Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-92-14

Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM

Johannes Schwagereit

September 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

Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM

Johannes Schwagereit

DFKI-D-92-14

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

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

Inhaltsverzeichnis

Kapitel 1

Motivation

In jedem größeren Industrieunternehmen erfolgt die Abwicklung der Produktion unter Einsatz von EDV[1]-Anlagen. Immer mehr strebt man hierbei eine integrierte Aufgabendurchführung im Rahmen eines CIM[2]-Konzeptes an [HaLä 90].

CIM ist nach [Karg 90] eine Konzeption für eine umfassende Integration der DV-Unterstützung in den betriebswirtschaftlichen und technischen Bereichen einer Unternehmung. Bereiche, in denen bereits EDV-unterstützt aber in Insellösungen gearbeitet wird, sollen in einen datenintegrierten Verbund zusammengefaßt werden.

Zwei Teilgebiete der betrieblichen Leistungserstellung sind die *Konstruktion* und die *Arbeitsplanung* [Zäp 89]. Für die rechnergestützte Konstruktion hat sich der Begriff CAD[3] durchgesetzt, für das elektronische Erstellen eines Arbeitsplanes der Begriff CAPP[4]. CAD und CAPP werden in [Zäp 89] unter dem Begriff CAE[5] zusammengefasst.
CAD dient der Erstellung von Konstruktionszeichnungen und der Durchführung von Konstruktionsberechnungen. Die mit CAD entwickelte konstruktive Lösung für ein Objekt ist der Ausgangspunkt für die Arbeitsplanung. Dort wird ein Plan für die Herstellung des Objektes erstellt, also u.a. eine Ermittlung der Arbeitsvorgangsfolge und der Maschinenauswahl [Eve 89].

Die Komplexität und die wirtschaftliche Bedeutung der Arbeitsplanung hatte bereits eine Vielzahl von Ansätzen zur Folge, diese computerunterstützt durchzuführen. Auch der Einsatz von Methoden der KI[6] [BKL 91c] wurde untersucht. Einige dieser Ansätze werden in [Bec 91] vorgestellt.

In [BKL 91c, Bec 91] wird eine *Feature-basierte Arbeitsplanung* vorgeschlagen. Grundidee ist die Identifikation von Werkstückbereichen, aus denen nach dem Prinzip der heuristischen Klassifikation Informationen für die Arbeitsplanung resultieren. Als Beschreibungselement dienen dem Experten dabei *Feature*, die auf den geometrischen und technologischen Daten eines Produktes basieren. Jedem Feature sind ein

[1] Elektronische Datenverarbeitung
[2] Computer Integrated Manufacturing
[3] Computer Aided Design
[4] Computer Aided Process Planning
[5] Computer Aided Engineering
[6] Künstliche Intelligenz

oder mehr *Skelettpläne*[7] zugeordnet. Wenn ein Werkstück durch Feature beschrieben wird, kann aus den zugehörigen Sklettplänen ein Arbeitsplan generiert werden.

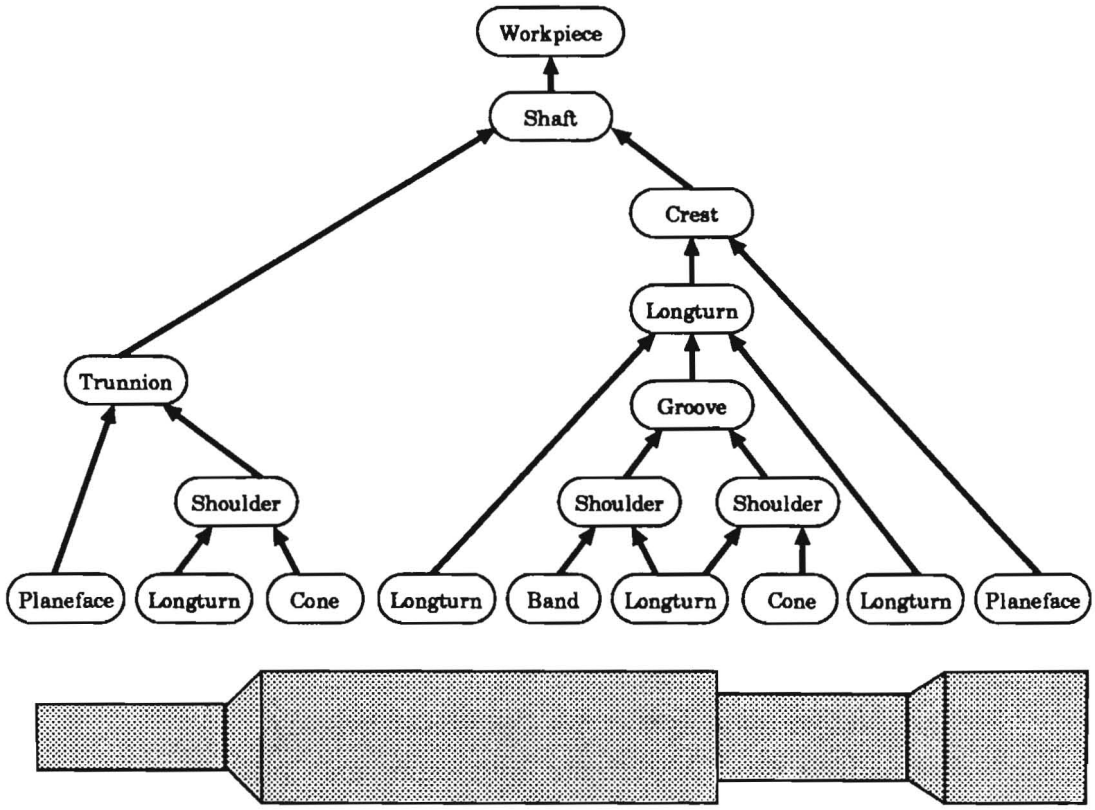


Abbildung 1.1: Beschreibung einer Welle durch Feature

Um die Beschreibung eines Werkstücks durch Feature automatisch erzeugen zu können, werden im ARC-TEC-Projekt[8] unter anderem *attributierte Graph-Grammatiken* verwendet. Durch Repräsentation der Feature als *Produktionen* dieser Grammatik und des Werkstücks als *Graph* können Feature durch den Rechner erkannt werden. Ein dafür spezialisierter Parser namens *GraPaKL* wird in [Mau 92] beschrieben.
Möglich ist auch der umgekehrte Weg: Die Konstruktion (Generierung) eines Werkstücks mittels der Feature.

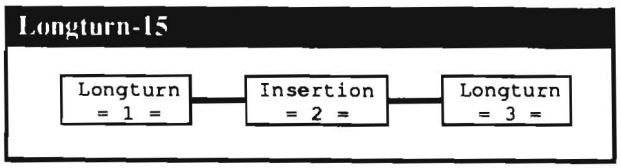
1.1 Aufgabenstellung

In einer Umgebung, in der Expertenwissen durch Feature ausgedrückt werden soll, kommt der Repräsentation dieser Feature eine wichtige Rolle zu.

Es war ein System zu entwickeln, in dem die Definitionen der Feature eingegeben, dargestellt und effizient verwaltet werden. Im einzelnen waren folgende Probleme zu lösen:

[7] abstrahierte Teil-Arbeitspläne

[8] ARC-TEC ist ein Projekt am DFKI, das sich in erster Linie mit dem Einsatz der KI in CIM beschäftigt.




```
((Featurename     Longturn)
 (Featurekind     geometrical)
 (Featurerules
   (
   ((Rule-Id     Longturn-15)
    (Nodes
      (
      ((Node-number 1)
       (Node-sort   Longturn))
      ((Node-number 2)
       (Node-sort   Insertion))
      ((Node-number 3)
       (Node-sort   Longturn))
     ))
    (Edges
       (((2) (1)) ((3) (2))))
   ))))
```

Abbildung 1.2: Darstellung eines Feature als Graph und seine Repräsentation in FEAT-REP

- Spezifizierung einer Datenstruktur für eine Graph–Grammatik, die maßgeschneidert für Feature in CIM ist, und dadurch die besonderen Charakteristika der Feature berücksichtigt.
- Entwurf einer integrierten Entwicklungsumgebung für Featuregrammatiken.
  - Tests auf korrekte Syntax der Featuredefinitionen.
  - Einbindung von TAXON zur Entdeckung und Verwaltung von Ähnlichkeiten zwischen Featuredefinitionen.
  - Tests zur Vollständigkeit und Wohlgeformtheit [Dro 89] der Grammatik.
- Entwicklung einer einfachen graphischen Benutzeroberfläche für diese Entwicklungsumgebung.
- Bereitstellung von Schnittstellen, die anderen Programmen den Zugriff auf die Repräsentation erlauben.
- Ein- und Ausgabe von Dateien im FEAT-REP–Format, einer Darstellung von Featuredefinitionen in Textform [BKL 91b].
- Implementierung eines Compilers, der die interne Repräsentation der Grammatik in die des Graph-Parsers GraPaKL überführt.

1.2 Gliederung der Arbeit und Überblick

Nachdem in diesem Kapitel eine Einführung in das zu bearbeitende Gebiet gegeben wurde und die Aufgabenstellung erläutert wurde, werden im **zweiten** Kapitel attributierte Grammatiken und Graph–Grammatiken definiert. Außerdem wird der Begriff der Wohlgeformtheit einer Grammatik erklärt.

Im **dritten** Kapitel werden zwei existierende Systeme zur Eingabe von Graph–Grammatiken vorgestellt. Speziell werden die Mächtigkeit der verwalteten Graph–Grammatiken und die Benutzerschnittstelle analysiert.

Das **vierte** Kapitel geht näher auf Feature in CIM ein. Ihre Definition und ihre Verwendung im ARC-TEC-Projekt werden aufgezeigt. Kurz wird die Vorgehensweise beim Parsen und Generieren von Werkstücken diskutiert.

Im **fünften** Kapitel wird eine attributierte Graph–Grammatik zur Repräsentation von Feature definiert. Sie kombiniert und erweitert die Definitionen aus Kapitel drei um die in Kapitel vier genannten Charakteristika der Feature berücksichtigen zu können.

Taxonomien dienen zum Aufbau einer Hierarchie auf den eingegebenen Featuredefinitionen. Sie bilden damit ein wichtiges Werkzeug zur Effizienzsteigerung bei der Benutzung von Features und zur Strukturierung der Regelbasis. Die Definition und Verwendung von Taxonomien, sowie das dafür verwendete Programm TAXON werden im **sechsten** Kapitel erläutert.

Das in dieser Arbeit erstellte System zur Entwicklung der Feature-Grammatiken erhielt den Namen GGD[9]. Sein Konzept wird in Kapitel **sieben** vorgestellt.

Kapitel **acht** zeigt, wie das Konzept aus Kapitel sieben umgesetzt wurde und erläutert die Implementierung des GGD. An Hand eines Beispiels wird die Erstellung einer Grammatik mit dem GGD gezeigt. Im **neunten** Kapitel wird die Frage diskutiert, ob statt des GGD auch eines der in Kapitel drei vorgestellten Systeme verwendet oder angepasst werden könnte, um Feature zu repräsentieren.

In Kapitel **zehn** werden schließlich die Resultate dieser Arbeit zusammengefaßt. Außerdem werden einige ungelöste Probleme angesprochen und Vorschläge für Erweiterungen des GGD gemacht.

Anhang A wendet sich an den Benutzer des GGD, also den Wissensingenieur, der eine Grammatik entwickelt. Auch wenn großer Wert auf die Entwicklung einer benutzerfreundlichen Oberfläche gelegt wurde, ist bei diesem komplexen Programm ein solches Manual notwendig.

Anhang B erläutert Details der Implementierung, um zukünftige Wartungsarbeiten und Erweiterungen zu ermöglichen. Es werden auch die Schnittstellen zu anderen Programmen dokumentiert.

[9]Graph Grammar Developer

Kapitel 2

Attributierte Grammatiken und Graph–Grammatiken

In diesem Kapitel sollen die formalen Grundlagen für die in dieser Arbeit verwendeten attributierten Graph–Grammatiken gelegt werden. An Hand der Literatur werden attributierte String-Grammatiken und allgemeine Graph–Grammatiken vorgestellt. In Kapitel 5 wird dann, aufbauend auf diesen Definitionen eine Attributierte Graph–Grammatik entwickelt, die maßgeschneidert für die Repräsentation von Featuredefinitionen ist.

2.1 Attributierte Grammatiken

Es soll zunächst die nicht-attributierte Grammatik definiert werden, wie sie in der Theorie der formalen Sprachen genutzt werden. Üblicherweise werden (kontextfreie) Grammatiken wie folgt definiert [Dro 89]:

Definition 2.1 (Grammatik)
Eine Grammatik[1] *G ist ein 4-Tupel $G = (T, N, S, P)$. Dabei ist T die nichtleere, endlich Menge der Terminalsymbole, N die nichtleere, endliche Menge der Non-Terminalsymbole und $S \in N$ das Startsymbol, auch Axiom genannt. Sei $V = T \cup N$. P ist die Menge der Produktionen (Regeln), jede Produktion wird als $A \rightarrow X_1, \ldots, X_n$ dargestellt mit $A \in N$, $X_i \in V$ für alle $i \in [1, \ldots, n]$. S darf nicht auf der rechten Seite einer Produktion verwendet werden.*

Definition 2.2 (Sprache) *Die durch eine Grammatik G definierte Sprache, also die Menge der durch diese Grammatik definierten Wörter, wird als $L(G)$ bezeichnet.*

Attributierte Grammatiken sind Erweiterungen der oben definierten Grammatiken [Mah 88]. Sie sind ein anerkanntes Mittel zur Beschreibung der statischen Semantik von Programmiersprachen, und werden bei der Syntaxanalyse von Programmen eingesetzt [Knu 68].

[1] Um diese Grammatik von Graph–Grammatiken zu unterscheiden, wird sie im folgenden auch 'String Grammatik' genannt.

Jeder Knoten des beim Parsen oder Generieren entstehenden Strukturbaumes wird mit Attributen 'dekoriert', die die Eigenschaften dieses Knotens spezifizieren. Zu den Aufgaben der Analyse gehört es, die Werte der Attribute zu bestimmen - man spricht von Attributauswertung - und deren Konsistenz zu prüfen. Die Werte der Attribute ergeben sich i.allg. aus Attributwerten der im Strukturbaum benachbarten Knoten. Auch bei der Generierung eines Wortes können Attribute hilfreich sein, indem sie nicht kontextfreie Informationen bereitstellen.

Die formale Definition von Attributierten Grammatiken lautet (nach [Mah 88, AlMe 91]):

Definition 2.3 (Attributierte Grammatiken) *Ein Attributierte Grammatik AG ist ein 4-Tupel $AG = (G, A, R, B)$ mit*

> *$G = (T, N, P, Z)$ ist eine Grammatik wie in Definition 2.1 beschrieben. Sei $V = T \cup N$.*
>
> *$A = \bigcup_{x \in V} A(x)$ ist eine endliche Menge von Attributen, wobei $A(x)$ die Menge der Attribute eines $x \in V$ ist.*
>
> *$R = \bigcup_{p \in P} R(p)$ ist die Menge der Attributierungsregeln, die den Produktionen zugeordnet sind. $R(p)$ sind die Attributierungsregeln eines $p \in P$.*
>
> *$B = \bigcup_{p \in P} B(p)$ ist die Menge von Kontextbedingungen, die jeweils den Produktionen zugeordnet sind. $B(p)$ sind die Kontextbedingungen eines $p \in P$.*

Jedem $x \in V$ ist also eine (möglicherweise leere) Menge von Attributen zugeordnet. Durch Attributierungsregeln werden diesen Attributen Werte zugewiesen.

Definiert sind Attributierungsregeln als $(a_{j_0}, p, k_0) \leftarrow f((a_{j_1}, p, k_1), \ldots, (a_{j_n}, p, k_n))$, wobei (a_{j_i}, p, k_i) ein Attribut a_{j_i} des Symbols k_i in der Produktion p ist (mit $i \in [0, \ldots, n]$). Eine Attributierungsregel ist somit eine Abbildung aus der Wertemenge verschiedener Attribute in die Wertemenge eines bestimmten Attributs.

Zu beachten ist, das sich Attribute aus Attributwerten beliebiger Knoten der Produktion berechnen, und daß auch Attributen von Knoten auf der rechten Seite einer Produktion ein Wert zugewiesen werden kann. Es wird daher unterschieden in

'Abgeleitete' (synthetisierte) Attribute sind Attribute des Knotens der linken Seite der Produktion. Sie berechnen sich aus den Attributen der Symbole der rechten Seite.

'Ererbte' Attribute sind Attribute der Knoten der rechten Seite der Produktion, die sich aus den Attributen des Nonterminals auf der linken Seite berechnen. Sie dienen zum Ausdruck von Kontext–Abhängigkeiten.

'Lexikalische' Attribute, die vorgegebenen Attribute der Terminalsymbole.

Die Kontextbedingung ist eine boolsche Funktion, die aus den Attributwerten einer Produktion die Anwendbarkeit dieser Produktion bestimmt. Sie ist definiert

als $f((a_0, p, k_0), \ldots, (a_m, p, k_m)) \rightarrow \{\texttt{true}|\texttt{false}\}$. Liefert eine Kontextbedingung `false`, so kann diese Produktion nicht verwendet werden, es ist also ein semantischer Fehler entdeckt worden.

Die Theorie der Attributierten Grammatiken beschäftigt sich mit den Problemen, die sich aus diesen Definitionen ergeben (Bestimmung von Auswertereihenfolgen, Vermeidung von zyklischen Attributsdefinitionen) [Mah 88].

Obwohl Attributsauswertungen und die Kontextbedingungen einen Bezug zum Kontext herstellen, wird eine solche Grammatik als kontextfreie attributierte Grammatik bezeichnet. Entscheidend für die Bezeichnung 'kontextfrei' ist die Form der Produktionen der Grammatik.

2.2 Graph–Grammatiken

Graph–Grammatiken sind seit etwa 20 Jahren ein Gegenstand der Forschung, motiviert durch die theoretische Attraktivität und zahlreiche praktische Anwendungen [EKG 79-91]. So werden Graph–Grammatiken derzeit in der Mustererkennung, zur Modellierung biologischer Strukturen, im Software Engineering und zur Wissensrepräsentation eingesetzt. Mit einer in [BKL 91b] vorgestellten Anwendung, dem Einsatz in einem CIM-Konzept, beschäftigt sich auch diese Arbeit.

Der wesentliche Unterschied zwischen einer Grammatik, wie sie meist in der Theorie formaler Sprachen verwendet wird, und einer **Graph** Grammatik liegt darin, daß nicht mit Strings gearbeitet wird, sondern mit (i.allg. knotenbeschrifteten) *Graphen.*

Definition 2.4 (Graph) *Ein knotenbeschrifteter Graph G wird definiert als 4-Tupel $G = (V, E, \Sigma, \varphi)$, wobei V eine nichtleere Menge von* Knoten, *$E \subseteq V \times V$ die Menge der* Kanten, *Σ ein endliches nichtleeres Alphabet der* Knoten-Label, *und $\varphi : V \rightarrow \Sigma$ die* Labelfunktion *sind. Für ein $v \in V$ ist $\varphi(v)$ der* Label von v.

Für ein Alphabet Σ wird die Menge aller Graphen, deren Knoten-Label Elemente von Σ sind, als G_Σ bezeichnet.

LEARRE Graph–Grammatik

Das einfachste Modell einer Graph–Grammatik ist eine LEARRE[2] Grammatik [Roz 86]. Dabei wird eine Produktion als geordnetes Paar (α, β) definiert. α ist die *linke Seite* der Produktion, β die *rechte Seite.* α und β sind Graphen.

Für die Anwendung einer Produktion werden folgende Bezeichner definiert:

Muttergraph: Bezeichnung für den Teilgraph, der zur linken Seite der Produktion isomorph ist, und daher ersetzt wird.

Restgraph: Der Teil des Graphen, der nicht zum Muttergraph gehört, wird *Restgraph* genannt.

[2]Ein Akronym für 'Locate, establish Embedding Area, Remove, Replace, Embed'.

Einbettungsgebiet: Der Teil des Restgraphen, der zum Tochtergraphen benachbart sein wird.

Tochtergraph: Die rechte Seite der Produktion, die eingefügt wird.

Prinzipiell läuft die Anwendung einer Produktion auf einen Graph in fünf Schritten ab:

1. Im Ausgangsgraphen wird ein zur linken Seite der Produktion isomorpher Teilgraph bestimmt, der *Muttergraph.*
2. Es wird eine Teilmenge der Knoten des Restgraphen bestimmt, das *Einbettungsgebiet.* Das Verfahren zu dessen Ermittlung ist abhängig vom spezifischen Typ der Grammatik.
3. Der Muttergraph wird entfernt (zusammen mit allen Kanten, die mindestens ein Ende im Muttergraph haben).
4. Die rechte Seite der Produktion, der *Tochtergraph* wird eingefügt.
5. Der Tochtergraph wird eingebettet, d.h. es werden Kanten gezogen zwischen den Knoten des Einbettungsgebiets und einigen Knoten des Tochtergraphen. Die genaue Festlegung der zu ziehenden Kanten ist abhängig vom spezifischen Typ der Grammatik.

