graphen definiert, so definiert eine ANLGG-Produktion p die Klasse H_p der durch sie spezifizierten Graphen. Graph-Isomorphie ist hier wegen der Verwendung von drei verschiedenen Relationen in rechten Regelseiten (Kante, Nonkante, Überlappung), die überdies auch noch „tief", d.h. zwischen den Komponenten von Knoten vorkommen können, kein passender Begriff. Außerdem sind Spezifikationen ihres prozeduralen Charakters wegen zumindest für die Entwicklung und Beschreibung des Parsers handlicher, weil direkt umsetzbar.

- *Einbettung:* Kanten werden nie entfernt oder gesplittet. Die Kanten zwischen ersetzten Knoten bleiben erhalten, weil ersetzte Knoten als Komponenten des ersetzenden Knotens „weiterleben". Einbettung des ersetzenden Knotens: er ist zu *allen* Knoten der 1-Nachbarschaft des ersetzten Graphen benachbart, dabei entstehen keine Mehrfachkanten. Diese geometrisch motivierte Einbettungsregel ist nur bei Analyse (Regelanwendung von rechts nach links) eindeutig definiert.

- *Knotenlabel:* Die Knoten tragen Label, die hier Sorten genannt werden. Die Sorten sind Elemente einer Sortenhierarchie.

- *Knotenattribute:* Zwischen den Knotenattributen der Knoten einer rechten Regelseite und den Attributen des Knotens der linken Regelseite können beliebige funktionale Beziehungen formuliert werden. Die Sorte des Knotens der linken Regelseite wird dabei ebenfalls als Knotenattribut betrachtet, d.h. die Sorte der Instanz einer linken Seite ist nicht a priori vorhersagbar, sondern steht in funktionalem Zusammenhang mit der jeweiligen Instanz der rechten Regelseite.

- *Knotenkomponenten:* Ein Knoten enthält ggf. andere, durch ihn ersetzte Knoten (Komponenten des Knotens), hat also selbst Graphstruktur. Die Kanten eines Graphen sind nicht nur zwischen seinen Knoten, sondern auch zwischen deren Komponenten definiert.

- *Drei Arten von Kanten:* In der Spezifikation einer rechten Regelseite werden drei Arten von Relationen verwendet, um Kanten, verbotene Kanten und Überlappungen zwischen Knoten oder deren Komponenten zu spezifizieren.

2.2.2 Vergleich mit anderen Formalismen

Der ANLGG-Formalismus ist u.a. aus den Erfahrungen mit dem unifikationsbasierten FEAT-PATR [BKS 91] hervorgegangen. Der Beziehung einer ANLGG-Produktion p zur Klasse H_p der durch sie spezifizierten Graphen entspricht in unifikationsbasierten Grammatik-Formalismen die Beziehung eines Terms t zur Klasse H_t der mit ihm unifizierbaren Terme. Unifikation mußte aber hier als constraintlösende Basisoperation fallengelassen werden, weil die Beschränkung auf reine Gleichheitsconstraints in der anvisierten technischen Domäne zu restriktiv ist.

Die ANLGGs haben eine gewisse Ähnlichkeit mit den sogenannten Hyperedge-Replacement Systemen [DrKr 90].

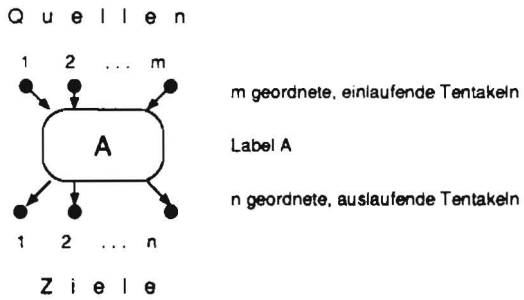


Abbildung 2.6: Hyperkante

Eine gerichtete Kante verbindet normalerweise zwei Knoten miteinander, d.h. einen Quell- und einen Zielknoten. Die Erweiterung auf m Quell- und n Zielknoten führt zum Begriff der Hyperkante (siehe Abb.2.6). Kontextfreie Hyperkanten-Regeln geben an, wie ein aus solchen Hyperkanten bestehender Graph durch eine einzige Hyperkante ersetzt werden kann. Der Graph muß mit seinen externen „Tentakeln" dieselben Quellen und Ziele berühren, wie die ersetzende Hyperkante. Die Einbettung ist also ähnlich einfach, wie bei ANLGGs, mit dem Unterschied, daß bei ANLGGs die Zahlen m und n nicht durch die Regeln, sondern durch den Kantengrad im jeweiligen Graphen festgelegt sind. Im Gegensatz zu Hyperedge-Replacement ist daher bei ANLGGs die Regelanwendung nur von rechts nach links eindeutig definiert.

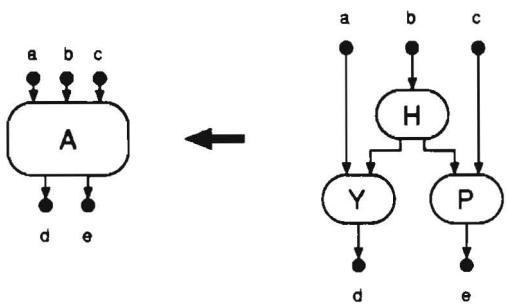


Abbildung 2.7: Beispiel einer Hyperkanten Regel

Die Hyperedge Regel aus Abb.2.7 darf z.B. von rechts nach links auf einen Graphen angewendet werden, der einen zur rechten Seite isomorphen Teilgraphen enthält, der *genau* drei einlaufende und zwei auslaufende Tentakeln hat. In diesem Fall darf dieser Teilgraph

durch eine Hyperkante mit Label A ersetzt werden. Diese Operation ist eindeutig umkehrbar. Mit Hyperedge-Grammatiken lassen sich z.B. Flußdiagramme sowohl analysieren als auch generieren. Hyperedge-Replacement ist in den 80er Jahren intensiv theoretisch untersucht worden, Literaturhinweise dazu finden sich in [DrKr 90].

Hyperedge-Grammatiken sind auch unter dem Synonym Plex-Grammatiken bekannt, nur daß die Hyperedge-Kanten in der Plex-Terminologie Knoten heißen und umgekehrt. In [BuHa 89] wird ein Parser für Plex-Grammatiken angegeben. Der Graph-Parser von Lutz [Lut 89] läßt sich ebenfalls als Plex- bzw. Hyperedge-Parser auffassen. Auf ihm basiert der im nächsten Kapitel beschriebene ANLGG-Parser.

NLC-Grammatiken [EnRo 90] sind nur sehr oberflächlich verwandt mit ANLGGs, d.h. die Gemeinsamkeiten erschöpfen sich bereits darin, daß beide Formalismen mit kontextfreien Graph-Grammatik-Regeln und belabelten Knoten arbeiten. NLC steht für Node Label Controlled und deutet an, daß die Einbettung des ersetzenden Graphen g von den Labeln seiner Knoten und den Labeln der 1-Nachbarschaft des ersetzten Knotens v abhängt. Dabei können in Abhängigkeit von den entsprechenden Label-Paaren Kanten entfernt, gesplittet oder hinzugefügt werden. In ANLGGs wird die Einbettung gerade nicht von den Knotenlabeln kontrolliert. Sie wird in diesem Sinne überhaupt nicht kontrolliert: Der Knoten v wird einfach mit *allen* Knoten der 1-Nachbarschaft von g verbunden.

Graph Parser für allgemeine kontextfreie Graph-Grammatiken können wegen der NP-Vollständigkeit des Graph-Isomorphie-Tests keine bessere Komplexität haben. Man ist daher an geeigneten Einschränkungen interessiert, die Parsebarkeit mit linearem oder polynomialen Aufwand garantieren, ähnlich, wie für kontextfreie String-Grammatiken durch LL-, LR- und Präzedenz-Grammatiken lineare Parsebarkeit erreicht wird (siehe [AU 72]). In [Kau 86] wird argumentiert, daß sich von den drei genannten Techniken nur der Präzendenz-Ansatz auf Graph-Grammatiken überträgen läßt, weil der LL- und der LR-Ansatz wesentlichen Gebrauch von der linearen Ordnung der Eingabestrings macht.

In [Kau 86] wird eine Klasse von NLC-ähnlichen sogenannten Präzedenz-Graph-Grammatiken (PGGen) vorgestellt, für die der ebenfalls angegebene Parser eine Zeitkomplexität $O(n^2)$ in der Zahl n der Knoten des Eingabegraphen hat. Die grundlegende Idee ist dabei die folgende: Für eine gegebene Graph-Grammatik mit Knotenlabel Alphabet S wird eine sogenannte Präzedenzrelation $<\subset S \times S$ berechnet. Der Präzedenz-Graph-Parser ordnet dann aufgrund dieser Präzedenzrelation jedem Graph der Graphsprache eine eindeutig bestimmte, hierarchische Struktur zu.

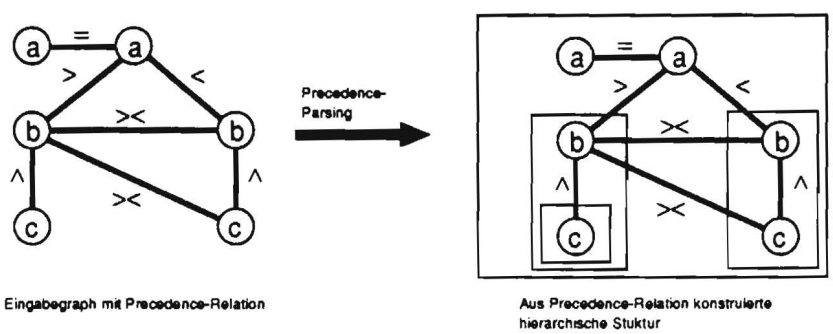


Abbildung 2.8: Präzedenz-Graph und seine Ableitung mit einer PGG

Die Suche wird dabei an einem beliebigen Knoten begonnen und entlang gleicher oder

aufsteigender Präzedenz so lange wie möglich fortgesetzt. Ist ein Teilgraph gefunden, dessen Knoten alle gleiche Präzedenz haben, so wird der Teilgraph durch Anwendung einer eindeutig bestimmten Produktion durch einen einzigen Knoten ersetzt und ähnlich wie bei NLC-Grammatiken eingebettet. Die Suche wird rekursiv bis zur Erzeugung eines Zielknotens fortgesetzt. Das Beispiel aus Abb.2.8 stammt aus [Kau 86].

Dieser Ansatz läßt sich so leider nicht auf die sehr allgemeine Klasse der hier verwendeten Werkstückgraphen (TEC-REP, [BKL 91a]) übertragen und wird hier auch nicht weiter verfolgt. Die Idee, die Komplexität des Graph-Parsens zu reduzieren, indem man die Reihenfolge der möglichen Reduktionsschritte durch eine Ordnung des Sortenalphabets einschränkt, läßt sich aber vielleicht zur Konstruktion der heuristischen Steuerung des ANLGG-Parsers (speziell der $<_A$-Heuristik, siehe 3.2.2) verwenden.

Die Repräsentationssprache TEC-REP [BKL 91a] verwendet uneingeschränkte, zusammenhängende Graphen zur Repräsentation von Werkstücken. Die Komplexität des Feature-Erkennungs (Graph-Parsing) Problems ließe sich reduzieren, wenn es gelänge, eine Einschränkung der benötigten Graphen, z.B. auf planare Graphen oder auf Graphen mit beschränktem Kantengrad zu finden.

In Abschnitt 3.4 wird noch auf eine überraschende Beziehung zwischen Ableitungen in ANLGGs und Klauselgraph Resolution hingewiesen. Dabei entspricht der Umstand, daß durch die ANLGG-Einbettung Kanten zu *jedem* Knoten der 1-Nachbarschaft des ersetzten Graphen entstehen, der Tatsache, daß eine Klausel mit *jeder* anderen Klausel resolvieren kann, die ein komplementäres Literal enthält. In Klauselgraphen sind das genau die Klauseln der 1-Nachbarschaft einer Klausel.

Kapitel 3

Ein Parser für ANLGGs

In diesem Kapitel wird der Parser für ANLGGs vorgestellt. Der verwendete Grammatik-Formalismus ANLGG erlaubt die Analyse von Graphen mit attributierten Knoten. Dabei können zwischen den Attributen der Knoten beliebige funktionale Beziehungen auftreten, nicht nur Gleichheit, wie in den meisten unifikationsbasierten Formalismen (z.B. D-PATR [Kar 86]). Außerdem können Überlappungen von Teilgraphen spezifiziert werden, d.h. bestimmte Teile des Eingabegraphen können mehrfach geparst werden.

Der Parser arbeitet chartbasiert und bottom-up. Die Suche wird durch Heuristiken geleitet. Der Parser zählt alle Parse auf, die Heuristiken sorgen dafür, daß „gute" Parse zuerst gefunden werden. Das Verfahren ist vollständig, d.h. schlechte Heuristiken verlängern zwar die Suche, jeder Parse wird aber schließlich gefunden, und terminiert, selbst wenn kein Parse existiert, falls es für die ANLGG eine Reduktionsordnung gibt.

3.1 Datenstrukturen: Bindepunkt, Patch, Agenda und Chart

In diesem Abschnitt werden die wichtigsten vom Parser benutzten Datenstrukturen beschrieben. Wir nehmen an, daß $g_0 = (V_0, E_0, S, \varphi_0)$ ein zu parsender terminaler Graph ist, und $gg = (S, \leq_S, P, R, \text{sort}, \text{metasort})$ die ANLGG, mit der geparst wird. Die Komponenten von zusammengesetzten Objekten werden hier durch Punkte referenziert, also $g_0.V_0$, $gg.sort$, $r\text{-}spec.r$, $r\text{-}spec.C^{\approx}$ usw. Wo Verwechslungen ausgeschlossen sind, ist die Objektbezeichnung weggelassen, z.B. beziehen sich V_0 und E_0 immer auf den terminalen Eingabegraphen g_0.

3.1.1 Bindepunkte

Um Kanten einfacher handhaben zu können, wählen wir für sie eine andere, praktischere Repräsentation. Bei jedem Ableitungsschritt werden der alten Kantenmenge i. allg. neue Kanten hinzugefügt: $E_{n+1} := E_n \cup d_n$. Die einfache Einbettungsregel der ANLGGs macht es möglich, die Rekursion aufzulösen und für alle aus g_0 abgeleiteten Graphen g_{n+1} die zugehörige Kantenmenge E_{n+1} direkt in Abhängigkeit von der initialen Kantenmenge E_0 darzustellen.

Definition 3.1 *Sei* $g_n = (V_n, E_n, S, \varphi_n)$ *ein Graph, der durch* n *Anwendungen von Produktionen auf den terminalen Graph* $g_0 = (V_0, E_0, S, \varphi_0)$ *entstanden ist. Dann ist* $\tau : E_0 \times V_0 \rightarrow T$ *eine Funktion, die jeder Kante* $e = (v_1, v_2) \in E_0$ *zwei unterscheidbare Kantenenden, die sogenannten* **Bindepunkte** *der Kante* e *zuordnet und* T *die Menge aller Bindepunkte in* E_0*. Die Funktion* **complement** *ordnet jedem Bindepunkt* tp *den jeweils gegenüberliegenden Bindepunkt zu, d.h. für alle* $e = (v_1, v_2) \in E_0 : \tau(e, v_1) =$ complement$(\tau(e, v_2))$ *. Für beliebige Knoten* $v \in V_n^*$ *ist:*

$$\mathbf{tps}(v) := \{tp \in T \mid \exists e = (v^*, v_j^*) \in E_0 : tp = \tau(e, v^*), v^* \in cast^*(v), v_j^* \notin cast^*(v)\}$$

Die Menge tps(v)[1] ist die Menge der Bindepunkte der Kanten, die aus v „herausragen", also der Kanten, über die v mit anderen Knoten verbunden sein kann. Diese externen Kanten entsprechen den Tentakeln der Hyperkanten, siehe Abschnitt 2.2.2. Der Parser erzeugt die Kantenmengen $E_0, E_1, E_2, \ldots$ nicht explizit, sondern berechnet zu jedem neu erzeugten Knoten v die Menge tps(v). Die Frage, ob zwei Knoten v_1 und v_2 miteinander verbunden sind, d.h. ob $(v_1, v_2) \in E_n$, läßt sich dann beantworten, indem man testet, ob tps(v_1) und tps(v_2) ein Paar von komplementären Kantenenden enthalten, denn es gilt die folgende Behauptung:

$$(v_1, v_2) \in E_n \quad gdw. \quad \exists tp_1 \in \text{tps}(v_1), tp_2 \in \text{tps}(v_2) : tp_1 = \text{complement}(tp_2)$$

Auf einen Beweis wird hier verzichtet[2].

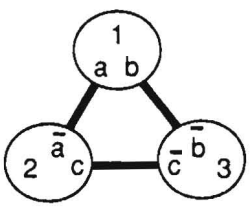


Abbildung 3.1: Graph mit sechs Bindepunkten

Abb.3.1 zeigt noch einmal den Graphen von Seite 7. Den drei Kanten sind sechs Bindepunkte $\{a, \bar{a}, b, \bar{b}, c, \bar{c}\}$ zugeordnet. Damit ist z.B. tps(1) $= \{a, \bar{b}\}$ und tps(2) $= \{\bar{a}, \bar{c}\}$. Die Knoten 1 und 2 sind miteinander verbunden, denn tps(1) und tps(2) enthalten mit a und $\bar{a}$ ein Paar komplementärer Bindepunkte.

3.1.2 Patches

Eine der Heuristiken des Parsers besteht in einer Ordnung der Rollenspezifikationen der Produktionen. Der Parser sucht nach Rollenwerten für die Rollen einer Produktion in der

[1] tiepoint set

[2] Beweisskizze: Für die Anwendung einer Produktion war die neue Kantenmenge E_{n+1} rekursiv definiert durch:

$$E_{n+1} = E_n \cup \{(v, v_j^*) \mid \exists (v^*, v_j^*) \in E_n : \quad v^* \in \text{cast}^*(v) \text{ und } v_j^* \in V_n^* \setminus \text{cast}^*(v)\}.$$

Dabei ist v der durch Anwendung der Produktion hinzugekommene Knoten. Durch Induktion über n zeigt man, daß sich E_{n+1} auch direkt angeben läßt, d.h. man zeigt Gleichheit zur Kantenmenge:

$$E_{n+1}^{\circ} = \{(v_i, v_j) \mid \exists (v_i^*, v_j^*) \in E_0 : \quad v_i^* \in \text{cast}^*(v_i), v_j^* \in \text{cast}^*(v_j) \text{ und } v_i^* \notin \text{cast}^*(v_j), v_j^* \notin \text{cast}^*(v_i)\}$$

E_0 ist dabei die Kantenmenge des terminalen Eingabegraphen.

durch diese Ordnung gegebenen Reihenfolge. Das wird weiter unten noch präzisiert. Der Suchvorgang führt über teilweise geglückte Instantierungen (Hypothesen) zu vollständigen Instanzen (Fakten) einer Produktion.

Ein **Patch** ist eine partielle (pp, Hypothese) oder vollständige (cp, Fakt) Instanz einer geordneten Produktion und besteht aus vier Komponenten:

prod Die Produktion, deren Instanz das Patch repräsentiert.

sort Die Sorte des Patches

tps Die Menge der Bindepunkte des Patches

r-spec Die Rollenspezifikation der ersten unbesetzten Rolle, falls das Patch ein pp ist, empty, falls es ein cp ist.

Die vier Komponenten werden wieder durch Punkte referenziert, also z.B. cp.tps, cp.sort usw. Ein Patch kann außerdem Attribute und Rollen haben, genau wie ein attributierter Knoten. Rollenwerte sind dabei vollständige Patches, z.B. ist $cp^* = pp(r)$ der Rollenwert der Rolle r im partiellen Patch pp. Die Funktionen cast und cast* sind entsprechend für Patches definiert, d.h. cast(p) liefert alle Rollenwerte, und cast*(p) alle Komponenten des Patches p (vgl. Def. 2.1 auf Seite 7).

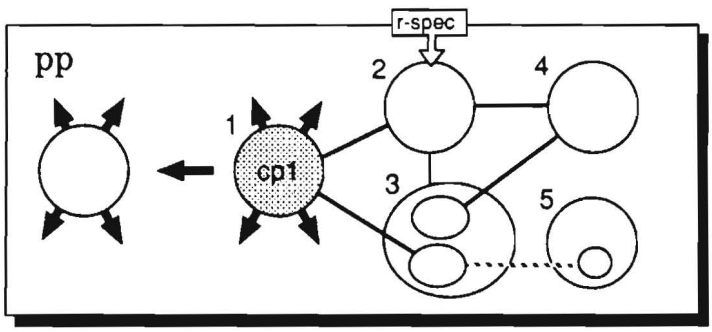


Abbildung 3.2: Ein partielles Patch mit einer besetzten Rolle

Abb.3.2 zeigt ein partielles Patch, das Instanz einer Produktion mit fünf Rollen ist. Mit cp1 ist ein Rollenwert für die erste Rolle bereits gefunden. Das pp sucht also einen Rollenwert für die zweite, durch r-spec spezifizierte Rolle. Die Pfadconstraints der Rollenspezifikationen der Produktion sind im Bild durch dünne schwarze (für Nachbarschaft) und schraffierte (für Überlappung) Kanten gegeben. Die dicken Kanten stehen für Verbindungen zwischen den Rollenwerten (Paare komplementärer Bindepunkte, siehe z.B. die Verbindung zwischen cp1 und cp2 in Abb.3.3), also für Kanten im zu parsenden Graphen. Die kurzen Pfeile und Kanten des Knotens links vom Regelpfeil stehen für die Bindepunkte im tps des Patches. Das tps des Patches aus Abb.3.2 enthält also z.B. vier Bindepunkte. Die Pfeilspitzen bedeuten dabei, daß der jeweilige Bindepunkt aktiv ist, d.h. entlang dieses Bindepunktes sucht das Patch nach Rollenwerten für seine nächste Rolle. Das wird weiter unten noch genauer erklärt.

Abb.3.3 zeigt ein Patch, in dem bereits drei Rollen besetzt sind, es wird also nach Rollenwerten für die vierte Rolle gesucht, und zwar entlang der aktiven Bindepunkte a oder b und c.

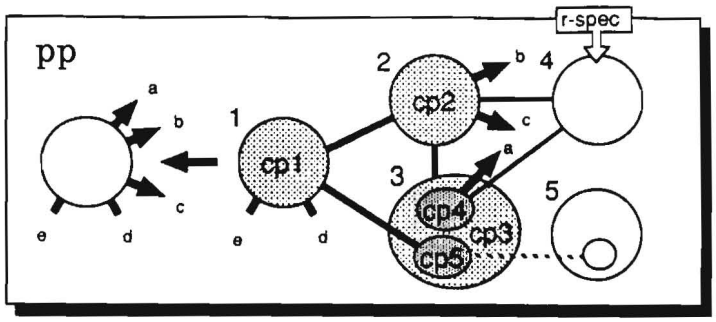


Abbildung 3.3: Ein partielles Patch mit drei besetzten Rollen

Definition 3.2 *Zwei Patches p1 und p2 sind miteinander* **verbunden**, *wenn sie mindestens ein komplementäres Paar von Bindepunkten haben, also wenn es ein tp* $\in$ *p1.tps gibt, so daß complement(tp)* $\in$ *p2.tps.*

Im zweiten Kapitel wurde definiert, unter welchen Bedingungen ein Knoten v_i eine Rollenspezifikation in einem Knoten v erfüllt. Im Kontext des Parsers entspricht v_i einem vollständigen Patch cp und v einem Patch p, daß sich durch Kombination von cp mit einem geeigneten pp ergibt. Das neu erzeugte Patch np enthält einen Rollenwert mehr, als pp, nämlich cp, d.h. cp = np(pp.r-spec.r).

Definition 3.3 *Ein cp ist mit einem pp* **kombinierbar**, *gdw. die folgenden Bedingungen erfüllt sind:*

1. *cp.sort* $\leq_S$ *sort(pp.r-spec)*

2. *Jeder Pfadconstraint* $(\pi_i = (r_0, r_1, \ldots, r_n), \pi_j)$ *in pp.r-spec referenziert zwei vollständige Patches* cp_i^* *und* cp_j^* *wie folgt:* $cp_i^* = cp$, *falls* $n = 0$ *und* $cp((r_1, \ldots, r_n))$ *sonst,* $cp_j^* = pp(\pi_j)$.

   Die Pfadconstraints müssen von cp erfüllt werden. Ein Pfadconstraint (π_i, π_j) *ist von cp erfüllt, wenn die Patches* cp_i^* *und* cp_j^*, *falls* (π_i, π_j) *aus ...*

   ...pp.r-spec.$C^{\approx}$*: miteinander verbunden sind*

   ...pp.r-spec.$C^{\not\approx}$*: nicht miteinander verbunden sind*

   ...pp.r-spec.$C^{=}$*: identisch sind, d.h.* $cp_i^* = cp_j^*$

3. *Alle Prädikate aus pp.r-spec.*C^{func} *müssen erfüllt sein. Die funktionalen Constraints und Sortenconstraints sind immer erfüllbar, indem man sie wahr macht, d.h. als Zuweisung an entsprechende Attribute des neuen Patches np bzw. an die Patchsorte np.sort auffaßt.*

Ein cp ist also mit einem pp kombinierbar, wenn es die erste freie Rolle in pp besetzen kann.

3.1.3 Agenda und Chart

Der Parser erzeugt während eines Parserlaufs Patches. Die erzeugten Patches befinden sich jeweils in der Agenda oder im Chart. Die **Agenda** ist eine Menge von partiellen oder vollständigen Patches. Der **Chart** enthält alle Patches, die nicht in der Agenda sind. Er ist so organisiert, daß man für jedes Patch in der Agenda sehr schnell alle Patches im Chart bestimmen kann, die mit diesem Patch kombinierbar sind. Dabei ist die Aufgabe des Charts nicht, genau diese Menge zu liefern, sondern eine möglichst kleine, aber vollständige Übermenge davon. Die zuviel gelieferten, d.h. nicht kombinierbaren Patches werden dann durch Evaluieren der verschiedenen Constraints bestimmt und ausgesiebt.

Den m Kanten $E = \{e_1, e_2, \ldots, e_m\}$ des Eingabegraphen sind, wie oben beschrieben, Bindepunkte $T = \{tp_1, \overline{tp}_1, tp_2, \overline{tp}_2, \ldots, tp_m, \overline{tp}_m\}$ zugeordnet. Diese Bindepunkte dienen im Chart als Bezugspunkte, als Indizes, über die gezielt auf Patches zugegriffen werden kann. Das funktioniert deswegen so gut, weil sich alle Kanten, auch die in abgeleiteten Graphen, über die konstante Menge T der Bindepunkte des Eingabegraphen definieren lassen.

Die Patches im Chart können mit folgenden Zugriffsfunktionen gelesen werden:

CP(tp,sort) = {cp ∈ Chart | cp.sort = sort und tp ∈ cp.tps}
liefert die Menge aller vollständigen Patches cp der Sorte sort im Chart, die den Bindepunkt tp in ihrem tps enthalten.