NLC Graph–Grammatik

Die meisten Ansätze für Graph–Grammatiken basieren auf dem LEARRE-Modell. Häufig ist die Einschränkung, daß bei der Generierung eines Graphen jeweils nur ein einzelner Knoten ersetzt wird, die linke Seite der Produktion (der Muttergraph) ist also ein einzelner Knoten [EnRo 91]. Die Ersetzung von Knoten wird durch eine endliche Zahl von Produktionen, die Einbettung durch eine endliche Zahl von *Einbettungs-Anweisungen* spezifiziert. Oft sind diese Produktionen und die Einbettungs-Anweisungen kombiniert in Ersetzungsregeln, die jeweils aus einer Produktion und endlich vielen Einbettungs-Anweisungen bestehen. Diese Ersetzungsregeln bilden dann den Hauptbestandteil einer Graph–Grammatik.

Ein typischer Vertreter dieser Art von Grammatiken sind NLC[3] Graph–Grammatiken. Die Produktionen haben einen einzelnen Knoten auf ihrer linken Seite, und einen ungerichteten Graph auf ihrer rechten Seite. Die Einbettungs-Anweisungen verbinden den Tochtergraphen mit einer Teilmenge der Knoten, zu denen der Muttergraph benachbart war. Die Ersetzung und Einbettung basiert auf den Label der beteiligten Knoten. Die Einbettung erfolgt **lokal**, d.h. im Gegensatz zur LEARRE Grammatik werden neue Kanten nur zu den zum Mutterknoten benachbarten Knoten gezogen.

Definition 2.5 (NLC Graph–Grammatik) *Eine NLC Graph–Grammatik ist definiert durch* $GG = (N, T, P, C, S)$, *mit*
N *ist eine nicht leere, endliche Menge, die* Nonterminale.

[3]Node Label Controlled

T ist eine nicht leere Menge, die Terminale. *Sei* $V = N \cup T$
P ist eine endliche Menge von Paaren der Form (α, β) *mit* $\alpha \in V$, $\beta \in G_\Sigma$, *die Menge der* Produktionen.
C ist eine Teilmenge von $V \times V$, *die* Verbindungsrelation.
$S \in \Sigma$ *ist der Startsymbol.*

NLC Graph–Grammatiken sind *kontext-frei* in dem Sinne, daß es keine Anwendungsbedingungen gibt und Ersetzungen lokal sind. Andererseits ist, bedingt durch die Einbettung, die Reihenfolge der Graphersetzungen entscheidend[4] für den resultierenden Graph[5].

In kontextfreien Graph–Grammatiken in bei der *Generierung* eines Graphen die Anwendung einer Produktion *nicht* Reihenfolge-erhaltend, ein Unterschied zu String-Grammatiken.

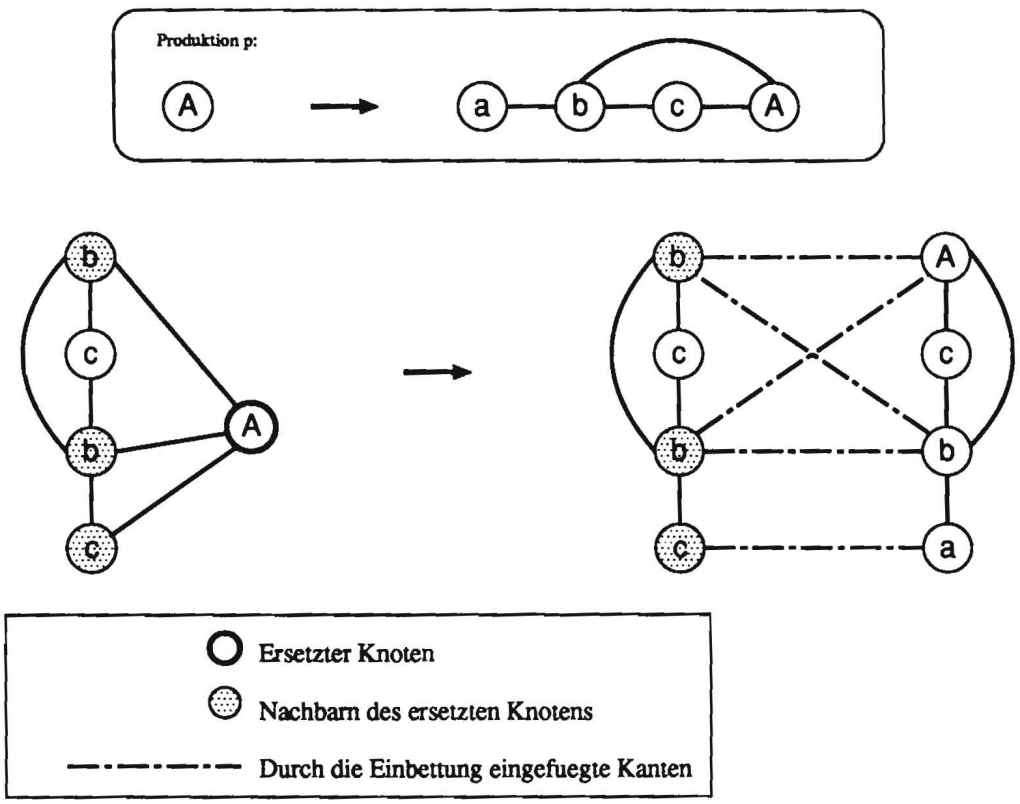


Abbildung 2.1: Beispiel für die Anwendung einer Produktion in einer NLC GG

Ein Beispiel für die Anwendung einer Produktion zeigt Abb. 2.1. Die Verbindungsrelation sei gegeben durch $C = \{(a,c), (b,b), (A,b)\}$. Dann ergibt sich aus dem linken Graph durch Anwendung der Produktion p der rechte Graph.

[4]Dieses ist nicht zu verwechseln mit der Eigenschaft *Konfluenz*. Natürlich sind auch Graph-Grammatiken im allgemeinen *nicht* konfluent.

[5]In [JaRo 83] werden allerdings *Neighbourhood-uniform NLC grammars* vorgestellt, bei denen die Reihenfolge der Anwendung ohne Belang ist. Diese Grammatiken sind allerdings nicht so mächtig wie NLC Grammatiken.

RNLC Graph–Grammatik

Eine Abwandlung von NLC Graph–Grammatiken sind solche mit regelabhängiger Einbettung, dort hat jede Produktion eine eigene Einbettungsrelation.

Definition 2.6 (RNLC[6] Graph–Grammatiken) *Eine RNLC Graph–Grammatik ist definiert durch* $E = (\Sigma, T, P, Z)$, *wobei* Σ, T *und* Z *wie in Def. 2.5 definiert sind.* P *ist eine Menge von Tripeln* (α, β, C) *mit* $\alpha \in \Sigma$, $\beta \in G_\Sigma$ *und* $C \subseteq \Sigma \times \Sigma$.

In [JaRo 80] wird gezeigt, daß die Menge der möglichen NLC Grammatiken gleich der Menge der RNLC Grammatiken ist. In dem Beweis wird zu jeder RNLC Grammatik eine äquivalente NLC Grammatik konstruiert, die umgekehrte Richtung ist trivial.

1-NCE Graph–Grammatik

Ähnlich der RNLC ist die *1-NCE[7] Graph–Grammatik* definiert[8] [EMR 84]. Im Unterschied zur RNLC werden in der Einbettungsvorschrift C nicht die Label der zu verbindenen Knoten angegeben. Es werden zu tatsächlich vorhandenen Knoten der rechten Seite die Label der Knoten angegeben, mit denen sie verbunden werden können.

Da es auf der rechten Seite einer Produktion auch mehrere Knoten mit dem gleichen Label geben kann, lassen sich mit 1-NCE Graph–Grammatiken detailliertere Einbettungsvorschriften als mit RNLC formulieren. In [EMR 84] wird gezeigt, daß die Menge der NLC Grammatiken und die Menge der 1-NCE Grammatiken gleich ist.

dNCE Graph–Grammatik

Eine andere Erweiterung von NLC Graph–Grammatiken erreicht man bei der Verwendung gerichteter Kanten, also Pfeilen. In diesen sogenannten *dNLC Grammatiken*[9] [EnRo 91] werden jeweils zwei Verbindungsrelationen spezifiziert, C_{in} und C_{out}. Die Elemente dieser beiden Mengen geben an, für welche vom ersetzten Knoten ein- bzw. ausgehenden Kanten neue Kanten gezogen werden. Dabei ist diese Ersetzung richtungserhaltend, d.h. Kanten einer bestimmten Richtung werden immer nur durch Kanten der gleichen Richtung ersetzt.

Die in dieser Arbeit verwendete Graph–Grammatik 'ANCEGG'

Die in Kapitel 5 definierte Grammatik kombiniert einige der hier vorgestellten Definitionen. Es handelt sich um eine NCE GG, für die wie für eine attributierte Grammatik Attribute, Berechnungsvorschriften und Bedingungen definiert sind. Außerdem sind zwei verschiedene Mengen von Kanten definiert, 'Nachbarschaften' und 'Überlappungen'.

---

[7] Neighbourhood-Controlled Embedding

[8] Wobei 1-NCE Grammatiken wiederum nur ein Spezialfall der NCE Grammatiken sind. In NCE besteht die linke Seite einer Produktion aus einem allgemeinen Graphen, bei 1-NCE nur aus einem Knoten.

[9] directed NLC grammars

2.3 Wohlgeformtheit einer Grammatik

Das in Kapitel 7 vorgestellte System GGD zur Entwicklung einer Graph-Grammatik ist in der Lage, die *Wohlgeformtheit* einer Grammatik zu überprüfen. Dieser Begriff soll im folgenden definiert werden.

Die meisten Definitionen, die für String-Grammatiken angegeben werden, lassen sich wegen der Ähnlichkeit der Konzepte auf Graph-Grammatiken übertragen. Entsprechend wurden auch hier die Definitionen zur Wohlgeformtheit aus [Dro 89] an Graph-Grammatiken angepaßt.

Definition 2.7 *Ein Symbol Y ist* **sinnlos**, *wenn gilt:* $\neg\exists\, i \in \mathbb{N}$ *mit* $Y \Rightarrow^i d$ *und d ein Graph über T. Y läßt sich also nicht zu einem Graphen expandieren, dessen Knoten alle Terminalsymbole $\in T$ sind.*

Ein Symbol $Y \in N \cup T$ ist **unerreichbar**, *wenn $\forall d \in L(G)$ gilt: $Y \notin d$.*

Eine Grammatik ist **λ-frei**, *wenn sie keine Produktion der Form $X \rightarrow \lambda$ enthält, wobei X ein Nonterminal ist und λ für den leeren Graphen steht.*

Eine **Einzelproduktion** *ist eine Produktion der Form $X \rightarrow Y$, wobei sowohl X als auch Y Nonterminale sind.*

Definition 2.8 (Wohlgeformte Grammatik) *Eine Grammatik $G = (N, T, P, Z)$ ist wohlgeformt*[10]*, wenn sie die folgenden Bedingungen erfüllt:*

1. *Sie enthält keine* sinnlosen *Symbole.*
2. *Sie enthält keine* unerreichbaren *Symbole.*
3. *Sie ist λ-frei.*
4. *Sie enthält keine* Einzel-Produktionen.

Durch Überprüfung der Wohlgeformtheit einer Grammatik läßt sich testen, ob Fehler in der Grammatik die Analyse eines allgemeinen Graphen möglicherweise verhindern, und ob es Produktionen gibt, die nicht zur Anwendung kommen können. Bei der Generierung mit einer nicht wohlgeformten Grammatik läßt sich unter Umständen kein Graph erzeugen, dessen Knoten nur Terminalsymbole sind.

Durch das Verbot von Einzelproduktionen erreicht man, daß ein Parser immer terminiert, da mit jedem Schritt eines Parse-Vorgangs die Zahl der Knoten kleiner wird[11].

[10] engl.: proper

[11] Dennoch überprüft das System GGD diese Bedingung nicht. Zur Begründung siehe 7.5.2.

Kapitel 3

Existierende Systeme zur Eingabe von Graphen und Graph–Grammatiken

Da Graph–Grammatiken seit etwa zwanzig Jahren ein wichtiger Forschungsschwerpunkt sind [EKG 79-91], wurden auch Systeme entwickelt, um Graphen und Graph–Grammatiken einzugeben [Nag 79]. Die meisten dieser Systeme unterstützen den Benutzer durch eine graphische Oberfläche da Darstellungen von Graphen in Textform für Menschen in der Regel nur schwer zu verstehen sind.

In der Literatur sind verschiedene Systeme um Graphen einzugeben und zu editieren, beschrieben [ZiNa 79, New 88]. Einige Programme sind auf spezielle Anwendungen zugeschnitten, andere lassen sich flexibel anpassen. *Graph–Grammatiken* werden allerdings nur selten berücksichtigt.

Es sollen hier zwei für diese Arbeit interessante Systeme vorgestellt werden, mit denen sich auch Graph–Grammatiken entwickeln lassen. Es handelt sich um die beiden einzigen Systeme für Graph–Grammatiken, die in der mir zugänglichen Literatur in ausreichender Tiefe beschrieben sind. Von besonderem Interesse ist vor allem das System GraphEd.

3.1 GraphEd — Ein interaktiver Graph–Editor

GraphEd wurde 1988 von M. Himsolt an der Universität Passau entwickelt [Him 89]. Das Programm ist in C implementiert und läuft unter Unix und SunView. Es ist in erster Linie ein Editor, um *Graphen* einzugeben. Als Erweiterung lassen sich aber auch *Graph–Grammatiken* eingeben.

Anwendung

GraphEd versteht sich als System, um Graphen und Graph–Grammatiken zu entwickeln. Es ist nicht auf eine bestimmte Anwendung festgelegt, sondern bietet die Voraussetzungen für den Einsatz in verschiedene Bereichen. Der Autor führt u.a.

Petri-Netze, Datenbank-Design, elektronische Schaltungen und Datenstrukturen als Beispiele auf.

Mächtigkeit der Graph-Grammatik

GraphEd wird in [Him 91] als erstes System zur interaktiven Eingabe von Graph-Grammatiken überhaupt bezeichnet.

Um den Anforderungen möglichst vieler verschiedener Anwendungen gerecht zu werden, versucht das System, möglichst viele Arten von Graphen repräsentieren zu können. So erlaubt es gerichtete und ungerichtete Kanten, Sorten für Knoten und Kanten und mehrere Kanten zwischen zwei Knoten.

GraphEd bietet die Möglichkeit, Graph-Grammatiken, aber nicht *attributierte Graph-Grammatiken* zu repräsentieren. Jede Produktion hat einen Knoten auf ihrer linken Seite, einen beliebigen Graph auf ihrer rechten Seite. Graph-Grammatiken, in deren Produktionen die linke Seite auch mehrere Knoten enthalten darf, können nicht repräsentiert werden.

Als Regelung zur Einbettung von ersetzten Graphen[1] verwendet GraphEd eine Variante von (gerichteten) 1-NCE Graph-Grammatiken [Him 89].

Die Einbettung wird durch

$$\text{embedding: } SORT \times \{in, out\} \mapsto NODE$$

beschrieben. Dabei ist $NODE$ eine Menge von Knoten der rechten Seite und $LABEL$ eine Menge von Knotenlabels. *in* bzw. *out* bezeichnen die Richtung der jeweiligen Kante.

$embedding(l, d) = \{n_1, \ldots, n_m\}$ bedeutet, daß wenn es Kanten in Richtung d zwischen dem ersetzten Knoten und Knoten mit Sorte l gab, dann wird jede dieser Kanten ersetzt durch m neue Kanten, die diese Knoten mit Sorte l mit den Knoten n_1 bis n_m verbinden. Die neuen Kanten haben die gleiche Richtung wie die ersetzten Kanten.

Beispiel: $embedding('A', in) = \{2, 3\}$ heißt: Wenn es Kanten gab von Knoten mit Sorte *'A'* zu dem ersetzten Knoten, dann wird jede dieser Kanten ersetzt durch Kanten von den *'A'*-Knoten zu den Knoten 2 und 3 des eingefügten Knotens (2 und 3 sind Nummern zur Identifizierung der Knoten der rechten Seite der Produktion).

Abb. 3.1 zeigt eine beispielhafte Produktion. Die linke Seite hat das Label *'A'*, die rechte Seite besteht aus den drei Knoten *'X'*, *'b'* und *'d'*. Die beiden Knoten links spezifizieren die Einbettung. Bei Anwendung der Produktion wird jede Kante von einem *'B'*-Knoten zu dem zu expandierenden *'A'*-Knoten durch eine Kante zum *'X'*-Knoten der rechten Seite ersetzt, analoges gilt für Kanten von diesem *'A'*- zu *'c'*-Knoten.

[1] Es ist in erster Linie an eine Verwendung der Produktionen zur Graphgenerierung gedacht.

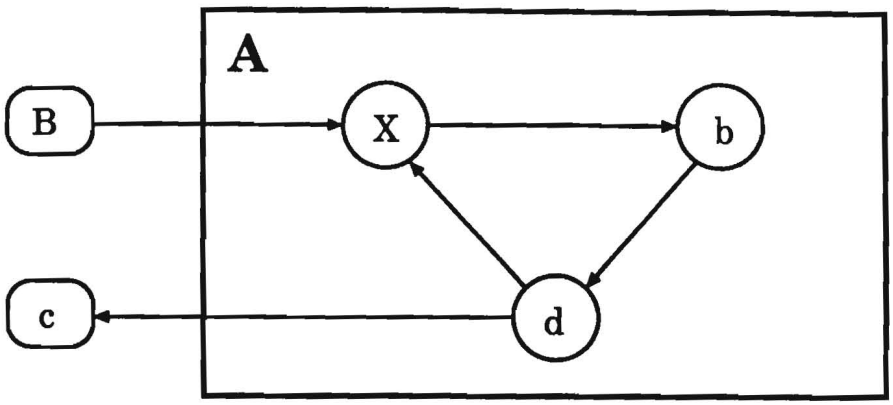


Abbildung 3.1: Beispiel einer Produktion in GraphEd

Graphische Oberfläche

GraphEd's Benutzeroberfläche bietet alle Operationen, um Graphen zeichnen und manipulieren zu können. Kanten und Knoten können eingefügt, gelöscht und verändert werden. Knoten beliebiger Form und Größe, und Kanten in verschiedenen Stilen und Biegungen sind möglich. Knoten können zu Gruppen zusammengefasst werden und zusammen modifiziert werden. Operationen zum Ausschneiden und Einfügen stehen zur Verfügung. Der Benutzer wird durch Algorithmen unterstützt, die das Layout eines Graphen optimieren (Minimierung von Kantenüberkreuzungen) oder Tests ausführen (wie einen Test auf Zusammenhang des Graphen). Es können mehrere Graphen zur gleichen Zeit bearbeitet werden.

Knoten und Kanten können zwar mit Attributen versehen werden, diese Attribute spezifizieren aber lediglich die Darstellung dieser Objekte in der GraphEd-Oberfläche (wie Größe, Form, Font des Labels).

GraphEd bedient sich einer eleganten Methode, Produktionen darzustellen: Der Knoten, der die linke Seite der Produktion darstellt, wird besonders groß gezeichnet, die gesamte rechte Seite wird in diesen Knoten gezeichnet. Knoten zur Spezifizierung der Einbettung werden außerhalb der linken Seite dargestellt. Der Graph wird also ähnlich wie in Abb. 3.1 dargestellt.

Der Benutzer kann in jedem dargestellten Graphen einen Knoten expandieren. Dazu wählt er erst eine Produktion aus, und dann den zu expandierenden Knoten. Dieser wird dann durch die rechte Seite der Produktion ersetzt und automatisch eingebettet.

Integration in andere Systeme

Andere Programme haben zwei Möglichkeiten, mit GraphEd zu kommunizieren. Graphen und Produktionen werden in einem dokumentierten Dateiformat gespeichert, diese Dateien können durch andere Programme gelesen und manupiliert werden. Alternativ können benutzerdefinierte Module in der Programmiersprache 'C' direkt auf die GraphEd-Datenstrukturen zugreifen.

3.2 PAGGED und NPAGGImp — Entwicklung attributierter programmierter Graph–Grammatiken

In [Göt 88] behandelt H. Göttler die Verwendung von Graph–Grammatiken zur Softwareentwicklung und beschreibt die Implementierung eines entsprechenden Systems an der Universität Nürnberg-Erlangen 1987-88. Das System besteht aus zwei Teilen, einem Softwarepaket zur Repräsentation und Anwendung von Graph–Grammatiken ('NPAGGImp') sowie einem Editor, um Produktionen eingeben zu können ('PAGGED'). Das System ist in LISP auf einem CADMUS-Rechner implementiert.

Anwendung

In dieser Arbeit werden attributierte, 'programmierte' Graph–Grammatiken zur Softwareentwicklung eingesetzt. Unter Verwendung dieser Grammatik werden Programme geschrieben. Der Zustand eines Programms ist durch einen Graph gegeben, jede Produktion beschreibt eine Änderung dieses Zustands.

Mit einer speziellen Programmiersprache (PGA[2]) kann der Kontrollfluß für die Anwendung der Produktionen spezifiziert werden. Es läßt sich angeben, welche Produktionen in welcher Reihenfolge unter welchen Bedingungen auszuwerten sind.

Mächtigkeit der Graph–Grammatik

Produktionen der Graph–Grammatik werden durch einen Graph mit drei Komponenten dargestellt: Erstens die 'linke Seite' der Regel, zweitens die 'rechte Seite', und drittens eine Komponente, die als Einbettungsvorschrift dient. Im Gegensatz zu vielen anderen Definitionen kann auch die linke Seite aus einem Graph mit mehr als nur einem Knoten bestehen. Jeder Knoten und jede Kante wird mit einem Label versehen.