PP(tp,sort) = {pp ∈ Chart | sort(pp.r-spec) = sort und ...}
liefert eine Menge von pp (nicht unbedingt alle pp) im Chart, die ein cp der Sorte sort als nächsten Rollenwert suchen, so daß es in pp.r-spec einen Verbindungsconstraint gibt, der von einem cp erfüllt wird, falls tp ∈ cp.tps. Anders gesagt: Für jeden tp ∈ cp.tps liefert PP(tp, cp.sort) partielle Patches, für deren nächste Rolle cp mindestens einen Verbindungsconstraint erfüllt. Es werden nicht alle pp des Charts geliefert, für die das zutrifft. Es ist aber garantiert, daß jedes cp, das alle Verbindungsconstraints eines gegebenen pp's erfüllt *mindestens einen* Bindepunkt tp in seinem tps hat, so daß für diesen Bindepunkt pp ∈ PP(tp, pp.sort). Wenn die Funktionen als Wertetabellen realisiert werden, verhindert diese Einschränkung überflüssige Mehrfacheintragungen von partiellen Patches.

Die folgenden beiden Funktionen dienen dazu, Patches zu finden, die Überlappungsconstraints erfüllen:

CP_{ov}(cp*) = {cp ∈ Chart | cp* ∈ cast(cp)}
liefert alle vollständigen Patches des Charts, die cp* als direkten Rollenwert benutzen.

PP_{ov}(cp*) = {(pp, π) | cp* = pp(π_j) und (π, π_j) ∈ pp.r-spec.$C^{=}$}
liefert alle 2-Tupel (pp,π) von pp des Charts, wobei pp ein cp sucht, daß sich an der Stelle cp* = cp(π) mit pp überlappt.

Außerdem enthält der Chart noch zwei Mengen $\mathbf{CP_0}$ und $\mathbf{PP_0}$. Sie enthalten alle vollständigen bzw. partiellen Patches des Charts, deren tps leer ist. CP_0 enthält alle gefundenen Parse.

Der Chart wird vorzugsweise als Array mit fester Stelligkeit, d.h. mit einem Eintrag für jede der m Kanten des Eingabegraphen, implementiert. Jeder Eintrag enthält vier Hashtabellen, die zusammen die Funktionen CP(tp,sort) und PP(tp,sort) realisieren, sort dient dabei als Hash-Schlüssel. Die Funktionen $CP_{ov}(cp^*)$ und $PP_{ov}(cp^*)$ lassen sich durch jeweils eine Hashtabelle realisieren, wobei jeweils cp^* als Schlüssel dient. CP_0 kann als einfache Liste realisiert werden, PP_0 wird vom Parser nicht benutzt, kann aber interessant sein, wenn man an teilweisen Parsen interessiert ist.

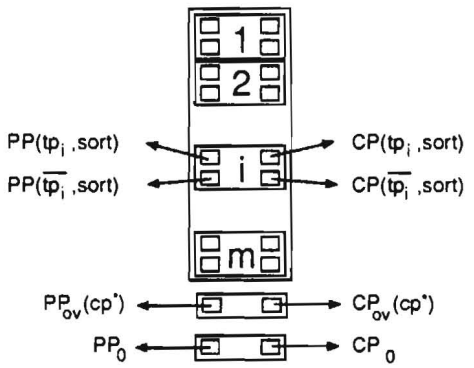


Abbildung 3.4: Organisation des Charts

Das folgende Beispiel soll die Benutzung des Charts durch den Parser illustrieren: Abb.3.3 zeigt ein partielles Patch, einen „Enkel" des Patches aus Abb. 3.2. Die dünnen Kanten stehen für Pfadconstraints, die dicken Kanten für Kanten des Graphen, die Bindepunkte des tps sind mit a,b,c,d,e markiert. Das pp sucht nach einem Rollenwert für die vierte, durch pp.r-spec spezifizierte Rolle entlang der Kantenenden a, b und c. Die Menge CP(a,sort) $\cap$ (CP(b,sort) $\cup$ CP(c,sort)) enthält alle cp der Sorte sort des Charts, die die geforderten Verbindungsconstraints erfüllen. Ähnlich lassen sich auch für alle anderen Suchprobleme des Parsers Ausdrücke angeben, die die gewünschten Kandidatenmengen liefern. Ein weiteres Beispiel: Patches, die durch Kombination des pp mit einem der gerade für die Besetzung der vierten Rolle gefundenen cp entstanden sind, stehen vor dem Problem, Kandidaten für die fünfte Rolle zu finden. Für diese ist nur ein Constraint spezifiziert, nämlich eine Überlappung (dünne schraffierte Kante) in cp5. CP_{ov}(cp5) liefert alle cp, die cp5 als Rollenwert enthalten, also Kandidaten für die fünfte Rolle sind.

3.2 Heuristiken

Der ANLGG-Formalismus und der hier vorgestellte Parser für ANLGGs sind mächtig genug, um die Isomorphie zweier Graphen testen zu können. Zwei Graphen g_1 und g_2 sind isomorph, falls es eine Bijektion $V_1 \rightarrow V_2$ zwischen ihren Knoten gibt, die sorten- und nachbarschaftserhaltend ist. Im ANLGG-Formalismus ist das ein Spezialfall der Frage, ob eine Produktion p in einem Graphen g anwendbar ist. Graph-Isomorphie ist im allgemeinen ein NP-vollständiges Problem. Damit kann auch ein allgemeiner ANLGG-Parser keine bessere Komplexität haben. Der damit verbundenen kombinatorischen Explosion, d.h. der rasant wachsenden Anzahl der beim Parsen erzeugten Hypothesen wird hier mit Heuristiken begegnet. Es wird also versucht, durch den Einsatz von Heuristiken trotz theoretisch schlechter Problemklasse (NP) praktikable Laufzeiten zu erreichen.

3.2.1 Erste Heuristik: Seltene Rollen zuerst

Die erste Heuristik des Parsers besteht in einer Ordnung der Rollenspezifikationen. Jeder Produktion $p = \{r\text{-}spec_1, \ldots, r\text{-}spec_n\}$ wird eine totale Ordnung $<_p$ der Rollenspezifikationen in p zugewiesen. Die Ordnung kodiert die Strategie, mit der der Parser eine Instanz der Produktion zu erzeugen versucht, d.h. er sucht nach Rollenwerten für die Rollen von p in der durch $<_p$ angegebenen Reihenfolge. Der Effekt ist erstens, daß Instanzen einer Produktion nur auf genau einem Weg erzeugt und das heißt für den Parser: nicht mehrmals erzeugt weden können. Die Heuristik sollte die jeweils schwerer erfüllbaren Rollenspezifikationen vorziehen. Der Effekt ist dann zweitens, daß leichter zu findende Rollenwerte erst gesucht werden, wenn die seltener anzutreffenden bereits gefunden sind, d.h. mit einiger Wahrscheinlichkeit angenommem werden kann, daß die vollständige Instantierung der Produktion gelingt. Dabei kann „schwer findbar" z.B. über die Sorten (seltene Sorten) und/oder über die Constraints (Anzahl der Constraints, schwer erfüllbare Constraints) definiert werden.

Wir schreiben Produktionen ab hier als Sequenz $p = (r\text{-}spec_1, \ldots, r\text{-}spec_n)$ und nehmen an, daß $r\text{-}spec_i <_p r\text{-}spec_j$ für $1 \leq i < j \leq n$. Nach Rollenwerten für $r\text{-}spec_1$ wird jeweils zuerst gesucht. Die Produktionen können nach dem Ordnen ihrer Rollenspezifikationen auch als Prozeduren angesehen werden, die vom Parser parallel interpretiert werden.

Die Pfadconstraints (siehe Def.2.10) treten innerhalb einer Produktion immer paarweise auf, und n-stellige Prädikate (siehe Def.2.12) entsprechend n mal. Beim Ordnen der Rollenspezifikationen einer Produktion werden alle mehrfach vorkommenden Constraints in $<_p$-kleineren Rollenspezifikationen entfernt: Dadurch wird erreicht, daß Pfadconstraints und Prädikate nur einmal und jeweils *so früh wie möglich* evaluiert werden (early pruning) d.h. sobald alle für ihre Auswertung benötigten Argumente gefunden sind.

Die funktionalen Constraints und Sortenconstraints sind immer erfüllbar und sollten auch nur einmal, aber zur Vermeidung überflüssiger Auswertungen *so spät wie möglich* evaluiert werden.

Für eine gegebene Ordnung $<_p$ der Rollenspezifikationen einer Produktion p läßt sich eine günstige Zuordnung der Constraints zu den einzelnen Rollenspezifikationen automatisch berechnen, das wird hier aber nicht weiter vertieft.

Die Ordnung $<_p$ ist durch folgende Bedingung eingeschränkt: Produktionen waren so definiert, daß die n Rollenspezifikationen zusammenhängen, $<_p$ muß diese Rollenspezifikationen so ordnen, daß auch alle Teilsequenzen $(r\text{-}spec_1, \ldots, r\text{-}spec_i)$ für alle $1 < i \leq n$ zusammenhängen. Das bedeutet nicht, daß jede Rollenspezifikation mit ihrem unmittelbaren Vorgänger verbunden ist, es reicht die Verbindung mit irgendeinem der Vorgänger.

Zum Beispiel sind die beiden pp aus Abb.3.2 und Abb.3.3 Instanzen einer Produktion, deren Rollenspezifikationen (1, 2, 3, 4, 5) geordnet sind. Das ist erlaubt: 5 ist zwar nicht mit 4 verbunden, dafür aber mit 3. Nicht erlaubt sind hier z.B. alle Ordnungen, die mit (1, 2, 5, ...) anfangen, weil 5 weder mit 1, noch mit 2 verbunden ist.

Ordnungen mit der genannten Einschränkung lassen sich immer finden, weil jeder zusammenhängende Graph einen aufspannenden Baum hat [Har 72], sie sind aber nicht immer mit der heuristisch motivierten Ordnung nach Seltenheit verträglich.

3.2.2 Zweite Heuristik: Gute Patches zuerst

Die zweite für das Laufzeitverhalten des Parsers wichtigste Heuristik besteht in einer totalen Ordnung $<_A$ der Patches; $<_A$ ordnet die Patches der Agenda nach heuristischer Priorität. Das $<_A$-größte Patch wird jeweils als nächstes bearbeitet. Der Parser arbeitet bottom-up, d.h. datengetrieben. Die Ordnung $<_A$ soll die zunächst blinde Suche auf das Ziel hin ausrichten. Patches, die mit einiger Wahrscheinlichkeit zum Auffinden eines Parses beitragen, sollen eine hohe Priorität bekommen. Jede Ordnung der Patches ist erlaubt, d.h. eine schlechte Heuristik verlängert zwar die Suche, jeder Parse wird aber schließlich gefunden, falls die ANLGG eine Reduktionsordnung hat. Die Ordnung $<_A$ kann z.B. über die Attribute der Patches, über eine unterliegende Ordnung der Produktionen oder der Sorten oder eine gewichtete Kombination von allem erfolgen.

Die global wirksame Heuristik $<_A$ ist wichtiger als die lokale Ordung $<_p$ der Rollen, in dem Sinne, daß sich mit $<_A$ die größeren Gewinne erzielen lassen: Eine optimale Heuristik $<_A$ erzeugt die möglichen Parse auf dem kürzesten überhaupt denkbaren Weg (und zwar unabhängig von $<_p$), d.h. erzeugt nur genau die Komponenten, die Komponenten eines Parses sind. Eine optimale Heuristik $<_p$ unterstützt zwar lokal die Erzeugung einzelner Komponenten mit minimalem Aufwand, kann aber nicht verhindern, daß eine suboptimale oder kontraproduktive Heuristik $<_A$ immer gerade die „falschen" Patches zur Bearbeitung vorschlägt.

3.3 Der Parsing Algorithmus

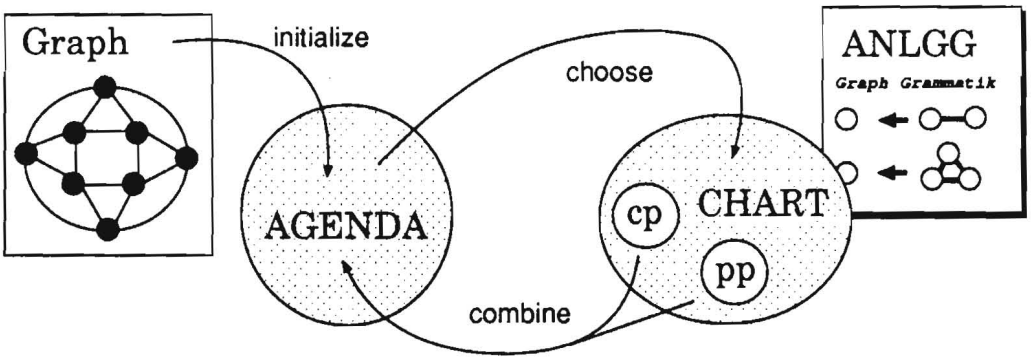


Abbildung 3.5: Aufbau des Graph Parsers

Der Parser besteht im wesentlichen aus drei Regeln **initialize**, **choose** und **combine**, die auf den beiden Patchmengen Agenda und Chart arbeiten. Die Agenda wird von initialize mit einem cp je Knoten des Eingabegraphen initialisiert, der Chart ist anfangs leer. Danach werden choose und combine solange abwechselnd angewendet, bis die Agenda leer ist. Der Chart enthält dann alle möglichen Parse.

3.3.1 Der Basis Algorithmus

In diesem Abschnitt wird der Kernstück des Parsers beschrieben, im nächsten Abschnitt folgen dann die Details der schnellen Suche im Chart. In Abschnitt 3.3.3 wird beschrieben,

wie zwei miteinander kombinierbare Patches zu einem neuen Patch kombiniert werden.

```
parse graph:
     initialize chart and agenda
     until the agenda is empty do
          choose best patch from agenda
          combine patch
          add patch to chart
     enduntil

initialize:
     agenda = empty
     for each node in graph.V do
          add a cp to agenda
     endfor
     chart = empty
```

`choose` liefert das vielversprechendste, d.h. $<_A$-größte Patch der Agenda, $<_A$ verkörpert dabei die für das Laufzeitverhalten des Parsers entscheidende Heuristik des Parsers, siehe dazu Abschnitt 3.2.2.

`combine patch` kombiniert das Patch `patch` mit allen Patches des Charts, für die das möglich ist (siehe Definition 3.3 von „kombinierbar"). Dabei kombiniert immer jeweils ein pp mit einem cp, indem das cp die erste freie Rolle in pp besetzt. Weder das pp noch das cp werden dabei verändert. Die neu erzeugten Patches werden in die Agenda aufgenommen.

```
combine patch:
     if patch is complete
          then propose patch
               continue-cp patch
          else continue-pp patch
     endif
```

Produktionen können als partielle Patches aufgefaßt werden, in denen keine einzige Rolle besetzt ist, `propose cp` kombiniert ein vollständiges Patch mit allen Produktionen, deren erste Rolle es besetzen kann. Dabei liefert `predict(sort)` alle Produktionen der Grammatik, deren erste Rolle von einem cp der Sorte `sort` besetzt werden kann, also:

$$\text{predict}(s) = \{p \in gg.P \mid s \leq_S gg.\text{sort}(\text{first}(p))\}$$

Die entsprechenden Produktionenmengen können für jede ANLGG unabhängig vom Eingabegraphen in einer Compilierungsphase vorausberechnet werden.

```
propose patch
     for each production p in predict(cp.sort)
          try to add a new patch np to agenda such that
          np.tps = cp.tps
     endfor
```

`continue-cp cp` und `continue-pp pp` kombinieren ein gegebenes cp oder pp mit allen pp bzw. cp des Charts, für die das möglich ist, erzeugen ein neues Patch für jede solche Kombination und fügen die erzeugten Patches in die Agenda ein. Für die Vollständigkeit des Parsers ist entscheidend, daß dabei wirklich *alle* möglichen Kombinationen entdeckt und erzeugt werden. Der Chart enthält i.allg. sehr viele Patches. In Testläufen für Grammatiken mit 20 bis 40 Regeln und Graphen mit bis zu 50 Knoten befanden sich beim Auffinden des ersten Parses z.B. einige Tausend Patches im Chart. Dabei wurden allerdings keine, oder nur sehr einfache Heuristiken verwendet.

Da die Auswertung der Constraints zeitaufwendig ist, ist eine Indizierung der Patches im Chart in jedem Fall wünschenswert. Damit sollen möglichst schnell alle Patches des Charts identifiziert werden können, die mit einiger Wahrscheinlichkeit mit einem gegebenen Patch kombinierbar sind. Bei dieser schnellen Vorauswahl mit Hilfe passend gewählter Indizes ist wieder entscheidend, daß mindestens alle kombinierbaren Patches erreicht werden. Die per Vorauswahl aus dem Chart geholte Kandidatenmenge soll also einerseits möglichst klein sein, um unnötige Constraintauswertung zu vermeiden, muß andrerseits aber vollständig sein, d.h. mindestens alle kombinierbaren Patches enthalten. Als Indizes werden hier Bindepunkte, Sorten und überlappende Patches verwendet. Die Kandidaten werden mittels der oben definierten Zugriffsfunktionen aus dem Chart geholt, die Details sind ziemlich technisch und werden im nächsten Abschnitt erklärt.

3.3.2 Schnelles Durchsuchen des Charts durch Indizierung

In diesem Abschnitt wird beschrieben, wie mit Hilfe der Chart Zugriffsfunktionen eine kleine, aber vollständige Menge von Kandidaten für die Kombination mit einem gegebenen Patch bestimmt wird.

`continue-cp cp` kombiniert ein gegebenes cp mit allen pp des Charts, für die das möglich ist (d.h. wieder, wenn cp alle Constraints für pp's nächste Rolle erfüllt, also mit pp kombinierbar ist) indem es für jede Kombination (mit **merge**) je ein neues Patch erzeugt und in die Agenda einfügt. Um eine Instanz einer Produktion mit n Rollen zu erzeugen, sind n solche Kombinationen notwendig. Die n-1 Vorstufen einer solchen Instanz sind partielle Patches. Sie werden i. allg. mehrfach benutzt, um alternative Instanzen zu bilden.

```
continue-cp cp
     pp-candidates = union(tp-seekers(cp), ov-seekers(cp))
     for each pp in pp-candidates
          if no-missing-tp   (cp, pp) and
             no-missing-ov   (cp, pp) and
             no-forbidden-tp (cp, pp) and
             no-forbidden-ov (cp, pp)
          then merge(cp, pp)
     endfor
```

`tp-seekers(cp)` liefert die Menge aller pp im Chart, für deren nächste Rolle cp mindestens einen Verbindungsconstraint erfüllt und für deren nächste Rolle cp eine passende Sorte hat, d.h. cp.sort $\leq_S$ *gg*.sort(pp.r-spec). Diese Kandidatenmenge kann mit Hilfe der

Chart-Zugriffsfunktion PP(tp,sort) als Vereinigung der für die Sorte von cp und für deren Übersorten gelieferten Patchmengen leicht bestimmt werden:

$$\texttt{tp-seekers(cp)} = \cup_{cp.sort \leq_S s \in S} \cup_{tp \in cp.tps} \mathrm{PP}(tp, s)$$

`ov-seekers(cp)` liefert die Menge aller pp im Chart, für deren nächste Rolle cp mindestens einen Überlappungsconstraint erfüllt und für deren nächste Rolle cp eine passende Sorte hat, d.h. wieder cp.sort $\leq_S$ *gg*.sort(pp.r-spec). Diese Kandidatenmenge ist mit Hilfe der Chart-Zugriffsfunktion PP_{ov} bestimmbar. Für jede Komponente $cp^* \in cast^*(cp)$ liefert $\mathrm{PP}_{ov}(cp^*)$ eine Menge von (pp,π) Paaren. Diese Paare waren so definiert (siehe 3.1.3), daß jedes cp mit $cp^* = cp(\pi)$ mindestens einen Überlappungsconstraint für pp's nächste Rolle erfüllt:

$$\begin{aligned}\texttt{ov-seekers(cp)} = \{\mathrm{pp} \mid\ & (\mathrm{pp}, \pi) \in \mathrm{PP}_{ov}(cp^*), \\ & cp(\pi) = cp^* \in cast^*(cp), \\ & cp.sort \leq_S sort(pp.r\text{-}spec)\}\end{aligned}$$

`continue-pp pp` kombiniert ein gegebenes pp mit allen cp des Charts, für die das möglich ist, indem es für jede Kombination (mit `merge`) je ein neues Patch erzeugt und in die Agenda einfügt.

```
continue-pp pp
     cp-candidates = intersection(tp-fillers(pp), ov-fillers(pp))
     for each cp in cp-candidates
          if no-forbidden-tp (cp, pp) and
             no-forbidden-ov (cp, pp)
          then merge(cp, pp)
     endfor
```

`tp-fillers(pp)` liefert die Menge aller cp im Chart, die alle Verbindungsconstraints für pp's nächste Rolle erfüllen und eine passende Sorte $s \leq_S$ sort(pp.r-spec) haben. Falls keine Verbindungsconstraints spezifiziert sind, d.h. $C.^{\approx} = \emptyset$, werden *alle* cp des Charts mit einer passender Sorte geliefert. Andernfalls ergibt sich die gelieferte Menge als Schnittmenge der von CP(tp,sort) für die „aktiven" Bindepunkte von pp gelieferten Patchmengen:

$$\texttt{tp-fillers(pp)} = \cup_{s \leq_S \mathrm{sort(pp.r\text{-}spec))}} \cup_{pp_i \in active(pp)} \cap_{tp \in active\text{-}tp(pp_i.tps)} \mathrm{CP}(tp, s)$$

Dabei wird diesmal über alle Subsorten der Sorte von pp's nächster Rolle vereinigt, also in der Sortenhierarchie herabgestiegen. Bei `tp-seekers` mußten dagegen alle Übersorten der Sorte eines gegebenen cp's betrachtet werden.

Die aktiven Nachbarn *active*(pp) eines pp's sind dabei die bereits gefundenen Rollenwerte des pp oder Komponenten davon, die mit pp's nächster Rolle durch einen Verbindungsconstraint verbunden sind. Die Menge der aktiven Bindepunkte *active-tp*(pp_i) eines solchen Nachbarn pp_i ergibt sich als Schnittmenge seines tps mit dem des pp. Beispielsweise

hat die nächste Rolle des pp's in Abb.3.2 nur einen aktiven Nachbarn, cp1, der vier aktive Bindepunkte hat. Das pp in Abb.3.3 hat dagegen zwei aktive Nachbarn, cp2 mit aktiven Bindepunkten b und c und cp4 mit aktivem Bindepunkt a. Die aktiven Bindepunkte eines pp sind also die Bindepunkte, entlang denen nach Rollenwerten für die nächste Rolle gesucht wird.

`ov-fillers(pp)` liefert die Menge aller cp im Chart, die alle Überlappungsconstraints für pp's nächste Rolle erfüllen und eine passende Sorte $s \leq_S$ sort(pp.r-spec) haben. Falls keine Überlappungsconstraints spezifiziert sind, d.h. $C.^{=} = \emptyset$, werden *alle* cp des Charts mit einer passenden Sorte geliefert. Andernfalls ergibt sich die gelieferte Menge als Vereinigung von Schnittmengen , ähnlich wie oben. Für jeden Überlappungsconstraint (π, π_j) für pp's nächste Rolle sind die Überlappungen $cp^* = pp(\pi)$ bekannt, denn sie sind Komponenten von pp. $CP_{ov}(cp^*)$ liefert alle vollständigen Patches des Charts, die eine solche Überlappung als Rollenwert benutzen. Rekursiv wiederholte Anwendung von CP_{ov} liefern alle cp, die cp^* in beliebiger Tiefe als Komponente enthalten. Falls für ein solches cp gilt $cp^* = cp(\pi_j)$, dann ist cp ein vollständiges Patch, das den Überlappungsconstraint erfüllt.

`no-forbidden-tp (cp, pp)` prüft, ob keine verbotenen Verbindungen bestehen, d.h. ob die durch die Pfadconstraints $pp.C^{\not=}$ spezifizierten Komponenten unverbunden sind. Dieser Test kann entfallen, falls $pp.C^{\not=} = \emptyset$.

`no-forbidden-ov (cp, pp)` prüft entsprechend, ob cp und pp keine unerlaubten Überlappungen enthalten, d.h. ob die Schnittmenge $cast^*(cp) \cap cast^*(pp)$ genau die Menge der Komponenten enthält, die durch die Überlappungsconstraints in $pp.C^{=}$ spezifiziert sind[3]. Dieser Test muß in jedem Fall durchgeführt werden, d.h. selbst, wenn $pp.C^{=} = \emptyset$, denn alle Überlappungen, die nicht explizit erlaubt werden, gelten als verboten.

3.3.3 Kombination zweier Patches

Die im vorigen Abschnitt beschriebenen Funktionen liefern für ein gegebenes Patch in einer schnellen Vorauswahl alle Patches des Charts, mit denen es möglicherweise kombinierbar ist. Für jeden dieser Kandidaten muß dann geprüft werden, ob er tatsächlich alle gegebenen Constraints erfüllt. In diesem Fall wird dann aus dem Patch und dem Kandidaten ein neues Patch erzeugt. Diese Verschmelzungsoperation entspricht in unifikationsbasierten Formalismen einem Unifikationsschritt.

`merge(cp, pp)` testet, ob cp mit pp kombinierbar ist, d.h. ob cp alle Prädikate aus $pp.r\text{-}spec.C^{func}$ erfüllt. In diesem Fall wird ein neues Patch np erzeugt, das mit cp einen Rollenwert mehr enthält als pp. Die Sorte des neuen Patches ist zunächst die von pp und wird gegebenenfalls durch einen Sortenconstraint spezialisiert. Außerdem enthält np gegebenenfalls Attribute, so daß alle funktionalen Constraints aus $pp.r\text{-}spec.C^{func}$ erfüllt sind, d.h. diese Constraints werden wie Zuweisungen an die dadurch in np definierten Attribute behandelt.

[3]Der Test ist etwas komplizierter, als hier ausgeführt. Das Problem ist, daß die Schnittmenge nicht nur die expliziet spezifizierten Überlappungen enthält, sondern auch deren Komponenten. Für jede solche Komponente muß geprüft werden, ob sie Komponente einer spezifizierten Überlappung ist oder (vielleicht zusätzlich) zu einer unerlaubten Überlappung gehört.

```
merge(cp, pp)
     if combinable(cp,pp) then
        make np as copy of pp with np(pp.r-spec.r) = cp
        satisfy functional constraints by making them true
        np.tps = merge-tps(cp,pp)
        add-to-agenda np
```

`merge-tps(cp,pp)` liefert die Menge der Bindepunkte eines als Kombination von cp mit pp neu erzeugten Patches. Da die Bindpunkte hier die Kanten repräsentieren, entspricht dies der Einbettung, das heißt, für eine Produktion mit n Rollenspezifikationen sollten die n zugehörigen merge-tps Berechnungen dem Übergang von E_n nach E_{n+1} entsprechen. Um Korrektheit zu zeigen, müßte die entsprechende Äquivalenz nachgewiesen werden, darauf wird hier verzichtet. Das neue tps ergibt sich als Vereinigung von cp.tps mit pp.tps, abzüglich komplementärer Bindepunkte, d.h. ohne Kanten, die cp mit pp verbinden. Zusätzlich werden alle Bindepunkte entfernt, die im tps einer Überlappung zwischen cp und pp, aber nicht gleichzeitig in cp.tps und in pp.tps vorkommen.

Diese „Vererbungsregel" für Bindepunkte steht in einem überraschenden Bezug zum Resolutions-Kalkül: Bindepunkte entsprechen dabei Literalen. Die Korrektheit der Regel wird hier, wie gesagt, nicht gezeigt, die dahinterstehende Intention aber in Abschnitt 3.4 an einem Beispiel verdeutlicht.

`add-to-chart patch` fügt das vollständige oder partielle Patch `patch` in den Chart ein, so daß es von den vier in 3.1.3 definierten Chart-Zugriffsfunktionen korrekt erreicht wird. Faßt man die Zugriffsfunktionen als Wertetabellen auf, so wird jedes Patch i. allg. mehrmals eingetragen, ein cp zum Beispiel für jeden seiner Bindepunkte und ein pp für jeden der aktiven Kanten *eines* der aktiven Nachbarn seiner nächsten Rolle. CP_0 enthält alle gefundenen und, wenn die Agenda leer wird, auch alle findbaren Parse.

3.4 Ein Beispiellauf des Parsers

In diesem Abschnitt wird das Verhalten des ANLGG-Parsers anhand eines Parserlaufs für den Eingabegraphen aus Abb.2.1 (Seite 7) und der ANLGG aus Abb.2.3 (Seite 15) illustriert. Die Parser-Heuristik $<_p$ ordnet die Rollenspezifikationen in p_1 und p_2 so, daß nach big-face, bzw. nach a1 zuerst gesucht wird. Die Heuristik $<_A$ ist so gewählt, daß sich erschöpfende Breitensuche ergibt, d.h. **choose** wählt immer das am längsten wartende Patch aus der Agenda.

```
TO_AGENDA   1
TO_AGENDA   2
TO_AGENDA   3

CHOOSE      1
PROPOSE     1
TO_AGENDA   4
CONT_CP     1
TO_CHART    1

CHOOSE      2
PROPOSE     2
TO_AGENDA   5
CONT_CP     2
TO_CHART    2

CHOOSE      3
PROPOSE     3
TO_AGENDA   6
CONT_CP     3
TO_CHART    3

CHOOSE      4
CONT_PP     4
TO_AGENDA   7
TO_AGENDA   8
TO_CHART    4

CHOOSE      5
CONT_PP     5
TO_AGENDA   9
TO_CHART    5

CHOOSE      6
CONT_PP     6
TO_CHART    6

CHOOSE      7
PROPOSE     7
TO_AGENDA  10
CONT_CP     7
TO_CHART    7

CHOOSE      8
PROPOSE     8
TO_AGENDA  11
CONT_CP     8
TO_CHART    8

CHOOSE      9
PROPOSE     9
TO_AGENDA  12
CONT_CP     9
TO_CHART    9

CHOOSE     10
CONT_PP    10
TO_PARSES  13
TO_CHART   10
```

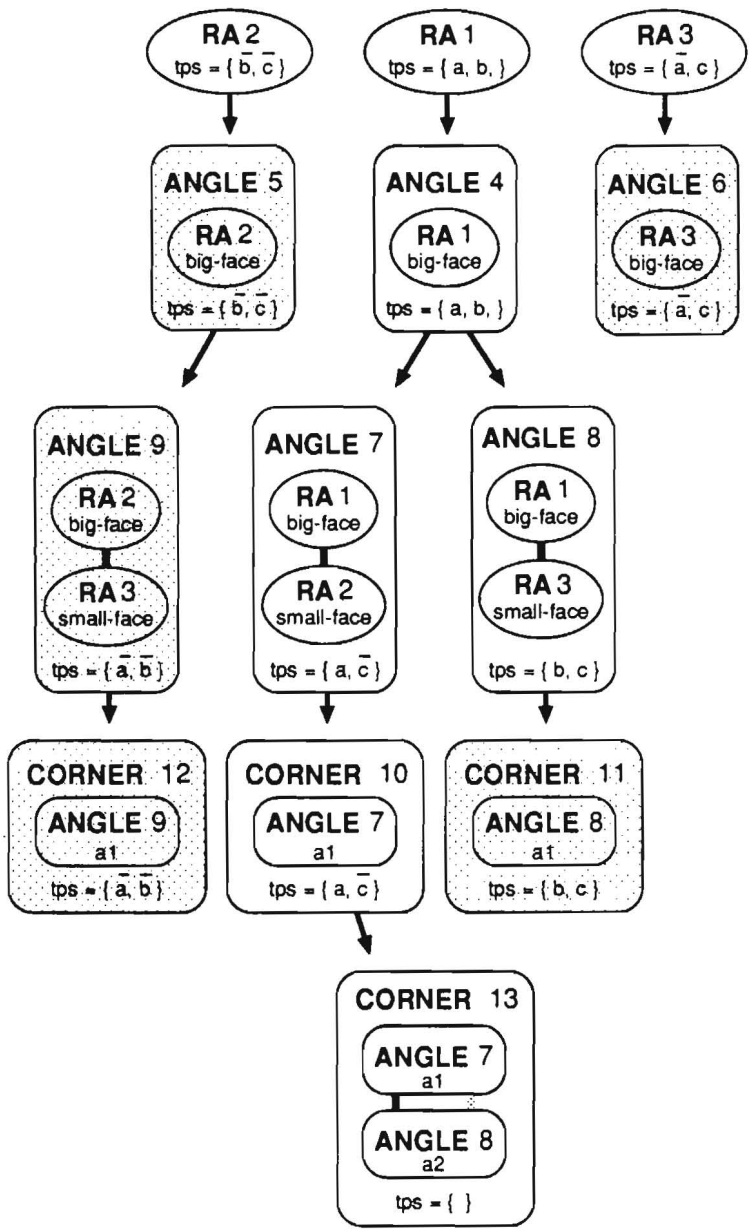


Abbildung 3.6: Ein Beispiellauf des Parsers

Die Abb.3.6 zeigt das Parser-Protokoll bis zum Auffinden des ersten Parses. Den drei Kanten des Eingabegraphen sind Bindepunkte zugeordnet, wie in Abb3.1 auf Seite 24. Mit der im vorigen Abschnitt (**merge-tps**) angegebenen „Vererbungsregel" für Bindepunkte

ergibt sich z.B. das tps von ANGLE_7 aus den tps des pp's ANGLE_4 und des cp's RA_2 wie folgt:

$$\frac{\{a, b\}, \quad \{\overline{b}, \overline{c}\}}{\{a, \overline{c}\}}$$

Das resultierende tps läßt sich also auch als Resolvente eines Resolutionsschritts deuten, wobei Bindepunkte für Literale stehen: Der Parsing-Algorithmus ähnelt dann Kowalski's Connection-Graph (Klausel-Graph) Verfahren für den Resolutionskalkül, wie es z.B. in [BlBü 87] Seite 49-64 beschrieben wird. Das tps eines Parses muß leer sein: Das entspricht der leeren Klausel!

Die Analogie gilt allerdings nur, solange keine Überlappungen berücksichtigt werden müssen. Ein Beispiel dafür ist das cp CORNER_{13}: Es entsteht durch Kombination von pp CORNER_{10} mit dem cp ANGLE_8, wobei die Überlappung in RA_1 zu beachten ist. Ohne deren Berücksichtigung erhielte man (durch „Resolution" wie oben):

$$\frac{\{a, \overline{c}\}, \quad \{b, c\}}{\{a, b\}}$$

Dieser Übergang wäre aber im ANLGG-Sinne nicht korrekt, d.h. simuliert nicht die Einbettung von CORNER_{13} in den (hier leeren) Restgraphen. Dazu müssen zusätzlich die Bindepunkte $\text{RA}_1.\text{tps} \setminus (\text{CORNER}_{10}.\text{tps} \cap \text{ANGLE}_8.\text{tps})$ entfernt werden. Damit erhält man hier $\text{CORNER}_{13}.\text{tps} = \emptyset$.

Die grau unterlegten Patches tragen zum ersten Parse nichts bei und wären von einer optimalen Heuristik $<_A$ gar nicht erst erzeugt worden. Beachte, daß das partielle Patch ANGLE_4 zweimal verwendet wird, einmal zur Bildung von ANGLE_7 und einmal für ANGLE_8. Das cp CORNER_{13} repräsentiert einen von insgesamt zwei hier möglichen Parsen und entspricht dem Feature-Baum auf Seite 17 bzw. der Ableitung auf Seite 18.

Die bottom-up Suche des Parsers kann man auch als forward-reasoning auffaßen, siehe dazu z.B. [Hin 91]. Prolog verwendet in seiner backward-reasonig Beweisprozedur genau die entgegengesetzte Strategie. Der Parser erzeugt alle aus den gegebenen Fakten (Knoten des Eingabegraphen) und Regeln (Produktionen der ANLGG) herleitbaren Konsequenzen. In forward-reasoning Systemen werden ganz ähnliche Techniken verwendet wie hier, z.B. entspricht dem Chart des Chart-Parsers in [Hin 91] ein sogenannter retain stack.

3.5 Einige Eigenschaften des Parsers

In den vorigen Abschnitten wurde der Parser für ANLGGs vorgestellt. Die Frage ist jetzt, welche Eigenschaften er hat, insbesondere möchte man wissen:

- Terminiert der Parser immer, d.h. sind Endlosschleifen ausgeschlossen ?
- Findet der Parser garantiert alle Parse (Vollständigkeit) ?
- Welche Rolle spielen dabei die Heuristiken. Ist jede Heuristik erlaubt ?

3.5.1 Termination

Die Antwort auf die erste Frage ist: Nein. Terminierung kann für den Parser in dieser Allgemeinheit nicht garantiert werden, z.B. können Grammatiken, die Regeln der Form

$$\begin{aligned} a &\leftarrow b \\ b &\leftarrow a \end{aligned}$$

enthalten, zu einem endlosen Parserlauf führen, wenn der zu parsende Graph a- oder b-Koten enthält.

Um die Terminierung des Parsers für alle Eingabegraphen garantieren zu können, muß man von der ANLGG fordern, daß sie die sogenannte Kettenbedingung erfüllt.

Die Kettenbedingung ist beispielsweise erfüllt, wenn man für die GG eine Noethersche Partialordnung $>_R$ auf einer Menge M angeben kann, und eine Funktion γ, die Graphen auf Elemente aus M abbildet, so daß für jede Anwendung $g_n \xrightarrow{p} g_{n+1}$ einer Produktion p der GG gilt: $\gamma(g_n) >_R \gamma(g_{n+1})$. Eine Ordnung $>$ auf einer Menge M heißt Noethersch, falls es keine unendliche Kette $x_1 > x_2 > x_3 > \ldots$ in M gibt. Generell ist es unentscheidbar, ob ein Regelsystem die Kettenbedigung erfüllt (siehe dazu z.B. [BlBü 87] Seite 118).

Intuitiv erfüllt eine GG die Kettenbedingung, wenn jede Anwendung einer Produktion den Graphen reduziert, d.h. in irgendeiner Form kleiner macht, so daß endlich viele solcher Verkleinerungen schließlich zu einem kleinsten, irreduziblen Graphen führen.

Für GGs, die die Kettenbedingung erfüllen, terminiert der Parser für beliebige Eingabegraphen nach endlicher Zeit. Beweisskizze:

> Die Agenda wird Anfangs mit je einem cp für jeden der (endlich vielen) Knoten des Eingabegraphen initialisiert, `choose` entfernt jeweils ein Patch aus der Agenda und fügt es in den Chart ein, `combine` erzeugt dafür jeweils eine endliche Menge von Nachfolgern und fügt sie in die Agenda ein.
>
> Die Anzahl dieser Nachfolger ist endlich: Sie ist kleiner oder gleich dem Produkt aus Anzahl der Produktionen der GG und der Anzahl der Patches im Chart (endlicher Verzweigungsgrad). Es kann auch keine unendliche Kette von Patches geben (mit `combine` als Nachfolger-Funktion), denn dann gäbe es eine unendlich lange Ableitungskette, Widerspruch dazu, daß die GG die Kettenbedingung erfüllt (kein

unendlich langer Ast). Dann ist der Baum, der (wieder mit `combine` als Nachfolger-Funktion) alle kombinatorisch möglichen Nachfolger eines Eingabeknotens (mit dem Knoten als Wurzel) enthält, endlich, denn er hat keinen Knoten mit unendlich vielen Söhnen und keinen unendlich langen Ast (Lemma von König).

Damit ist auch die Anzahl der kombinatorisch aus den endlich vielen Knoten des Eingabegraphen erzeugbaren Patches endlich.

Weil außerdem der Parser kein Patch zweimal erzeugt und jede Anwendung von `choose` ein Patch unwiderbringlich aus der Agenda entfernt, ist die Agenda nach endlich vielen paarweisen Anwendungen von `choose/combine` leer.

3.5.2 Vollständigkeit

Wenn die Agenda schließlich leer ist, enthält der Chart alle mit Produktionen der GG herleitbaren Parse. Beweisskizze:

Jede paarweise Anwendung von `choose` und `combine` erhält die folgende Chartinvariante:

> Für zwei beliebige Patches cp und pp des Charts ist ein daraus mit `combine` bildbarer Nachfolger tatsächlich bereits erzeugt und befindet sich in der Agenda oder im Chart.

Die Invariante gilt, weil `combine` gerade so definiert ist, daß es *alle* mit dem von `choose` gewählten Patch und den Patches im Chart erzeugbaren Nachfolger tatsächlich erzeugt, bevor das gewählte Patch in den Chart wandert.

Wenn die Agenda schließlich leer ist, folgt daraus: Jeder direkte Nachfolger eines Patches des Charts ist selber im Chart und durch Induktion: Alle rekursiv aus den Knoten des Eingabegraphen herleitbaren Patches sind im Chart, insbesondere alle Parse.

Wenn die Agenda leer ist, hat der Parser also nicht nur alle Parse, sondern alle kombinatorisch möglichen Patches erzeugt. Für große Eingabegraphen und/oder umfangreiche GGs ist diese Zahl aber astronomisch groß. Erschöpfende Suche ist dann praktisch unmöglich. Daraus folgt sofort die Notwendigkeit einer heuristischen Steuerung von `choose` (Siehe Abschnitt 3.2.2 Gute Patches zuerst). Weder der Beweis für Vollständigkeit, noch der für Termination machen einschränkende Annahmen über die Reihenfolge, in der `choose` Patches aus der Agenda holt, d.h. jede Heuristik dafür ist erlaubt. Schlechte Heuristiken verlängern zwar die Suche, jeder Parse wird aber schließlich gefunden, falls die GG die Kettenbedingung erfüllt.

3.5.3 Komplexität

Der ANLGG-Formalismus und der hier vorgestellte Parser für ANLGGs sind mächtig genug, um die Isomorphie zweier Graphen testen zu können. Zwei Graphen g_1 und g_2 sind isomorph, falls es eine Bijektion $V_1 \rightarrow V_2$ zwischen ihren Knoten gibt, die sorten- und

nachbarschaftserhaltend ist. Für ANLGGs ist das ein Spezialfall der Frage, ob eine Produktion p in einem Graphen g anwendbar ist. Graph-Isomorphie ist ein NP-vollständiges Problem. Damit kann auch ein allgemeiner ANLGG-Parser keine bessere Komplexität haben.

Es gibt ANLGGs, für die die vom Parser benötigte Rechenzeit und der Speicherbedarf exponentiell mit der Zahl n der Knoten des Eingabegraphen wachsen.

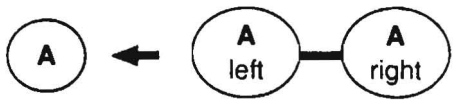


Abbildung 3.7: ANLGG mit nur einer Produktion

Betrachte dazu die ANLGG aus Abb.3.7 mit nur einer Produktion. Dabei ist $S = \{\mathrm{A}, \mathrm{B}\}$, metasort $= (\overset{\mathrm{A}}{_{goal}}, \overset{\mathrm{B}}{_{terminal}})$ und $\mathrm{B} \leq_S \mathrm{A}$. Die von dieser GG erzeugte Sprache ist die Menge aller zusammenhängenden Graphen. Wieviel verschiedene Parse ergeben sich damit für einen vollständigen Graphen mit n Knoten ?

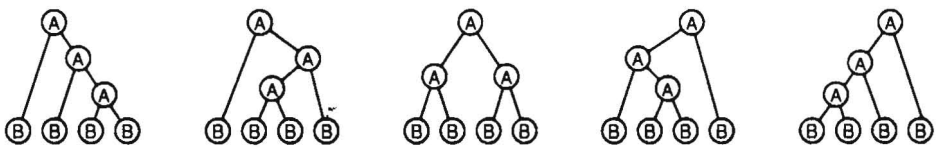


Abbildung 3.8: Die fünf möglichen Binärbäume (Parse) für $n = 4$, ohne Permutationen

Die Parse müssen Binärbäume mit n Blättern sein. Der linke Teilbaum eines Knotens steht jeweils für den Wert der Rolle *left*, der rechte für den Rollenwert von *right*. Die Zahl $\mathrm{f}(n)$ der bis auf Permutation der Blätter verschiedenen Binärbäume mit n Blättern läßt sich rekursiv angeben:

$$\begin{aligned} \mathrm{f}(1) &= 1 \\ \mathrm{f}(n) &= \sum_{i=1}^{n-1} \mathrm{f}(i)\ \mathrm{f}(n-i) \end{aligned}$$

Das sind genau die catalanschen Zahlen. Die Binärbäume lassen sich auch als Klammerungen ihrer Blätter auffassen. Damit erhält man:

$$\mathrm{f}(n) = \frac{1}{n}\binom{2(n-1)}{n-1}$$

Berücksichtigung der $n!$ Permutationen der Blätter liefert die Anzahl $\mathrm{Parse}(n)$:

$$\mathrm{Parse}(n) = n!\ \mathrm{f}(n) = \ldots = \frac{(2n-2)!}{(n-1)!}$$

Das läßt sich mit der Formel von Stirling[4] für $n > 1$ abschätzen zu:

[4]Formel von Stirling, $n > 1$: $\sqrt{2\pi n}\,(\frac{n}{e})^n \leq n! \leq \sqrt{2\pi n}\,(\frac{n}{e})^n\, e^{\frac{1}{12\,(n-1)}}$

$$\text{Parse}(n) \geq n^{n-1}$$

Die Tabelle zeigt die Anzahl der möglichen Parse für vollständige Graphen mit n Knoten in Abhängigkeit von n:

n	Parse(n)
1	1
2	2
3	12
4	120
5	1 680
6	30 240
7	665 280
8	17 297 280
9	518 918 400
10	17 643 226 000

Für $n = 8$ gibt es bereits mehr als 17 Millionen verschiedene Parse! Wegen der Vollständigkeit des Parsers werden alle diese Parse bis zum Terminieren des Parsers erzeugt, unabhängig von der verwendeten Heuristik. Selbst wenn man für den Aufwand zum Erzeugen und Speichern eines Patches konstante Werte annimmt, wächst also der bis zum Terminieren benötigte Speicherplatz und die Rechenzeit für einen vollständigen Eingabegraphen exponentiell mit der Zahl n seiner Knoten. Das Verhalten ist hier deswegen exponentiell, weil die Zahl der Parse exponentiell mit n wächst. Die Komplexität des Parsers sollte aber fairerweise am Aufwand zum Bestimmen eines ersten Parses gemessen werden. Zum Beispiel wird die Komplexität des Earley-Parsers für ambiguente Grammatiken mit $O(|G|^2|n|^3)$ angegeben [AU 72], obwohl es kontextfreie Grammatiken gibt, für die bestimmte Eingaben $n!$ Parse erzeugen.

Die folgende Überlegung zeigt aber, daß bei ungünstiger Heuristik $<_A$ auch der Aufwand bis zum Erzeugen des ersten Parses exponentiell mit n wächst. Im ungünstigsten Fall holt nämlich $<_A$ die Patches so aus der Agenda, daß alle kombinatorisch überhaupt erzeugbaren partiellen und vollständigen Patches, die keine Parse sind, vor dem ersten Parse erzeugt werden. Durch Abzählen aller vollständigen Patches cp mit $n-1$ Blättern, wie oben, zeigt man, daß diese Zahl mindestens wie $(n-1)^{(n-2)}$ wächst.

Das Problem mit der obigen Grammatik ist nicht, daß sie so ambiguent ist. Ambiguenz erleichtert die heuristisch gesteuerte Tiefensuche, denn wenn in vielen Ästen des Suchbaumes Parse zu finden sind, erhöht das die Chance, auch mit suboptimaler Heuristik rechtzeitig einen Parse zu finden. Das Problem ist vielmehr, daß die Zahl der überhaupt erzeugbaren Patches sehr schnell mit der Zahl n der Eingabe Graphen wachsen kann. In solchen Situationen können dem ANLGG-Parser nur starke, d.h. domänenspezifische Heuristiken helfen, zum Ziel zu kommen. Erschöpfende Suche kann dann schon bei sehr kleinen Problemen unmöglich sein.

Möglicherweise läßt sich für die Unterklasse der ANLGG-String-Grammatiken, wie sie zur Beschreibung rotationssymmetrischer Drehteile verwendet werden, eine polynomiale obere Schranke angeben. Der Parser verhält sich für String-Grammatiken wie ein Insel-Parser, wie in [SFP 88] (ohne Komplexitätsanalyse) beschrieben.

3.6 ANLGGs und Feature Erkennung

In diesem Abschnitt wird untersucht, ob und wie der ANLGG-Formalismus und der Parser die Forderungen erfüllen, die sich aus der anvisierten Anwendung auf das Feature-Erkennungsproblem ergeben.

In der Aufgabenstellung auf Seite 4 waren dazu einige Forderungen formuliert worden, die größtenteils aus [BKL 91b] stammen:

1. *Überlappung*

   ANLGGs erlauben neben der Formulierung von Nachbarschaft auch die explizite Formulierung von Überlappungen der Komponenten eines Features. In der Produktion aus Abb.1.2 wird z.B. eine Ecke als ein Feature definiert, das sich aus zwei Winkeln a1 und a2 zusammensetzt, die sich gegenseitig überlappen, d.h. eine gemeinsame Fläche haben.

2. *Dimensionsabhängigkeit*

   ANLGGs arbeiten mit attributierten Knoten. Sie erlauben die Formulierung beliebiger funktionaler Beziehungen zwischen den Attributen. Die Attribute können z.B. geometrische und technologische „Dimensionen" kodieren. In der Produktion aus Abb.1.1 wird ein Winkel als ein Feature definiert, das ein Attribut *area* hat, das die Gesamtoberfläche des Winkels angibt. Das Attribut steht in funktionaler Beziehung zu den beiden Komponenten des Winkels, denn der Attributwert ergibt sich als Summe der Flächen der beiden Rechtecke, aus denen der Winkel besteht.

3. *Hierarchie*

   Der ANLGG-Parser liefert einen Feature-Baum. Überlappungen sind dabei durch Überlappung (structure sharing) der entsprechenden Teilbäume realisiert, der Feature-Baum ist also streng genommen gar kein Baum, sondern nur ein dag (directed acyclic tree) mit Wurzel. Abb.2.4 zeigt z.B. so einen Feature-Baum. Der Knoten RA_1 hat darin zwei Väter, $ANGLE_4$ und $ANGLE_5$ und ist damit ein Beispiel für einen überlappenden Teilbaum der Tiefe 1.

4. *Qualitative Beschreibung*

   Jede Grammatik-Regel faßt i. allg. (Die Ausnahme bilden Regeln der Form $a \leftarrow b$) mehrere Knoten zu einem einzigen zusammen, das Parsen kann so als ständiges Kompaktifizieren oder Abstrahieren des ursprünglichen Werkstückmodells aufgefaßt werden. Natürlich hängt die Qualität der am Ende gelieferten Beschreibung von der Qualität des in der verwendeten ANLGG kodierten Wissens ab.

5. *Ambiguenz*

   Der ANLGG-Parser ermöglicht durch die Verwendung einer Agenda die heuristisch gesteuerte Suche nach einem ersten Parse. Anders als bei erschöpfender Breitensuche kann so schon sehr früh ein Parse gefunden werden (wenn man eine gute Heuristik hat). Die hohe Ambiguenz des ANLGG-Formalismus wirkt sich dabei vorteilhaft aus: Je mehr Parse in den einzelnen Zweigen des Suchbaumes vorkommen, desto größer die Chance, daß bei heuristisch gesteuerter Tiefensuche auch mit einer suboptimalen Heuristik ein Parse gefunden wird.

6. *Ähnliche Feature*

   Man kann hier zwei Arten von Ähnlichkeiten unterscheiden: Die Produktionen und Rollenspezifikationen einer ANLGG bilden eine *has-parts* Hierarchie; die erste Ähnlichkeit ist eine Ähnlichkeit bezüglich dieser Hierarchie. Die Sortenhierarchie repräsentiert taxonomisches Wissen und ist eine *is-a* Hiearchie; auf diese Hierarchie bezieht sich die zweite Form der Ähnlichkeit von Featuren.

   - Feature können einander ähnlich sein in dem Sinne, daß sie teilweise aus denselben Komponenten bestehen. Durch Verwendung eines Charts wird diese Art von Ähnlichkeit effizient ausgenutzt: Scheitert die Instantierung einer Feature-Produktion, dann sind alle vollständig erkannten Komponenten des Features zur Instantierung ähnlicher Feature weiterverwendbar. In ähnlicher Weise werden auch alle unvollständigen Instantierungen einer Produktion zu Erzeugung alternativer, vollständiger Instanzen wieder oder mehrfach verwendet. Wiederverwendbarkeit alleine reicht aber noch nicht. Die in einer gegebenen Situation wiederverwendbaren Komponenten sollten auch schnell gefunden werden können, um die eingesparte Herleitungsarbeit nicht durch langes Suchen im typischerweise sehr großen Chart wieder zu verlieren. In Earleys's Chart-Parser [Ear 70] dienen die Anfangs- und die Endposition einer gesuchten Komponente im Eingabestring als Suchindex. Im ANLGG-Parser werden die Kanten des Eingabegraphen, die Sorten und die Komponenten selbst als Index benutzt.
   - Die zweite Art von Ähnlichkeit bezieht sich auf Spezialfälle und Varianten: Feature können dieselben Komponenten haben und trotzdem verschieden sein, z.B. können zwei Rechteckflächen ein Winkel-Feature ANGLE bilden und gleichzeitig auch den rechtwinkligen Speziallfall R-ANGLE.

     Durch die Verwendung von Sortenhierarchien in ANLGGs und deren Integration in den Parsing-Algorithmus wird die Beschreibung solcher Varianten und Spezialfälle effizient unterstützt. Effizient heißt hier: Weniger Regeln zur Beschreibung einer gegebenen Domäne, dadurch kleinere, kompaktere Grammatiken und dadurch effizienteres Parsen. Wenn z.B. R-ANGLE eine Subsorte von ANGLE ist, dann wird der Parser ein einmal entdecktes R-ANGLE Feature automatisch an allen Stellen verwenden, an denen nur ein ANGLE Feature verlangt wird.
     Die Produktion CORNER ← ANGLE, ANGLE aus Abb.1.2 braucht also nur einmal in der ANLGG enthalten sein, und nicht etwa in drei weiteren Spezialversionen für den Fall, daß die erste oder zweite ANGLE Komponente oder alle beide R-ANGLE sind. Außerdem sind ANGLE und R-ANGLE durch eine einzige ANLGG-Produktion der Form ANGLE ← RA, RA mit entsprechendem Sortenconstraint für R-ANGLE beschreibar.

Kapitel 4

Anwendung auf das Feature-Erkennungsproblem

Der im vorigen Kapitel beschriebene Parser wurde im Rahmen dieser Arbeit in Common Lisp implementiert (siehe Anhang A und B) und läuft auf SPARC stations mit ivory boards.

Außerdem wurden zwei größere ANLG-Grammatiken entwickelt, die Expertenwissen aus der Domäne der Dreh- bzw. der Frästeilfertigung repräsentieren. Diese beiden Grammatiken und die damit gemachten Erfahrungen werden in diesem Kapitel beschrieben.