Außerdem kann jeder Knoten attributiert werden, zu jeder Produktion lassen sich Attributsberechungsvorschriften angeben. Zusätzlich können Vor– und Nachbedingungen spezifiziert werden, die zur Anwendung der Produktion erfüllt sein müssen.

Eine Produktion läßt sich in einem sog. Y-Diagramm darstellen. Abb. 3.2 zeigt ein Beispiel für eine Produktion (ohne Kantenlabel).

An diesem Bespiel soll gezeigt werden, wie eine Produktion angewendet wird. Im linken Graph in Abb. 3.3 soll der Knoten mit Label 'a' durch Anwendung dieser Produktion durch einen Graph zweier Knoten mit den Label 'e' und 'f' ersetzt werden. Jede Kante von dem ersetzten Knoten zu Knoten mit Label 'b' wird durch Kanten von 'e' und 'f' zu 'b' ersetzt. Jede Kante von 'a' zu einem Knoten mit Label 'c', der wiederum mit einem Knoten mit Label 'd' verbunden ist, wird durch eine Kante von 'd' nach 'f' ersetzt. Kantenlabel wurden in diesem Beispiel nicht berücksichtigt.

[2]Programmierte GraphoperationsAnwendung

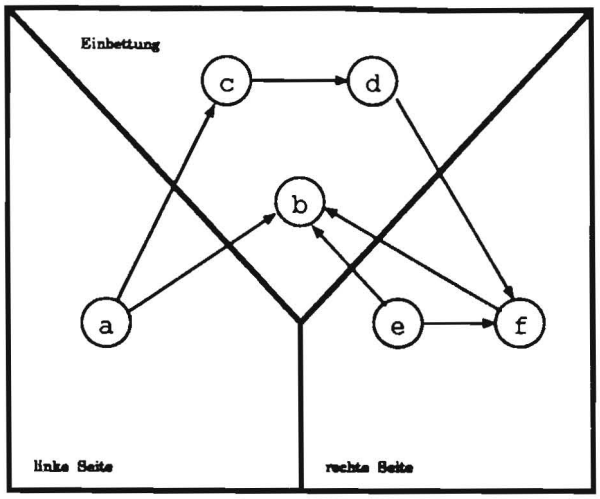


Abbildung 3.2: Beispiel einer Produktion im PAGGED

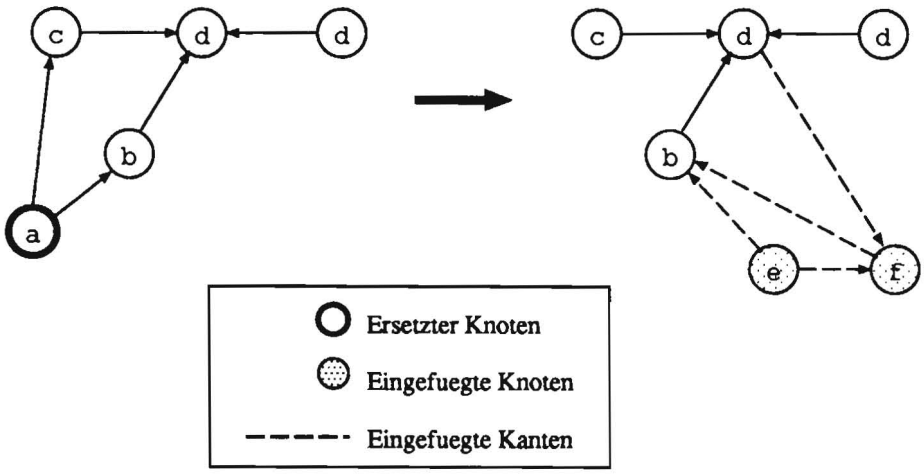


Abbildung 3.3: Beispiel für die Anwendung einer Produktion im PAGGED

Es fällt auf, daß diese Form der Einbettung deutlich mächtiger ist als die des GraphEd. Kanten können ihre Richtung ändern, und können flexibel ergänzt und gelöscht werden.

Die hier beschriebene Dreiteilung einer Produktion kann in einigen Fällen sehr ineffizient sein (Beispiel: Für das Löschen von Kanten sind linke und rechte Seite der Produktion identisch. Um die Produktion anzuwenden, ist das Löschen und Neuerzeugen der beteiligten Knoten erforderlich.). In der aktuellen Implementierung dieses Systems ist eine Produktion daher in vier Komponenten aufgeteilt (Als vierte Komponente gibt es Knoten, die vorhanden sein müssen, aber nicht gelöscht werden). Die sich daraus ergebende, sehr komplexe Produktionsanwendung soll hier nicht näher beschrieben werden.

Graphische Oberfläche

Als Benutzerschnittstelle dient der Grapheditor PAGGED. Mit ihm können Produktionen direkt in ein Y-Diagramm eingegeben werden. Die Graphen werden also wie

in Abb. 3.2 (Seite 19) dargestellt[3].

Es stehen natürlich alle notwendigen Operationen zum Erzeugen, Verschieben und Löschen von Knoten und Kanten zur Verfügung. Dazu kommen die Verwaltungsoperationen zum Erzeugen und Löschen von Produktionen. Für Attribute und Bedingungen wird ein Texteditor aufgerufen.

Einbindung in andere Systeme

Die mit PAGGED erstellten Produktionen werden an das System NPAGGImp weitergegeben, der sie zur Auswertung der PAG-Programme nutzt. Die Produktionen sind in Dateien mit LISP-artiger Syntax gespeichert, je Produktion wird eine Datei verwendet. Auch andere Programme könnten auf die Dateien zugreifen.

[3]In der aktuellen Implementierung wird ein X-Diagramm verwendnet, um die vier Komponenten einer Produktion darzustellen.

Kapitel 4

Feature in CIM

Ein zentraler Begriff dieser Arbeit sind *Feature*. Im ARC-TEC–Projekt bilden sie die Einheit, mit der ein Experte in seiner Domäne Wissen über ein Produkt formuliert.

In diesem Kapitel wird der Begriff des Feature definiert, die Bedeutung von Feature für diese Arbeit erklärt und die Anwendung von Feature aufgezeigt.

4.1 Was sind Feature?

Die meisten bisherigen CAD-Systeme beschreiben Werkstücke in Begriffen einer niedrigeren Abstraktionsebene wie Linien, Punkte, Oberflächen oder Toleranzen. Mit solchen Daten läßt sich ein Werkstück zwar quantitativ beschreiben, nicht aber auf einer höheren, qualitativen Ebene. Solche qualitativen Daten sind aber notwendig, um etwa die Fertigung eines Werkstücks planen zu können.

Im ARC-TEC–Projekt werden diese qualitativ höherwertigen Informationen durch sogenannte *Feature* repräsentiert. Die Feature können als Sprache betrachtet werden, in der ein Experte ein Werkstück beschreiben kann.

Die genaue Definition eines Feature ist umstritten, in [BKL 91b] findet sich die folgende Formulierung, die auch für diese Arbeit Gültigkeit haben soll:

> Ein **Feature** ist ein auf den technischen und geometrischen Daten eines Produkts basierendes Beschreibungselement, mit dem ein Experte in einer Domäne bestimmte Informationen assoziiert.

Verschiedene Experten werden also auch unterschiedliche Feature verwenden, um damit ein Werkstück zu beschreiben. Zum Beispiel sieht ein Konstrukteur ein Produkt aus einem anderen Blickwinkel als ein Experte für die Fertigung. Jeder Experte wird seine Erfahrung, sein Wissen und seine Terminologie in die Definition der für ihn relevanten Feature einfließen lassen.

Feature können nach Art und Anwendung unterschieden werden. Verschiedene Arten eines Feature sind *funktionale* (wie Lagersitz), *qualitative* (wie Stangenteil oder stabiles Teil) oder *geometrische* (wie Schulter, Nut oder Bohrung) Feature. Diese Einteilung findet sich auch in [SaFi 90].

In der Anwendung kann man beispielsweise unterscheiden, ob es sich um ein *Design*- oder *Fertigungs*-bezogenes Feature handelt.

Wenn Werkstücke nicht allein auf einer geometrischen Ebene beschrieben werden sollen, sondern auch durch Feature, stellt sich die Frage nach einer geeigneten Repräsentation. Für Feature und Werkstücke ist eine Repräsentation zu finden, die es erlaubt,

- Feature in einer einfachen, möglichst natürlichen Weise zu formulieren
- sich aus der Anwendung ergebende speziellen Charakteristika von Feature zu berücksichtigen
- das Problem effizient zu lösen, die Beschreibung eines Werkstücks durch geometrische und technische Primitive in eine Beschreibung durch Feature zu überführen
- die Beschreibung eines Werkstücks aus Featuredefinitionen zu generieren.

Im ARC-TEC–Projekt wird ein Ansatz verfolgt, der auf Graphen bzw. Graph–Grammatiken basiert. Diese Idee findet sich u.a. auch in [JoCh 88].

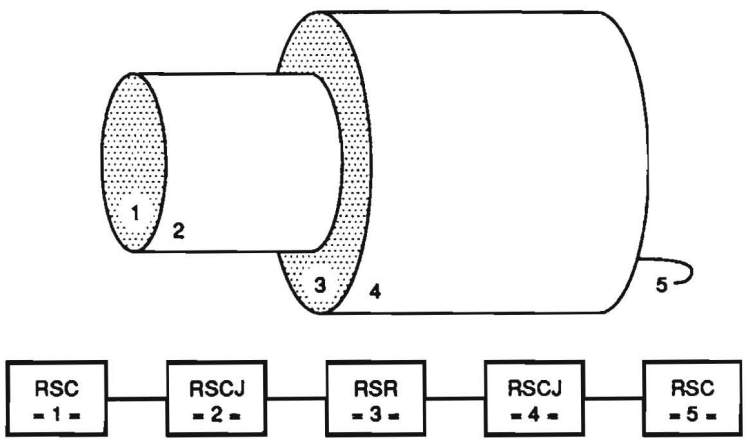


Abbildung 4.1: Eine einfache Welle und der dazugehörige Graph

Werkstücke werden als Graphen repräsentiert, dabei stehen die Knoten i.allg. für elementare Flächen (wie Zylindermantelflächen, Rechteckflächen), die Kanten für Nachbarschaftsbeziehungen zwischen den Flächen. Abb. 4.1 zeigt ein einfaches Werkstück (eine Welle), und den zugehörigen Graph. Nicht dargestellt sind hier die *Attribute*, die zu den Flächen noch Angaben wie Radius, Länge und Oberflächenbeschaffenheit machen.

Die Feature werden durch Produktionen einer Graph–Grammatik repräsentiert. Die rechte Seite einer Produktion beschreibt jeweils ein Feature, die Menge aller Feature (also die Grammatik) repräsentiert das eingegebene Wissen eines Experten. Als lesbarer Text wird diese Grammatik in der dafür maßgeschneiderten Sprache FEATREP dargestellt.

Ein Beispiel[1] für ein (geometrisches) Feature zeigt Abb. 4.2. Das Feature *Schulter* wird hier durch zwei benachbarte Flächen, eine *Langdrehfläche* und einen *Ring*

[1] aus der Grammatik für rotationssymmtrische Werkstücke '`turning.feat-rep`' (vereinfacht).

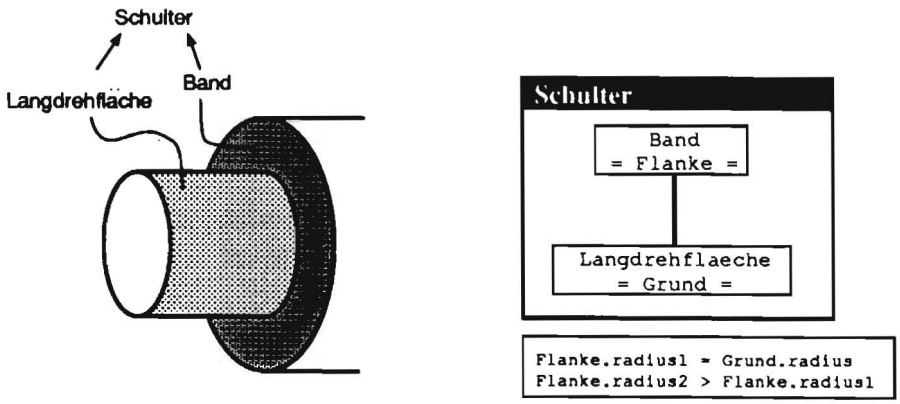


Abbildung 4.2: Beispiel für ein einfaches Feature

charakterisiert. Außerdem muß durch entsprechende Bedingungen festgelegt werden, daß der Innenradius des Ringes gleich dem Radius der Langdrehfläche und der Außenradius des Ringes größer dem Innenradius sein müssen. Langdrehfläche und Ring haben in diesem Beispiel Namen (*'Label'*) erhalten, die im nächsten Beispiel verwendet werden.

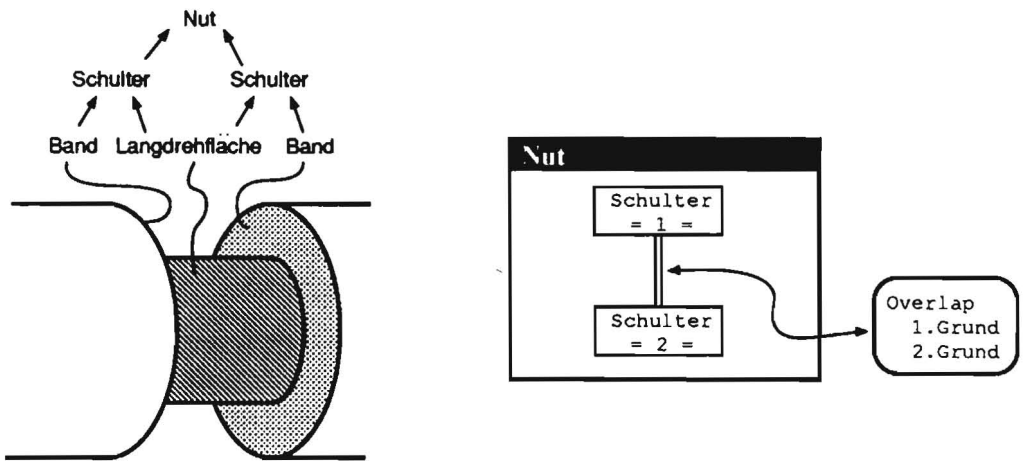


Abbildung 4.3: Beispiel eines Features mit überlappenden Flächen

Ein weiteres Beispiel (Abb. 4.3) zeigt, wie Feature zur Definition neuer Feature genutzt werden können. Zwei Schultern wie aus Abb. 4.2 definieren eine *Nut*, wenn sich die beiden Schultern mit ihrer mit *Grund* bezeichneten Fläche überlappen. *Überlappung* heißt, daß es sich um *identische* Flächen handeln muß, d.h. die beiden Schultern 'teilen' sich den Grund.

Für ein und dasselbe Feature gibt es in der Regel mehrere Definitionen (z.B. wird es in einer Feature-Beschreibung für Drehteile verschiedene Definitionen einer *Schulter* geben). In der entsprechenden Graph–Grammatiken gibt es dann mehrere Produktionen der gleichen Sorte (z.B. der Sorte *Schulter*).

Featurebeschreibungen für die beschriebene technische Anwendung weisen verschiedene Charakteristika auf, diese müssen in der Sprache FEAT-REP oder jeder anderen Repräsentation von Feature Berücksichtigung finden:

Dimensionsabhängigkeit: In Abhängigkeit von ihren Abmessungen können identische Strukturen unterschiedliche Feature repräsentieren. Dies macht es notwendig, zusätzliche Bedingungen (*Constraints*) in eine Featuredefinition einzufügen, und führt zu Featurebeschreibungen, die sich nur durch ihre Constraints unterscheiden.

Überlappungen: Teile von Feature können sich überlappen, d.h. einzelne Flächen können Bestandteil mehrerer, benachbarter Feature sein.

'Tiefe' Nachbarschaften: Die Nachbarschaft von Feature sollte auch 'tief' spezifiziert werden können, d.h. es sollte anzugeben sein, welche *Bestandteile* zweier Feature sich berühren.

Zahlreiche ähnliche Definitionen: In einer praktischen Anwendung wird die Regelbasis sehr groß sein. Zahlreiche Feature-Definitionen sind ähnlich zueinander. Eine gute Repräsentation sollte diese Tatsache ausnutzen, z.B. durch die kompakte Formulierbarkeit von Varianten und Spezialfällen, und durch die Einführung von Vererbung.

Sinnvoll erscheint es, eine Hierarchie aller Featuredefinitionen zu berechnen, in der aufgezeigt wird, welche Feature eine Spezialisierung anderer Feature, und welche Feature (möglicherweise unbeabsichtigt) identisch sind.

Kontextsensitivität: Einige Feature müssen in einem bestimmten Kontext stehen, d.h. sie werden nur dann erkannt, wenn sie zu bestimmten anderen Feature benachbart sind. Bespielsweise wird eine *Langdrehfläche* nur in einem bestimmten Kontext als *Nutgrund* bezeichnet.

Mehrdeutigkeit: Ein Werkstück läßt sich häufig durch syntaktisch unterschiedliche, semantisch aber häufig ähnliche Featuremengen beschreiben (Interessiert ist man dann meistens aber nur an einer Beschreibung).

Sortenhierarchie: Um die Wissensbasis zu strukturieren, und die Zahl der Featuredefinitionen zu minimieren, kann eine *isa*–Hierarchie auf den Sorten definiert werden. Mit ihr läßt sich angeben, daß jedes Feature einer bestimmten Sorte auch ein Feature einer anderen Sorte definiert (*Beispiel:* Jede linke Schulter ist auch eine Schulter).

Aus diesen Charakteristika ergeben sich die folgenden Anforderungen für eine Repräsentation von Features:

- Jede Featuredefinition definiert ein Feature einer bestimmten Sorte.
- Ein nichtleeren, zusammenhängenden Graph spezifiziert die Geometrie des Features. Die Kanten des Graphen sind *Nachbarschaften* oder *Überlappungen.*
- Nachbarschaften können auch 'tief' (s.o.) angegeben werden.
- Neben Nachbarschaften müssen *Überlappungen* kodiert werden können.
- Jedem Knoten sind Attribute zugeordnet. Berechnet werden bei der Analyse eines Graphen aber nur die in 2.1 definierten *synthetisierten* Attribute, also diejenigen der linken Seite der Produktion.

- Es gibt *Anwendungsbedingungen*, also Prädikate auf den Attributen, die die Gültigkeit einer Regel ausdrücken.
- Für das Parsen eines Graphen soll eine Standardeinbettung verwendet werden, zu der innerhalb einer Featuredefinition keine weiteren Angaben gemacht werden müssen.
- Bei der Generierung ist erforderlich, detailliertere Angaben zur Einbettung zu machen. Diese werden in den einzelnen Definitionen durch Kontextknoten spezifiziert.

4.2 Einsatz von Feature

Beim Einsatz von Features zur Beschreibung von Werkstücken gibt es zwei grundlegende Probleme:

1. Wie läßt sich ein durch Primitive wie Flächen und Nachbarschaften gegebenes Werkstück überführen in eine Beschreibung durch Features: das Problem der *Feature-Erkennung.*
2. Genauso wie sich eine Beschreibung durch Features aus den Werkstückdaten gewinnen läßt, ist auch der umgekehrte Weg denkbar: Die Erzeugung einer geometrischen Beschreibung des Werkstücks.

Im ARC-TEC-Projekt wird ein auf Graph-Grammatiken basierender Formalismus verfolgt. Basierend darauf wurde im Rahmen dieses Projekts die Repräsentationssprache TEC-REP[2] [BKL 91a] entwickelt: Werkstücke werden als attributierte Graphen repräsentiert, dabei kodieren die Knoten einzelne Flächen, die Knotensorte bestimmt den Flächentyp (wie *Rechteck, Zylindermantel*). Jedem Knoten sind Attribute zugeordnet, die geometrische und technologische Informationen (wie Radius, Länge, Oberflächengüte etc.) liefern. Die Kanten des Graphen kodieren die Topologie des Werkstücks: Zwei Knoten sind benachbart, wenn sie durch eine Kante miteinander verbunden sind.

4.2.1 Parsen von Werkstückdaten

Ein **Parser**[3] löst das sogenannte **Wortproblem** [Zim 82]. Es ist das Problem, für ein Wort x und eine Grammatik G festzustellen, ob $x \in L(G)$, und falls dieses der Fall ist, einen Ableitungsbaum zu finden.

Für unser Problem heißt dies: Es ist festzustellen, ob sich ein Werkstück durch die verwendete Feature-Grammatik vollständig erkennen läßt. Eine erfolgreiche Analyse liefert (über den Ableitungsbaum) eine Beschreibung des Objektes durch Features.

Ein Bottum-Up Parser für Graphen geht im Prinzip nach folgendem Algorithmus vor (angelehnt an [ASU 86]): Durch Reduktionsschritte wird der Graph in einen

[2]Eine B-Rep Darstellung, d.h. nur die Außenflächen des Körpers werden repräsentiert.
[3]deutsch: Analyseprogramm

Zielknoten überführt. In jedem dieser Schritte wird ein Teil des Graphen, der der Beschreibung durch die rechte Seite einer Produktion entspricht durch den Knoten der linken Seite dieser Produktion ersetzt. Falls in jedem Reduktionsschritt der geeignete Teilgraph ersetzt wird, erhält man schließlich einen Zielknoten und damit einen Parse des Graphen.
Eine Implementierung eines Parsers muß aber auch die folgenden Probleme lösen: Nicht jeder ausgeführte Reduktionsschritt führt zu einer Lösung. Bei Benutzung einer umfangreichen Grammatik kann der Suchraum sehr groß werden ('Kombinatorische Explosion').