4.1 turning.gg: Eine ANLGG für Drehteile

Rotationssymmetrische Werkstücke werden in TEC-REP [BKL 91a] als lineare Kette von attributierten Knoten repräsentiert. Man hat es also beim Parsen in dieser Domäne nicht mit allgemeinen Graphen, sondern nur mit der Unterklasse der Strings zu tun. Die Feature können durch eine String-Grammatik beschrieben werden.

Der Parser verhält sich dann wie ein Insel-Parser[1], wie er z.B. in [SFP 88] beschrieben wird, d.h. er arbeitet sich nicht sequentiell von links nach rechts durch den Eingabestring, (wie z.B. FEAT-PATR alias D-PATR mit seiner top-down, breath-first Strategie), sondern arbeitet simultan an allen Stellen des Eingabestrings zugleich, wobei die jeweils aktive Stelle durch die $<_A$ Heuristik bestimmt wird. Beim Parsen von Strings kann kein erzeugbares Patch mehr als zwei Bindepunkte in seinem tps haben. Da der Parser für das Parsen allgemeiner Graphen ausgelegt ist, wird dieser Umstand hier aber nicht ausgenutzt.

Die Feature-Grammatik für Drehteile aus [BKS 91] wurde in den ANLGG-Formalismus übertragen und von Christoph Klauck noch einmal erheblich erweitert. Sie enthält momentan (1.Mai 92) 64 Produktionen mit zusammen 187 Rollenspezifikationen und eine Sortenhierarchie mit 43 Sorten. Abb.4.1 ist eine Zusammenstellung einiger rotationssymmetrischer Werkstücke, deren TEC-REP Graphen sich mit dieser turning.gg genannten

[1] Insel-Parser bieten für die Analyse natürlicher Sprache einerseits die Möglichkeit, teilweise Parse zu erzeugen, also maximale parsebare „Inseln" im Eingabestring zu identifizieren. Andrerseits bieten sie die Möglichkeit, den Suchraum klein zu halten, indem man mit dem Parsen nicht am Rand des Satzes anfängt, sondern im meist höher informierten Zentrum (Head) eines Satzes, bzw. einer Phrase.

ANLGG ableiten lassen. Die Zeichnungen wurden mit einem Grafikprogramm von Johannes Schwagereit automatisch aus den TEC-REP Graphen erzeugt.

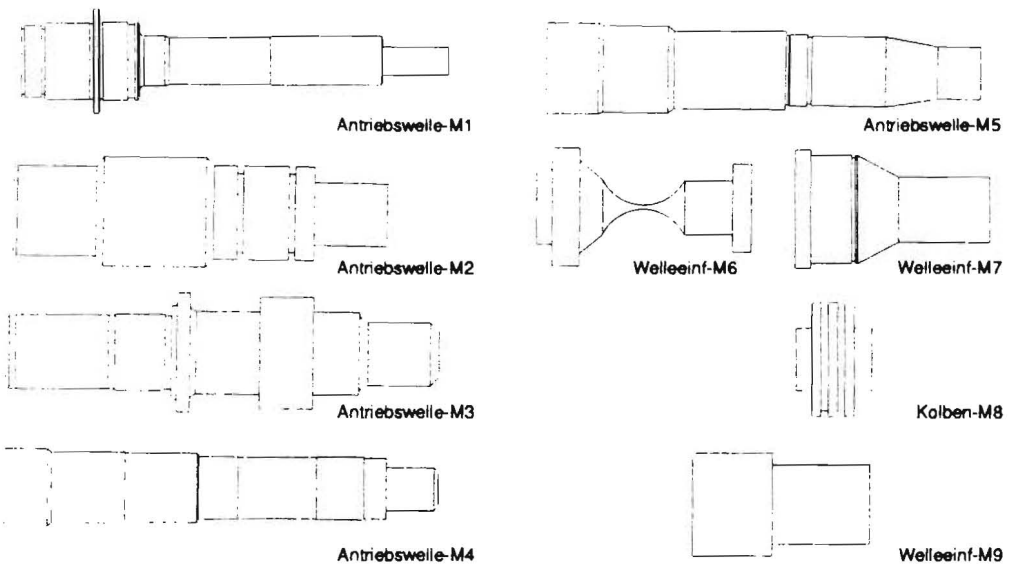


Abbildung 4.1: Rotationssymmetrische Werkstücke

Die Antriebswelle-M2 ist z.B. durch einen Graphen mit 18 Knoten RSC_1, $RSEC_2$, ..., RSC_{18} und 17 Kanten repräsentiert. Die Kürzel RSC, RSEC usw. sind TEC-REP Sorten, wie in [BKL 91a] definiert. Zum Beispiel bedeuten:

RSC	rotational symmetric circular surface, Kreisfläche
RSEC	rotational symmetric envelope of a cone, Kegelmantelfläche
RSCJ	rotational symmetric cylinder jacket, Zylindermantelfläche
RSR	rotational symmetric ring, Kreisringfläche

Zu jedem Knoten gehört noch ein Satz Attribute für die geometrischen und technologischen Daten eines Flächenelements. Abb.4.2 zeigt neben den 18 terminalen Knoten des TEC-REP Eingabegraphen auch einen Feature-Baum, der vom Parser mit der turning.gg abgeleitet wurde. Der Feature-Baum interpretiert den Graphen als Shaft (Welle), der sich aus einem Middlestep (Mittelabsatz) und zwei Trunnions (Zapfen) zusammensetzt. Ein Mittelabsatz ist dabei so definiert, daß er die dickste Stelle einer Welle enthält, die Konturline der Welle also rechts und links davon monoton abfällt. Ein Zapfen ist ein Wellenende, das, anders als ein Crest (Kragen), nicht die dickste Stelle der Welle enthält. Diese (hier verkürzt wiedergegebenen) Feature-Definitionen sind von einem Maschinenbauer, H.-W.Höpper, der eine Ausbildung als Dreher (3 Jahre) und ein Jahr praktische Erfahrung absolviert hat, im Rahmen einer Studienarbeit zusammengestellt worden. Die turning.gg ist von ihrer Sortenhierarchie her auf die Unterscheidung von Innen- und Außenbearbeitung vorbereitet, Regeln für Innenbearbeitung sind aber weggelassen. Die auf -OUT endenden Sorten sind also Subsorten von allgemeineren Sorten, z.B. Ist LONGTURN-OUT eine Subsorte von LONGTURN, und steht für den Speziallfall einer Zylindermantelfläche, deren Material sich im Zylinderinneren befindet. LONGTURN-IN steht entsprechend für zylinderförmige Löcher, Bohrungen usw.

Beachte, daß im Feature-Baum für die Welle M2 einige Knoten des Eingabegraphen, z.B. $RSEC_4$ überlappen, also zu mehreren Featuren gleichzeitig gehören. Beim Parsen der

Welle wurde noch ein anderer, sehr ähnlicher Parse gefunden. Beide Parse wurden in weniger als 3.5 Sekunden erzeugt.

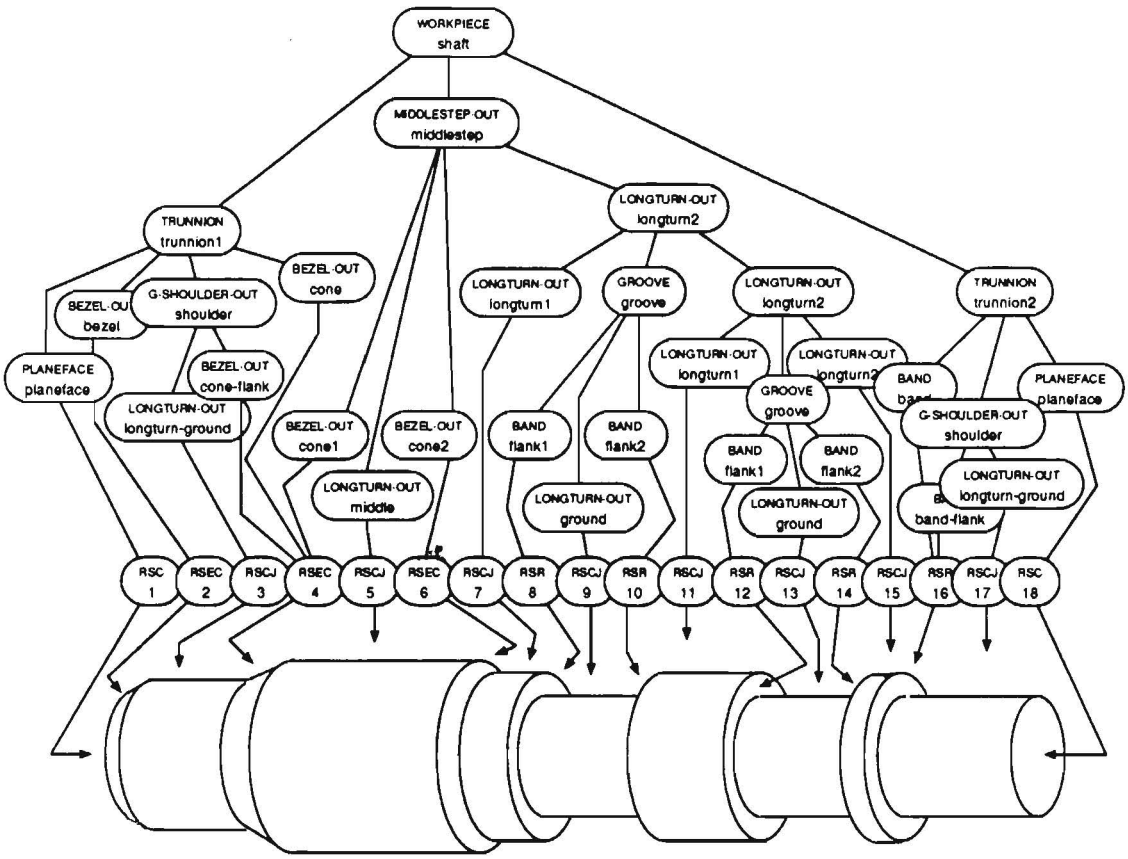


Abbildung 4.2: Antriebswelle-M2 und zugehöriger Feature-Baum

Die folgende Tabelle zeigt das Laufzeitverhalten des Parsers beim Parsen mit der turning.gg für die Werkstücke aus Abb.4.1. Dabei wurde als Suchstrategie Breitensuche gewählt, d.h. $<_A$ wählt jeweils das am längsten wartende Patch aus der Agenda. Die Laufzeit[2] ist die Zeit bis zum Terminieren mit leerer Agenda, es wurden also bis dahin nicht nur alle Parse, sondern *alle* kombinatorisch erzeugbaren Patches erzeugt. In der Tabelle gibt Patches die Zahl der insgesamt erzeugten Patches an, einschließlich der Patches, die die Knoten und Kanten des Eingabegraphen repräsentieren. Die gemessenen Laufzeiten sind um einen Faktor 4 bis 10 besser, als die Laufzeiten des D-PATRs [BKS 91] für dieselben Werkstücke.

[2]angegeben ist hier die „elapsed time", das ist ungefähr die reale Antwortzeit des Lisp-Systems (gemessen bei Einbenutzerbetrieb), also die reine CPU-Zeit plus Zeit für die Speicherverwaltung (page-faults, fetches) plus Zeit für Prozeßumschaltungen usw.

Werkstück	Knoten	Kanten	Parse	Patches	Zeit[sec]
Antriebswelle-M1	37	36	1	688	13.20
Antriebswelle-M2	18	17	2	243	3.36
Antriebswelle-M3	27	26	4	592	10.04
Antriebswelle-M4	19	18	3	448	6.97
Antriebswelle-M5	17	16	1	272	3.78
Welleeinf-M6	11	10	1	101	1.35
Welleeinf-M7	11	10	1	135	1.72
Kolben-M8	17	16	2	203	2.75
Welleeinf-M9	5	4	1	51	0.89

Leider stehen zur Zeit geeignete domänenspezifische Heuristiken nicht zur Verfügung. Die hier gemessenen Laufzeiten stellen den worst-case dar, d.h. entsprechen dem Verhalten des Parsers, wenn man nach dem ersten gefundenen Parse abbrechen würde, aber mit $<_A$ die schlechteste überhaupt denkbare Heuristik benutzt. Auch die schlechteste Heuristik kann wegen der Vollständigkeit des Parsers nicht verhindern, daß alle Parse schließlich gefunden werden. Sie sorgt aber dafür, daß alle kombinatorisch überhaupt erzeugbaren Patches vor dem ersten Parse erzeugt werden.

4.2 milling.gg: Eine ANLGG für Frästeile

Frästeile werden in TEC-REP [BKL 91a] durch Graphen mit attributierten Knoten repräsentiert. Unter Frästeilen werden hier alle Werkstücke verstanden, die sich durch Bohr-, Säge- und Fräs-Operationen (Fragmentierungen) aus einem quaderförmigen Rohling herstellen lassen. Ein solcher Rohling oder Block ließe sich im ANLGG-Formalismus durch eine Produktion wie in Abb.4.3 beschreiben.

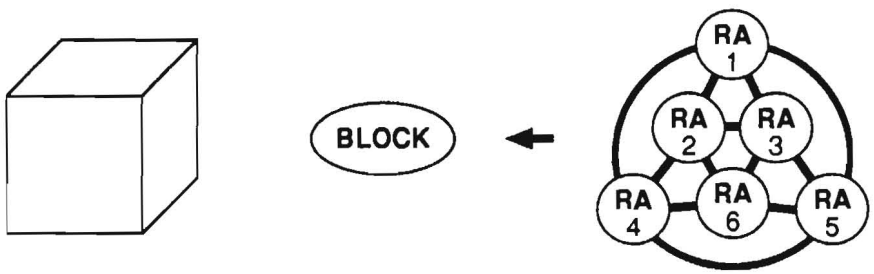


Abbildung 4.3: Produktion für einen unfragmentierten Block

Die sechs RA-Knoten[3] stehen dabei für die sechs Rechteckflächen des Blocks. Diese naheliegende Definition des unfragmentierten Blocks durch eine einzige Graph-Produktion wirft aber für die Beschreibung von fragmentierten Varianten schwere Probleme auf.

Solche Fragmentierungen sind natürlich in der Praxis häufig anzutreffen. Das Problem ist, daß an jeder der acht Ecken und zwölf Winkelkanten eines Blocks Fragmentierungen auftreten können, und zwar in fast beliebigen, sehr vielen Kombinationen. In milling.gg, der ANLGG für Frästeile, werden Blöcke daher schrittweise zusammengesetzt, auf jeder

[3]RA wie Rectangular Surface, eine TEC-REP Sorte.

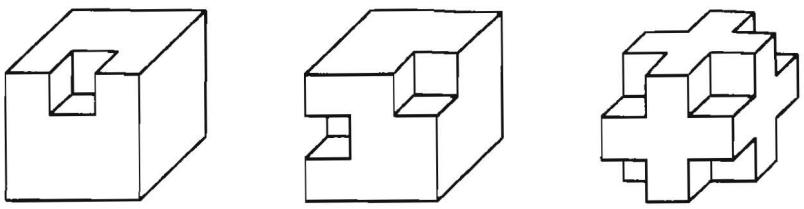


Abbildung 4.4: Einige fragmentierte Blöcke

Ebene können dann eventuelle Fragmentierungen berücksichtigt werden. Auf der untersten Ebene hat man Feature der Sorte PLANEFACE. Zwei PLANEFACEs bilden einen Winkel ANGLE. Winkel sind entweder konvex (ANX) oder konkav (ANV), je nachdem, ob sie Material ein- oder ausschließen. In der Sortenhierarchie sind ANX und ANV also Subsorten von ANGLE. RECT-ANGLE, RECT-ANX usw. sind Subsorten, die für die entsprechenden rechtwinkligen Spezialfälle stehen. Auf der Ebene der ANGLE können Fragmentierungen zwischen zwei benachbarten Flächen erkannt werden, z.B. ist die Fragmentierung eines konvexen Winkels durch eine Bucht durch eine Produktion der Form FRAG-ANX ← ANX, OPEN-GROOVE beschreibbar.

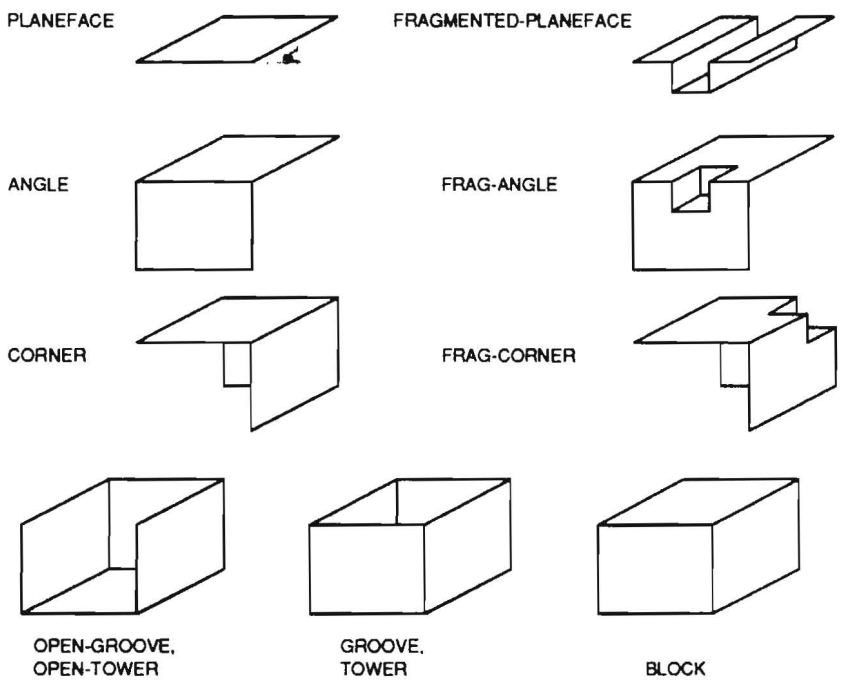


Abbildung 4.5: Einige Feature der milling.gg

Die nächst höhere Ebene ist die Ebene der Ecken (CORNER). Eine Ecke entsteht durch geeignte Überlappung dreier, ggf. fragmentierter Winkel. Fragmentierungen zwischen zwei Flächen brauchen dabei nicht mehr berücksichtigt werden, weil sie bereits auf der Ebene der Winkel erkannt wurden. Auf der Ebene der Ecken können Fragmentierungen zwischen drei benachbarten Flächen erkannt werden, z.B. die Fragmentierung eines konvexen Winkels durch einen kleineren konkaven Winkel.

Auf der nächst höheren Ebene bilden zwei ggf. fragmentierte Ecken eine OPEN-GROOVE, bzw. einen OPEN-TOWER, je nachdem ob die beiden Ecken konkav oder

konvex sind. Dabei können Fragmentierungen zwischen zwei gegenüberliegenden, nicht benachbarten Flächen erkannt werden, beispielsweise ein Bohrloch, das die beiden gegenüberliegenden Flächen eines OPEN-TOWERs verbindet.

Noch eine Ebene höher erhält man Turm (TOWER) und Nut (GROOVE). Beide können ein PLANEFACE fragmentieren: Hier schließt sich die Rekursion, d.h. die hier erhaltene fragmentierte Fläche kann nun ihrerseits wieder zur Bildung von Winkeln, Ecken usw. beitragen. Zwei Türme bilden schließlich einen BLOCK.

Dieses grammatische Grundgerüst ist durch Verfeinern der Sortenhierarchie und Hinzunahme entsprechender Sortenconstraints zu den bestehenden Produktionen oder Hinzunahme neuer Produktionen erweiterbar. In milling.gg hat z.B. PLANEFACE noch eine Untersorte STEP. Diese Feature wird durch drei benachbarte Flächen top, border und bottom gebildet, wobei top und bottom parallel sein müssen.

Abb.4.6 zeigt den Feature-Baum, der vom Parser mit der milling.gg für eine U-Führung abgeleitet wurde. Dargestellt ist nur der interessanteste Teil des Feature-Baums, in dem die Fragmentierung des quadrischen Grundkörpers beschrieben wird: Der BLOCK enthält einen rechten Winkel FRAGMENTED-RECT-ANGLE, der durch eine Bucht RECT-GROOVE fragmentiert wird. Eine der beide Flächen des Winkels ist ein FRAGMENTED-PLANEFACE, das durch Überlappung zweier gleichtiefer, rechtwinkliger Stufen RECT-STEP entsteht.

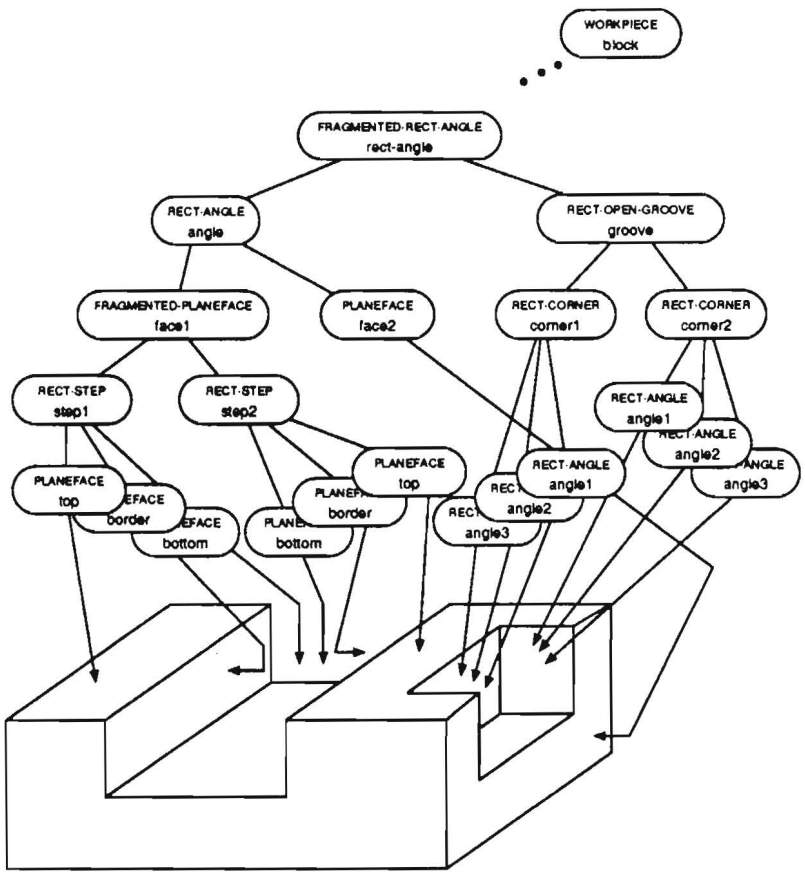


Abbildung 4.6: U-Führung und zugehöriger Feature-Baum

Abb.4.7 zeigt den Feature-Baum für ein Multispanblock genanntes Frästeil. Das

FRAGMENTED-PLANEFACE und das STEP Feature haben Attribute, die hier nicht dargestellt sind, und die seine „virtuelle" Fläche (Ortsvektor und zwei Richtungsvektoren) angeben, also die Rekonstruktion der unfragmentierten Fläche.

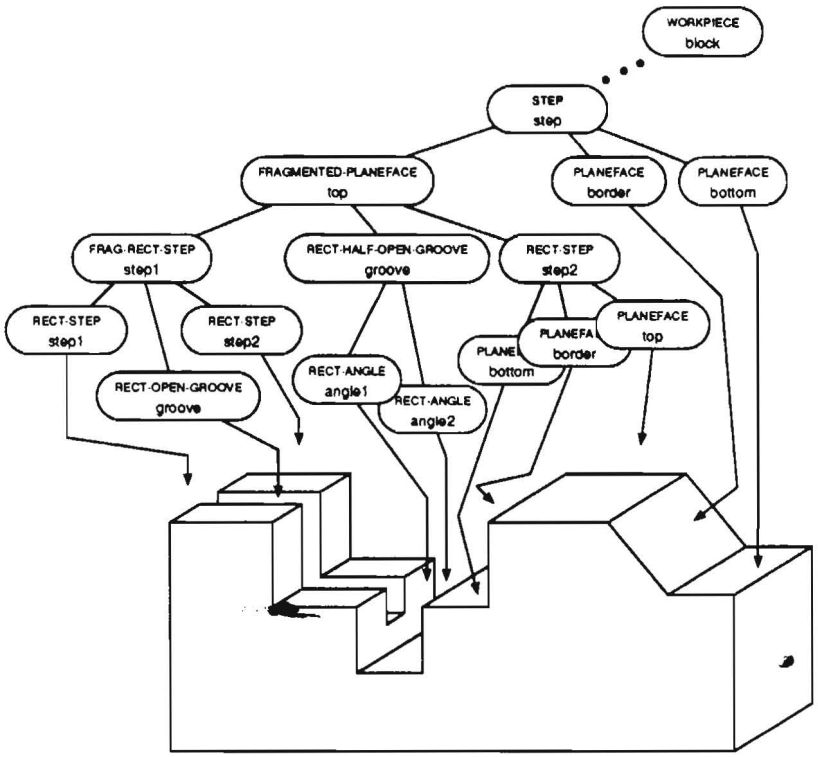


Abbildung 4.7: Multispanblock und zugehöriger Feature-Baum

Die folgende Tabelle gibt das Laufzeitverhalten des Parsers bis zum Auffinden des ersten Parses an. Als Suchstrategie wurde wieder Breitensuche verwendet. Patches gibt hier die Zahl der bis zum Finden des ersten Parses erzeugten Patches an.

Werkstück	Knoten	Kanten	Patches	Zeit[sec]
Quader	6	12	294	5.9
U-Führung	28	52	738	24.1
Multispanblock	50	100	1224	77.3

Die milling.gg wird noch weiterentwickelt, geschildert ist hier der vorläufige Stand der Entwicklung. Die meisten der Feature-Definitionen (Winkel, Bucht, Nut, Turm usw.) wurden von einem Maschinenbauer, Andreas Müller, zusammengestellt. Die Grammatik bildet aber beim jetzigen Stand nur eine allgemeine Plattform für eine weitere Verfeinerung unter fertigungsrelevanten Gesichtspunkten.

Kapitel 5

Zusammenfassung und Ausblick

Im ersten Abschnitt dieses Kapitels werden die wesentlichen Resultate dieser Arbeit zusammengefaßt. Dabei wird besonders auf offen gebliebene Fragen hingewiesen. Die wichtigste offene Frage betrifft wohl die heuristische Steuerung des Parsers oder allgemeiner: Die Frage nach denkbaren Strategien zur Verhinderung der kombinatorischen Explosion beim Graph-Parsen. Im zweiten Abschnitt werden dazu zwei mögliche Lösungswege skizziert.

5.1 Zusammenfassung

In dieser Arbeit wurde ein ANLGG genannter Graph-Grammatik-Formalismus definiert und ein Graph-Parser für ANLGGs entwickelt, implementiert und an Beispielen getestet.

Der ANLGG-Formalismus wurde speziell für die Repräsentation von Featuren in CIM-Systemen entwickelt. Dabei wurden eine Reihe Feature-spezifischer Charakteristika berücksichtigt (siehe 3.6): ANLGGs erlauben die Spezifikation von relationalen Nachbarschafts- und Überlappungsbeziehungen zwischen den Komponenten eines Features, die Definition von Feature-Attributen und die Formulierung beliebiger funktionaler Beziehungen zwischen den Attributen eines Features und denen seiner Komponenten.

Von den aus der Literatur [EKG 79-90] bekannten Graph-Grammatik-Formalismen sind die ANLGGs den sogenannten Hyperedge- oder Plex-Grammatiken [DrKr 90] am ähnlichsten. (Siehe dazu und für eine genauere graph-grammatische Charakterisierung der ANLGGs Abschnitt 2.2).