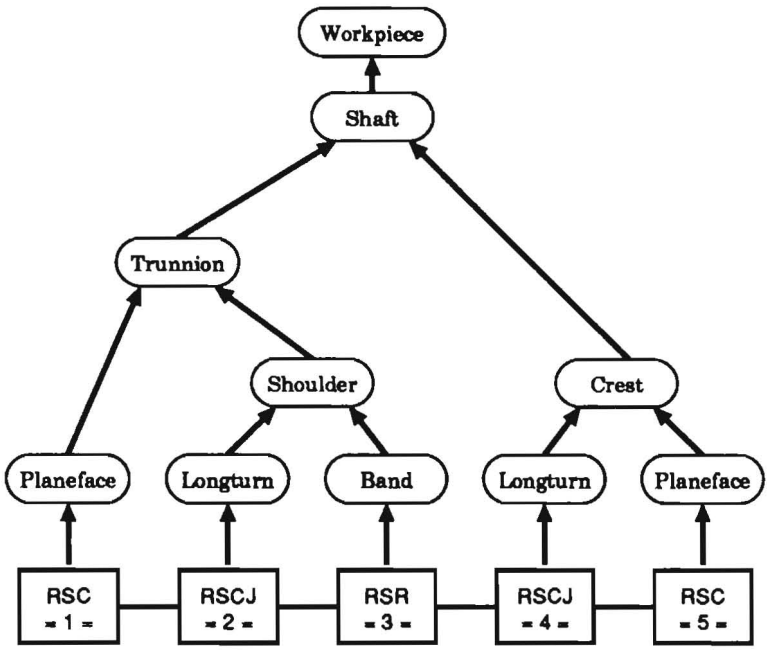


Abbildung 4.4: Featurebaum einer einfachen Welle.

Auch ein Parser muß die speziellen Charakteristika der Feature berücksichtigen können. Im ARC-TEC-Projekt wurde daher ein Parser für attributierte Graph-Grammatiken, GraPaKL[4] [Mau 92], entwickelt. Als *Graph*-Parser kann er nicht nur für Drehteile, sondern auch für Frästeile[5] eingesetzt werden.

4.2.2 Generierung von Werkstückdaten

Bei der Generierung wird versucht, an Hand von Featurebeschreibungen die Geometrie und Topologie eines Werkstücks zu erzeugen. Ensprechend der in [BKL 91a] beschriebenen Repräsentation erhält man einen Graph, dessen Knoten die Flächen des Werkstücks und dessen Kanten die Nachbarschaften kodieren.

Ein Generator arbeitet ähnlich dem in 2.2 beschriebenen LEARRE-Algorithmus: ausgehend von einem Startknoten werden Knoten expandiert, d.h. durch die rechte

[4]Graph Parser KaiserLautern

[5]Werkstücke, die sich durch Bohr–, Säge– und Fräsoperationen aus einem quaderförmigen Rohling herstellen lassen.

Seite einer Produktion ersetzt. Dieses wird solange wiederholt, bis der generierte Graph nur aus Knoten besteht, deren Sorten Terminale sind.

Die Implementierung eines Generators ist im ARC-TEC Projekt bisher nur geplant [BKLSS 92]. Die in dieser Arbeit beschriebene Repräsentation von Feature sollte aber genauso für die *Generierung* von Werkstückdaten geeignet sein, wie sie es für das *Parsen* ist. Es wurde daher darauf geachtet, daß es zumindest prinzipiell möglich ist, jede Feature-Grammatik sowohl zum Parsen wie auch zum Generieren benutzten zu können.

Ein Algorithmus zur Generierung eines Werkstückgraphen muß Überlappungen, 'tiefe' Nachbarschaften, Kontextknoten sowie Sortenhierachien berücksichtigten. *Anwendungsbedingungen* und *Attributsberechnungsvorschriften* sollten ebenfalls ausgewertet werden.

Die genaue Semantik der Kontextknoten bei der Generierung ist noch nicht geklärt. Die folgende Definition sollte aber ausreichend sein:

> Ein 'Kontextknoten' ist ein spezieller Knoten der rechten Seite einer Produktion, der eine Liste von Sorten spezifiziert. Ein Generierungs-Algorithmus verwendet ihn, um die Einbettung eines Tochtergraphen festzulegen.

Eine Charakteristik unserer Anwendung ist, daß bei der Expandierung eines Knotens keine Kanten 'verloren' gehen. Jeder Nachbar des expandierten Knotens ist auch mit (mindestens) einem Knoten des Tochtergraphen benachbart. Hätte die Anwendung einer Produktion zur Folge, daß diese Forderung nicht erfüllt würde, könnte sie der Generierungs-Algorithmus nicht verwenden.

Beispiel

Das Beispiel illustriert die Verwendung von Kontextknoten. Die Menge der Terminale sei T={RSC, RSCJ, RSR }, die der Nonterminale sei N={ Shaft, Trunnion, Shoulder, Crest, Planeface, Longturn, Band}. Der Startknoten Z ist 'Shaft'[6]. Die Produktionen sind in Abb. 4.5 graphisch dargestellt.

Abb. 4.6 zeigt die Expansion des Startknotens 'Shaft'. Da die Grammatik keine Überlappungen oder 'tiefe' Nachbarschaften enthält, wurde eine Darstellungsart gewählt, die den entstehenden Graph zeigt.

Ausgehend von 'B' konnte nur *Trunnion* expandiert werden, da nach Anwendnung der Produktion für *Crest* der Graph nicht mehr zusammenhängend gewesen wäre[7]. In Graph 'C' konnte sowohl *Shoulder* als auch *Crest* expandiert werden, wodurch 'D' bzw. 'E' generiert wurden. Von 'F' nach 'G' wurden fünf Produktionen anwendet, auf jeden Knoten des Graphen eine.

Fügt man der Grammatik aus Abb. 4.5 die einzelne Regel aus Abb. 4.7 zu, könnten auch beliebig große Graphen erzeugt werden. Allerdings ist dieses in unserer Anwendung semantisch nicht sinnvoll, da es keine beliebig detaillierten Werkstücke geben kann.

[6] engl. für Welle.

[7] Begründet ist dies in den nicht ausreichend spezifizierten Kontextknoten.

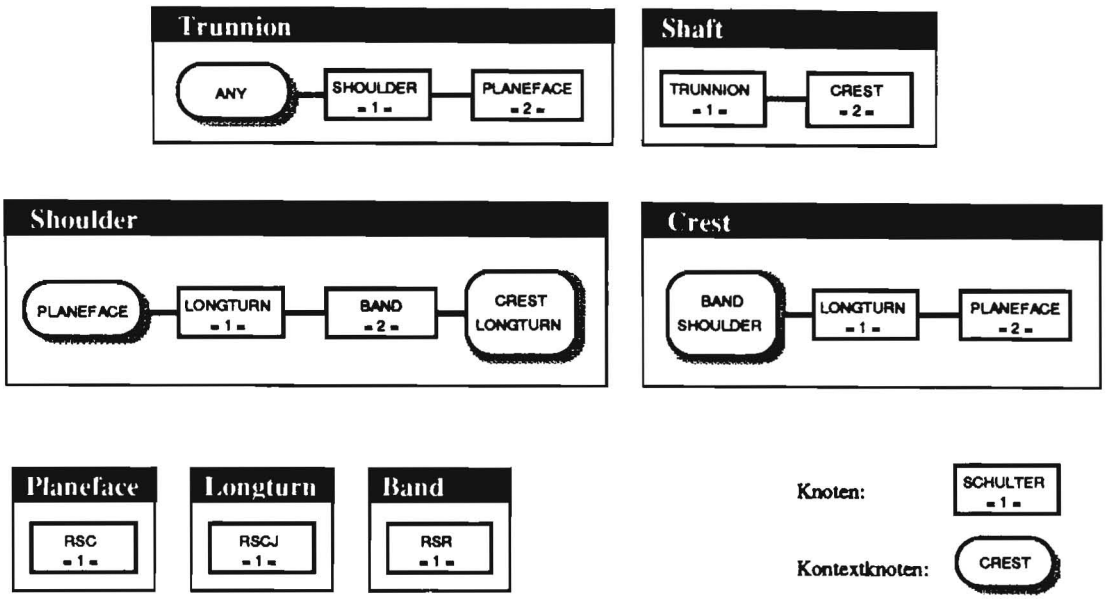


Abbildung 4.5: Grammatik zum Beispiel

4.3 Verwendung von Feature in der Literatur

In der Literatur finden sich verschiedene Ansätze, Feature in KI-basierten Systemen einzusetzen. Zwei Einsatzgebiete lassen sich erkennen: Zum Einen die Entwicklung von intelligenten CAD-Systemen, die featurebasiertes Design erlauben, zum Anderen die automatische Arbeitsplangenerierung, bei der aus der CAD-Darstellung eines Werkstücks Feature extrahiert werden, und so dessen Fertigung geplant werden kann.

Auch in [JoCh 88] werden Features eingesetzt, um aus den CAD-Daten eines Werkstücks eine qualitative Beschreibung zu gewinnen, die für die Automation der Arbeitsplanung verwendet werden kann. Wie in [BKL 91b] soll das Wissen zur Herstellung eines Features verwendet werden, um einen Plan zur Fertigung des Werkstücks zu generieren. Als wichtige Probleme werden die Wahl einer geeigneten Repräsentation und die Entwicklung eines Parsers gesehen.

Wie in TEC-REP [BKL 91a] wird ein Werkstück als Graph seiner Flächen repräsentiert. Allerdings sind nicht die Knoten, sondern die Kanten attributiert. Durch ein Attribut wird unterschieden, ob die durch eine Kante verbundenen Flächen einen konkaven oder konvexen Winkel bilden. Weitere Attribute sind nicht definiert. Auch die Feature werden als Graphen repräsentiert. Wie im GGD werden diese entsprechend ihrer Komplexität in eine Hierarchie eingeordnet; man verspricht sich davon Effizienz beim Parsen und eine kompakte Repräsentation.

Allerdings wird in [JoCh 88] nicht versucht, unter Verwendung der Theorie der Graph–Grammatik eine Beschreibung eines kompletten Werkstücks zu finden. Der ein Feature beschreibende Graph enthält nur Terminale, es wird kein Feature in der Beschreibung eines anderen verwendet.

Durch diesen eingeschränkten Einsatz von Features und die recht einfache Repräsentation ergeben sich mehrere Probleme. Bearbeitet werden können nur Werkstücke

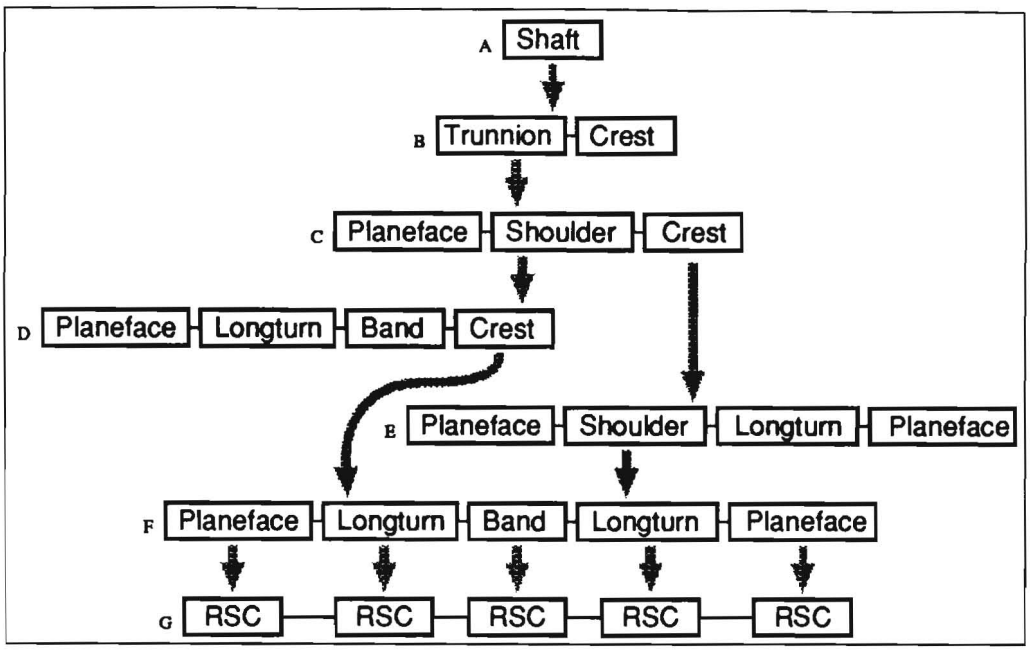


Abbildung 4.6: Beispiel für die Anwendung eines Algorithmus zur Generierung

Abbildung 4.7: Zusätzliche Regel zur Grammatik aus Abb. 4.5.

mit planaren Flächen. Andere, z.B. zylindrische Flächen sind nicht vorgesehen. Verschiedene Feature, die sich zwar in ihrer Funktion, nicht aber in ihren Graphen unterscheiden, lassen sich nicht eindeutig unterscheiden.

In [UnRa 88] wird angenommen, daß *Feature* den Schlüssel zu einer vollautomatischen Arbeitsplanung darstellen. In dem dort beschriebenen Ansatz wird aber nicht eine Beschreibung durch *Features* aus CAD-Daten gewonnen, sondern Feature werden in erster Linie vom *Designer* des Werkstücks verwendet. Die Beschreibung des Werkstücks enthält dann nicht nur geometrische Daten, sondern auch Informationen über Toleranzen und Topologie.

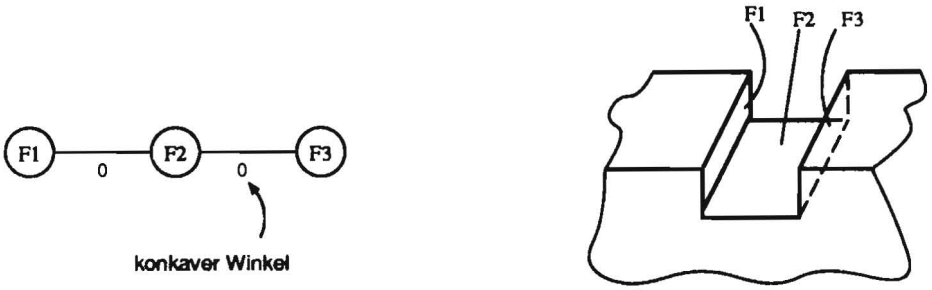


Abbildung 4.8: Eine Beschreibung des Feature 'Pocket' in [JoCh 88].

In dem Projekt wurde daher ein System entwickelt, mit dem das Design eines Werkstücks *feature*basiert erfolgen kann. Es wurden ein CAD-System und ein System zur automatischen Arbeitsplan-Erstellung miteinander gekoppelt. Der Designer erhält so direkte Hinweise zur Herstellbarkeit des von ihm entworfenen Werkstücks. In einem bereits eingegebenen CAD-Modell lassen sich Features manuell identifizieren oder bereits bekannte anzeigen und manipulieren. Alternativ läßt sich ein Werkstück direkt mit den (herstellungbezogenen) Feature konstruieren. Aus den eingegebenen Daten kann dann ein Arbeitsplan generiert werden.

Im Gegensatz zum in [BKL 91b] beschriebenen Ansatz ist nicht vorgesehen, aus dem CAD-Modell eines Werkstücks automatisch eine vollständige Feature-Beschreibung zu generieren. Es fehlt eine formale Definition von Feature, wie sie Graph–Grammatiken zur Verfügung stellen.

In [SaFi 90] und [FFPR 90] werden Feature als Abstraktionsmechanismus und Mittel zur Kommunikation zwischen Experten bezeichnet. Wie im ARC-TEC Projekt werden Graph–Grammatiken verwendet, um Feature zu beschreiben. Da auch das Werkstück als Graph repräsentiert wird, läßt sich mit einem entsprechenden Programm aus CAD-Daten eine qualitative Beschreibung durch Features erzeugen.

Die Anwendung für die Verwendung von Feature wird in der Entwicklung einer intelligenten CAD-Umgebung gesehen, weniger in der automatischen Plangenerierung. Schon während des Designs sollen die Anforderungen, die sich aus dem Material, der Herstellung oder der Wartung ergeben, berücksichtigt werden können. Jedes einzelne Tool der CAD-Umgebung verwendet seine spezifischen Feature, und hat somit eine eigene Sicht auf das Werkstück. Alle Tools arbeiten aber auf einer gemeinsamen, einheitlichen Repräsentation des Werkstücks.

Die Feature werden ähnlich wie in [JoCh 88] repräsentiert: Die Knoten sind nicht attributiert, bei den Kanten werden konvexe und konkave Knanten unterschieden.

Zur Effizienzsteigerung bei der Verwendung von Feature wird die Grammatik *compiliert.* Die Knoten werden klassifiziert nach Zahl und Art der von ihnen ausgehenden Kanten, danach entsprechend der Kardinalität der Klassen und der Zahl der Kanten in eine Ordnung eingefügt. Ziel ist es, einem Parser eine Heuristik für die Reihenfolge der zu suchenden Knoten zu liefern[8].

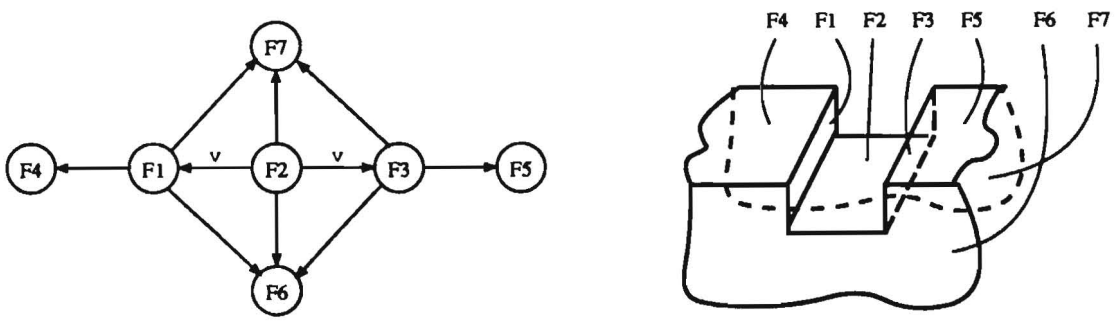


Abbildung 4.9: Beschreibung eines Features

[8]Auch im GGD werden die Knoten für den Parser in eine Reihenfolge gebracht, allerdings auch aus technischen Gründen, und nicht automatisch, sondern durch den menschlichen Experten.

Im Beispiel in Abb. 4.9 wurden die Knoten bereits in eine Ordnung gebracht, die Kanten sind in Richtung niedrigerer Priorität gerichtet. Knoten 'F2' hat die höchste Priorität, weil er der einzige seiner Art (mit je zwei konvexen und konkaven Kanten) ist. Ein Parser würde seine Suche mit diesem Knoten beginnen. Die niedrigste Priorität haben 'F4' und 'F5', weil sie jeweils nur einen Nachbarn haben.

Die Repräsentation des Wissens kann aber nicht nur Feature und die Produktgeometrie enthalten, es müssen auch Anforderungen aus der Herstellung, durch Industrienormen oder physikalische Gesetze berücksichtigt werden. In [FFPR 90] werden daher *Constraints* verwendet, diese werden aber nicht wie in [BKL 91b] einzelnen Feature zugeordnet. Die vorgeschlagenen Algorithmen überprüfen diese Anforderungen an das Design, unterstützen aber auch den Ingenieur bei der Entwicklung von Design-Alternativen.

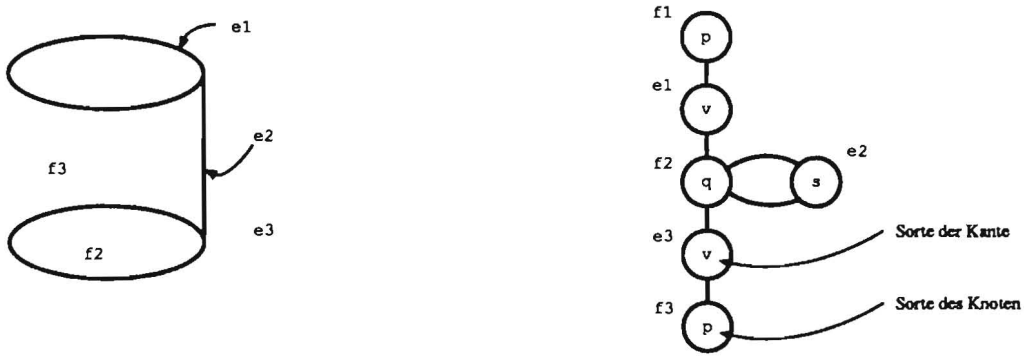


Abbildung 4.10: Zylinder und zugehöriger Web-Graph nach [ChHe 91]

In [ChHe 91] werden Web Grammatiken, eine spezielle Graph-Grammatik zur Featureerkennung verwendet. Das Werkstück und die Feature werden wieder als Graph repräsentiert, allerdings mit einem wichtigen Unterschied zu anderen Ansätzen: Nicht nur die Flächen, auch die Kanten des Werkstücks sind Knoten des Graphen. Auf diese Weise lassen sich verschiedene Kanten (wie Linie, Kurve, Kreis etc.) unterscheiden.

Besondere Aufmerksamkeit finden in [ChHe 91] ineinander geschachtelte Feature (wie die Bohrung in der Bohrung) und sich überschneidende Feature (wie sich überlappende Bohrungen). Für geschachtelte Feature werden *rekursive* Beschreibungen verwendet, bei denen das definierte Feature wieder in seine Beschreibung eingeht. Sich überschneidende Feature werden durch spezielle Feature definiert. Diese Lösungen entsprechen somit jenen, die auch in unserer Repräsentation gewählt würden. Allerdings kennt die Grammatik in [ChHe 91] keine Attribute, so daß Abmessungen, Dimensionen und Richtungen nicht berücksichtigt werden können.

Kapitel 5

Attributierte Graph–Grammatiken zur Featurerepräsentation

Graph–Grammatiken sind ein adäquater Formalismus, um Feature in einer CIM–Umgebung zu repräsentieren: Feature haben die Charakteristik der Produktionen einer Graph–Grammatik[1].

Feature lassen sich im wesentlichen durch ihre Bestandteile sowie die Nachbarschaften zwischen diesen Teilen beschreiben. Die Übertragung einer solchen Beschreibung in eine Produktion einer Graph–Grammatik ist einsichtig und technisch einfach.

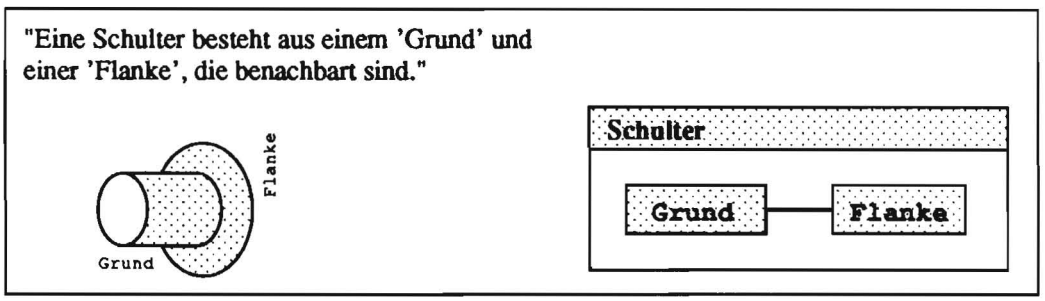


Abbildung 5.1: Analogie zwischen Featurebeschreibung und Produktion.

Da sich auch ein Werkstück durch einen Graph beschreiben läßt, kann das Problem, eine Beschreibung durch Feature zu finden oder ein Werkstück aus Featurebeschreibungen zu generieren, durch die Verwendung von Graph–Grammatiken gelöst werden.

Die theoretische Fundierung von Graph–Grammatiken erlaubt, bereits bekannte Ergebnisse und Definitionen (wie Vollständigkeit oder Termination) anzuwenden. Bestehende Algorithmen (wie Parser oder Generator) können übernommen werden.

Allerdings bereitet es Probleme, mit den bisher beschriebenen Graph–Grammatiken (siehe 2.2) alle Eigenschaften wiederzugeben, die unsere Feature auszeichnen. Vorzu-

[1]Zur Begriffsdefinition: Dem Begriff *Feature*, wie er in [BKL 91b] definiert wurde, entspricht die *Sorte* im GGD. Eine *Feature-Regel* wird hier als *Produktion* oder *Featuredefinition* bezeichnet.

ziehen wäre ein Formalismus, der den Charakteristika der Feature besser entspricht (siehe 4.1).

Um diese Anforderungen zu erfüllen, wurde eine Grammatik definiert, die im folgenden **ANCEGG**[2] genannt werden soll.

Der bei der Beschreibung von Graph–Grammatiken (2.2) verwendete Begriff *Label* diente dazu, jedem Knoten eines Graphen ein Symbol zuzuweisen. Durch dieses Symbol gab es verschiedene Arten von Knoten (die z.B. bei Graphersetzungen von Interesse waren), innerhalb einer Regel konnte es aber mehrere Knoten mit dem gleichen Label geben.
Für die Definition der ANCEGG wird der Begriff *Sorte* die Rolle des *Labels* aus 2.2 übernehmen! Als *Label* wird jener Name bezeichnet, den ein Knoten zu seiner *eindeutigen* Identifikation innerhalb einer Produktion erhält! Jedem Knoten wird in einer ANCEGG somit sowohl eine Sorte als auch ein Label zugewiesen.

Produktionen in dieser Grammatik haben grundsätzlich nur einen Knoten auf ihrer linken Seite, einen Graph mit mindestens einem Knoten auf ihrer rechten Seite. Eine (echte) Teilmenge dieser Knoten kann als *Kontextknoten* deklariert werden, die bei der Einbettung eine besondere Rolle spielen.
Der Graph auf der rechten Seite einer Produktion ist zusammenhängend. Die Knoten des Graphs sind durch zwei Arten von Kanten verbunden: *Nachbarschaften* und *Überlappungen*. Es sind sog. 'tiefe' Nachbarschaften erlaubt, bei denen angegeben wird, welche Teile der Knoten benachbart sind. Zu Überlappungen muß jeweils angegeben werden, welche Teile eines Knotens sich überlappen. Alle Kanten sind ungerichtet und nicht attributiert.

Es gibt *Attributsberechnungsvorschriften* für die Attribute der linken Seite, die sich aus Attributen der rechten und linken Seite berechnen. Die Auswertereihenfolge der Berechnungsvorschriften ist vorgegeben. Zu einer Produktion können *Anwendungsbedingungen* angegeben werden.

Sowohl beim Parsen als auch beim Generieren eines Graphen ergibt sich das Problem, ein eingesetzten Knoten bzw. Graphen in den Restgraphen einzubetten.

Beim Parsen werden von dem eingesetzten Knoten Kanten eingesetzt zu allen Knoten, zu denen auch der Muttergraph benachbart war [Mau 92]. Die Kontextknoten müssen hier nicht beachtet werden. Andere Methoden sind denkbar, erscheinen aber für unsere Anwendung nicht sinnvoll.

Beim Generieren eines Graphs sind Angaben zur Einbettung des Tochtergraphen notwendig[3]. Einige bekannte Methoden für die Einbettung wurden in 2.2 vorgestellt, die vorliegende Arbeit orientiert sich an NCE Grammatiken. Zu diesem Zweck können zu den Knoten der rechten Seite einer Produktion zusätzlich *Kontextknoten* angegeben werden. Bei der Anwendung einer Produktion wird dann der Tochtergraph durch Kanten mit Knoten des Restgraphen verbunden, deren Sorten den benachbarten Kontextknoten entsprechen (Abschnitt 4.2.2 geht näher auf die Problematik der Graphgenerierung ein).

[2] Attributed Neighbourhood Controlled Embedding Graph Grammar

[3] Es ist im allgemeinen semantisch falsch, einfach jeden Knoten des Tochtergraphen mit jedem Nachbarn des expandierten Knotens zu verbinden.

Das Sortenalphabet Σ ist durch eine Partialordung $<_\Sigma$ strukturiert, die Sortenhierarchie.

Definition 5.1 *Sei Σ eine Menge von Sorten und $<_\Sigma$ eine Partialordnung auf Σ. Dann ist $<_\Sigma$ eine* **Sortenhierarchie** *für Σ und für zwei Sorten $s_1, s_2 \in \Sigma$ ist s_1 eine* **Subsorte** *von s_2, falls $s_1 <_\Sigma s_2$.*

Für eine ausgezeichnete Sorte namens **ANY** $\in \Sigma$ *gilt $\forall x \in \Sigma : x <_\Sigma ANY$.*

Definition 5.2 (ANCEGG)
Eine ANCEGG[4] wird definiert als $GG = (N, T, P, S, <_\Sigma, L)$. Dabei ist N die nichtleere, endlich Menge der Nonterminalsymbole, T die nichtleere, endlich Menge der Terminalsymbole, und $S \in N$ das Startsymbol, auch Axiom genannt. Sei $V = T \cup N$. P ist die Menge der Produktionen. $<_\Sigma$ ist die oben definierte Partialordung auf Σ. L die Menge der Label, die zur Identifizierung der Knoten dienen.

Definition 5.3 *Eine* **Produktion** *$p \in P$ ist definiert als $p = (\alpha, \beta, A, R, B)$.*
α steht für die linke Seite der Produktion und ist ein einzelner Knoten $\in N$, β für die rechte Seite der Produktion, einem Graph über V, genannt Spezifikation. *Sei $V(\alpha)$ die ein-elementige Menge der Knoten der linken Seite, $V(\beta)$ die Menge der Knoten der Spezifikation.*

$A = \bigcup_x A(x)$ ist eine endliche Menge von Attributen, wobei $A(x)$ die Menge der Attribute eines $x \in V(\alpha) \cup V(\beta)$ ist. R ist die Menge der Attributsberechnungsvorschriften, und B die der Anwendungsbedingungen.

Ein $r \in R$ ist eine Funktion $k \leftarrow f(l_1, \ldots, l_n)$, wobei $k \in A(x), x \in V(\alpha))$ und $l_i \in A(y), y \in V(\beta)$ ist. Ist eine Funktion f nicht berechenbar, kann die Produktion nicht angewendet werden.

Jedes $b \in B$ ist ein Prädikat, es ist definiert als $P(l_1, \ldots, l_n) \rightarrow \{\texttt{true}|\texttt{false}\}$, mit $l_i \in A(x), x \in V(\beta)$. Liefert ein b `false`*, so kann diese Produktion nicht angewendet werden.*

Definition 5.4 *Eine* **Spezifikation** *β in einer Produktion p ist definiert als $\beta = (V, C, E_N, E_{\bar{U}}, \sigma, \xi)$.*
V ist die Menge der Knoten $= V(\beta)$, C die der Kontextknoten. E_N ist die Menge der Nachbarschaften, $E_{\bar{U}}$ die der Überlappungen.
$\sigma : V \cup C \rightarrow \Sigma$ ordnet jedem $x \in V \cup C$ eine Sorte zu. Die Bijektion $\xi : V \cup C \rightarrow L$ ordnet jedem $x \in V \cup C$ ein Label zu, mit $x \neq y \Leftrightarrow \xi(x) \neq \xi(y)$ für alle $x, y \in V \cup C$.

Eine Pfadspezifikation *sei definiert als $(l_0, l_1, \ldots, l_m)$ mit $l_0 = \xi(v), v \in V \cup C$, $l_i \in L$, $0 \leq m < \infty$. Eine Pfadspezifikation ist also der Label eines der Knoten des Graphen, gefolgt von einer (möglicherweise leeren) Sequenz beliebiger anderer Labels.*

Jedes Element von E_N und $E_{\bar{U}}$ ist ein Paar von Pfadspezifikationen.

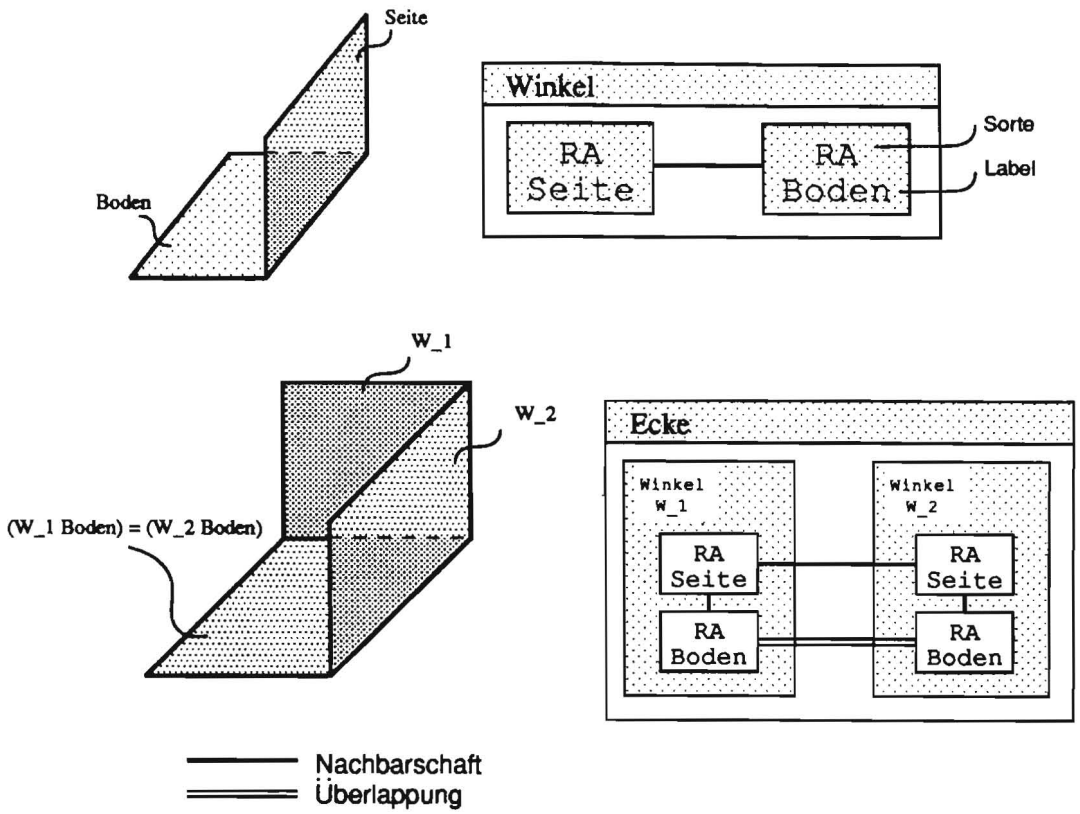


Abbildung 5.2: Zwei Featuredefinitionen, die zweite mit 'tiefer' Nachbarschaft und Überlappung

Zu jeder Überlappung und Nachbarschaft wird durch die Pfadspezifikation angegeben, welche Komponenten eines Knoten sich überlappen bzw. benachbart sind, sie werden dabei durch die Label benannt.

Um den Einsatz von Pfadspezifikationen zu illustrieren, zeigt Abb. 5.2 zwei Featuredefinitionen. Die beiden 'Winkel', die die 'Ecke' bilden, sind nur mit ihren Knoten 'Seite' benachbart. Damit wäre $E_N = ((W\_1\ Seite)(W\_2\ Seite))$. Analog ist $E_{\bar{U}} = ((W\_1\ Boden)(W\_2\ Boden))$.

Label werden also zur Darstellung von Überlappungen und 'tiefen' Nachbarschaften, aber auch in Anwendungsbedingungen und Attributsberechnungsvorschriften zur Identifizierung von Knoten verwendet.

Beispiel einer ANCEGG

Es soll eine Grammatik spezifiziert werden, die 'Rechtecke', 'Winkel', 'Rechter_Winkel' und 'Ecken' kennt. Ein Winkel besteht aus zwei benachbarten Rechtecken; eine Ecke setzt sich aus zwei 'Rechten_Winkeln' zusammen, von denen sich jeweils ein Rechteck überlappt, die beiden anderen Rechtecke sind benachbart. Die Grammatik hat drei Produktionen, für 'Winkel', 'Rechter_Winkel' und 'Ecke'. Auf Attribute und deren Berechnung wird hier verzichtet.

[4]Genaugenommen müßte es A 1-NCE GG heißen, da jede Produktion nur einen Knoten auf ihrer linken Seite aufweist.

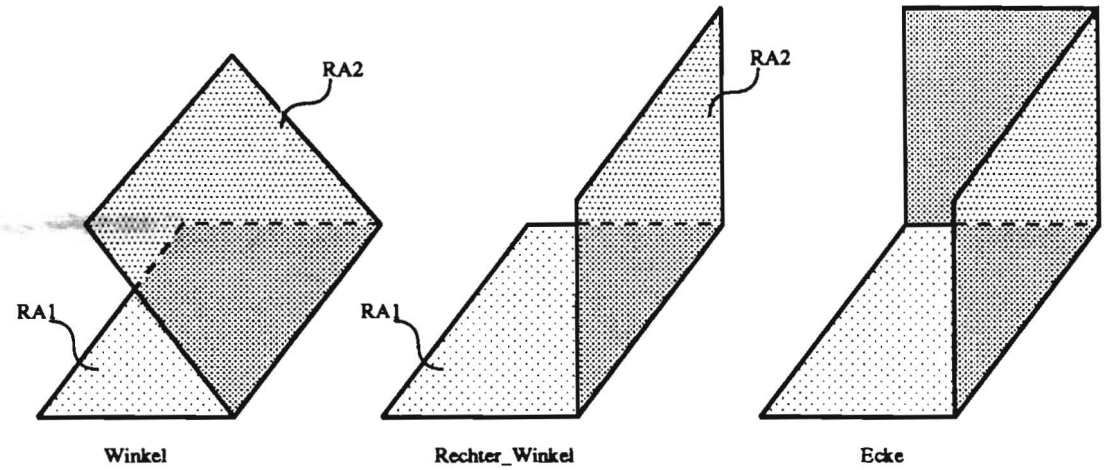


Abbildung 5.3: Die Elemente der Beispielgrammatik.

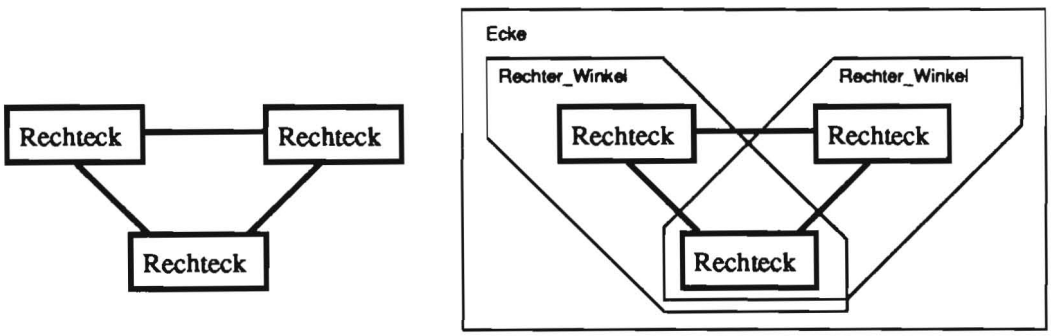


Abbildung 5.4: Graph einer 'Ecke' und seine Struktur.

Es ist $N = \{Winkel, Rechter\_Winkel, Ecke\}$, $V = \{Rechteck\}$, $Z = Ecke$[5]. Es sei $Rechter\_Winkel <_\Sigma Winkel$, jeder 'Rechte_Winkel' ist also auch ein 'Winkel'.

1. *Produktion:* $\alpha = Winkel.\ A = \emptyset.\ R = \emptyset.\ B = \emptyset.$
   Die Spezifikation β: $V = \{x_1, x_2\}$, $\sigma(x_1) = Rechteck$, $\sigma(x_2) = Rechteck$, $L = \{RA1, RA2\}$, $\xi(x_1) = RA1$, $\xi(x_2) = RA2$, $E_N = \{((RA1), (RA2))\}$, $E_{\tilde{U}} = \emptyset$.
2. *Produktion:* $\alpha = Rechter\_Winkel.\ A = \emptyset.\ R = \emptyset.$
   $B = \{rechtwinklig(RA1, RA2)\}$. Die Spezifikation β: $V = \{x_1, x_2\}$, $\sigma(x_1) = Rechteck$, $\sigma(x_2) = Rechteck$, $L = \{RA1, RA2\}$, $\xi(x_1) = RA1$, $\xi(x_2) = RA2$, $E_N = \{((RA1), (RA2))\}$, $E_{\tilde{U}} = \emptyset$.
3. *Produktion:* $\alpha = Ecke.\ R = \emptyset.\ B = \emptyset.\ A = \emptyset.$
   Die Spezifikation β: $V = \{x_1, x_2\}$, $\sigma(x_1) = Rechter\_Winkel$, $\sigma(x_2) = Rechter\_Winkel$, $L = \{Winkel1, Winkel2\}$, $\xi(x_1) = Winkel1$, $\xi(x_2) = Winkel2$, $E_N = \{((Winkel1\ RA1), (Winkel2\ RA1))$[6]$\}$, $E_{\tilde{U}} = \{((Winkel1\ RA2), (Winkel2\ RA2))\}$.

[5]Man beachte aber, daß eine Ecke alleine noch keinen geschlossenen geometrischen Körper beschreibt.

[6]Ein Beispiel für eine 'tiefe' Nachbarschaft.

Kapitel 6

Integration von Taxonomien

6.1 Motivation

In 4.1 wurde auch auf eine wichtige Eigenschaft der Graph–Grammatik zur Feature–Repräsentation hingewiesen: Die Zahl der Produktionen kann zwar sehr groß sein, andererseits sind aber viele der Produktionen ähnlich zueinander. Es erscheint wichtig, diese Eigenschaft auszunutzen, sei es zur Unterstützung bei der Entwicklung einer Grammatik, sei es zur Effizienzsteigerung von Programmen, die diese Grammatik verwenden.

Im System GGD geschieht das durch automatische Einordnung der Featuredefinitionen in eine Hierarchie, die deren Ähnlichkeit aufzeigt. Die errechnete Hierarchie steht dem Wissensingenieur und anderen Programmen zur Verfügung.

Im folgenden soll zunächst TAXON vorgestellt werden, dieses System wird in dieser Arbeit zur Entdeckung und Verwaltung von Ähnlichkeiten eingesetzt. Es wird dann gezeigt, warum es sinnvoll ist, Ähnlichkeiten auszuwerten, wie dieses bewerkstelligt wird, und welche Rolle TAXON dabei spielt.

6.2 TAXON – Ein terminologisches Wissensrepräsentationssystem

TAXON ist ein System zur Repräsentation von terminologischem, d.h. begrifflichem Wissen. Es weist syntaktisch und semantisch Ähnlichkeiten mit KL–ONE auf, und steht somit in der Tradition semantischer Netze und Frames.