Die Einbeziehung von Sortenhierarchien in den ANLGG-Formalismus hat sich bei der Entwicklung der Dreh- und Frästeilgrammatiken (siehe Kapitel 4) sehr bewährt. Man hat damit die Möglichkeit, klar zwischen is-a Hierarchien (z.B. ANGLE ← R-ANGLE) und has-parts Hierachien (z.B. ANGLE ← FACE1, FACE2) zu unterscheiden. Das führt einerseit zu lesbareren Grammatiken, andrerseits wirkt sich die Einsparung von Produktionen durch Verwendung einer Sortenhierarchie vorteilhaft auf das Laufzeitverhalten des Parsers aus.

Die Entwicklung, Implementierung und Beschreibung des ANLGG-Parsers bildet den Kern dieser Arbeit. Er kann als „maßgeschneidertes" Werkzeug für die Feature-Erkennung in CIM-Systemen angesehen werden. Der Parser arbeitet chartbasiert und bottom-up. Er zählt alle Parse auf (Vollständigkeit, 3.5.2) und terminiert, falls die Grammatik terminiert (siehe 3.5.1). Der Parsingvorgang kann auch als forward reasoning aufgefaßt werden (siehe dazu 3.4).

Die Korrektheit des Parsers wird in dieser Arbeit nicht nachgewiesen. Der Parser wäre in der Art, wie er hier präsentiert wird und die sich stark an der Implementierung anlehnt, auch schwer diskutierbar. Ein Korrektheitsbeweis müßte erstens den Nachweis erbringen, daß die Simulation der ANLGG-Einbettungsregel durch Bindepunkte korrekt ist. Der Beweis dazu wurde hier nur skizziert (siehe 3.1.1). Zweitens wäre zu zeigen, daß die „Vererbungsregel" für Bindepunkte (bei Kombinieren zweier Patches, siehe 3.3.3) im ANLGG-Sinne korrekt ist. Das wurde hier nur am Beispiel plausibel gemacht (siehe 3.4). Vor allem die Art der Berücksichtigung von Überlappungen ist etwas überraschend und nicht unmittelbar einzusehen und bedürfte daher eines formalen Beweises.

Die grundlegende Idee des ANLGG-Parsers ist dieselbe wie bei Earley's Chart-Parser [Ear 70] für String-Grammatiken. Die günstige Komplexität des Earley-Parsers kann aber für den ANLGG-Parser nicht mehr garantiert werden. Stattdessen wird hier versucht, trotz der theoretisch schlechten Problemklasse (NP) durch den Einsatz von Heuristiken praktikable Laufzeiten zu erreichen.

Im Rahmen dieser Arbeit wurden aber keine wirklich domänenspezifischen Heuristiken für den Parser entwickelt und getestet. Es ist daher eine offene Frage, ob sich starke Heuristiken finden oder automatisch aquirieren lassen, die die Anwendung des Parsers auch für „schwierige" Graphen und Grammatiken praktikabel machen. Die Erfahrungen vor allem mit der Frästeilgrammatik (milling.gg, siehe 4.2) zeigen jedenfalls, daß für die Behandlung echter, d.h. in der Praxis anzutreffender Werkstücke die Verwendung starker Heuristiken notwendig wird.

5.2 Zwei Strategien zur Verhinderung der kombinatorischen Explosion

Eine ANLGG für die Feature-Erkennung beispielsweise in der Domäne der Frästeile wird typischerweise durch „naives" Hinschreiben sehr vieler Feature-Definitionen (in Form von Produktionen einer ANLGG) durch einen Experten in der gegebenen Domäne entstehen. Die Erfahrungen mit verschiedenen „naiven" Vorstufen der milling.gg zeigen aber, daß sich mit einer solchen GG keine praktikablen Parser-Laufzeiten erzielen lassen. Die GG muß dazu entweder von Hand optimiert werden, was genaue Kenntniss des Parsing-Ablaufs voraussetzt, oder es müssen starke Heuristiken zur Verfügung gestellt werden. Beides kann von dem menschlichen Grammatik-Entwickler nicht erwartet werden.

Um unter diesen Voraussetzung trotzdem zum Erfolg zu kommen ist folgendes Vorgehen denkbar:

1. Strategie

- „naives" Hinschreiben einer ANLGG
- automatische Aquisition von Heuristiken mit Hilfe von
  - konnektionistischen Methoden
  - genetischen Algorithmen

Dabei ist mit *naives Hinschreiben* gemeint, daß der Experte Feature-Definitionen in Form von ANLGG-Produktionen und Sortenhierarchie niederschreibt, ohne sich dabei mit Überlegungen zu belasten, die die Effizienz des Parsing-Vorgangs betreffen. Für die resultierende ANLGG kann dann evtl. mit Hilfe konnektionistischer oder genetischer Methoden eine Heuristik automatisch aquiriert werden. Mit Heuristik ist hier die für das Laufzeitverhalten des Parsers entscheidende $<_A$-Heuristik (siehe 3.2.2) gemeint, die das jeweils vielversprechendste Patch aus der Agenda auswählt.

In [SuEr 90] wird ein konnektionistisches System vorgestellt, daß Heuristiken für ein heuristisch gesteuertes Deduktionssystem automatisch aquiriert. Auf den ANLGG-Parser läßt sich die Idee wie folgt übertragen:

1. Identifizieren einer Reihe lokaler und globaler Indikatoren, die den Zustand eines gegebenen Patches (z.B. seine Sorte, Anzahl seiner Komponenten usw.) bzw. den globalen Zustand des Parsers (z.B. Anzahl der Patches der Agenda, des Charts usw.) beschreiben.

2. Die $<_A$-Heuristik wird durch ein neuronales Netz realisiert, das mit Hilfe dieser Indikatoren jedem Patch eine heuristische Priorität zuordnet.

3. Damit werden viele verschiedene Werkstückgraphen der jeweiligen Domäne sehr oft geparst. Parserläufe mit kurzen Laufzeiten liefern positive, die übrigen liefern negative Trainingsdaten.

4. Das neuronale Netz wird mit den damit identifizierbaren „guten" und „schlechten" heuristischen Patchbewertungen trainiert.

Eine ähnliche Methode zur automatischen Aquirierung der $<_A$-Heuristik basiert auf genetischen Algorithmen [Gol 89]: Dabei wird nicht wie oben eine Heuristik schrittweise verbessert (hill-climbing, Gefahr lokaler Maxima), sondern ein ganzes „Rudel" von Heuristiken beobachtet, die genetisch, d.h. durch symbolische Baupläne repräsentiert werden. Die jeweils erfolgreichsten Heuristiken einer solchen Population paaren sich und zeugen durch Mischung (crossing over) ihrer Baupläne (bei evtl. zusätzlicher „Mutation") neue Heuristiken, die günstigstenfalls die besten Eigenschaften ihrer Eltern in sich vereinigen.

Eine zweite Möglichkeit, die nicht auf Heuristiken basiert, ist die folgende:

2. Strategie

- Entwicklung einer unspezifischen Basisgrammatik (Common-Sense Wissen)
- Optimieren dieser Grammatik, bis praktikable Laufzeiten erreicht werden
- Definition von Verfeinerungsoperationen für ANLGGs, die das Laufzeitverhalten garantiert nicht verschlechtern
- Domänenspezifisches Verfeinern der Basisgrammatik (Expertenwissen)

Die unspezifische Basisgrammatik repräsentiert dabei kein domänenspezifisches Expertenwissen, sondern geometrische Grundkonzepte (Fläche, Winkel, Ecke, Turm, Quader usw.) Diese Basisgrammatik wird dann von Hand und unter Ausnutzung der parserinternen Gegebenheiten solange optimiert, bis sich alle praktisch vorkommenden Werkstückgraphen damit parsen, d.h. also: mit den geometrischen Begriffen eines Nicht-Experten beschreiben lassen. Diese Grammatik braucht nur einmal entwickelt zu werden, d.h. sie ist wiederverwendbar.

Außerdem werden für ANLGGs Verfeinerungsoperationen definiert, deren Anwendung auf eine gegebene ANLGG das Laufzeitverhalten des Parsers garantiert nicht verschlechtert. Solche Operationen sind z.B. das Hinzufügen zusätzlicher Prädikate und funktionaler Constraints, das Verfeinern der Sortenhierarchie und die Spezifikation entsprechender Sortenconstraints in den Produktionen usw.

Bei der Grammatik-Entwicklung wendet der jeweilige Experte ausschließlich diese Operationen auf die Basisgrammatik an und erzeugt dadurch eine domänenspezifische ANLGG, die sein Expertenwissen repräsentiert, dabei aber garantiert das günstige Laufzeitverhalten der ursprünglichen Basisgrammatik hat.

Diese Strategie wurde bei der Entwicklung der milling.gg verfolgt. Diese stellt in dem hier beschriebenen Stadium ihrer Entwicklung (siehe 4.2) eine noch wenig optimierte Basisgrammatik dar. Die beiden genannten Strategien sind miteinander kombinierbar.

Anhang A

GraPaKL Handbuch

GraPaKL [1] ist eine Entwicklungsumgebung für Graph-Grammatiken, die auf dem ANLGG-Formalismus basieren. GraPaKL besteht aus den folgenden Komponenten:

- Grammatik-Compiler zur Compilierung von ANLGGs
- Chart-Parser für compilierte ANLGGs und attributierte Graphen
- Graphik-Tracer für die Fehlersuche in ANLGGs

Die folgenden Seiten geben eine kurze Einführung in die Arbeit mit GraPaKL.

A.1 Das GraPaKL Graph Grammatik Format

GraPaKL ist in Common Lisp implementiert. Grammatiken, Sortendefinitionen, Regeln und Graphen sind daher der Einfachheit halber selbst Lisp-Ausdrücke. Sie können deswegen wie Lisp Programme kommentiert werden. Die Syntax für GraPaKL Grammatiken, wie sie z.B. von GraPaKL's Grammatik-Compiler akzeptiert werden, ist [2]:

```
<gg> ::= (make-gg  :name <string>
                   { :default-graph <string> }
                   { :include <string>       }
                   { :sorts '(<sort-def>* )  }
                   { :rules '(<rule>* )      } )
```

Dabei haben die einzelnen Komponenten folgende Bedeutung:

name Der Name der Grammatik, ein String (ohne Suffix .gg, Gross- wird von Kleinschreibung unterschieden), dient mit Suffix .gg als Filename für die GG, mit Suffix .lisp als Filename für die benutzerdefinierten Prädikate und Funktionen der GG.

[1] Graph Parser Kaiserslautern

[2] Dabei bedeutet wie üblich {A} kein- oder einmaliges Auftreten von A, A|B entweder A oder B, A+ ein- oder mehrmaliges und A* kein-, ein- oder mehrmaliges Auftreten von A.

default-graph String, Filename des Graphen, der zusammen mit dieser GG geladen werden soll. Der angegebene Graph sollte mit der GG parsebar sein.

include String, Name einer GG, die vor dem Laden dieser GG geladen werden soll, d.h. die Sorten und Regeln der angegebenen GG werden mit dieser GG zu einer einzigen GG vereinigt. Dabei kann die hinzugenommene GG selber eine GG includen. Dadurch wird das modulare Aufbauen großer GGs unterstützt, also z.B. die Mehrfachverwendung einzelner, bereits ausgetesteter Module. Nützliche Module sind z.B. TEC-REP und RS-TEC-REP. Sie enthalten die Definitionen der terminalen Sorten zum Aufbau von Frästeil- bzw. Drehteilgrammatiken, wie sie in [BKL 91a] definiert sind.

sorts Liste mit den Sortendefinitionen der GG.

rules Liste der Regeln (Graph-Grammatik-Produktionen) der GG.

A.1.1 Sorten

Die von GraPaKL behandelten Graphen bestehen aus attributierten, gelabelten Knoten. Die Knotenlabel heißen Sorten. Die Knoten haben zwei Arten von Attributen, die einen liefern atomare Werte (Zahlen, Vektoren, Strings usw.), die Werte der anderen sind selber wieder Knoten. Diese Attribute werden Rollen genannt. Die Definition einer Sorte ordnet jeder Sorte eine Metasorte zu und definiert die möglichen Rollen und Attribute, die sogenanten Slots, die in einem Knoten dieser Sorte auftauchen können. Dabei müssen in einem Knoten nicht immer alle Slots besetzt sein, die in der Definition seiner Sorte angegeben sind. Die Sortendefinition bündelt Information, die andernfalls in den Regeln der GG verteilt repräsentiert würde. Sie stellt so etwas wie eine semantische Schnittstelle dar: GraPaKL interpretiert alle Knoten, Rollen- und Attributwerte in Abhängigkeit von deren Slot- und Metasorten. Im Kontext der intendierten Anwendung von GraPaKL, der Arbeitsplanung im Maschinenbau, verstärkt sich dieser semantiktragende Charakter der Sorten noch, denn den Sorten werden dann zusätzlich ungefähre Arbeitspläne, sogenannte Skelettpläne zugeordnet.

Die Syntax für Sortendefinitionen ist:

```
      <sort-def> ::= (<head> <slot-def>+)
          <head> ::= (<metasort> <sort>
                         { (include  . <sort>)  }
                         { (subsorts   <sort>+) } )
      <metasort> ::= terminal | optional | global | preterminal |
                     nonterminal | goal | edge
      <slot-def> ::= (<slot-sort> . <slot-label>)
    <slot-label> ::= <role-label> | <attribute-label>
     <slot-sort> ::= <role-sort> | <attribute-sort>
     <role-sort> ::= <rule-sort> | preterminal-sort |
                     terminal-sort | optional-sort
     <rule-sort> ::= nonterminal-sort | goal-sort
<attribute-sort> ::= attr | node | ref | ref-1 | ref-2 | ref-list
```

Dabei bedeuten die Komponenten im einzelnen folgendes:

head Enthält die zu definierende Sorte sort und deren Metasorte, optional ist noch ein include einer anderen Sortendefinition und die Deklaration von Untersorten mittels subsort möglich.

metasort Jede Sorte hat eine Metasorte, diese bedeuten im einzelnen folgendes (in Klammern sind jeweils typische TEC-REP Sorten und Feature der entsprechenden Metasorte angegeben):

- **terminal** Terminale Sorte der GG, das ist die „normale" Sorte von Knoten der zu parsenden Eingabegraphen (RSCJ, RSEC, RA, ...).
- **optional** Sorte von Knoten des Eingabegraphen, die für einen oder zwei Knoten des Eingabegraphen zusätzliche Informationen liefern (ISOF, ISOFD, TH, ...).
- **global** Sorte von Knoten des Eingabegraphen, die globale Information tragen, sich also auf den gesammten Graphen beziehen (WP, GTM). Ein Graph darf höchstens einen Knoten je global-Sorte enthalten, also z.B. höchstens einen WP- und höchstens einen GTM-Knoten.
- **preterminal** Sorte von Knoten des Eingabegraphen, die andere terminale oder preterminale Knoten des Eingabegraphen zusammenfassen (CPS, PS).
- **nonterminal** Sorte eines Knotens, der durch die Anwendung einer Regel der GG dem Graphen hinzugefügt worden ist, aber kein Zielknoten ist. Die Anwendung einer Regel faßt einen oder mehrere Knoten zusammen, und erzeugt daraus einen neuen Knoten (SHOULDER, CREST, TRUNNION, ...).
- **goal** Startsymbol der GG. Ein Parse ist ein Knoten, der alle Knoten des Eingabegraphen zusammenfaßt und dessen Sorte die Metasorte goal hat.
- **edge** Die Kanten des Eingabegraphen sind als „spezielle" Knoten repräsentiert, deren Sorte die Metasorte edge hat. Knoten mit einer edge-Sorte haben immer mindestens zwei Attribute, durch deren Werte die zwei terminalen oder preterminalen Knoten bezeichnet werden, die durch die Kante verbunden sind (NR).

include Die slot-defs der angegebenen Sorte werden in diese Sortendefinition übernommen. Damit können z.B. in Subsorten einfach die Slots der jeweiligen Übersorte einkopiert werden. Bei Änderungen an der Slotstruktur der Übersorte ändern sich dann automatische die Slots der entsprechenden Untersorten. Die Verwendung von include deklariert aber keine Sorten-Untersortenbeziehung, sondern erleichtert nur den Aufbau von Sortenhierarchien mit Vererbung. Die mittels include einkopierte Sorte muß bereits definiert sein, die Reihenfolge der Sortendefinitionen ist hier nicht egal.

subsort Aufbau einer Sortenhierarchie. Die aufgezählten Sorten sind Untersorten dieser Sorte. Dabei brauchen hier nicht rekursiv alle Untersorten der Sorte angegeben zu werden, sondern nur die direkten Nachfolger in der Sortenhierarchie. Bsp.: a<b<c und a<d<c. Dann reicht es, in der Sortendefinition von c (subsorts b d) anzugeben und in den Definitionen von b und d jeweils (subsorts a). Damit ist a auch automatisch Untersorte von c. Untersorten werden vom Parser wie Synonyme für ihre

entsprechenden Übersorten behandelt. Es werden aber keine Annahmen darüber gemacht, welche Mengenbeziehungen zwischen ihren Attributen und Rollen bestehen: Eine Untersorte muß nicht notwendigerweise alle Rollen- und Attribut-Slots haben, die ihre Übersorten haben.

slot-def Durch eine Slotdefinition wird erstens ein Attribut oder eine Rolle deklariert, d.h. Knoten dieser Sorte können, müssen aber nicht einen Wert für den deklarierten Slot haben. Das slot-label ist der Name des Attributs oder der Rolle, wie er innerhalb einer Regel, in Pfaden usw. benutzt wird. Innerhalb einer GG dürfen alle slot-label mehrfach vergeben werden, innerhalb einer Sortendefinition allerdings nur einmal. Zweitens wird durch eine Slotdefinition mittels slot-sort die Sorte möglicher Slotwerte festgelegt.

role-sort Die entsprechende Rolle darf nur mit Knoten dieser Sorte oder einer Untersorte davon besetzt werden.

attribute-sort Die Sorte des Attributes kodiert die Bedeutung, den Typ, die Semantik des Attributes. Folgende Werte werden von GraPaKL unterschieden:

attr Beliebiges atomares Attribut, z.B. ein Vektor, eine Zahl, ein String.

node Attribut, das einen Knoten des Eingabegraphen eindeutig bezeichnet, üblicherweise eine Nummer. Die ref-, ref-1, ref-2 und ref-list Attribute beziehen sich auf diese Bezeichnung. Ein Knoten darf daher nicht mehr als ein Attribut der Sorte node enthalten.

ref Attribut, z.B. in einem Knoten mit Metasorte optional, das den Bezugsknoten bezeichnet. Ein Attribut der Sorte ref wird immer als Bezeichner für einen Bezugsknoten interpretiert.

ref-1, ref-2 Treten immer paarweise auf und bezeichnen zwei Bezugsknoten. Beispielsweise haben Sorten mit Metasorte edge immer zwei Attribute mit Sorte ref-1 und ref-2, um die beiden Knoten zu bezeichnen, die durch die Kante verbunden werden. In TEC-REP hat jeder NR-Knoten zwei Attribute surface_number_1 und surface_number_2 mit Attribut-Sorten ref-1 und ref-2, NR hat die Metasorte edge.

ref-list Attribut, das eine Liste von Bezugsknoten bezeichnet (in TEC-REP z.B. Attribute wie planar_surfaces in CPS oder set_of_intersecting_surfaces in PS).

A.1.2 Regeln

Die Syntax für Regeln ist:

```
            <rule> ::= (<rule-sort> (<role>+))
            <role> ::= (<role-sort> <role-label> { (<constraint>+) } )
      <constraint> ::= <path-constraint> | <predicate>
<path-constraint> ::= (nr    <path> <path>) |
                      (ov    <path> <path>) |
                      (no-nr <path> <path>)
```

```
                <path> ::= (% <role-label>+) |
                           (% <role-label>* <attribute-label>) |
                           (% <role-label>* sort) |
                           (%-global <global-sort> <attribute-label>)
           <predicate> ::= <form>
                <form> ::= <path> | <assignment> |
                           <subsort-declaration> | <lisp-form>
          <assignment> ::= <attribute-assignment> | <role-assignment>
<attribute-assignment> ::= (:= <attribute-label> <form>)
     <role-assignment> ::= (== <role-label> <form>)
 <subsort-declaration> ::= (subsort <form>)
```

Dabei bedeuten die Komponenten in einzelnen folgendes:

rule-sort Die Sorte des Knotens der linken Seite der Regel, oder kurz: die Sorte der Regel. Die Metasorte von rule-sort muß nonterminal oder goal sein. Eine Sorte hat i.A. mehrere Regeln. Die Sortendefinition bündelt Informationen, die andernfalls in jeder Regel erneut aufgeführt würden, die Rollen- und Attributlabel haben in allen Regeln einer Sorte dieselbe Bedeutung, weil sie an derselben Stelle und nur einmal definiert sind.

role Spezifiziert einen Knoten der rechten Seite der Regel durch Angabe seiner Sorte, seiner Nachbarschaft im Graphen, Überlappung mit andern Knoten und durch zusätzliche Prädikate, die auf den Knoten-Attributen und -Rollen definiert sind. Die Reihenfolge, in der die Rollen innerhalb einer Regel spezifiziert werden, kodiert die Strategie, mit der der Parser die Regel zu instantieren versucht, d.h. nach einem Rollentäger für die erste angegebe Rolle wird zuerst gesucht.

path-constraint Entspricht den $C^{\approx}, C^{\not=}, C^{=}$ Constraints in ANLGGs, nr steht für neighborhood und ov für overlap, no-nr bezeichnet explizit verbotene Nachbarschaft, d.h. die durch die beiden Pfade referenzierten Knoten dürfen nicht benachbart sein.

path Pfade bezeichnen Rollen und Attribute in Rollenwerten und deren Komponenten. Ein Pfad, der mit dem Schlüsselwort sort endet, liefert die Sorte des Rollenwertes bzw. der bezeichneten Komponente. Pfade der Form (%-global <global-sort> <attribute-label>) liefern den bezeichneten Attributwert des Knotens mit der angegebenen global-Sorte. Undefinierte Pfade, d.h. Pfade, die Rollen oder Attribute bezeichnen, die im jeweiligen Bezugsknoten nicht besetzt sind, liefern nil.

predicate Prädikate entsprechen den C^{func} Constraints im ANLGG-Formalismus. Sie gelten als erfüllt, wenn sie nicht nil liefern, dabei gelten subsort-declaration, attribute- und role-assignment immer als erfüllt.

attribute-assignment Weist einem Attribut einen entsprechenden Wert zu. Der Wert kann aus den Rollen- und Attributwerten bereits gefundener Rollenwerte berechnet werden, oft mit Hilfe benutzerdefinierter Funktionen. Beispielsweise wird im Beispiel unten eine benutzerdefinierte Funktion area benutzt, um in ANGLE-Knoten ein Attribut area zu definieren, das die Gesammtoberfläche angibt.

role-assignment Weist einer Rolle einen Wert zu. Das angegebene role-label wird damit praktisch als Abkürzung für den jeweiligen Pfad definiert, nur sinnvoll, falls der Pfad definiert ist, d.h. falls die entsprechende Rolle bereits spezifiziert wurde.

subsort-declaration Spezialisiert die Regel-Sorte auf eine berechnete Untersorte. Angenommen, die Sorte ANGLE in Beispiel unten hätte eine Untersorte RA-ANGLE, für spezielle, nämlich rechtwinklige Winkel. Außerdem sei rectangular noch ein benutzerdefiniertes Prädikat, das testet, ob zwei gegebene RAs rechtwinklig aufeinanderstehen. Man kann dann die Regel für Winkel aus dem Beispiel unten durch Hinzunahme einer Subsorten-Deklaration so modifizieren, daß die Regel ANGLE- und RA-ANGLE erkennt, ohne Subsorten-Deklaration bräuchte man dafür zwei, fast gleiche Regeln.

```
(ANGLE
   ((RA big-face)
    (RA small-face
        ((nr (% big-face) (% small-face))
         (convex (% big-face) (% small-face))
         (<= (area (% small-face)) (area (% big-face)))

         (subsort (if (rectangular (% big-face) (% small-face))
                      'RA-ANGLE ;then
                      'ANGLE))  ;else

         (:= area (+ (area (% small-face)) (area (% big-face))))))))
```

lisp-form Ein beliebiger Lisp-Ausdruck, wie z.B. `(if ...)` und `(<= ...)` im obigen Beispiel. Neben allen Standard Common Lisp Funktionen sind folgende Prädikate für Vergleiche in der Sortenhierarchie verwendbar:

> `(subsort` $s1$ $s2$) Erfüllt, falls $s1$ Untersorte von $s2$ ist, d.h. wenn $s1 \leq_S s2$.
>
> `(samesort` $s1$ $s2$) Erfüllt, falls $s1$ in der Sortenhierarchie mit $s2$ vergleichbar ist, also wenn $s1 \leq_S s2$ oder $s2 \leq_S s1$.

Die Prädikate `nr`, `ov` und `no-nr` dürfen beliebig tief in andere Funktionen eingebettet sein, z.B. `(if (or (nr ...) (ov ...)) ...)`, solche eingebetteten path-constraints werden aber beim Test, ob die Rollen einer Regel zusammenhängend, also paarweise durch entsprechende path-constraints verbunden sind, nicht berücksichtigt.

Beim Laden einer GG wird getestet, ob ein entsprechendes .lisp File mit benutzerdefinierten Prädikaten und Funktionen (im Beispiel oben convex, area und rectangular) existiert und gegebenenfalls geladen.