Begriffliches Wissen läßt sich in Form von *Rollen* und *Konzepten* darstellen. Außerdem können Aussagen (*Assertion*) und deren Zusammenhang zum begrifflichen Wissen repräsentiert werden.
In TAXON ist diese Aufgabe in zwei Bereiche unterteilt: Die *T-Box* enthält das terminologische Wissen, die *A-Box* das Aussagewissen.

In der **T-Box** werden die Begriffe erklärt, die die allgemeine Struktur der Welt beschreiben. Es gibt zwei Typen von Ausdrücken, die *Konzepte* und die *Rollen*.

Der zentrale Begriff ist das *Konzept.* Ein Konzept ist eine einstellige Relation, eine Beschreibung der Objekte der Welt.
Beispiel: Konzepte wären 'Zylinder', 'Winkel', oder 'Nut'.

Rollen sind zweistellige Relationen, die Beziehungen beschreiben, die zwischen den Objekten bestehen.
Beispiel: Die Rolle 'Bestandteil-sein' setzt Konzept in Beziehung zu anderen, komplexen Konzepten.

Konzepte lassen sich in *primitive* und *komplexe* Konzepte unterscheiden.
Mit primitiven Konzepten werden grundlegende Begriffe repräsentiert, die als nicht weiter definierbar angenommen werden.
Beispiel: 'Winkel', 'rechtwinklig'.

Komplexe Konzepte werden aus primitiven bzw. bereits definierten Konzepten durch Kombination und Restriktion aufgebaut (mit Hilfe von Rollen und konzeptformenden Operatoren). Der T-Box–Formalismus wird daher auch als *Konzeptbeschreibungssprache* bezeichnet.
Beispiel: Ein 'Rechter-Winkel' ist ein 'rechtwinkliger' 'Winkel'. Ein 'Nutgrund' ist eine 'Langdrehfläche', der zu einer 'Nut' in der 'Bestandteil-sein'–Beziehung steht.

Die so definierten Konzepte werden in eine Begriffshierarchie eingeordnet, genannt *Taxonomie.* Diese wird von TAXON berechnet und verwaltet. In dieser Hierarchie steht ein Konzept über dem anderen, wenn ein Konzept ein anderes *subsumiert*, es ist also eine Instanz des einen immer auch eine Instanz des anderen Konzepts.
Beispiel: 'Nutgrund' wird von 'Langdrehfläche' subsumiert, jeder 'Nutgrund' ist per se auch eine 'Langdrehfläche'.

Eine Ergänzung von TAXON gegenüber KL-ONE liegt in der Möglichkeit, *konkrete Bereiche* zu definieren. Prädikate können dann einen konkreten Bereich referenzieren und deren Objekte und Operatoren benutzen. In technischen Anwendungen ist es sinnvoll, auf diese Weise Zugriff auf reele Arithmetik zu haben. Bespielsweise lassen sich dann Radien vergleichen oder Abstände überprüfen.
Beispiel: Jeder 'Kegel' habe zwei Radien 'Rad-1' und 'Rad-2'. Eine 'Kegel' ist genau dann ein 'Zylinder', wenn (= 'Rad-1' 'Rad-2') gilt.

In der **A-Box** kann ein konkreter Zustand der Welt beschrieben werden. Mit Hilfe von *Assertions* (Zusicherungen) instantiiert man die in in der T-Box definierten Rollen und Konzepte.

Es können Instanzen eines Konzepts definiert werden, d.h. es werden Objekte erzeugt, die durch dieses Konzept beschrieben werden. Es können aber auch Instanzen von Rollen definiert werden, diese setzten zwei Objekte bezüglich einer in der T-Box definierten Rolle in Verbindung.
Beispiel: 'Teil-1' ist eine 'Nut' und 'Teil-2' eine 'Langdrehfläche'. 'Teil-2 ist 'Bestandteil' von 'Teil-1'. TAXON kann dann selber feststellen, daß 'Teil-2' ein 'Nutgrund' ist.

6.3 Taxonomien und Ähnlichkeiten von Features

Im System GGD wird die Ähnlichkeit von Produktionen der Grammatik ausgenutzt. Informal gesagt, werden zwei Produktionen als ähnlich angesehen, wenn ihre rechten Seiten eine ähnliche Struktur haben, d.h. die Zahl und Sorten ihrer Knoten sind gleich oder ähnlich, die Kanten verbinden die Knoten zu ähnlichen Graphen, und die *Bedingungen* zur Anwendung sind ähnlich.

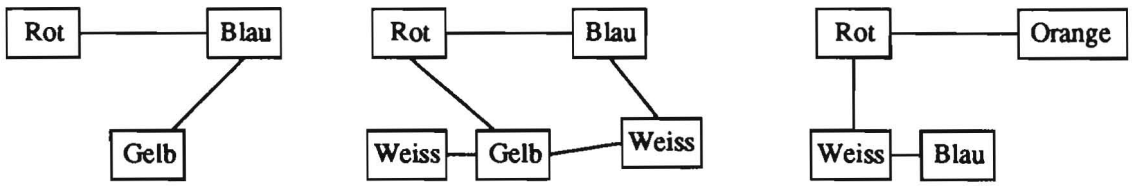


Abbildung 6.1: Drei Graphen, A und B sind ähnlich zueinander, nicht jedoch C.

Um den Begriff der Ähnlichkeit handhabbar zu machen, wird eine Partialordnung $\leq_E$ auf den Produktionen definiert.

Durch diese Ordnung läßt sich eine Hierarchie auf den Feature aufbauen, in der p_1 unter p_2 steht, wenn $p_1 <_E p_2$ gilt. Es ist kein spezielles TOP-Element[1] definiert, das größer als alle anderen Elemente ist.

Wann ist nun eine Produktion eine Erweiterung einer anderen? Ziel der Definition war es, eine Relation zu finden, die der nicht näher definierbaren Einteilung $<_{Experte}$ durch einen Experten möglichst nahe kommt.
Gesucht war eine Ordnung $<_E$ für die gilt: $p_1 <_{Experte} p_2 \rightarrow p_1 <_E p_2$. Ideal, aber wohl nicht zu erreichen, wäre diese Ordnung, wenn gleichzeitig $p_1 <_E p_2 \rightarrow p_1 <_{Experte} p_2$ gelten würde.
Es wurde demnach versucht, die Ordnung möglichst nahe an die Einschätzung durch den Experten zu approximieren, ohne $p_1 <_{Experte} p_2 \rightarrow p_1 <_E p_2$ zu verletzen.

Zwei wesentliche Merkmale einer Produktion sind der *Graph*, der durch die Spezifikation kodiert wird, und die Menge der *Bedingungen*. Diese beiden Merkmale werden zur Definition der Ähnlichkeit herangezogen.

Definition 6.1 *p_1, p_2 seien zwei Produktionen einer ANCEGG. Sei $p_1 \leq_G p_2$ eine Partialordnung auf den Graphen, die durch die Spezifikationen dieser Produktionen festgelegt sind. Sei $p_1 \leq_C p_2$ eine Partialordnung auf den Bedingungen der Produktionen.*

Für zwei Produktionen $p_1, p_2 \in P$ gelte $p_1 \leq_E p_2$ genau dann, wenn $p_1 \leq_C p_2 \wedge p_1 \leq_G p_2$. Es sei $p_1 <_E p_2$, gdw. $p_1 \leq_E p_2 \wedge (p_2 \leq_E p_1)$.

Es gibt zahlreiche Möglichkeiten, auf Graphen eine Partialordnung zu definieren, wie es durch $\leq_G$ geschehen soll. So könnte man einen Graphen als kleiner bezeichnen, wenn er ein Teilgraph eines anderen Graphen ist. Allerdings ist diese Relation für

[1] Wie 'THING' in KL-ONE.

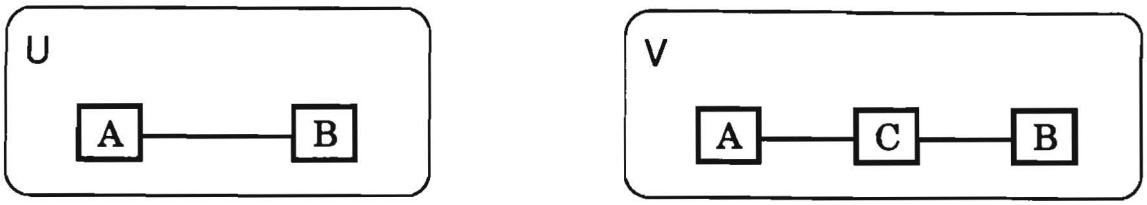


Abbildung 6.2: Zwei einfache Graphen, für die $U \leq_G V$ gelten soll.

unsere Zwecke zu restriktiv. Beispielsweise wären die beiden Graphen in Abb. 6.2 nicht vergleichbar, weil im rechten Graphen die Kante zwischen 'A' und 'B' fehlt.

Eine weitere naheliegende Möglichkeit der Definition der Relation $\leq_G$ ist der Test, ob sich der 'kleinere' Graph durch Zufügen von Kanten und Knoten zum 'größeren' Graphen erweitern lässt. Dazu soll der Begriff der *Erweiterung* definiert werden.

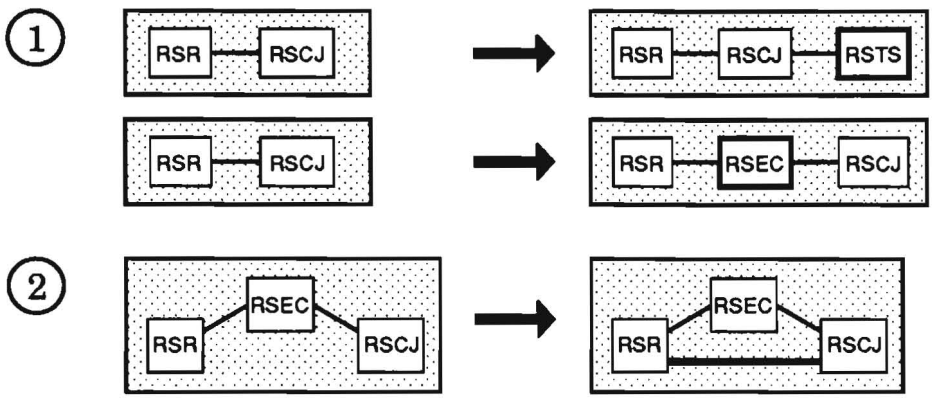


Abbildung 6.3: Verschiedene *Erweiterungen* von Graphen

Definition 6.2 *Ein Graph gilt als* **Erweiterung** *eines anderen, wenn er durch eine Folge dieser Operationen aus dem anderen Graph generiert werden könnte:*

1. *Das Einfügen eines zusätzlichen Knotens mit einer Kante, die diesen mit dem Restgraph verbindet. Ein Sonderfall ist das Teilen einer bestehenden Kante durch einen neuen Knoten am Punkt der Teilung.*
2. *Das Hinzufügen einer zusätzlichen Kante.*

Definition 6.3 *Für Produktionen $p_1, p_2 \in P$ gelte $p_1 \leq_G p_2$ genau dann, wenn der Graph der rechten Seite von p2 eine Erweiterung des Graphen von p1 darstellt.*

Durch diese Definition sind diejenigen Graphen vergleichbar, die unserer Einschätzung nach auch von einem Experten als ähnlich eingestuft würden.

Abb. 6.4 zeigt ein Beispiel aus dem CIM-Bereich: Eine Hierarchie verschiedener *Schultern* für Drehteile. Links wird jeweils eine schematische Zeichnung der entsprechenden Schulter angegeben, rechts deren Darstellung als Graph. *Shoulder-4* ist ein Beispiel für einen Graph, der eine Erweiterung mehrerer Graphen darstellt.

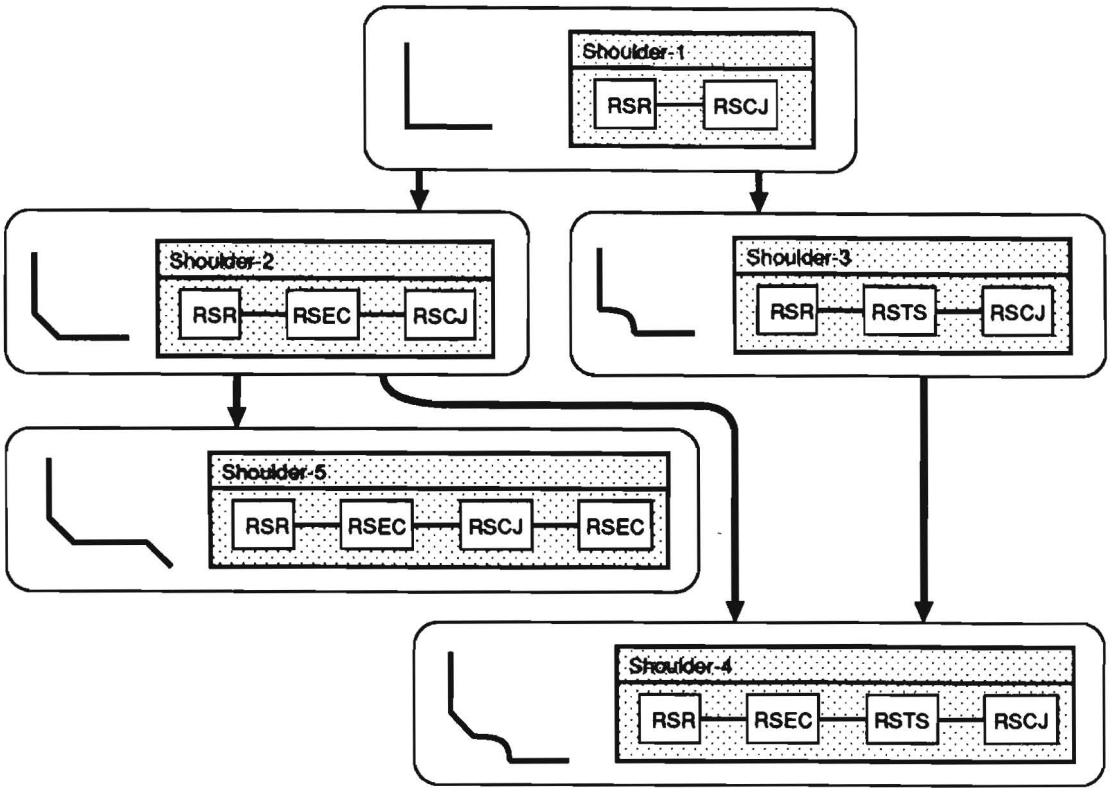


Abbildung 6.4: Eine Hierarchie von Graphen der Sorte 'Schulter'.

Die Definition einer Partialordnung $c_1 \leq_C c_2$ auf den Bedingungen von Produktionen stellt ein Problem dar, für das in dieser Arbeit keine befriedigende Lösung gefunden werden konnte. Eine solche Ordnung wäre aber fast von gleichem Interesse wie die Ordnung über die Graphen der rechten Seiten.

Bei zahlreichen Feature ist ein großer Teil des kodierten Wissens in den Bedingungen enthalten, während der Graph sehr einfach ist (z.B. aneinandergrenzende Rechtecke, die nur unter einem bestimmten Winkel und einer bestimmten Lage als Feature erkannt werden). Nutzt man die in den Bedingungen enthaltenen Informationen nicht, schränkt das den Nutzen einer Partialordung auf allen Feature ein.

Manche Feature unterscheiden sich durch die Bedingungen, unter denen sie angewendet werden können (z.B. Dimensionen), weisen aber identische oder ähnliche Graphen auf. Ohne Betrachtung der Bedingungen ist nicht feststellbar, ob die Feature unter ähnlichen Bedingungen angewendet werden können.

Möchte man aber Bedingungen für eine Partialordnung heranziehen, reicht es nicht, nur die Syntax deren zu vergleichen. Nötig wäre, die Bedingungen auch in ihrer Semantik zu analysieren und zu verstehen. Da für die Bedingungen allgemeine LISP-Ausdrücke zugelassen sind, ist es i.allg. kaum entscheidbar, ob für zwei Bedingungen $c_1, c_2 : c1 \vdash c_2$ gilt, ob also c_2 durch c_1 bedingt wird, oder ob sie widersprüchlich sind.

Einfach realisierbar wäre ein Vergleich der verwendeten Attribute, etwa in der Form: $p_1 \leq_C p_2$, wenn die Menge der in p_1 verwendeten Attribute eine Teilmenge der in p_2 verwendeten ist. Da aber diese Relation nicht sehr aussagekräftig ist, wurde auf ihre Verwendung verzichtet.

In dem Beispiel in Abb. 6.4 gehörten alle Produktionen nur einer Sorte an. Dies läßt sich im allgemeinen Fall aber nicht sagen. Auch Produktionen verschiedener Sorten können bezüglich der Relation $\leq_E$ vergleichbar sein.

Beispielsweise könnten Schultern mit einer bestimmten Eigenschaft als Produktionen der Sorte `Complex-Shoulder` definiert werden, in der Produktions-Taxonomie würden die dennoch als Erweiterung 'normaler' Schultern eingeordnet.

Jedoch sind sortenübergreifende Abhängigkeiten nicht immer erwünscht. Es kann vorkommen, daß Produktionen unterschiedlicher Feature als ähnlich erkannt werden, ohne daß dies vom Benutzer erwünscht wäre. Das System, das die Partialordnung verwaltet sollte daher die Möglichkeit bieten, wahlweise nur die 'ähnlichen' Produktionen der **gleichen** Sorte auszugeben.

Noch besser wäre es, könnte der Benutzer bestimmte Beziehungen verbieten, also *Disjunktionen* einführen. Leider erlaubt dies TAXON in seiner aktuellen Version[2] nicht.

6.3.1 TAXON im GGD

Das Programm TAXON ist in die Graph–Grammatik–Entwicklungsumgebung GGD integriert worden. Es verwaltet dort eine Partialordnung der Produktionen. Jede neue oder geänderte Produktion wird in ein (komplexes) TAXON-Konzept umgesetzt[3], dieses Konzept wird dann an TAXON zur Klassifizierung übergeben. Allerdings erfolgt die Klassifizierung aus Implementierungsgründen nicht gemäß Definition 6.1 (siehe auch 8.4.3, Seite 57).

Der Benutzer kann jederzeit die aktuelle Taxonomie abfragen. TAXON ist so eingebunden, daß sich zu einer bestimmten Produktion alle größeren, kleineren oder äquivalenten Produktionen ermitteln lassen, wahlweise der gleichen oder einer beliebigen Sorte.

6.3.2 Ähnlichkeiten aus der Sicht des Wissensingenieurs

Die Benutzung von Taxonomien auf Features kommt dem Wissensingenieur, der die Aufgabe hat, eine Regelbasis zu entwerfen, bzw. zu verändern, in vielerlei Hinsicht zu Gute.

- Die Benutzung von Taxonomien erleichtert die Eingabe von neuen Sorten und deren Produktionen. Bei einer Sorte mit zahlreichen ähnlichen Produktionen ist es sinnvoll, mit den einfacheren Beschreibungen anzufangen, und diese dann schrittweise um kompliziertere Fälle zu ergänzen. Man erreicht so eine Beschleunigung der Eingabe und behält die Übersicht über bereits eingegebene Produktionen. TAXON dient dabei zur Kontrolle, ob neue Produktionen korrekt eingegeben und klassifiziert wurden.

[2] An der Implementierung von Disjunktionen in TAXON wird aber gearbeitet.
[3] siehe auch B.3.

- Taxonomien sind ein starkes Werkzeug, um Mängel und Fehler der Regelbasis zu erkennen. Der Wissensingenieur kann leicht erkennen, wenn
  - eine Produktion in ihrer Struktur äquivalent zu einer anderen ist. Der Verdacht liegt dann nahe, daß eine Produktion versehentlich zweimal eingegeben wurde oder daß in der Regelbasis gleiche Produktionen für namentlich verschiedene Sorten definiert wurden.
  - eine Regel in eine Hierarchie eingeordnet wird, in der dieses nicht vorgesehen war. Wenn eine Produktion fälschlicherweise einer anderen Produktion über- oder untergeordnet wurde, liegt entweder ein Eingabefehler oder ein Fehler beim Entwurf der Regelbasis vor.
  - eine Regel nicht wie erwünscht in eine Hierarchie eingeordnet wird. Auch hier kann z.B. ein Eingabefehler vorliegen.
- Taxonomien können als Mittel der Strukturierung dienen. Gerade in größen Wissensbasen ist es schwierig, den Überblick über die bereits eingegebenen Produktionen zu behalten. Die Hierarchien, die durch TAXON berechnet und verwaltet werden, geben ein gutes Bild der Struktur der Wissensbasis.

6.3.3 Ausnutzung von Taxonomien durch andere Programme

Die durch TAXON berechnete Hierarchie kommt nicht nur dem Wissensingenieur zu Gute, der eine Regelbasis eingibt oder bearbeitet. Für die meisten Programme, die auf Features in der im GGD verwendeten Form zugreifen, ist ein sinnvoller Einsatz der Hierarchien denkbar.

Ein Beispiel wäre ein Parser, der, nachdem er dabei gescheitert ist, eine bestimmte Regel zu instanziieren, es mit einer einfacheren, aber ähnlichen Regel versucht[4]. Er kann selber entscheiden, ob diese einfachere Regel der gleichen Sorte angehören muß oder nicht. Bereits gewonnenes Wissen über die Struktur des Werkstücks kann dadurch möglicherweise erneut genutzt werden.

Sinnvoll wäre die Hierarchie möglicherweise für ein Programm, das Werkstücke aus den Produktionen generiert. Es könnte die Ähnlichkeiten von Produktionen benutzen, um Entscheidungen über auszuwählende Produktionen zu treffen und so möglichst einfache oder besonders komplizierte Werkstücke zu generieren.