A.1.3 Beispiel

Abb.A.1 zeigt die grafische Repräsentation einer ANLGG mit nur zwei Regeln. Hier die gleiche Grammatik im GraPaKL-Format:

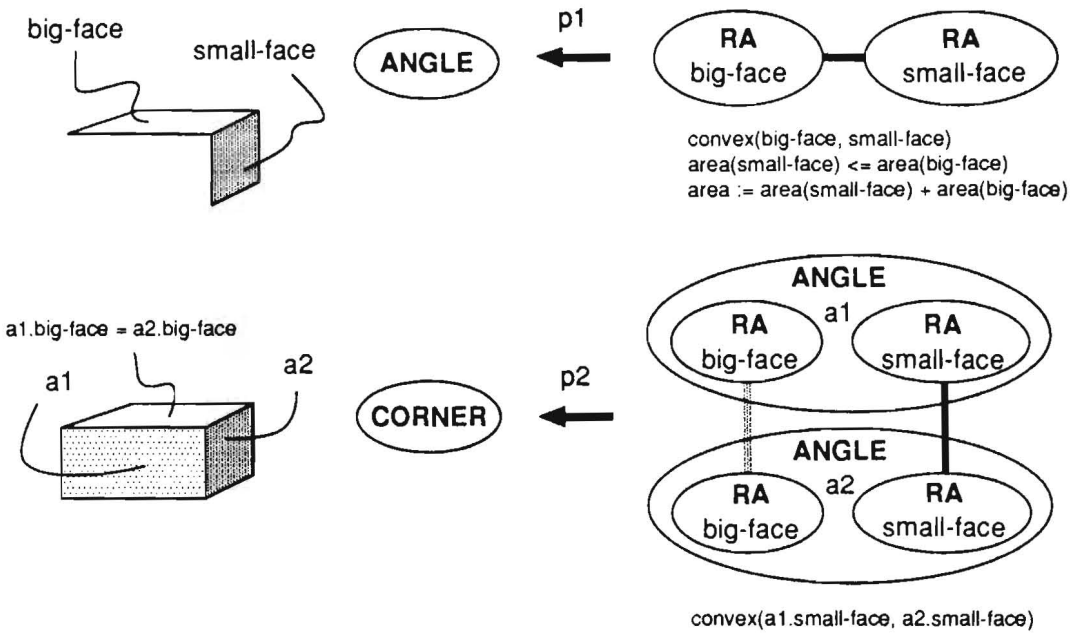


Abbildung A.1: Eine ANLGG mit zwei Regeln

```
(make-gg
 :name "corner"
 :default-graph "corner"
 :sorts '(

        ((edge   NR)     ;NeighboRhood, Kante
         (node . id)
        (ref-1 . surface_number1)
        (ref-2 . surface_number2))

    ((terminal   RA)     ;RectAngle, Rechteckflaeche
         (node . id)
         (attr . reference_point)
         (attr . direction_vector1)
         (attr . direction_vector2)
         (attr . direction_of_material))

 ((nonterminal   ANGLE)  ;Winkel, bestehend aus zwei RA
           (RA . big-face)
           (RA . small-face)
         (attr . area))

        ((goal   CORNER) ;Ecke, bestehend aus zwei ANGLE
        (ANGLE . a1)
        (ANGLE . a2)))

:rules '(

  (ANGLE
      ((RA big-face)
       (RA small-face
           ((nr (% big-face) (% small-face))
            (convex (% big-face) (% small-face))
            (<= (area (% small-face)) (area (% big-face)))
            (:= area (+ (area (% small-face)) (area (% big-face))))))))

  (CORNER
      ((ANGLE a1)
       (ANGLE a2
           ((nr (% a1 small-face) (% a2 small-face))
            (ov (% a1 big-face) (% a2 big-face))
            (convex (% a1 small-face) (% a2 small-face)))))))
```

A.1.4 Benutzerdefinierte Prädikate und Funktionen

Zur Formulierung der Constraints und Definition von Attributwerten können neben den vordefinierten Common Lisp Funktionen auch spezielle, vom Benutzer definierte Lisp-Funktionen verwendet werden. Die entsprechenden Funktionsdefinition einer GG **name.gg** müssen dabei in einem File **name.lisp** im selben Directory stehen.

Bei der Formulierung der Funktionsdefinitionen dürfen Pfade in der (% ...) Syntax nicht verwendet werden. Stattdessen werden zum Zugriff auf Rollen- und Attributwerte Zugriffsmacros verwendet, die beim Compilieren einer GG automatisch für die Slots aller Sorten erzeugt werden. Diese Macros haben für alle Sorten mit Ausnahme der optionalen und globalen Sorten die Form **(sort-slot v)**. Beispielsweise sind für Knoten der Sorte RA wie oben definiert nach dem Compilieren der GG die Zugriffsmacros **RA-id, RA-reference_point,...RA-direction_of_material** verfügbar. Angewandt auf einen Knoten der Sorte RA liefern sie den entsprechenden Attributwert, falls der Slot im Knoten existiert, nil sonst. Damit läßt sich z.B. die oben benutzte Funktion **area** definieren, die für Knoten der Sorte RA die Oberfläche der durch den Knoten repräsentierten Recheckfläche liefert:

```
(defun area (ra)
  (* (vec-abs (RA-direction_vector1 ra))
     (vec-abs (RA-direction_vector2 ra))))

(defun vec-abs (vector)
  (sqrt (+ (expt (first  vector) 2)
           (expt (second vector) 2)
           (expt (third  vector) 2))))
```

Für globale Sorten werden entsprechend nullstellige Macros erzeugt. Wenn z.B.

```
      ((global  WP)
        (node . id)
        (attr . name)
        (attr . info))
```

die Definition einer globalen Sorte in einer GG ist, dann werden beim Compilieren der GG nullstellige Zugriffsmacros **WP-id, WP-name, WP-info** erzeugt, die die entsprechenden Attributwerte im WP-Knoten eines Graphen liefern, falls der Graph einen WP-Knoten hat und in diesem ein entsprechendes Attribut definiert ist, nil sonst.

Für optionale Sorten werden entsprechende zweistellige Macros (sort-slot n v) definiert, die den Slotwert im n-ten Vorkommen der Option eines Knotens v liefern, $n \leq 0$. Beispiel: Seien in der obigen GG zusätzlich zu den Sorten NR,RA,ANGLE und CORNER die Sorten TH (wie thread, Gewinde) und DOUBLE-THREAD definiert:

```
  ((optional   TH) ;THread, Gewinde
       (node . id)
        (ref . surface_number)
       (attr . reference_point1)
       (attr . reference_point2)
       (attr . thread_name))

((nonterminal   DOUBLE-THREAD) ; RA mit zwei Gewinden
         (RA . face)
         (TH . thread-1)
         (TH . thread-2))
```

In einem Graphen können nun jedem RA-Knoten optionale TH-Knoten in beliebiger Anzahl zugeordnet sein, d.h.: jede Rechteckfläche kann beliebig viele Gewindebohrungen aufweisen. Die folgende Regel definiert dann das Feature „Rechteckfläche mit genau zwei M4-Gewinden":

```
(DOUBLE-THREAD
    ((RA face
        ((= 2 (length (% face TH)))   ;genau zwei Gewinde
         (equal "M4" (TH-thread_name 0 (% face)))
         (equal "M4" (TH-thread_name 1 (% face)))
         (== thread-1 (first (% face TH)))
         (== thread-2 (second (% face TH)))))))
```

A.1.5 Graphen

Die zu parsenden Eingabegraphen haben die folgende Syntax:

```
        <graph> ::= <node>*
         <node> ::= (node-sort <id> <attribute>*)
        <label> ::= terminal-sort | preterminal-sort |
                    optional-sort | edge-sort
           <id> ::= <number>
    <attribute> ::= (attribute-label attribute-value)
```

Dabei ist <id> eine innerhalb des Graphen nur einmal vergebene positive Zahl. Diese Zahlen numerieren die Knoten des Graphen beginnend mit Null. Die Anzahl der Knoten ist um eins größer als die größte vorkommende id. Die Reihenfolge der Knoten spielt keine Rolle, sie brauchen z.B. nicht nach id auf- oder absteigend sortiert sein.

In der Sortendefinition einer GG wird jeder terminalen-, preterminalen-, optionalen- und jeder edge-Sorte genau ein Attribut der Attributsorte node zugeordnet. Dieses Attribut liefert jeweils die id der Knoten eines Eingabegraphen.

Ergänzend werden für Graphen bezüglich einer gegebenen GG noch folgende Eigenschaften gefordert:

- Für alle Knotenlabel existiert in der GG eine entsprechende Sortendefinition.
- Die Attribute der Knoten sind eine (unechte) Teilmenge der Attribute, die in der zugehörigen Sortendefinition deklariert sind.
- Die Bezugsknoten für edge-Knoten existieren und sind terminal-Knoten.
- Die Bezugsknoten für optional- und preterminal-Knoten existieren und sind terminal- oder preterminal-Knoten.

Eine syntaktische Repräsentation des Graphen in Abb.A.2 ist zum Beispiel:

```
(RA 1
    (reference_point   (0 2 0))
    (direction_vector1 (0 0 2))
    (direction_vector2 (3 0 0))
    (direction_of_material +  ))
```

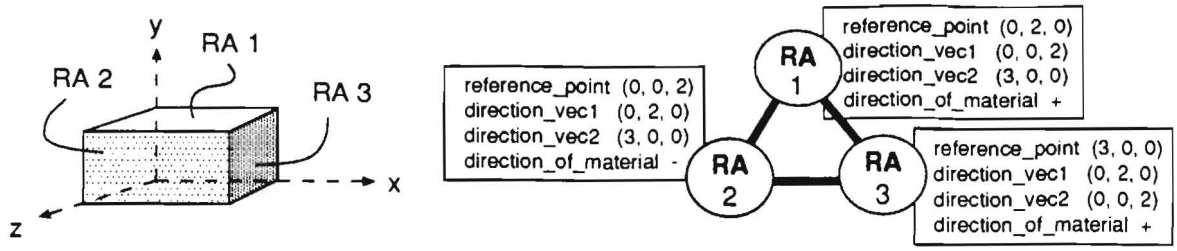


Abbildung A.2: Ein einfacher Eingabegraph

```
(RA 2
    (reference_point   (0 0 2))
    (direction_vector1 (0 2 0))
    (direction_vector2 (3 0 0))
    (direction_of_material -  ))

(RA 3
    (reference_point   (3 0 0))
    (direction_vector1 (0 2 0))
    (direction_vector2 (0 0 2))
    (direction_of_material +  ))

(NR 0
    (surface_number1 1)
    (surface_number2 2))

(NR 4
    (surface_number1 1)
    (surface_number2 3))

(NR 5
    (surface_number1 2)
    (surface_number2 3))
```

A.2 Grammatik Entwicklung mit GraPaKL

Nach dem Laden von GraPaKL stehen die folgenden Funktionen zur Verfügung

- `(help)` liefert eine kommentierte Liste der verfügbaren Funktionen.
- `(print-dirs)` liefert die Namen der verfügbaren Graphen und Grammatiken.
- `(print-env)` liefert eine kommentierte Liste globaler Variablen und deren momentanen Wert. Mit `(setq <environment-variable> <new-value>)` lassen sich diese Werte ändern. Aus diese Weise kann man beispielsweise die Suchstrategie des Parsers festlegen (Breitensuche, Tiefensuche, heuristische gesteuerte Suche usw.), Abbruchereignisse definieren (z.B. Abbruch nach Finden des ersten Parses) oder die Directories für Graphen und Grammatiken setzen.
- `(compile-gg name)` lädt und compiliert die Grammatik `name.gg`, dabei ist `name` ein String ohne Suffix `.gg`, `(compile-gg)` lädt und compiliert die zuletzt geladene GG noch einmal, nützlich um eine fehlerbereinigte GG erneut zu compilieren.
- `(parse graph)` Startet den Parser mit dem Graphen `graph` und der zuvor compilierten GG, dabei ist `graph` der Filename des Graphen, ein String, `(parse)` startet den Parser mit dem Default-Graphen der zuvor compilierten GG.

Die Aktionen des Parsers können mit dem Grafik-Tracer Schritt für Schritt, vorwärts und rückwärts verfolgt werden. GraPaKL's Compiler, der Parser und der Grafik-Tracer werden in den folgenden drei Abschnitten vorgestellt.

A.2.1 Der Compiler

Der Compiler lädt eine Grammatik, bindet dabei die gegebenfalls mit include angegebenen Grammatiken ein, übersetzt die resultierende Grammatik in eine speicherresidente Repräsentation, die auf die Bedürfnisse des Parsers zugeschnitten ist, und erzeugt Zugriffsmacros entsprechend den Slotdefinitionen der Sorten.

Beim Compilieren einer Grammatik können die folgenden Fehler und entsprechenden Fehlermeldungen auftreten:

- `compile-gg: Die Grammatik enthaelt keine einzige GOAL Regel (kein Startsymbol).`
  Der Parser kann für diese Grammatik keinen Parse erzeugen, weil keine der Regeln einen Zielknoten erzeugt. Abhilfe: Eine Regel definieren, deren Sorte die Metasorte goal hat.

- `compile-gg: Unzusammenhaengende rechte Seite, Rolle r hat weder Kanten noch Ueberlappung.`
  Die rechte Seite einer Regelseite muß zusammenhängen, d.h. jede Rolle muß über einen Nachbarschafts- oder einen Überlappungs-Constraint mit einer bereits spezifizierten Rolle verbunden sein. Abhilfe: Passenden nr- oder ov-constraint einfügen oder die Reihenfolge der Rollen innerhalb der Regel ändern.

- `compile-sorts: Include Sorte s ist bisher nicht definiert.`
  Versuch, in einer Sortendefinition mittels (include s) die Slots einer Sorte s einzufügen, die an dieser Stelle noch nicht definiert ist. Abhilfe: Undefinierte Sorte definieren, bzw. so umordnen, daß sie nicht hinter der includenden Sortendefinition steht.

- `compile-sorts: Unbekannte Meta-Sorte m in Sorte s.`
  Fehler im Head einer Sortendefinition, erlaubte Metasorten sind `terminal`, `optional`, `preterminal`, `nonterminal`, `goal`, `global` und `edge`. (Die Lisp-Konstante `=meta-sorts=` liefert die Liste gültiger Metasorten).

- `compile-gg-rule: Sorte s der Regel p undefiniert.`
  Die linke Seite der Regel hat eine Sorte, zu der keine Sortendefinition existiert. Abhilfe: Regelsorte ändern oder entsprechende Sortendefinition einfügen.

- `compile-gg-rule: Unbestimmbare Rollensorte fuer Rolle r in Regel p.`
  Die Sorte ist unbestimmbar, weil entweder in p's Sortendefinition ein Rollenslot für r fehlt, oder weil die dort angegebene Sorte nicht definiert ist.

- `compile-gg-rule: Def's Sorte s1 ungleich Sorte s2 von Rolle r in Regel p.`

Unterschiedliche Sortenangaben für Rolle r in Sortendefinition und der Regel p. Rollensorten werden in GraPaKL mehrmals angegeben, einmal in der Sortendefinition und je einmal bei der Spezifikation der Rolle in einer Regel. Dadurch wird die GG für Menschen leichter lesbar.

- `extract: Ungueltiger nr, ov oder no-nr Constraint c in Rolle r.`
  Die beiden Pfade eines Pfadconstraint müssen bestimmten Forderungen genügen: Der eine muß ein Pfad in die gerade spezifizierte Rolle r sein, der andere muß in eine der bereits spezifizierten Rolle reichen. (Erste Rollen können also keine Pfadconstraints haben.) Die Reihenfolge der beiden Pfade ist egal.

- `extract-predicates: Unbekanntes Attribut a in pred`
  Das Prädikat pred enthält ein attribute-assignment für ein Attribut a, aber ein solches Attribut ist in der Definition der Regelsorte nicht definiert. Abhilfe: Attribut a oder Sortendefinition ändern.

- `subsorts-transitive: Unbekannte Subsorte s.`
  Im Head einer der Sortendefinitionen wird s als Untersorte deklariert, ist aber weder vorher noch nachher definiert.

Die Funktion (`print-cgg`) liefert die compilierte GG in tabellarischer Darstellung, auch für unvollständig compilierte GGs. Damit kann bei Fehlermeldungen des Compilers festgestellt werden, bis wohin die Compilierung geklappt hat und beim Compilieren welcher Regel und Rolle der Fehler aufgetreten ist.

A.2.2 Der Parser

Der Parser analysiert mit der zuletzt compilierten, speicherresidenten GG den angegebenen Graphen. Falls kein Graph angegeben ist, wird der Default-Graph der GG geparst.

Falls der zu parsende Graph mit der GG nicht kompatibel ist, werden beim Einlesen zu Beginn des Parsens entsprechende Fehlermeldungen erzeugt:

- `graph-to-agenda: Unerlaubte Sorte s von Knoten v.`
  Der eingelesene Knoten hat eine unbekannte Sorte oder eine Sorte mit Metasorte nonterminal oder goal.

- `node-to-patch: Ungueltige Knotenstruktur in v.`
  Ungültige Listenstruktur, der Knoten verletzt die für Graph-Knoten definierte Syntax.

- `node-to-patch: Zu viele Slots (...) in Knoten v.`
  Der Knoten hat mindestes ein Attribut mehr, als in seiner Sortendefinition deklariert.

- `add-to-global: Globaler s-Knoten existiert bereits.`
  Der Graph enthält unzulässigerweise mindestens zwei Knoten derselben global-Sorte.

- `add-option: Fehlender Knoten fuer Option v.`
  Ein Knoten mit optional-Sorte bezieht sich auf einen Knoten, der im Graphen nicht vorkommt.

- `add-edge: Fehlender Knoten fuer Kante e.`
  Ein Knoten mit edge-Sorte behauptet, zwei Knoten zu verbinden, aber mindestens einer davon kommt im Graphen nicht vor.

- `fill-ref: Preterminal's Referenzknoten v existiert nicht.`
  Ein Knoten mit preterminal-Sorte bezieht sich auf einen Knoten, den es im Graphen nicht gibt.

Der Algorithmus

Hier wird ganz kurz der Parsing Algorithmus beschrieben, wie er sich aus der Sicht des Grammatik-Entwicklers darstellt: Ein *Patch* ist eine teilweise (Hypothese, partielles patch, pp) oder vollständige (Fakt, complete patch, cp) Instanz einer Regel. Es enthält an Informationen die Regel, deren Instanz es ist, die Sorte der Regel und die Liste der bereits gefundenen Rollenwerte. Rollenwerte und die Knoten des zu parsenden Graphen sind immer vollständige Patches.

Der Parser arbeitet mit zwei Mengen von Patches: Agenda und Chart. Die Agenda wird mit einem cp je Knoten des Eingabegraphen initialisiert, der Chart ist anfangs leer. Es werden sechs Aktionen des Parsers unterschieden:

`CHOOSE` Holt das Patch mit der höchsten heuristischen Priorität aus der Agenda, bzw. bei Breitensuche das am längsten wartende und bei Tiefensuche das jeweils zuletzt erzeugte Patch.

`PROPOSE` Kombiniert ein aus der Agenda geholtes cp mit allen Regeln der GG, in denen es die erste Rolle besetzen kann, erzeugt für jede solche Kombination ein neues Patch.

`CONT-CP` Kombiniert das geholte cp mit allen pp des Charts, in denen es die nächste, d.h. erste unbesetzte Rolle besetzen kann, erzeugt für jede solche Kombination ein neues Patch.

`CONT-PP` Kombiniert ein aus der Agenda geholtes pp mit allen cp des Charts, die darin die nächste Rolle besetzen können, erzeugt für jede dieser Kombinationen eine entsprechendes Patch.

`TO-AGENDA` Fügt ein von `PROPOSE`, `CONT-CP` oder `CONT-PP` erzeugtes Patch in die Agenda ein.

`TO-CHART` Fügt ein mit `CHOOSE` aus der Agenda geholtes Patch in den Chart ein, nachdem via `PROPOSE`, `CONT-CP` oder `CONT-PP` alle möglichen Kombinationen mit den Patches des Charts erzeugt und in die Agenda eingefügt worden sind.

`TO-PARSES` Fügt ein Patch, daß den ganzen Eingabegraphen in seinen Rollenwerten zusammenfaßt und eine goal-Sorte hat, in die Liste der Parse ein.

Nachdem die Agenda via `TO-AGENDA` mit den Knoten des Eingabegraphen initialisiert worden ist, kreist der Parser also in folgender Schleife:

```
loop
    CHOOSE patch
    IF patch is complete
       then
           PROPOSE patch
           TO-AGENDA successors
           CONT-CP
           TO-AGENDA successors
       else
           CONT-PP patch
           TO-AGENDA successors
    TO-CHART patch
until agenda is empty
```

Spätestens, wenn die Agenda leer ist, sind alle Parse gefunden. In diesem Fall wären aber alle kombinatorisch überhaupt möglichen Patches erzeugt worden. Um das zu vermeiden und die Suche nach einem ersten Parse abzukürzen, können vom Grammatik-Entwickler Heuristiken definiert werden. Das ist im nächsten Abschnitt beschrieben.

Benutzerdefinierte Heuristiken

Der Parser arbeitet, wenn keine andere Strategie angegeben wird, mit Breitensuche, d.h. `CHOOSE` wählt immer das am längsten wartende Patch aus der Agenda aus. Mit `print-env` und `(setq *search-opt* h)` kann man dem Parser eine Heuristik-Funktion `h` übergeben, h ist dabei der Name einer einstelligen Lisp Funktion und liefert die heuristische Priorität eines gegebenen Patches. Die Priorität ist eine beliebige positive oder negative Zahl. `CHOOSE` wählt jeweils das Patch mit der größten Priorität. Die Heuristik-Funktion einer GG `name.gg` kann z.B. im File `name.lisp` definiert sein. Sie wird dann beim Laden der GG automatisch mitgeladen.

Die Heuristik-Funktion kann zur Berechnung der Priorität die Regel, die Sorte oder die Rollen- und Attributwerte des Patches heranziehen oder eine gewichtete Kombination von allem. Die Funktionen `patch--sort` und `patch--rule` liefern die Sorte und die Regel, deren Instanz das Patch ist. Als Beispiel hier die Definition einer `random-heuristic` genannten Heuristik-Funktion:

```
(defun random-heuristic (patch)
  (if (patch--rule patch)
      (random 1000)
      (values 1000)))
```

Die Funktion ordnet jedem Patch, das einen Knoten des Eingabegraphen repräsentiert (d.h. nicht Instanz einer Regel ist, also `patch--rule` liefert `nil`), die höchste Priorität 1000 zu. Für jedes andere Patch liefert sie einen Zufallswert zwischen 0 und 999. Diese Heuristik ist übrigens erstaunlich wirkungsvoll und markiert die Mindestleistung, die eine maßgeschneiderte, domänenspezifische Heuristik erreichen sollte.

Die Rollen- und Attributwerte eines Patches sind innerhalb einer Heuristik-Funktion erreichbar, wie im Abschnitt über benutzerdefinierte Funktionen und Prädikate erklärt.

A.2.3 Fehlersuche mit dem Grafik-Tracer

Der Grafik-Tracer dient dem Anzeigen gefundener Parse und der Fehlersuche in GGs. Abb.A.3 zeigt das Layout der Fenster des Tracers.

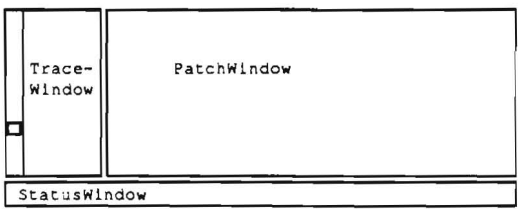


Abbildung A.3: Layout des Grafik-Tracers

Alle drei Fenster sind Maus-sensitiv. Im Statusfenster werden immer die an der aktuellen Mausposition möglichen Aktionen angezeigt. Die Fenster erfüllen im einzelnen folgende Aufgaben:

TraceWindow Zeigt einen Ausschnitt des Protokolls des jeweils letzten Parserlaufs. Protokolliert sind alle Aktionen des Parsers (siehe voriger Abschnitt) mit Ausnahme von `CHOOSE`. Auf das Protokollieren von `CHOOSE` kann verzichtet werden, weil darauf immer `PROPOSE` oder `CONT-PP` folgt und man an deren Argument sehen kann, welches Patch aus der Agenda geholt wurde.

Das kleine Quadrat am linken Rand zeigt die relative Position des sichtbaren Protokollausschnitts im gesamten Protokoll an. Klicken an eine Stelle am linken Rand verschiebt diesen Ausschnitt entsprechend. Um ihn in kleinen Schritten zu verschieben, kann man auch eine der angezeigten Aktionen anklicken. Sie rollt dann an die mit Querstrichen markierte Zeile des Auschnitts und wird neue aktive Aktion.

PatchWindow Zeigt das Argument der jeweils aktiven Aktion des TraceWindows, also ein Patch. Nur für `TO-AGENDA` Aktionen wird das Patch sofort als Baum dargestellt, d.h. rekursiv mit allen seinen Rollenwerten und Rollenwerten von Rollenwerten usw. Für alle anderen Aktionen wird zunächst nur die Wurzel des Baumes gezeigt. Anklicken des Wurzelknotens expandiert dann den zugehörigen Baum. Von den Kindern eines Knotens werden entweder alle oder, falls der Platz dazu nicht reicht, keins angezeigt. In diesem Fall wird für jedes fehlende Kind ein Kantenende an die Unterseite des Vaterknotens gezeichnet. Anklicken eines beliebigen Knotens des Baumes scrollt ihn nach oben und macht ihn zur Wurzel des neuen Baumes. Anklicken der aktiven Aktion im TraceWindow holt die ursprüngliche Wurzel zurück. Anklicken eines Knotens mit der rechten Maustaste öffnet ein kleines Fenster mit dessen Attributen und Attributwerten. Anklicken eines Attributfensters verschiebt oder schließt es wieder.

Die Knoten sind im PatchWindow in Blockschrift mit ihrer Sorte, in Kleinschrift mit ihrem Rollenlabel und mit der Patchnummer beschriftet. Die Nummer ist für

vollständige Patches (z.B. =123=) markiert, für partielle Patches dagegen nicht (z.B. 123).

StatusWindow Zeigt die Namen der geladenen GG und des Graphen, mögliche Mausaktionen und die Anzahl der Patches in Agenda und Chart zu dem Zeitpunkt an, der im TraceWindow gewählt wurde. Klicken der Maus im StatusWindow erzeugt ein Auswahlmenu:

`exit Graphics` Schließt den Grafik-Tracer. Neustart mit `(open-graphic)`.

`find TO-AGENDA` Springt an die Stelle im Parser Protokoll, an der das Patch, das momentan Wurzelknoten im PatchWindow ist, erzeugt und in die Agenda aufgenommen wurde.

`find TO-CHART` Springt entsprechend an die Stelle, an der das Patch aus der Agenda in den Chart gewechselt ist. An dieser Stelle hat es mit `PROPOSE`, `CONT-CP` oder `CONT-PP` gegebenenfalls Nachfolger erzeugt.

Die Benutzung der beiden letzten Funktionen setzt das Verstehen des Parser-Algs voraus, wie er im letzten Abschnitt beschrieben wurde. Sie sind dann sehr nützlich, um festzustellen, warum bestimmte Regeln nicht instantiert oder Parse nicht gefunden wurden.

Weitere Funktionen zur Fehlersuche

Zur Suche von Fehlern in GGs stehen zusätzlich folgende Funktionen zu Verfügung:

`(print-rule rule)` Liefert den Quelltext der angegebenen Regel `rule` einer GG, `rule` ist dabei der Nummer der Regel, wie sie z.B. in der mit `(print-cgg)` erzeugten Tabelle benutzt wird.

`(print-population)` Liefert eine Statistik der Patches des Charts, aufgeschlüsselt nach Sorten.

`(chart-sort-instances sort cp-pp &optional components)` Liefert die Liste der Patches des Charts der angegebenen Sorte, `cp-pp` = `=cp=` liefert nur vollständige, `=pp=` liefert nur partielle Patches. Mit `components` kann optional eine Liste mit Patches angegeben werden. Es werden dann nur die Patches geliefert, in denen alle angegebenen Komponenten in beliebiger Tiefe enthalten sind.

`(chart-rule-instances rule cp-pp &optional components)` Wie oben, liefert aber nur Instanzen der angegebenen Regel.

`(draw-tree patch)` Zeichnet das angegebene Patch ins PatchWindow. Dabei ist Patch die Nummer eines Patches, wie sie z.B. von `chart-sort-instances` geliefert wird.

Die Abb.A.4 liefert einen Eindruck von GraPaKL's Benutzeroberfläche: Das PatchWindow zeigt einen von zwei Parsen, die für die Antriebswelle-M2 mit der turning.gg gefunden wurden. Im TraceWindow sind die letzten zwölf Aktionen des Parsers zu sehen, bevor er mit leerer Agenda terminierte. Für vier Knoten des Feature-Baums wurden

durch Anklicken Attributfenster geöffnet. Der Knoten middlestep$_{179}$ ist aus Platzgründen ohne seine Kinder dargestellt. Anklicken des Knotens würde ihn zur Wurzel eines neuen Feature-Baums machen, der dann auch die fehlenden Kinder enthält.

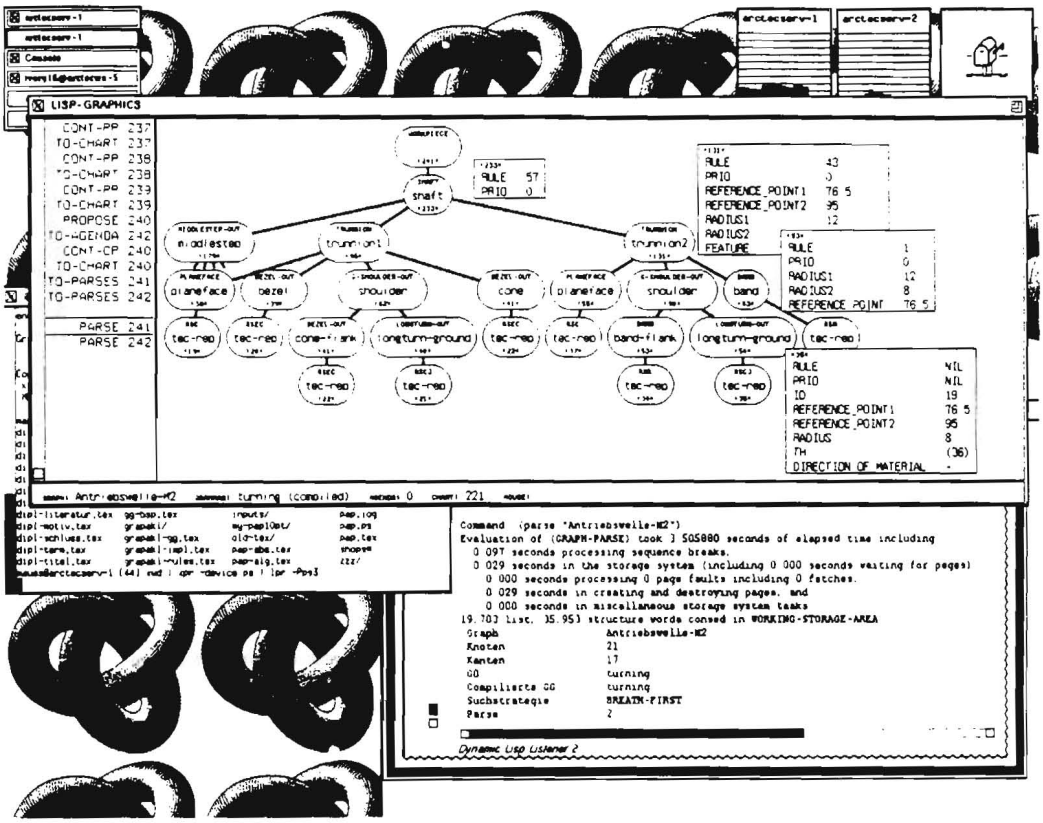


Abbildung A.4: GraPaKL's Benutzeroberfläche

Anhang B

GraPaKL Implementierung

GraPaKL wurde in Common Lisp implementiert und umfaßt alles zusammen ca. 3000 Zeilen kommentierten Quelltext.

B.1 Die Module im Überblick

Die folgenden Seiten geben einen kurzen Überblick über die einzelnen Module. Alle Datenstrukturen und fast alle Funktionen sind ausführlich im Quelltext kommentiert.

Modul	Kurzbeschreibung
stru	Datenstrukturen: `struct patch, sort, gg, cgg, rule, role, graph.`
mcro	Zugriffsfunktionen: für `patch, rule, role` und `sort.`
vars	Globale Variablen: `*patches*, *agenda*, *chart*, *cgg*, *graph*` ...
attr	Attributauswertung: Auswertung funktionaler Constraints.
comp	Grammatik Compiler: Umsetzung der ANLGG in die interne Repräsentation.
heur	Heuristiken: Prioritätsfunktionen für Patches der Agenda ($<_A$).
node	Umwandlung Graphknoten in Patch: Umwandlung eines TEC-REP Graphen.
pars	Chart Parser: Der heuristisch gesteuerte Parser für compilierte ANLGGs.
graf	Grafik Tracer: Patch-Browser und interaktive Fehlersuche in ANLGGs.
main	Ein- und Ausgabe: Benutzeroberfläche und Filehandling.

Jedes Modul entspricht dabei einem .lisp File. Die in den Modulen definierten Funktionen werden hier nicht nochmals dokumentiert, siehe dazu den Quelltext, sondern nur die Aufgaben der Module gegeneinander abgegrenzt, deren Zusammenspiel skizziert und die Namen der wichtigsten Funktionen genannt.

stru Definiert die folgenden Lisp Stukturen (`defstruct` ...):

- `patch` partielle (Hypothese) oder vollständige (Fakt) Instanz einer Produktion.
- `gg` ANLGG im GraPaKL Graph Grammatik Format (siehe Anhang A).
- `cgg` compilierte GG in GraPaKL's interner, tabellarischer Repräsentation.
- `sort` compilierte Sortendefinition.
- `rule` compilierte ANLGG Produktion.
- `role` compilierte ANLGG Rollenspezifikation (R-Spec).
- `graph` TEC-REP Repräsentation des zu parsenden Graphen.

Alle Datenstrukturen sind im Quelltext ausführlich dokumentiert. Patches sind die zentrale Datenstruktur des Parsers. (Sie entsprechen den dags im D-PATR.) Ein `patch` (cp oder pp) ist ein Lispstruct mit folgenden slots:

`rule`	Regel, dessen Instanz das Patch ist.
`sort`	Sorte des Patches.
`prio`	heuristische Priorität (für $<_A$).
`next-role`	nächste zu besetzende Rolle, nil gdw. das Patch ein cp ist.
`cast`	A-Liste der gefundenen Rollen- und berechneten Attributwerte.
`tps`	Liste der Bindepunkte des Patches (tie point set).

Alle erzeugten Patches werden im globalen Array `*patches*` gehalten. Der cast eines Patches ist eine Assoziationsliste, die dessen Rollen und Attribute mit den entsprechenden Attribut- und Rollenwerten assoziert. Rollenwerte sind immer vollständige Patches und werden durch ihren Index im Array `*patches*` repräsentiert. Attributwerte sind immer Listen mit genau einem Element.

mcro Definiert zusätzliche Zugriffsmacros für struct `patch`, `rule`, `role` und `sort`. Die Macros erhöhen die Lesbarkeit des Quelltextes und verbergen die Tatsache, daß die entsprechenden structe in globalen Arrays verwaltet werden.

vars Definiert alle globalen Variablen, die wichtigsten sind:

`*patches*`	dynamisch wachsender, eindimensionaler Array für Patches.
`*agenda*`	mit `patch.prio` heuristisch sortierte Liste von Patches.
`*chart*`	(n 2 2) stelliger Array, n = Zahl der Kanten im Eingabegraphen.
`*cp-chart*`	Hasharray, realisiert $CP_{ov}(cp^*)$.
`*pp-chart*`	Hasharray, realisiert $PP_{ov}(cp^*)$.
`*cgg*`	Compilierte GG: Arrays mit Sorten, Regeln und R-Specs.
`*options*`	Liste der Parser-Optionen.
`*graphic*`	t, falls das graf-Modul (Grafik-Tracer) geladen wurde.
`*parslist*`	Liste gefundener Parse

Durch Setzen der in `*options*` enthaltenen Variablen auf entsprechende Werte lassen sich z.B. die Suchstrategie (Heuristik $<_A$) des Parsers wählen, Abbruchereignisse definieren, z.B. bei Finden des n-ten Parses, Erzeugen des n-ten Patches usw. Die Funktion `print-env` liefert die aktuellen Einstellungen.

attr Dient dem automatischen Erzeugen von Zugriffsmacros und dem Auswerten funktionaler Constraints. Beim Compilieren der Sortendefinitionen einer Grammatik wird für jeden in einer Sortendefinition deklarierten Slot mit `def-sort-access-macros` ein entsprechendes Zugriffsmacro erzeugt. Mit diesen Macros können benutzerdefinierte Funktionen und Prädikate auf die entsprechenden Slotwerte in einem Patch zugreifen. Zweitens werden mit `eval-predicate-p` die durch eine Rollenspezifikation gegebenen funktionalen Constraints (C^{func} in ANLGGs) ausgewertet. Als Seiteneffekt sind im cast des Patches anschließend ggf. neue Attribut und Rollen definiert.

comp Compiliert mit `compile-gg`, `compile-sorts`, `compile-gg-rule` eine Grammatik im GraPaKL GG-Format, d.h. erzeugt eine speicherresidente, tabellarische Darstellung, die ganz auf den Parsevorgang zugeschnitten ist.

heur Dieses Modul enthält zwei exemplarische Heuristik-Funktionen zum heuristischen Ordnen der Patches in der Agenda.

node Enthält Funktionen zur Umwandlung von Knoten eines TEC-REP Eingabegraphen in ein vollständiges Patch, wichtigste Funktion ist `node-to-patch`. Alle erzeugten Patches werden ins Array `*patches*` eingetragen.

pars Der eigentliche Chart-Parser für ANLGGs. Die Bezeichnungen der Funktionen entsprechen weitgehend den in Kapitel 3 verwendeten. Wichtige Funktionen: `parse` und `parse-graph`, davon gerufen werden: `add-to-agenda` zum sortierten Eintragen in die Agenda (ruft das heur-Modul zum Sortieren), `propose`, `start-patch` zum Verschmelzen einer Regel mit einem cp, `continue-pp`, `continue-cp`, `merge-patches` zum Verschmelzen von einem pp mit einem cp. Dabei ruft `merge-patches` das attr-Modul, um die Constraints für pp's nächste Rolle zu evaluieren. `add-to-chart` trägt ein Patch in die entsprechenden Arrays des Charts ein, `add-to-parslist` trägt ein Patch in die Liste `*parslist*` der gefundenen Parse ein.

graf Dieses Modul enthält alle Funktionen zur Realisierung des Grafik-Tracers. Das Modul setzt auf den Grafik-Funktionen des Window-Toolkits von Andreas Becker auf. Alle anderen Module sind unabhängig vom graf-Modul lauffähig. Dazu muß die globale Variable `*graphic*` auf nil gesetzt werden. Diese Entkopplung erleichtert die schnelle Portierung in Common Lisp Umgebungen, in denen das Window-Toolkit nicht verfügbar ist.

main Dieses Modul realisiert die Schnittstelle zum Benutzer, wichtige Funktionen sind `help`, `print-dirs`, `print-status`, `print-population` usw. Außerdem finden sich hier die Funktionen zum Lesen von Graphen und Grammatiken `load-struct-graph`, `load-struct-gg` und zum Schreiben eines Parses in ein File `print-feature-tree`. Die globalen Variablen `*graph-pathname*` und `*gg-pathname*` geben die Pfade zu den beiden Directories für Graphen und Grammatiken an und sollten im Ladeprogram (siehe z.B. `grapakl-loader.lisp`) entsprechend belegt werden.

Anhang C

A Heuristic Driven Parser ... Papier für die ISAI '92 in México

Auf den folgenden Seiten wird ein Papier reproduziert, das zusammen mit Christoph Klauck geschrieben und im April 92 bei der *ISAI '92, 5th International Symposium on Artificial Intelligence on Applications in Manufacturing and Robotics* eingereicht wurde. Die ISAI '92 findet vom 7. bis 11. Dezember 1992 in Cancún, México statt.

Das Papier faßt die Arbeiten zusammen, die in [BKL 91b] begonnen und mit dieser Arbeit fortgesetzt wurden.

A Heuristic Driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM

Christoph Klauck and Jakob Mauss

May 4, 1992

Abstract

To integrate CA*-systems with other applications in the CIM world, one principal approach currently under development is the feature recognition process based on graph grammars. It enables any CIM component to recognize the higher-level entities - the so-called *features* - used in this component out of a lower-data exchange format, which might be the internal representation of a CAD system as well as some standard data exchange format. In this paper we present a 'made-to-measure' parsing algorithm for feature recognition. The heuristic driven chart based bottom up parser analyzes attributed node labeled graphs (representing workpieces) with a (feature-)specific attributed node labeled graph grammar (representing the feature definitions) yielding a high level (qualitative) description of the workpiece in terms of features.

1 Motivation

Research in feature-based CA*-systems like CAD (Computer Aided Design), CAPP (Computer Aided Process Planning) or CAM (Computer Aided Manufacturing), has been motivated by the realization that geometric models represent a workpiece in greater detail than can be utilized e.g. by a designer, strength calculator or process planner. When CA*-experts view a workpiece, they perceive it in terms of their own expertise. These terms, the so-called *features*, which are build upon a *syntax* (shape description: topological graph containing geometry and technology) and a *semantics* (description of related informations, e.g. skeletal plans in manufacturing or functional relations in design), provide an abstraction mechanism to facilitate e.g. the creation, manufacturing and analysis of workpieces. Features that are required e.g. for design may differ considerably from those required e.g. for manufacturing or assembly, even though they may be based on the same geometric and technological entities (cf. [Fi 90]).

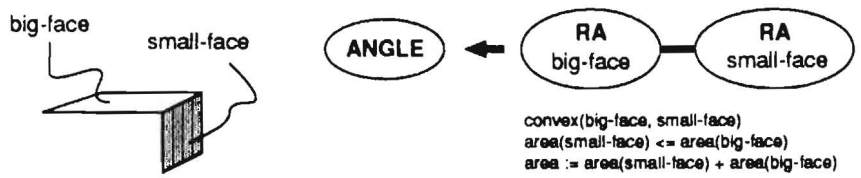


Figure 1: A sample feature definition

So representing features and recognizing them out of the (lower-level) workpiece description is a necessary step to bridge the gap between the several CA*-systems and an important step towards truly Computer Integrated Manufacturing (CIM). The expected advantages of a close coupling of CA*-systems are: The information interchange shall lead to better knowledge transfer, to shorter turnaround times and to improved feedback. In the end, higher flexibility and generally better results are expected.

In current research one method to represent and recognize features is based on graph grammars (cf. [Fi 90, Mu 91, Ri 91]). This area is a well established field of research and provides a powerful set of methods like parsing and knowledge about problems, their complexity and how they could be solved

efficiently. The use of graph grammars for feature descriptions facilitates the application of these results to the area of feature recognition. So in consideration of the feature characteristics 'made-to-measure' tools must be developed to make the recognition and representation process very efficient.

Out of this point of view we present in this paper a heuristic driven chart based parser for parsing attributed node labeled graphs representing workpieces with attributed node labeled graph grammars representing the feature definitions yielding a high level (qualitative) description of the workpiece in terms of the so-called features to support e.g. a feature based CAPP-system (cf. [Ch 90a, Ch 90b]). The nodes of a workpiece graph represent geometric primitive surfaces, the node label decode the type of the surface, the attributes carry detailed geometric and technologic information and the edges decode the topology of the workpiece, i.e. two nodes are adjacent if the corresponding surfaces touch each other.

In figure 1 a sample feature definition and its associated grammar rule is represented. An ANGLE consists of two connected atomic features (surfaces), called big-face and small-face within the ANGLE feature. Several constraints are specified, like *convex(big-face, small-face)* and a function is used to define the ANGLE attribute area.

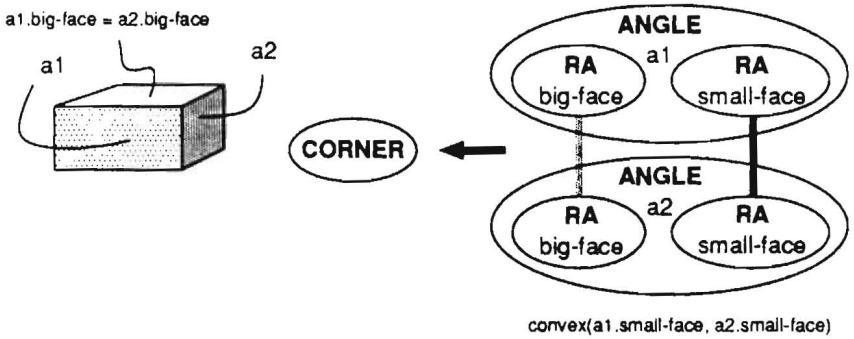


Figure 2: A feature definition with a deep connection and an overlap

In figure 2 the (complex) feature CORNER is defined by two overlapping ANGLE features. The two ANGLEs (a1 and a2) overlap with their big-face, say they share the same surface as big-face. Additional their small-faces must be in neighborhoodness. The grammar formalism introduced in the next section enables us to express three kinds of deep relations between the components of nodes. (black: neighborhoodness, black/cross: no neighborhoodness, grey: overlapping)

To become more closure with the effect of feature characteristics to our developed parsing algorithm, we briefly introduce now the term feature and the most important characteristics of its definitions. Detail description and the analogue to graph grammars can be found e.g. in [Kl 91].

We define the term **feature** as a description element based on geometrical and technological data of a product which an expert in a domain associates with certain informations.

In [Kl 91] several syntactical characteristics of features have been outlined.

1. *Interaction:* Areas of features can overlap. This must be taken into account by the parser.
2. *Dependence of Dimensions:* In dependence of the dimensions, the same structures may be identified as different features. This leads to several constraints included in the feature definition.
3. *Hierarchy:* A complete feature description of a workpiece forms a hierarchical structure of features: the result of our parser.
4. *Qualitative Description:* To describe a feature an expert uses only less geometrical and technological informations; he uses a qualitative description.
5. *Ambiguity:* A feature can often be derived in many syntactic different but semantic equivalent ways. We are interested in only one pregnant description of the workpiece.
6. *Similar Features:* Features differ often only in details. A terse formulation and an efficient treatment of variants and special cases is desired.

These characteristics lead us to the 'made-to-measure' grammar formalism, called ANLGG (Attributed Node Labeled Graph Grammar) introduced in the next section.

2 Attributed Node Labeled Graph Grammars

In this section we will briefly define the terminology of attributed node labeled graph grammars as used in this paper, shortly called ANLGG. Surveys and detailed introduction to graph grammars can be found e.g. in [Eh 79-90].

A **node** in our formalism ANLGG is a structure with two kinds of slots called **role** and **attribute**. A node can be seen as a partial function mapping slots to their corresponding values. An attribute value is an arbitrary atomic object, a role value is always a node. Let **cast**(v) be the set of all role values of the node v, and cast*(v) be the transitive closure of the function cast, i.e. the set of all role values of v and role values of role values of v and so on, including v. The nodes in cast(v) are assigned to v by an application of a graph grammar production as explained later in this section. There may be edges between the nodes in cast(v); therefor the node v can also be seen as a graph. The nodes in cast*(v) are called **components** of v and represent a hierarchy of features as mentioned in the previous section with v serving as root-node. The leaves (atomic features) of the hierarchy are given by the nodes of a terminal graph (representing e.g. a workpiece).

A **path** in a role r_0 is a finite sequence $\pi = [r_0 \ldots r_n]$ of roles and $v(\pi)$ denotes the component v_n of v where $v_0 := v(r_0), v_1 := v_0(r_1) \ldots v_n := v_{n-1}(r_n)$. The last element in π may be an attribute, in this case $v(\pi)$ denotes the corresponding attribute value. Occasionally we use a superscript notation like in v^* to indicate, that v^* is a component of v. For a given set V of nodes let V^* denote the union $\bigcup_{v \in V} \text{cast}^*(v)$.

In our paper the term (feature-)graph means an attributed finite undirected node labeled graph, in the sequel shortly called **graph**.

Such a graph g is formally given as a 4-tuple $g := (V, E, S, \varphi)$ where

V	is a finite set of nodes, $\|V\|$ is the number of nodes in g.
E	$\subseteq V^* \times V^*$ is a set of undirected edges
S	is a finite nonempty alphabet of node labels
φ	is a labeling function $\varphi : V^* \rightarrow S$.

For a given node v the label $\varphi(v)$ is called the **sort** of v. Note, that E and φ are defined for $V^* \supseteq V$, so the nodes itself possess graph structure. A **terminal graph** is a graph $g_0 = (V_0, E_0, S, \varphi_0)$ where $\forall v \in V_0 : \text{cast}(v) = \emptyset$ and metasort $(\varphi(v)) = terminal$.

The sorts in S are structured by a partial order $<_S$, the **sort hierarchy**, a subsumption hierarchy. If $s_1 <_S s_2$ we say s_1 is a **subsort** of s_2. The parser will treat any instance of s_1 like an instance of s_2, but not vice versa. Sort hierarchies are used here in order to reduce the number of productions needed to describe the domain.

In our paper an attributed node labeled (feature) **graph grammar**, or ANLGG for short, is a 5-tuple $gg := (S, P, R, \text{sort}, \text{metasort})$ where

S	is the same alphabet as above. S is structured by the sort hierarchy $<_S$.
P	is a finite set of productions
R	is the finite set of all role specifications of productions in P
sort	is a total function sort: $P \cup R \rightarrow S$ and
metasort	is a total function metasort: $S \rightarrow \{terminal, nonterminal, goal\}$

A **production** is a finite, nonempty set $p = \{r\text{-}spec_1, \ldots, r\text{-}spec_n\}$ of role specifications, defining the right hand side of p. Each role specification $\in p$ specifies a node such that the whole p specifies a connected graph with certain relational and functional constraints holding between the nodes. The specified graph may be replaced by a single node v, i.e. the production is applied to a graph g_n leading to a reduced graph g_{n+1} by

- identifying n nodes $v_i \in V_n^*$ with $\varphi(v_i) \leq_S \text{sort}(r\text{-}spec_i)$, that satisfy the relational and functional constraints given by $r\text{-}spec_i$, $1 \leq i \leq n$

- building V_{n+1} by removing these nodes from V_n and adding a new node v with $\varphi(v) \leq_S \text{sort}(p)$, that has one role r_i for each *r-spec*$_i$ with the corresponding removed node v_i serving as role value, i.e. $v(r_i) = v_i, \quad 1 \leq i \leq n$ and $V_{n+1} = \{v\} \cup V_n \setminus \text{cast}(v)$. v represents an instance of the left hand side of p.

Additional we need an embedding rule that specifies, what happens to the edges of the removed nodes $\{v_1, \ldots, v_n\} = \text{cast}(v)$ and how the added node v is connected to the remaining nodes $V_{n+1} \setminus \{v\}$ of the new graph g_{n+1}. We define:

- $E_{n+1} = E_n \cup \{(v, v_j^*) \mid \exists (v^*, v_j^*) \in E_n : \quad v^* \in \text{cast}^*(v) \text{ and } v_j^* \in V_n^* \setminus \text{cast}^*(v)\}$

This embedding simply states, that the added node shares an edge with all nodes, that have at least one connection with one of the removed nodes. Geometrical interpreted it states, that two composed objects touch each other, if at least two of their components do so. As a consequence of our embedding rule, the reverse application of a production (i.e. from left to right) is not uniquely defined.

A **role specification** is a 5-tuple r-spec $= (r, C^{\approx}, C^{\not\approx}, C^{=}, C^{func})$ where r is the specified role, $C^{\approx}, C^{\not\approx}$ and $C^{=}$ are finite sets of path constraints and C^{func} is a finite set of functional constraints and predicates (see e.g. figure 2). A role may have several specifications within a grammar, but not within a production. A **path constraint** for a role r in a production p is a 2-tuple (π, π_0) of pathes and denotes two nodes v^* and v_0^* as follows: Let v be a node added by an application of p and $v_0 = v(r)$. Then $v^* := v(\pi)$ and $v_0^* := v_0(\pi_0)$. The path constraint enforces v^* and v_0^* to be connected, not to be connected or to be identical (i.e. to overlap) depending on the type of the set of path constraints it occurs. The ability to express deep relations, i.e. relations between the components of the nodes in a graph enables us e.g. to express an alignment of components and thereby to reduce the ambiguity inherent to our grammar formalism.

A **functional constraint** of a r-spec of a production p is an expression of the form $a = f(\pi_1, \ldots, \pi_n)$, where a denotes an attribute value $v(a)$ of a node v added by p and f is applied to the (role or attribute) values $v(\pi_i)$. Analogous a **predicate** is an expression of the form $p(\pi_1, \ldots, \pi_n)$ and yields true when applied to the denoted values $v(\pi_i)$. A node $v_0 \in \text{cast}(v)$ satisfies a given r-spec of a role r if $\varphi(v_0) \leq_S$ sort(r-spec) and $v_0 = v(r)$ such that v satisfies all constraints and predicates given by r-spec.

Let g_0 be a terminal graph, $(g_0 \rightarrow g_1 \rightarrow \ldots \rightarrow g_n)$ a finite, nonempty sequence of successive applications of productions of a given gg such that g_n contains only one node v_g and $\text{metasort}(\varphi(v_g)) = goal$. Then that sequence is called a **derivation** of g_0 and the node v_g is called a **parse** or **feature tree** of g_0, where the nodes of this feature tree are given by $\text{cast}^*(v_g)$.

3 The Graph Parsing Procedure

Before we describe the algorithm to derive all parses from a given terminal graph some special data structures and orders will be defined.

To handle edges more easy by the parser we associate with every edge $(v_1, v_2) = e \in E_0$ of a given terminal graph g_0 a 2-tuple of **tiepoints** (tp_+, tp_-) corresponding to the two nodes v_1 and v_2 specifying the edge e. $\text{tps}(v)$ defines the set of all tiepoints of the node v. If tp_+ is a tiepoint then tp_- is its **complement** and vice versa. The embedding rule defined in the previous section introduces new edges at each application. But these new edges are defined in terms of the old ones, and depend by recursion on the set E_0 of edges of the initial terminal graph. The parser doesn't represent the added edges explicit, but calculates the connections using the tiepoints of the initial terminal graph.

As a first heuristic extension to the given grammar we add to every production p a total order $<_p$. So p becomes an ordered sequence [r-spec$_1$, ..., r-spec$_n$] of role specifications. The order decodes the strategy that the parser uses to build an instance of the production, i.e. he will try to find role values for the roles specified by p in the sequence given by $<_p$. The heuristic in our case should state that r-spec$_i$ $<_p$ r-spec$_j$ if r-spec$_i$ is more restricted than r-spec$_j$, this effects early pruning: if the parser can't find a role value for r-spec$_i$ he will never try to find a role value for r-spec$_j$. We constrain $<_p$ such that r-spec$_j$ specifies a connection or overlapping with at least one of the role values specified by r-spec$_i$, $1 \leq i < j$. This insures that the subgraph covered by a partial instantiation of a production is connected.

A **patch** is a partial (hypothesis, pp) or complete (fact, cp) instance of a production and consists of:

production	the production, whose partial or complete instance the patch is.
sort	the production's sort (or one of it's subsorts).
tps	the node's tiepoint set.
r-spec	the specification of the role that the patch is searching a role value for, empty, if the patch is complete.

Additional patches can have attributes and roles as defined above for nodes. The slot *sort* of a patch is treated like an attribute, i.e. it can be changed by a functional constraint to an appropriate subsort. We use a dotted notation, e.g. cp.tps denotes the tps of a complete patch cp. Two patches are **connected** if their tiepoint sets contain at least one complementary pair of tiepoints.

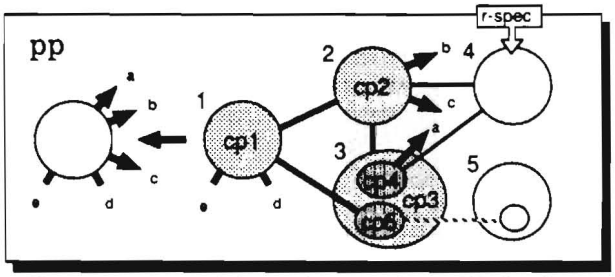


Figure 3: A partial patch with three filled roles

A cp is **combinable** with a pp iff

1) cp.sort $\leq_S$ sort(pp.r-spec)
2) For each path constraint $(\pi, \pi_0) \in$ pp.r-spec.$C^{\approx}$ the patches referenced by pp(π) and cp(π_0) are connected, and analogous for each path constraint $\in$ pp.r-spec.$C^{\not\approx}$ the referenced patches are not connected.
3) $(\pi, \pi_0) \in$ pp.r-spec.$C^{=}$ iff pp(π) = cp(π_0)
4) and all constraints $\in$ pp.r-spec.C^{func} are satisfied in the sense defined above. The functional constraints are always satisfiable by making them true, i.e. by treating them as an assignment of the function value to the thereby defined attribute.