Auch in [JoCh 88] wird festgestellt, daß es sinnvoll ist, Features in eine Hierarchie einzuordnen, die auf deren Geometrie basiert. Als Hauptvorteil wird die Reduzierung der Zeit zur Erkennung eines Features genannt. Ein weiterer Vorteil sei die kompaktere Repräsentation, die sich aus der Vererbung von Eigenschaften einer Klasse an ihre Subklassen ergebe.

[4]GraPaKL benutzt diese Möglichkeit bisher nicht, könnte sie aber in der Heuristik zur Auswahl von Patches benutzen.

Kapitel 7

Konzeption eines Systems zur Repräsentation und Entwicklung von Graph–Grammatiken

7.1 Anforderungen und Konzept für das System GGD

Das System GGD ist ein System zur Entwicklung und Verwaltung von attributierten Graph–Grammatiken. Das System ist maßgeschneidert für die Anforderungen, die sich für die Repräsentation von Features in CIM ergeben.

Schnittstellen zu anderen Programmen und Funktionen zur Ein- und Ausgabe ermöglichen den Einsatz des GGD als eine Komponente eines CIM-Systems. Insbesondere dient es als Wissensbasis für Experten und Programme zur Generierung und Analyse von Werkstücken.

Im einzelnen gab es für die Entwicklung des GGD folgende Anforderungen:

- Repräsentation einer Graph–Grammatik (GG). Dabei ist diese GG spezialisiert für Features in CAD/CAM, d.h. die besonderen Eigenschaften von Features werden berücksichtigt.
- Bereitstellung einer Entwicklungsumgebung für Graph–Grammatiken. Diese unterstützt einen Wissensingineurs bei der Entwicklung einer Regelbasis, also einer Menge von Produktionen. Diese Unterstützung besteht u.A. aus Tests zur Konsistenz der Regelbasis und aus der Einbindung von Tools wie TAXON.
- Eine graphische Benutzeroberfläche, um Feature in einfacher Weise eingeben zu können.
- Effiziente Speicherung und Verarbeitung der Informationen, bezogen auf Speicherplatz und Zugriffszeit. Gleichzeitig soll die Repräsentation so einfach sein, daß spätere Wartungsarbeiten am Programm nicht ausgeschlossen sind.
- Eingabe und Ausgabe der Regelbasis im FEAT-REP Dateiformat.

- Unterstützung des Graph Parsers GraPaKL und anderen Programmen.

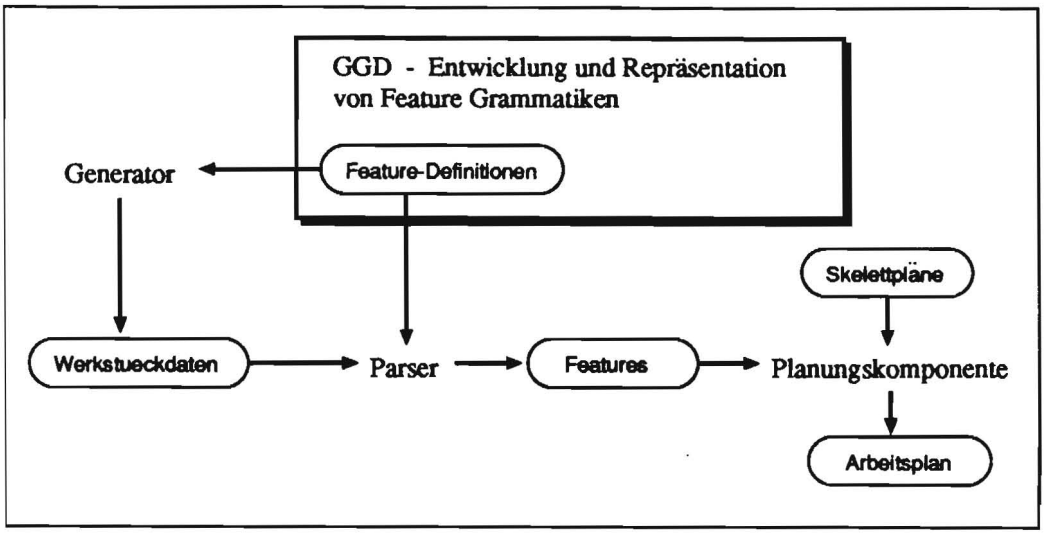


Abbildung 7.1: Einbettung des Systems GGD

Das Datenflußdiagramm in Abb. 7.1 zeigt schematisch, wie die Umgebung des GGD aussehen kann. Ein *Generator* kann unter Zugriff auf die Featuredefinitionen Werkstücke generieren. Ein *Parser* benutzt die Featuredefinitionen, um aus den geometrischen Daten eines Werkstücks eine Beschreibung durch Features zu gewinnen. Die untere Zeile in Abb. 7.1 (der Transformation der Werkstückdaten in einen Arbeitsplan) wird derzeit als ein integriertes System, PIM[1] [BKL 91c], implementiert. Der verwendete Parser ist das Programm 'GraPaKL' [Mau 92], als Planungskomponente dient 'Skippy' [Bec 91].

7.2 Komponenten des GGD

Der GGD selber setzt sich aus verschiedenen Komponenten zusammen, die jeweils bestimmte Aufgaben übernehmen. Der genaue Aufbau ist 8.1 zu entnehmen, an dieser Stelle soll nur ein grober Überblick gegeben werden, welche Komponenten vorhanden sein sollten.

Die Aufgaben der in Abb. 7.2 angegebenen Komponenten können wie folgt charakterisiert werden:

Visualisierung: Eine graphische Oberfläche, die dem Benutzer in einfacher Weise erlaubt, Graphen einzugeben, anzusehen, und zu verändern. Ein Editor zur Eingabe von Constraints ist Bestandteil dieser Komponente. Durch entsprechende Menüs können die meisten Funktionen der anderen Komponenten aufgerufen werden. Ziel war es, dem Benutzer die Entwicklung von Graph–Grammatiken möglichst einfach zu machen. GGD sollte aber auch ohne diese graphische Oberfläche benutzt werden können (beispielsweise wenn ein Drittprogramm auf den GGD zugreifen will).

[1] Planning In Manufacturing

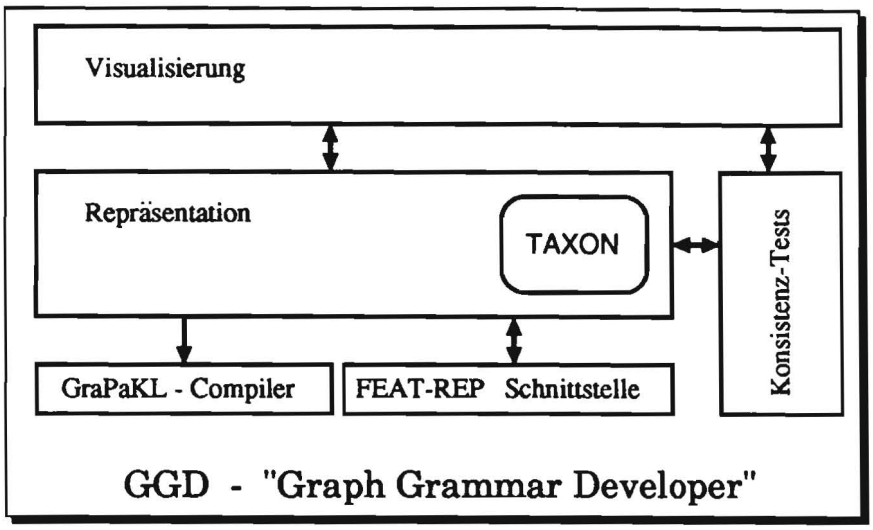


Abbildung 7.2: Aufbau des Systems GGD

Repräsentation: Die Komponente, in der das gesamte Wissen repräsentiert wird. Die Funktionen zum Zugriff und zur Modifikation der Regelbasis bilden einen wesentlichen Bestandteil. Eingebettet in diese Komponente ist das Wissensrepräsentationssytem TAXON.

Konsistenz-Tests: Unerlässlich bei der Entwicklung einer Graph–Grammatik ist die Möglichkeit, immer wieder Konsistenztests durchführen zu können. Die Konsistenz einzelner Produktionen wird, soweit möglich, zum Definitionszeitpunkt getestet, der Konsistenztest für die gesamte Regelbasis kann jederzeit aufgerufen werden. Für eine Übersicht über die einzelnen Tests siehe 7.5.1 und 7.5.2.

FEAT-REP Interface: FEAT-REP ist nach [BKL 91b] das Dateiformat, in dem alle Informationen und Daten der Features festgehalten werden. GGD ist in der Lage, Dateien dieses Formats zu erzeugen und zu lesen.

GraPaKL-Compiler: Der Graph-Parser GraPaKL verlangt die Definition einer Grammatik in einer speziellen Syntax. GGD übersetzt die in der Repräsentation-Komponente gespeicherten Informationen in Dateien, die von GraPaKL verarbeitet werden können.

7.3 Sorten, Produktionen und Hierarchien

7.3.1 Sorten

Der Begriff der Sorte wurde in Kapitel 5 eingeführt. Der **Name** einer Sorte dient zu ihrer Identifizierung, er muß daher eindeutig sein.

Wie in 4.1 erklärt, werden die Feature nach ihrer **Art** und ihrer **Anwendung** unterschieden. Diese Angaben werden zusammen mit der Sorte verwaltet[2].

[2]Art und Anwendung einer Produktion bestimmt sich somit aus der Sorte der Produktion.

Auf den Sorten ist eine Partialordnung, die Sortenhierarchie definiert. Der Benutzer soll die Möglichkeit haben, zu jeder Sorte Supersorten anzugeben. Durch Angabe von Superklassen läßt sich eine *isa*–Hierarchie aufbauen, wie man sie aus der objektorientierten Wissensrepräsenation kennt. Jede Sorte entspricht dabei einer *Klasse*. Jede Instanz (Produktion) einer Sorte ist gleichzeitig auch Instanz ihrer Supersorten.

7.3.2 Produktionen

Jede Produktion definiert genau ein Feature, Feature können nur durch Produktionen definiert werden. Eine Produktion, und damit das entsprechende Feature erhält einen eindeutigen, identifizierenden Namen. Jede Produktion gehört einer der Sorten an. Die Sorte einer Produktion bestimmt sich aus der Sorte des Knotens, der die linke Seite der Produktion bildet. GGD sollte erlauben, die Sorte einer Produktion zu ändern.

Jede Produktion kann mit einem Symbol als **Ordnungswert** versehen werden. GGD verwaltet außerdem eine Ordnung auf diesen Symbolen. Es besteht somit die Möglichkeit, eine Partialordung auf den Produktionen für eine benutzerdefinierte Heuristik anzugeben. Diese kann von anderen Programmen (wie einem Parser) verwendet werden.

Eine Produktion definiert die topologische Struktur eines Features. Die rechte Seite einer Produktion besteht aus (einem oder mehr) Knoten, die durch Nachbarschaften und Überlappungen verbunden sind. Dieser Graph muß zusammenhängend ist.
Zusätzlich können Kontextknoten definiert werden, die zu den regulären Knoten benachbart sind. Diese dienen zur Spezifizierung der Einbettung bei der Generierung eines Graphen.

Jeder Knoten der rechten Seite eines Graphen verfügt über zumindest zwei Attribute, einer **Sorte** und einer **Nummer**. Außerdem kann optional ein **Label** gesetzt werden.

Die Sorte eines Knotens muß einer der definierten Sorten entsprechen, kann aber der Sorte der Produktion entsprechen. Eine Produktion kann in ihrer rechten Seite mehrere Knoten der gleichen Sorte enthalten.
Eindeutig identifiziert wird jeder Knoten der rechten Seite einer Produktion durch die Nummer. Die Knoten werden durchgehend von 1 bis n (bei n Knoten) numeriert. Dadurch erhält man eine totale Ordnung auf den Knoten. Diese wird z.B. vom Graph-Parser GraPaKL verwendet, um die Reihenfolge der zu suchenden Knoten zu bestimmen. Die Numerierung der Knoten kann vom Benutzer geändert werden.
Der optionale Label erfüllt zwei Funktionen: Zum Ersten kann er als Beschreibung seiner Funktion innerhalb der Produktion dienen (Beispiel: *Flanke, Oberteil*). Zum Zweiten dient er als Referenz, wenn in anderen Produktionen auf Knoten der Produktion zugegriffen werden soll (z.B. durch Überlappungen). Die Knoten-Nummern können hier nicht verwendet werden, da dann eine Änderung dieser Nummern möglicherweise aufwendige Modifikationen anderer Produktionen bedeuten würden.

Je zwei Knoten der rechten Seite eines Graphen können durch eine Überlappung oder eine Nachbarschaft miteinander verbunden sein.
Nachbarschaften können auch *tief* spezifiziert werden, um zu erreichen, daß bestimmte Teile der in den Knoten benannten Features benachbart sind. Dazu werden

die Label des betroffenen Knoten angegeben.
Analog zu den tiefen Nachbarschaften spezifizeren bei *Überlappungen* Label die sich überlappenden Flächen.

Die bloße Definition der Topologie eines Feature ist im allgemeinen nicht ausreichend. Zusätzlich müssen *Attribute* berechnet und *Zusicherungen* überprüft werden. Für diesen Zweck lassen sich zu jeder Produktion *Attributsberechnungsvorschriften* und *Bedingungen* angeben, die im folgenden unter dem Begriff **Constraints** zusammengefaßt werden sollen.

Jede neue oder geänderte Produktion wird klassifiziert (siehe 6.3). Zu diesem Zweck wird sie in ein TAXON-Konzept umgesetzt, und von TAXON in die von diesem Programm verwaltete Hierarchie eingeordnet.

7.3.3 Hierarchien

Die relativ große Zahl unterschiedlicher Hierarchien mag verwirren. Es soll daher klargestellt werden, welche Hierachien verwaltet werden und wozu sie eingesetzt werden können.

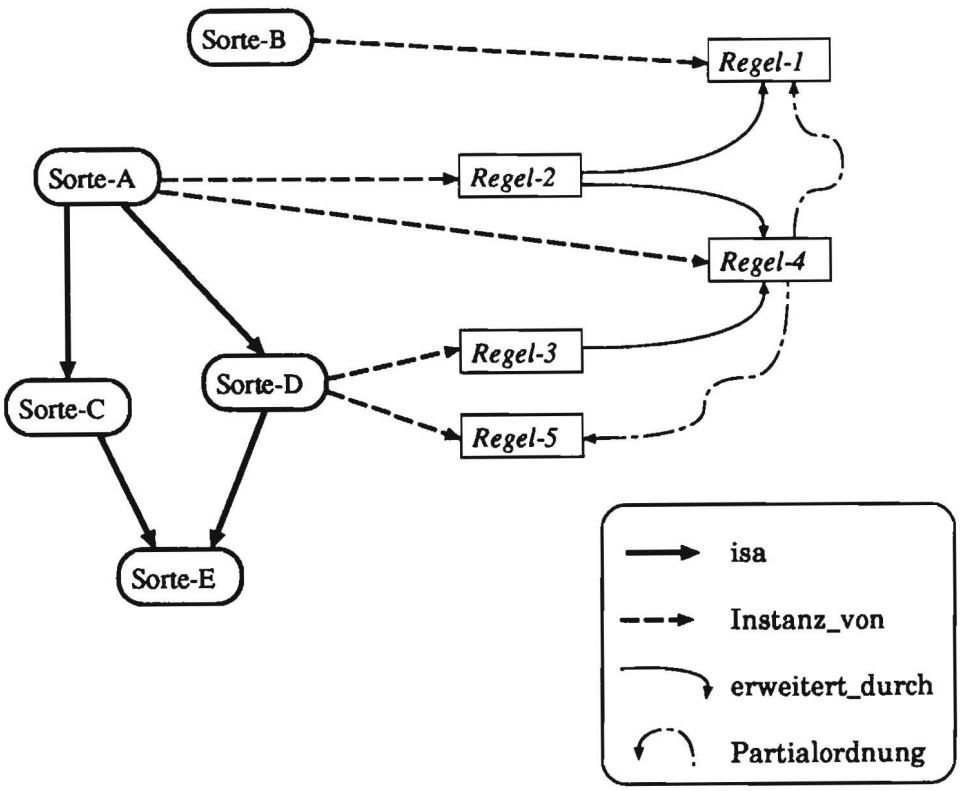


Abbildung 7.3: Durch GGD verwaltete Hierachien

1. Die Sorten sind in eine *isa*–Hierarchie eingeordnet, d.h. eine Sorte kann Sub- und Supersorten haben. Hier entsprechen Sorten den Klassen der objektorientierter Wissensrepräsentation.
2. Jede Produktion ist eine Instanz einer Sorte. Jede Produktion ist außerdem Instanz der Supersorten ihrer Sorte.

3. TAXON ordnet alle Produktionen entsprechend ihrer Ähnlichkeit zueinander in eine Hierarchie ein. In dieser Hierarchie wird eine Produktion einer anderen untergeordnet, wenn sie eine Erweiterung darstellt (siehe Kapitel 4).
4. Durch die Angabe eines Ordnungswertes läßt sich eine zusätzliche Partialordnung definieren.

Abb. 7.3 zeigt ein Beispiel, wie das Geflecht der Hierarchien aussehen könnte. Man beachte, daß die durch die Ähnlichkeit und den Benutzer definierten Ordnungen auch sortenübergreifend gelten.

7.4 Das Attributkonzept

Ein Konzept zur Verwaltung der Attribute und Label scheint erforderlich. Durch ein solches Konzept lassen sich verschiedene systemantische Fehler erkennen (z.B. Label und Attribute gleichen Namens, Label mit nicht eindeutiger Sorte). Zudem erfordert die Subsortenhierarchie ein Konzept zur Vererbung von Label und Attributen.

Im Gegensatz zur Grammatikdefinition in 5 ist im folgenden nicht nur der Name eines Labels von Interesse. Innerhalb einer Sorte und in Sortenhierarchien darf jedes Label nur Knoten einer bestimmter Sorte bezeichnen.

Attribute werden durch Attribut-Zuweisungen in den Constraints einer Produktion definiert und berechnet. Knoten der rechten Seite eines Graphen können mit *Label* versehen werden, die Label werden so in ihrer Sorte festgelegt. Label werden auch durch Rollenzuweisung (*role-assignment*, siehe A.2.3) eingeführt.

7.4.1 Label und Attribute in Produktionen

Jedes Label kann als Tupel (*Name*, *Sorte*) aufgefasst werden. Sei L_{Name} der Name, L_{Sorte} die Sorte eines Labels L. Für je zwei Label L_x, L_y einer Produktion mit $L_x \neq L_y$ muß dann $L_{x_{Name}} \neq L_{y_{Name}}$ gelten. Die Namen der Label müssen also paarweise verschieden sein.

Ein Attribut hat zwar bei der Verwendung der Features einen Wert, da aber der Wert nicht typgebunden ist, spielt er für diese Überlegungen keine Rolle. Ein Attribut sei daher einfach ein Name.

Sei $\mathcal{A}$ die Menge der Attribute, $\mathcal{L}$ die Menge der Label einer Produktion. Sei $\mathcal{L}_{\text{Name}}$ die Menge der Namen der Label in $\mathcal{L}$. Dann muß $\mathcal{A} \cap \mathcal{L}_{\text{Name}} = \emptyset$ gelten, kein Name darf also sowohl für ein Attribut wie für ein Label vergeben sein.

7.4.2 Label und Attribute in Sorten

Für zwei Produktionen S und R der **gleichen** Sorte muß gelten: Für zwei Label $L_S \in \mathcal{L}_S$ und $L_R \in \mathcal{L}_R$ gilt:

$$L_{S_{Name}} = L_{R_{Name}} \implies L_{S_{Sorte}} = L_{R_{Sorte}}$$

Label gleichen Namens müssen also in jeder Produktion auch die gleiche Sorte haben.

Sei $\mathcal{A}'$ die Vereinigung der $\mathcal{A}$ aller Produktionen dieser Sorte. Sei $\mathcal{L}'$ die Vereinigung der $\mathcal{L}$ aller Produktionen dieser Sorte. ' Dann muß $\mathcal{A}' \cap \mathcal{L}'_{\text{Name}} = \emptyset$ gelten. Der Name eines Labels in einer Produktion darf also nicht in einer anderen Produktion ein Attribut bezeichnen, und umgekehrt.

7.4.3 Vererbung von Label und Attributen bei Subsortenbeziehungen

In Subsortenbeziehungen gilt das in der objektorientierten Wissenrepräsenation übliche Verfahren: Jede Subklasse erbt alle Attribute aller ihrer Superklassen. Entsprechend gilt hier: Jede Subsorte erbt alle Attribute und Label ihrer Supersorten.

Erbt eine Klasse von zwei Superklassen, kann es zu *Vererbungskonflikten* kommen. Bei der Repräsentation von Features sind diese Konflikte möglich:

1. Ein Attribut einer der Supersorte ist ein Name eines Label der Sorte oder einer anderen Supersorte.
2. Der Name eines Label einer der Superklassen ist ein Attribut der Sorte oder einer anderen Supersorte.
3. Label gleichen Namens der Sorte und der Supersorten haben unterschiedliche Sorten.