The **agenda** is a set of patches. The **chart** contains all patches not contained in the agenda. Its purpose is, to retrieve quickly the set of all patches, that are combinable with a given patch. Its task is not, to retrieve exactly this set but a likewise small, but complete superset of this. The superfluous patches can then be eliminated by evaluating the constraints. The retrieval of patches is done by applying the following four careful chosen access functions:

CP(tp,sort) = {cp $\in$ chart | cp.sort = sort and tp $\in$ cp.tps}
yields the set of all complete patches cp of sort *sort* contained in the chart that contain the tiepoint tp in its tps.

PP(tp,sort) = {pp $\in$ chart | sort(pp.r-spec) = sort and ...}
yields a set of pp (not necessary all pp) of the chart, that are looking for a cp of sort *sort* as next role value such that the cp satisfies a connection constraint in pp.r-spec.$C^{\approx}$ if it contains tp in its tps. The function does not retrieve all pp with this property. However it is guaranteed that each cp combinable with a given pp contains at least one tp in its tps, such that for this tp pp $\in$ PP(tp,cp.sort). If the functions are implemented as (hash) tables this restriction avoids superfluous multiple entries of partial patches.

These functions may be implemented as an array of hash tables with tp serving as index and sort serving as key. The following two functions retrieve patches satisfying a given overlap constraint:

$\mathbf{CP}_{ov}(\mathbf{cp}^*) = \{cp \in chart \mid cp^* \in cast(cp)\}$
yields all complete patches of the chart that contain cp^* as direct role value.

$\mathbf{PP}_{ov}(\mathbf{cp}^*) = \{(pp, \pi_0) \mid cp^* = pp(\pi) \quad \text{and} \quad (\pi, \pi_0) \in pp.r\text{-}spec.C^=\}$
yields all 2-tuples (pp,π_0) of pp in chart such that pp is looking for a cp with an overlap $cp^* = cp(\pi_0)$ specified by pp.r-spec.$C^=$.

These functions may be implemented as hash tables with cp^* serving as key. Additional the chart contains two sets $\mathbf{CP}_0$ and $\mathbf{PP}_0$ containing all patches of the chart, whose tps is empty.

As a second and most important search guiding heuristic we add a total order $<_A$ for patches, its meaning will become clear below. The patch order $<_A$ may e.g. depend on the state of the agenda and an underlying ordering of the patch sorts or productions or attribute values or of a weighted combination of all. The weights may be determined by neural networks as proposed in [Er 90] or by genetic algorithms as proposed in [Bo 89].

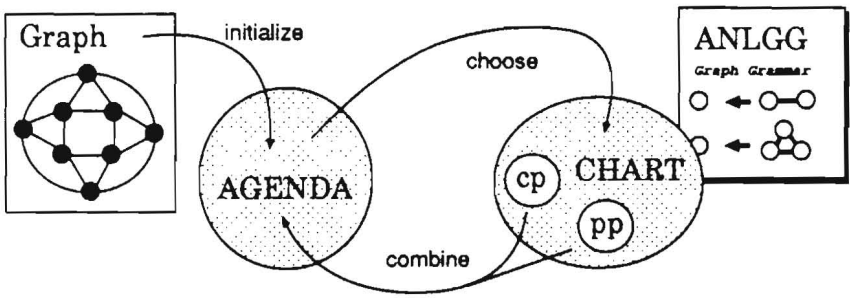


Figure 4: Architecture of the parser

Let us now describe the parsing procedure. The parser consists of three rules **initialize**, **choose** and **combine** that operate on two sets agenda and chart. The agenda is initialized with one patch for each of the graph's nodes and then choose and combine are applied alternately until the agenda runs empty. If this happens the chart i.e. CP_0 contains all possible parses. (see figure 4) In practise the parser will be stopped after the first parse is found.

```
parse graph:
     initialize chart and agenda
     until the agenda is empty do
          choose a best patch from agenda
          combine patch
          add patch to chart
     enduntil
```

initialize sets up an empty agenda and an empty chart, creates a cp for each node v of the input graph with cp.tps = tps(v) and adds it to the agenda.

choose picks up the most promising i.e. the $<_A$ greatest patch from the agenda, $<_A$ is completely free in ordering the patches, i.e. every order will finally lead to an empty agenda, however the workspace and time needed to encounter a first parse may crucially depend on the heuristic embodied by $<_A$.

```
combine patch:
     if patch is complete
        then propose patch
             continue-cp patch
        else continue-pp patch
     endif
```

propose combines a given cp with every production in `predict(cp.sort)` whose first r-spec is satisfied by the cp. For each such combination a new patch np is created with np.tps = cp.tps and is added to the

agenda. For each sort $s \in S$ `predict(s)` $\subseteq P$ yields the set of productions $\{p \mid s \leq_S \text{sort}(\text{first}(p))\}$. These sets depend on the given graph grammar only and can be precalculated.

`continue-cp` combines a given cp with all possible pp and `continue-pp` combines a given pp with all possible cp contained in the chart creating a new patch for every successful cp-pp combination and adds it to the agenda. For a production with n roles n such binary combinations yield a cp that is a complete instance of the production. The intermediate pp are generally used several times to build alternative instances, so no derivation work has to be done twice.

The four chart access functions are used to retrieve candidates for the combination with a given patch, i.e. the candidates can be received by simple union and intersecting operations on the sets received by single (in the case of CP_{ov} recursive iterated) applications of these functions. Take e.g. a look at figure 3: This pp is an instance of a production formed by 5 role specifications, values for the first three roles are already found. The pp is searching a role value for its fourth role r_4. The r-spec for this role states, that any candidate for the role needs a connection with r_2, occupied here by cp2 and a certain component of the value for the third role r_3, occupied here by cp4. The intersection of pp's tiepoint set pp.tps = $\{a, b, c, d, e\}$ with the tps of the two already determined neighbors of r_4 yields two sets of active tiepoints $\{a\}$ and $\{b, c\}$.

Then $\cup_{s \in \text{subsorts}(\text{sort}(\text{r-spec}))} \text{CP}(a, s) \cap (\text{CP}(b, s) \cup \text{CP}(c, s))$ yields the set of all cp in chart that satisfy all $C^{\approx}$ constraints given for the fourth role. In a similar fashion all other retrieval tasks can be done, using the four chart access functions defined above. E.g. for a given cp the set $\cup_{s \in \text{supersorts}(\text{cp.sort})} \cup_{tp \in \text{cp.tps}} \text{PP}(tp, s)$ contains all pp such that cp is of an appropriate sort and satisfys at least one of the $C^{\approx}$ constraints given for pp's next role.

Note that a patch p1 and a candidate p2 retrieved from chart might overlap, i.e. before combining them it has to be tested that the intersection $\text{cast}^*(p1) \cap \text{cast}^*(p2)$ contains exactly the set of overlappings specified by $C^{=}$, even in the case of $C^{=} = \emptyset$.

The tps of a newly created patch np is the union of cp.tps and pp.tps with complementary pairs of tiepoints removed (i.e. without connections between cp and pp). Additional all tiepoints are removed that are contained in the tps of an overlapping component but not simultaneously in cp.tps and pp.tps.

`add-to-chart patch` adds a partial or complete patch to chart such that the access functions work as specified above. If they are implemented as tables a patch causes in general several entries, e.g. a cp is entered in CP once for each of its tiepoints and in CP_{ov} once for each of its role values. Note that it is sufficient to enter a pp for the tiepoints of *one* of the already found neighbors of its next role, not for all, as mentioned in the specification of PP.

The parsing procedure presented here is an extension to the one introduced by R. Lutz in [Lu 89]. Main differences are the use of a search-guiding heuristic by ordering the agenda, the ability to specify overlappings, the ability to specify overlapping and neighborhood not only for nodes, but also for their components, and the use of sort hierarchies. Note also, that our total ordering of the productions right hand sides effects an early pruning and determines uniquely the order that the parser uses to build an instance, i.e. the instance is build once or never. In [Lu 89] an instance of a production whose rhs consists of n nodes will in the worst case be build in n! different ways leading to n! copies of the same patch.

4 Termination, Completeness and Complexity

The agenda is initialized with one patch for each node of the input graph. The number of patches that can be build from that by binary combinations via **combine** is finite, if the grammar does not allow infinite chains of production applications. Because every application of **choose** removes one patch from the agenda and **combine** generates no patch twice, the agenda will run empty after a finite number of pairwise **choose-combine** applications.

After each such pairwise application the following chart invariant holds: For two arbitrary patches cp and pp in chart the direct successor i.e. binary combination of cp with pp is in agenda or in chart. When finally the agenda runs empty this means: every direct successor of any patch in chart is already in chart and by induction: every successor, that can be generated by a finite sequence of binary combinations of the input nodes is in chart, in particular all possible parses.

Our formalism is expressive enough to cope with graph isomorphy which is well known to be np-complete. This doesn't mean, that every grammar formulated within our framework has worse runtime.

The strategy to avoid the combinatoric explosion inherent to the problem is here to incorporate domain-specific heuristics ($<_A$ and $<_p$) and sorts ($<_S$) to guide the parser's search.

5 Application to Feature Recognition in CIM

The parser has been implemented in Common Lisp and runs on SPARC stations with ivory boards. We developed graph grammars representing expert knowledge of our two experts in the turning and milling domain and tested the parser with several workpieces.

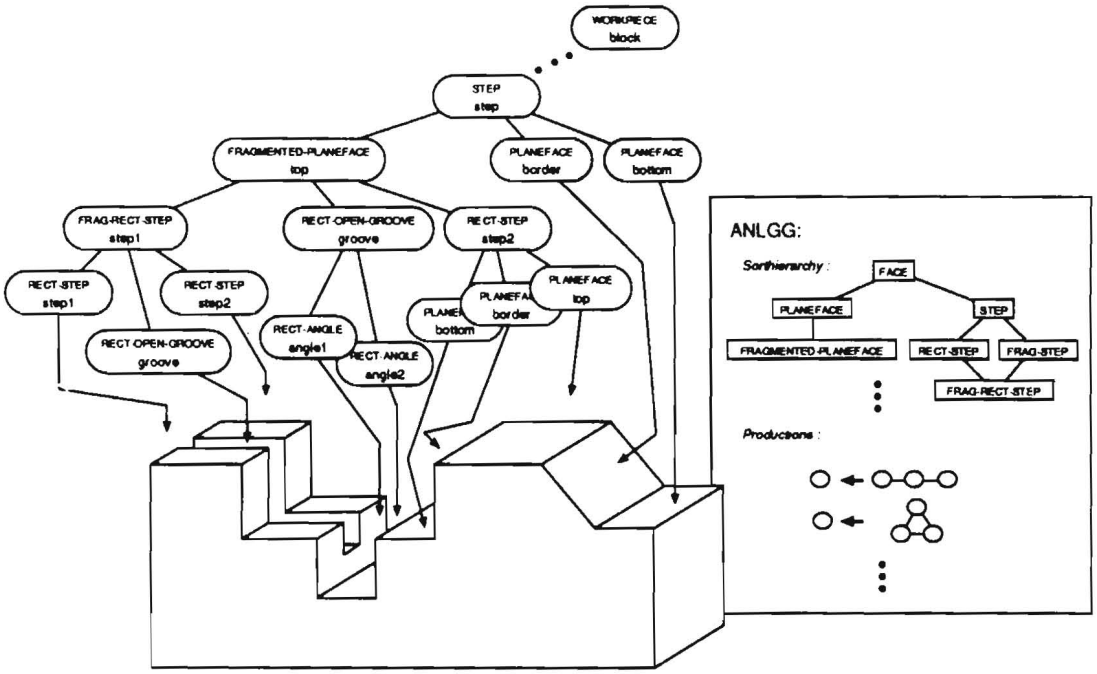


Figure 5: A workpiece and a part of the feature tree generated by the parser

The milling grammar, still under development, contains 19 productions and a sort hierarchy with 43 sorts. Sort hierarchies proved to be very useful both for efficiency and readability of the grammar.

The workpiece in figure 5 is represented by a graph containing 100 nodes and 50 edges. The parsing took 2.12 seconds (CPU time) to derive a first of 48 possible parses. 1224 patches were generated. Further work is needed to improve the heuristics guiding the parsers search in order to be able to deal with more complex workpieces and to complete the graph grammar.

The turning grammar contains 64 productions and a sort hierarchy with 27 sorts. One of the more complex shapes generates 587 patches to derive all 4 possible parses in 0.21 seconds (CPU time). In early days of the grammar the parser generates more then 2800 patches. The reduction to 587 was mainly caused by the $<_p$ heuristic.

In our system PIM (Planning In Manufacturing, see [Le 92]) the feature tree is used to guide the merging of skeletal plans associated with the sorts to get an overall plan for the manufacturing of the workpiece suitable for NC-machines.

6 Conclusion

In this paper we presented the 'made-to-measure' grammar formalism ANLGG and a heuristic driven chart based parser based on ANLGG for feature recognition in CIM. This tool supports the integration of any CA*-system via features ([Kl 92]) with any other CA*-system. The two possibilities to integrate heuristics as well as the use of a chart and sorts facilitates the effort to come quickly to a good solution.

This is necessary to support the communication between several CA* systems and to support the tasks of the systems ([Le 92]).

The presented algorithm and formalism can also be used in other domains that have the same or similar characteristics (from the point of view of graph grammars) as the features in CIM.

References

[Bo 89] Booker, L. B. et al: Classifier Systems and Genetic Algorithms. in: *Artificial Intelligence*. No. 40, pp. 235-282, 1989.

[Ch 90a] Chang, T.C.: Expert Process Planning for Manufacturing. *Addison-Wesley*, 1990.

[Ch 90b] Chang, T.C. et al: Feature extraction and feature based design approaches in the development of design interface for process planning. in: *Journal of Intelligent Manufacturing*, pp. 1-15, 1990.

[Ch 91] Chuang, S.-H. et al: Compound Feature Recognition by Web Grammar Parsing, in: *Research in Engineering Design*, Springer-Verlag, pp. 147-158, 1991.

[Eh 79-90] Ehrig, H. et al: Graph Grammars and Their Application to Computer Science. 1th - 4th International Workshop, *Springer Verlag*, LNCS 73, 153, 291, 532, 1979-1990.

[Er 90] Ertel, W. et al: Automatic Acquisition of Search Guiding Heuristics. in: *IJCAI'90*, pp. 470-484, 1990.

[Fi 90] Finger, S. et al: Parsing Features in Solid Geometric Models. in: *ECAI'90*, pp. 566-572, 1990.

[Kl 91] Klauck, Ch. et al: FEAT-REP: Representing Features in CAD/CAM. in: *IV International Symposium on Artificial Intelligence: Applications in Informatics*, pp. 245-251, 1991.

[Kl 92] Klauck, Ch. et al: Feature based Integration of CAD and CAPP. in: *CAD'92: Neue Konzepte zur Realisierung anwendungsorientierter CAD-Systeme*, Springer-Verlag, forthcoming 1992.

[Le 92] Legleitner, R. et al: PIM: Skeletal Plan based CAPP. in: *International Conference on Manufacturing Automation*, forthcoming 1992.

[Lu 89] Lutz, R. : Chart Parsing of Flowgraphs. in: *IJCAI-89*, pp 116-121, 1989.

[Ma 92] Mauss, J.: Ein heuristisch gesteuerter Chart Parser für attributierte Graph Grammatiken, *Diplomarbeit*, Universität Kaiserslautern, 1992.

[Mu 91] Mullins, S. et al: Grammatical Approaches to Engineering Design, Part I: An Introduction and Commentary. in: *Research in Engineering Design*, Springer-Verlag, pp. 121-135, 1991.

[Ri 91] Rinderle, R. : Grammatical Approaches to Engineering Design, Part II: Melding Configuration and Parametric Design Using Attribute Grammars. in: *Research in Engineering Design*, Springer-Verlag, pp. 137-146, 1991.

Literaturverzeichnis

[AU 72] A. Aho, J. Ullman: The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing, Prentice-Hall 1972.

[BKS 91] K. Becker, C. Klauck, J. Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM. Technical Memo TM-91-12, DFKI 1991.

[BKL 91a] A. Bernardi, C. Klauck, R. Legleitner: TEC-REP: Repräsentation vom Geometrie- und Technologieinformationen. Document D-91-07, DFKI 1991.

[BKL 91b] A. Bernardi, C. Klauck, R. Legleitner: FEAT-REP: Representing Features in CAD/CAM. in: IV International Symposium on Artificial Intelligence: Applications in Informatics, 1991.

[BKL 91c] A. Bernardi, C. Klauck, R. Legleitner: PIM: Skeletal Plan based CAPP. in: International Conference on Manufacturing Automation, Hongkong, 1992.

[BKL 92] A. Bernardi, C. Klauck, R. Legleitner: Feature based Integration of CAD and CAPP, in: CAD '92: Neue Konzepte zur Realisierung anwendungsorientierter CAD-Systeme, Springer Verlag 1992.

[BlBü 87] K. H. Bläsius, H.-J. Bürckert: Deduktionssysteme. Oldenbourg 1987.

[BGH 89] L.B. Booker, D.E. Goldberg, J.H. Holland: Classifier Systems and genetic algorithms, in: Artifical Intelligence Vol.40 (1989), pp 235-282.

[BuHa 89] H. Bunke, B. Haller: A parser for context free plex grammars, in: M. Nagl (Ed.): Graph Theoretic Concepts in Computer Science, 15'th Workshop 1989, pp136-150.

[BS 85] R.J. Brachman, J.G. Schmolze: An Overview of the KL-ONE Knowledge Representation System, in: Cognitive Science 9 (1985), pp 171-216.

[CH 91] S.H. Chuang, M.R. Henderson: Compound Feature Recognition by Web Grammar Parsing. Research in Engineering Design, Part II, Springer Verlag 1991, pp 147-158, 1991.

[DrKr 90] F. Drewes, H.-J. Kreowski: A Note on Hyperedge Replacement, in: [EKG 79-90], LNCS 532, pp 1-11.

[Ear 70] J. Earley: An Efficient Context-Free Parsing Algorithm, in: Communciations of the ACM 13(2), pp 94-102, 1970.

[EnRo 90] J. Engelfriet, G. Rozenberg: Graph grammars based on node rewriting: an introduction to NLC graph grammars., in: [EKG 79-90], LNCS 532, pp 12-23.

[EKG 79-90] H. Ehrig et.al.: Graph Grammars and their Application to Computer Science. 1th - 4th International Workshop, Springer Verlag LNCS 73,153,291,532, 1979-90.

[FeHi 90] J.C.E. Ferreira, S. Hinduja: Convex hull-based feature-recognition method for 2.5D components. Computer-Aided-Design, Vol. 22, No. 1, pp 41-49.

[FiSa 90] S. Finger, S. Safier: Parsing Features in Solid Geometric Models. ECAI '90, pp 566-572, 1990.

[FlBr 89] L. De Floriani, E. Bruzzone: Building a feature-based object description from a boundary model, Computer-Aided-Design, Vol. 21, No. 10, pp 602-610.

[GaHe 90] P. Gavankar, M.R. Henderson: Graph-based extraction of protrusions and depressions from boundary representations, Computer-Aided-Design, Vol. 22, No. 7, pp 442-450.

[Gol 89] D. E. Goldberg: Genetic Algorithms in Search, Optimisation, and Machine Learning. Addison-Wesley, Reading, 412 pages, 1989.

[Har 72] F. Harary: Graph Theory, Addison-Wesley, third printing, 1972.

[Hin 91] K. Hinkelmann: Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation, Technical Memo TM-91-02, DFKI 1991.

[JoCh 88] S. Joshi, T.C. Chang: Graph-based heuristics for recognition of machined features from a 3D solid model, Computer-Aided-Design, Vol. 20, No. 2, pp 58-67.

[Kau 86] M. Kaul: Syntaxanalyse von Graphen bei Präzedenz-Graph-Grammatiken. Dissertation, Universität Passau, 1986.

[Kar 86] L. Kartunen: D-PATR: A Development Environment for Unification-based Grammars. Report No. CLSI-86-68, Stanford CSLI 1986.

[Lut 89] R. Lutz: Chart Parsing of Flowgraphs. IJCAI 89, pp 116-121.

[MuRi 91] S. Mullins, R. Rinderle: Grammatical Approaches to Engineering Design, Part I: An Introduction and Commentary. Research in Engineering Design, Springer Verlag 1991, pp 121-135.

[Rin 91] R. Rinderle: Grammatical Approaches to Engineering Design, Part II: Melding Configuration and Parametric Design Using Attribute Grammars. Research in Engineering Design, Springer Verlag 1991, pp 137-146.

[Sel 87] P. Sells: Lectures on Contemporary Syntactic Theories, CSLI Lecture Notes, Number3, 1987.

[SuEr 90] C. Suttner, W. Ertel: Automatic Acquisition of Search Guiding Heuristics, IJCAI '90, pp 470-484.

[SFP 88] O. Stock, R. Falcone, P. Insinnamo: Island Parsing and Bidirectional Charts, in: COLING 88, pp 636-641.

[Win 83] T. Winograd: Language as a Cognitive Process, Vol. I: Syntax, Chart Parsing: pp 116-127, Addison-Weseley 1983.

[Woo 88] J.R. Woodwark: Some speculations on feature recognition, Computer-Aided-Design, Vol. 20, No. 4, pp 189-196.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all publisched papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist: WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta: RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke: A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel: Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, Stanley Peters: The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka: Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-14
Peter Breuer, Jürgen Müller: A Two Level Representation for Spatial Relations, Part I
27 pages

RR-91-15
Bernhard Nebel, Gert Smolka: Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann: Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel, Nelson M. Mattos: The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-18
John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein: A Diagnostic Tool for German Syntax
20 pages

RR-91-19
Munindar P. Singh: On the Commitments and Precommitments of Limited Agents
15 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner FEAT-Rep: Representing Features in CAD/CAM
48 pages

RR-91-21
Klaus Netter: Clause Union and Verb Raising Phenomena in German
38 pages

RR-91-22
Andreas Dengel: Self-Adapting Structuring and Representation of Space
27 pages

RR-91-23
Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-24
Jochen Heinsohn: A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder: Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger: Integrated Plan Generation and Recognition
- A Logic-Based Approach -
17 pages

RR-91-27
A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer: ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit: Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne: Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

RR-91-31
H.-U. Krieger, J. Nerbonne: Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz: Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader, Klaus Schulz: Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-34
Bernhard Nebel, Christer Bäckström: On the Computational Complexity of Temporal Projection and some related Problems
35 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

RR-92-01
Werner Nutt: Unification in Monoidal Theories is Solving Linear Equations over Semirings
57 pages

RR-92-02
Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Frank Fein, Michael Malburg: Π_{ODA}: The Paper Interface to ODA
53 pages

RR-92-03
Harold Boley: Extended Logic-plus-Functional Programming
28 pages

RR-92-04
John Nerbonne: Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

RR-92-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark: Feature based Integration of CAD and CAPP
19 pages

RR-92-06
Achim Schupetea: Main Topics od DAI: A Review
38 pages

RR-92-07
Michael Beetz: Decision-theoretic Transformational Planning
22 pages

RR-92-08
Gabriele Merziger: Approaches to Abductive Reasoning - An Overview -
46 pages

RR-92-09
Winfried Graf, Markus A. Thies:
Perspektiven zur Kombination von automatischem Animationsdesign und planbasierter Hilfe
15 Seiten

RR-92-11
Susane Biundo, Dietmar Dengler, Jana Koehler:
Deductive Planning and Plan Reuse in a Command Language Environment
13 pages

RR-92-13
Markus A. Thies, Frank Berger:
Planbasierte graphische Hilfe in objektorientierten Benutzungsoberflächen
13 Seiten

RR-92-14
Intelligent User Support in Graphical User Interfaces:
1. InCome: A System to Navigate through Interactions and Plans
   Thomas Fehrle, Markus A. Thies
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
   Markus A. Thies, Frank Berger

22 pages

RR-92-15
Winfried Graf: Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

RR-92-17
Hassan Aït-Kaci, Andreas Podelski, Gert Smolka:
A Feature-based Constraint System for Logic Programming with Entailment
23 pages

RR-92-18
John Nerbonne: Constraint-Based Semantics
21 pages

RR-92-19
Ralf Legleitner, Ansgar Bernardi, Christoph Klauck
PIM: Planning In Manufacturing using Skeletal Plans and Features
17 pages

RR-92-20
John Nerbonne: Representing Grammar, Meaning and Knowledge
18 pages

RR-92-22
Jörg Würtz: Unifying Cycles
24 pages

RR-92-24
Gabriele Schmidt: Knowledge Acquisition from Text in a Complex Domain
20 pages

DFKI Technical Memos

TM-91-08
Munindar P. Singh: Social and Psychological Commitments in Multiagent Systems
11 pages

TM-91-09
Munindar P. Singh: On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

TM-91-10
Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch: Tree Adjoining Grammars mit Unifikation
149 pages

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann:
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Bussmann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang:
Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

DFKI Documents

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.):
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
Thilo C. Horstmann: Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann:
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

D-91-14
Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS : Knowledge Representation and Inference System
- Benutzerhandbuch -
28 Seiten

D-91-15
Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Jörg Müller, Thomas Oltzen, Michael Sintek, Werner Stein, Frank Steinle:
µCAD2NC: A Declarative Lathe-Worplanning Model Transforming CAD-like Geometries into Abstract NC Programs
100 pages

D-91-16
Jörg Thoben, Franz Schmalhofer, Thomas Reinartz:
Wiederholungs-, Varianten- und Neuplanung bei der Fertigung rotationssymmetrischer Drehteile
134 Seiten

D-91-17
Andreas Becker:
Analyse der Planungsverfahren der KI im Hinblick auf ihre Eignung für die Abeitsplanung
86 Seiten

D-91-18
Thomas Reinartz: Definition von Problemklassen im Maschinenbau als eine Begriffsbildungsaufgabe
107 Seiten

D-91-19
Peter Wazinski: Objektlokalisation in graphischen Darstellungen
110 Seiten

D-92-01
Stefan Bussmann: Simulation Environment for Multi-Agent Worlds - Benutzeranleitung
50 Seiten

D-92-02
Wolfgang Maaß: Constraint-basierte Plazierung in multimodalen Dokumenten am Beispiel des Layout-Managers in WIP
111 Seiten

D-92-03
Wolfgan Maaß, Thomas Schiffmann, Dudung Soetopo, Winfried Graf: LAYLAB: Ein System zur automatischen Plazierung von Text-Bild-Kombinationen in multimodalen Dokumenten
41 Seiten

D-92-06
Hans Werner Höper: Systematik zur Beschreibung von Werkstücken in der Terminologie der Featuresprache
392 Seiten

D-92-08
Jochen Heinsohn, Bernhard Hollunder (Eds.):
DFKI Workshop on Taxonomic Reasoning Proceedings
56 pages

D-92-09
Gernod P. Laufkötter: Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes
86 Seiten

D-92-10
Jakob Mauss: Ein heuristisch gesteuerter Chrat-Parser für attributierte Graph-Grammatiken
87 Seiten

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht 1991
130 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken

Jakob Mauss

D-92-10
Document