Im GGD werden Vererbungskonflikte als Fehler betrachtet, und erfordern eine Modifikation der Wissensbasis. Es sind also keine Mehrfachvererbungen zugelassen. Dieser strenge Ansatz schränkt den Wissensingenieur ein, insbesondere durch die sortengebundenen Label. Andererseits erhält man eine konsistente Regelbasis, in der etliche subtile Fehler (z.B. durch Zugriff auf Label, deren Sorte nicht der erwarteten entspricht) nicht auftreten können.

7.5 Konsistenztests

7.5.1 Konsistenztests auf Produktionen

Sobald eine Produktion in der Repräsentation–Komponente gespeichert werden soll, werden einige Konsistenztests durchgeführt. Es handelt sich jeweils um *lokale* Tests, die ohne Kenntnis anderer Produktionen durchgeführt werden können. Zunächst wird die Syntax der Definition getestet, dann die Semantik. Die meisten Syntax–Tests können nicht fehlschlagen, wenn die graphische Benutzeroberfläche benutzt wird.

Folgende Bedingungen zur Semantik werden getestet:

- Jeder Label eines Knoten darf nur maximal einmal verwendet werden.

- `workpiece` darf nicht als Sorte eines Knotens der rechten Seite einer Produktion verwendet werden.
- Jeder in den Kanten oder Überlappungen referenzierte Knoten und Knoten-Label muß auch tatsächlich vorhanden sein.
- Die beiden Enden einer Kante oder Überlappung müssen verschieden sein.
- Die rechte Seite einer Produktion muß zusammenhängend sein, sowohl mit als auch ohne Berücksichtigung der Kontextknoten.
- Die Knoten müssen von 1 bis n numeriert sein (bei n Knoten). Die Numerierung muß einer GraPaKL-spezischen Bedingung genügen, nach der jeder Knoten (außer demjenigen mit Nummer 1) zu mindestens einem Knoten mit einer kleineren Nummer durch eine Kante verbunden sein muß.

7.5.2 Konsistenztests auf der Regelbasis

Der Benutzer kann selbst den Zeitpunkt bestimmen, an dem die Konsistenz der gesamten Regelbasis getestet werden soll. Vor dem Speichern als GraPaKL-Datei sollte dieser Test unbedingt ausgeführt werden.

Getestet wird, ob die Regelbasis konsistent und wohlgeformt (siehe 2.3) ist. Werden hier Fehler gefunden, ist die Regelbasis nicht zum Parsen oder Generieren von Werkstückdaten geeignet.

Die folgenden Test werden durchgeführt:

- Gibt es Zyklen in den Sub- bzw. Supersortenbeziehungen?
- Sind alle zu einer Sorte angegebenen Supersorten tatsächlich definiert?
- Gibt es ein Startsymbol?
- Enthält die Regelmenge sinnlose Symbole?
- Enthält die Regelmenge unerreichbare Symbole?
- Ist die Syntax der Constraints korrekt?
- Sind referenzierte Label definiert worden?
- Gibt es Fehler bezüglich des in 7.4 definierten Attributkonzeptes?

Es wird nicht getestet, ob eine Regelmenge 'Einzelproduktionen' (siehe 2.3) enthält, denn diese können in unserer Anwendung durchaus vorkommen (z.B. kann eine Langdrehfläche durch einen Zylindermantel definiert werden). Dennoch sollte ein Parser immer terminieren, da zirkuläre Definitionen für uns semantisch nicht sinnvoll sind.

Kapitel 8

Implementierung des Systems GGD

8.1 Module und Struktur des GGD

Das System GGD ist modular aufgebaut, die Funktionen für eine bestimmte Funktion finden sich in jeweils einem bestimmten Modul. Jedes Modul entspricht einer Datei der Implementierung. Abb. 8.1 zeigt die Module[1] und den Kontroll- und Datenfluß zwischen den Modulen.

Die einzelnen Module haben die folgende Aufgabe:

Check Durchführung der Konsistenztests.

Constraint-Editor Ein Editor, in dem die Contraints, also Bedingungen und Attributsberechnungsvorschriften eingegeben werden können.

Dags Verwaltung der DAGs (siehe 8.3).

Feat-Rep Ein- und Ausgabe von Dateien im Feat-Rep–Format.

GrPaKL-Output Compilierung der Datenbasis für den Graph–Parser GraPaKL.

Initialize Initialisierung der DAGs und einiger gloabler Variablen bei Arbeitsbeginn oder zur Löschung der Datenbasis.

Interface-V-R Die Schnittstelle zwischen der Repräsentations- und Visualisierungskomponente des GGD.

Loader Systemfunktionen zum Einlesen der anderen Module.

Macros Einige allgemeine Zugriffsmacros, die die Verwendung des Programms ohne Grafikoberfläche erleichtern.

Rules Repräsentation der Regeln (Produktionen), Funktionen zum Zugriff und zur Manipulation der Regeln.

Rule-Editor Verwaltung der Fenster, in denen die Struktur (Knoten und Kanten) einer Produktion eingegeben werden kann. Außerdem Funktionen zur Steuerung des Systems (Dateiverwaltung etc.)

[1] Die Kästchen mit den abgerundeten Ecken.

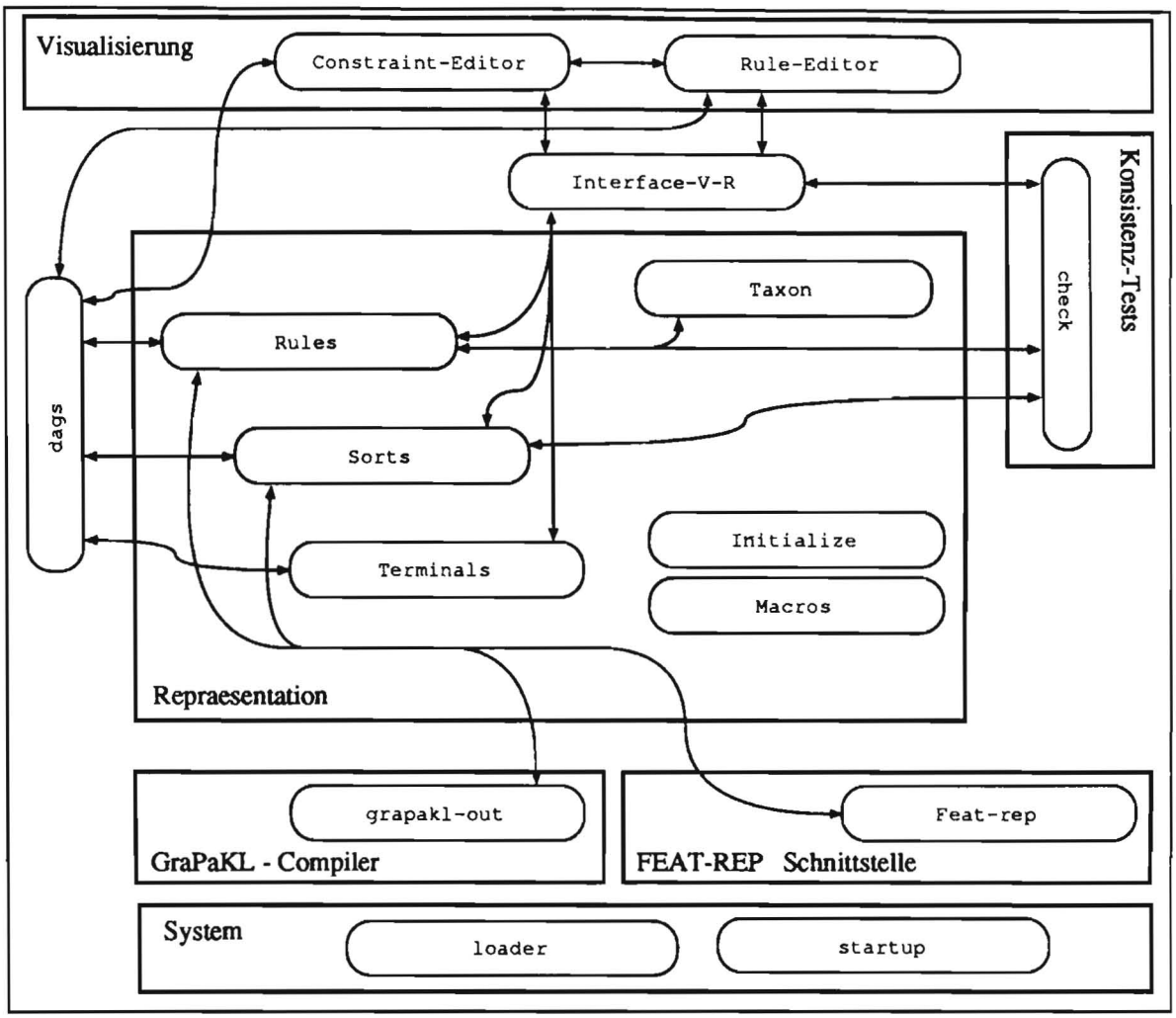


Abbildung 8.1: Die Module des GGD und der Daten- und Kontrollfluß.

Sorts Repräsentation der Sorten.

Startup Start und Initialierung der graphischen Oberfläche.

Taxon Die Einbindung des Systems TAXON.

Terminals Ein Teil der Sorten sind Terminalsymbole. Sie werden in diesem Modul eigens verwaltet.

8.2 Die graphische Benutzerschnittstelle

Die graphische Benutzerschnittstelle erlaubt, alle Sorten und Produktionen im Dialog mit dem System interaktiv eingeben und manipulieren zu können. Die Bedienung orientiert sich an anderen Programmen mit graphischer Oberfläche.

Für die Eingabe von Produktionen und Constraints stehen spezielle, maßgeschneiderte Editoren zur Verfügung. Dialoge und Fenster erlauben die Ein- und Ausgabe von Texten. Alle Eingaben können durch Benutzung der Maus erfolgen, nur Texte

(wie dem System nicht bekannte Namen) müssen über die Tastatur eingegeben werden.

Für alle Operationen (wie 'Erzeugen einer Produktion', oder 'Abspeichern einer Datei') stehen Menüs zur Verfügung. Durch Anwahl eines Objektes (wie einer Produktion oder eines Knotens) lassen sich Menüs öffnen, die die für dieses Objekt zur Verfügung stehenden Operationen zur Auswahl anbieten.

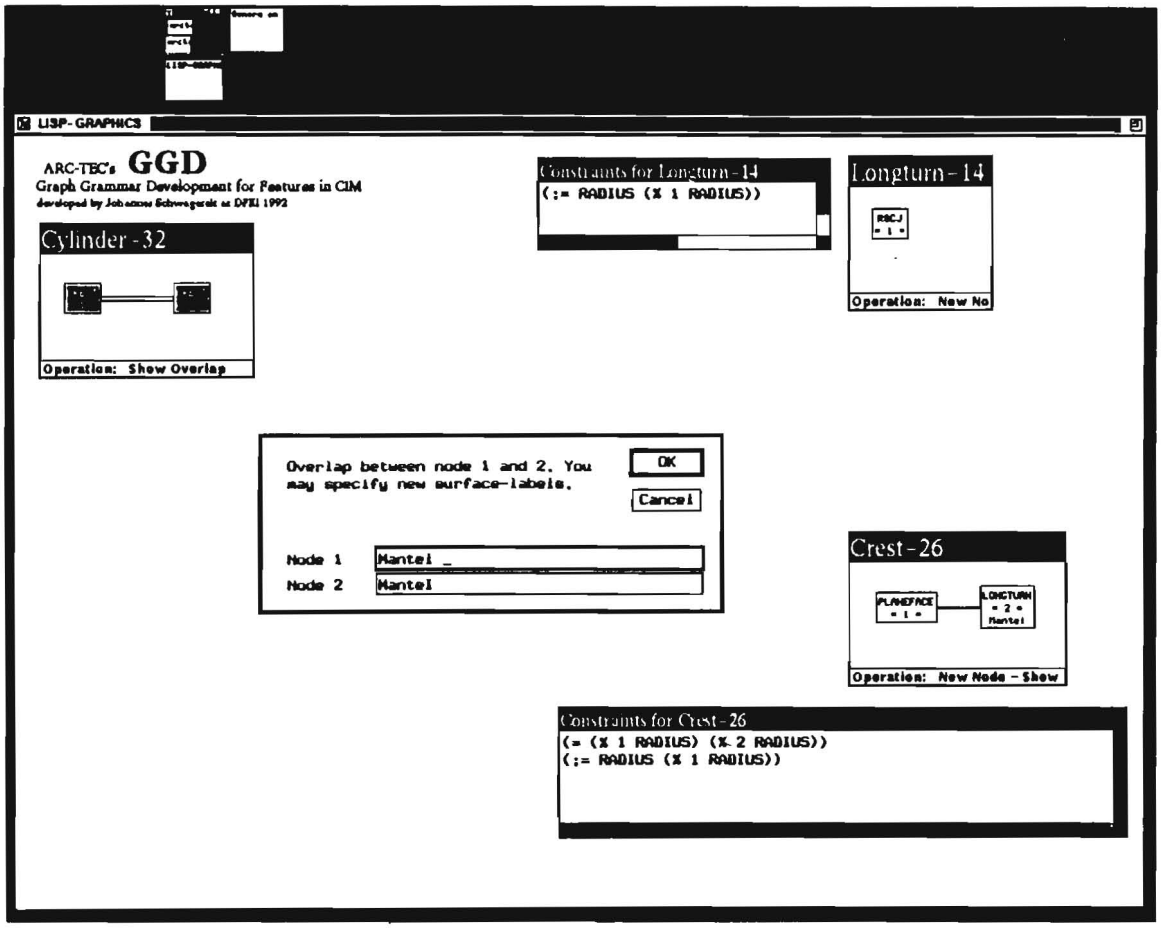


Abbildung 8.2: Die graphische Benutzeroberfläche des GGD

8.3 Datenstrukturen

Zentrale Datenstruktur im GGD sind gerichtete zusammenhängende azyklische Graphen (Directed Acyclic Graphs, DAG's). Alle Informationen im GGD werden durch DAG's repräsentiert, auch die Daten der in der Visualisierungs–Komponente dargestellten Objekte.

Formal betrachtet, können DAG's durch die folgende Definition beschrieben werden:

- Ein DAG ist ein gerichteter Graph Φ bestehend aus einer Menge V, den *Knoten* von Φ, und einer Menge E von geordneten Paaren $[\ \chi, \psi\ ]$ mit $\chi, \psi \in$ V, den Kanten von Φ.

- Zwei Knoten $\chi, \psi \in V$ heißen direkt verbunden, wenn $[\chi, \psi] \in E$ oder $[\psi, \chi] \in E$ gilt.
- Ein DAG ist *zusammenhängend*, d.h. für je zwei Knoten $\chi, \psi \in V$ gibt es eine Menge $\{\chi_0, \chi_1, \ldots, \chi_n\}$, wobei χ_i mit χ_{i-1} $(\forall\, i \in \{0, \ldots, n-1\})$, χ mit χ_0 und ψ mit χ_n direkt verbunden sind.
- Ein DAG ist *azyklisch*, d.h. er enthält keine Kantenfolge der Form $[\chi, \chi_0], \ldots, [\chi_n, \chi]$.

Eine übliche Notation für DAG's sind Matrizen von Attribut–Werte Paaren. Die Attribute sind markierte Kanten und die Werte sind Knoten eines DAG's.

Ein Beispiel aus dem GGD ist die Repräsentation eines Knotens einer Produktion:

Nummer	12
Name	Bottom
Sorte	RSC

Innerhalb eines DAG's ist eine Schachtelung möglich, das heißt ein Wert eines Attributs in einem DAG kann wieder ein DAG sein.

Als Beispiel diene die Darstellung einer Produktion im GGD:

Name	Left-Shoulder-21		
Sorte	Left-Shoulder		
Supersorte	Shoulder		
Order	4711		
Knoten	*Node-1*	*Nummer*	12
		Name	Bottom
		Sorte	RSC
	Node-2	*Nummer*	13
		Name	Top
		Sorte	RSCJ

Eine aus der Rekursion der DAG's resultierende Notation sind *Pfade*. Ein Pfad ist eine (endliche) Sequenz von Attributen und beschreibt einen (partiellen) Weg in einem DAG.

Der Pfad, um im Beispiel auf die Sorte des zweiten Knotens zuzugreifen, würde `(Knoten Node-2 Sorte)` lauten, und 'RSCJ' zurückliefern.

8.4 Eingabe von Sorten, Produktionen und Hierarchien

Das im GGD repräsentierte Wissen setzt sich aus Sorten, Produktionen, sowie Hierarchien über Sorten und Produktionen zusammen. Dieser Abschnitt soll zeigen, wie die Eingabe dieses Wissens in das System GGD implementiert ist.

8.4.1 Sorten

Die Sorten müssen explizit vom Benutzer des GGD definiert werden. Produktionen können nur dann einer Sorte zugeordnet werden, wenn diese bereits definiert wurde.

Die Attribute 'Art' und 'Anwendung' einer Sorte werden durch den Benutzer des Programms festgelegt. Diese Angaben werden im GGD repräsentiert, bisher aber nicht weiter ausgewertet.

Zu jeder Sorte können ihre **Supersorten** (auch mehr als eine) angegeben werden. Zyklische Subsortenbeziehungen sind aber nicht erlaubt. Diese Sortenhierarchie wird aktiv verwaltet, d.h. das System verhindert zyklische Beziehungen und die Angabe nicht vorhandener Sorten, und überwacht die Vererbung von Attributen (Zum Vererbungskonzept siehe 7.4.).

Eingegeben werden die Sorten in einem Dialog, der die oben genannten Werte abfragt. Bei der Eingabe von 'Name' und 'Art' wird sofort überprüft, ob es sich um erlaubte Werte handelt (z.B. kein bereits definierter Name).

Eine Sonderstellung nehmen *atomare* Features (Terminale) ein, die vordefiniert sind und daher nicht vom Benutzer eingegeben werden müssen.

Beispiel einer Sorte: Eine *Linke-Schulter* ist ein *geometrisches* (Art) Feature, das durch *Drehen* (Anwendung) hergestellt wird. Seine Supersorte ist *Schulter*.

8.4.2 Produktionen

GGD erlaubt, Produktionen (Regeln) zu erstellen und zu löschen. Zu jeder Produktion können ihre Sorte, ein Ordnungswert sowie ein Dokumentationsstring angegeben werden. Die Definition ähnlicher Produktionen wird unterstützt.

Knoten, Nachbarschaften und Überlappungen können gesetzt und entfernt werden. Kontextknoten werden analog zu regulären Knoten repräsentiert, in der graphischen Oberfläche allerdings gesondert dargestellt[2].

Constraints werden in einem speziellen Editor eingegeben, der für jede Produktion zur Verfügung steht. Attribute einer Regel werden implizit definiert, in dem ihnen in den Constraints ein Wert zugewiesen wird. Auch werden Label durch ihre Verwendung in einem Knoten definiert, der Benutzer muß sie nicht zusätzlich angeben[3].

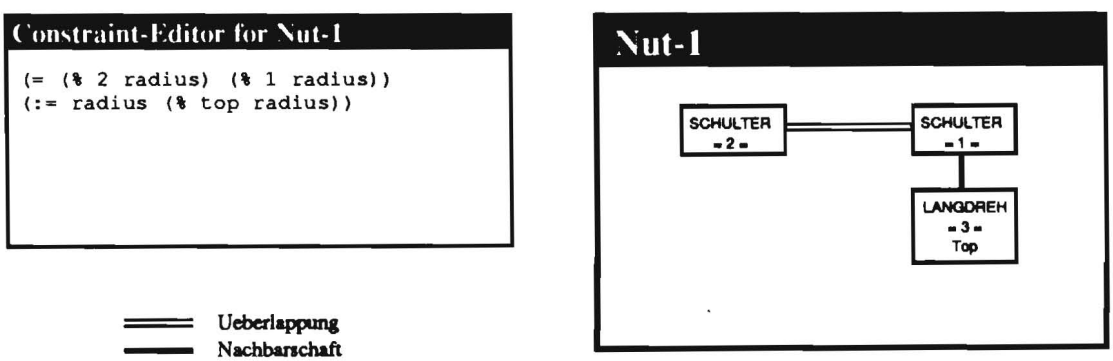


Abbildung 8.3: Produktion im GGD

[2]In 4.2.2 wurde gefordert, daß in einem Kontextknoten eine Liste von Sorten angegeben werden kann. In der aktuellen Implementierung kann aber nur eine Sorte spezifiziert werden.

[3]Ein Unterschied zur GraPaKL-Grammatik, dort müssen alle verwendeten Attribute und Label explizit definiert werden.

Abb. 8.3 zeigt die Darstellung einer typischen Produktion wie sie auf der interaktiven Graphikoberfläche des GGD dargestellt wird. Es handelt sich um eine *Nut*, die durch zwei Schultern und einer Langdrehfläche beschrieben wird. Nicht sichtbar ist hier die Angabe der sich überlappenden Flächen der Schultern.

In der Sicht der objektorientierten Wissensrepräsentation bilden die Produktionen die *Instanzen* ihrer Sorten. Damit ist dann jede Produktion einer Subklasse immer auch eine Instanz jeder ihrer Superklassen und, durch die Transitivität dieser Beziehung, auch derer Superklassen.

8.4.3 Die Verwendung von TAXON

Definition der verwendeten Partialordnung

Die Partialordnung $\leq_E$ ist nicht so implementiert, wie sie in Definition 6.1 in 6.3 spezifiziert wurde. Tatsächlich lautet die Definition:

Definition 8.1 *p_1, p_2 seien zwei Produktionen einer ANCEGG.*
$\beta_1 = (V_1, C_1, E_{N1}, E_{\ddot{U}1}, \sigma_1, \xi_1)$ und $\beta_2 = (V_2, C_2, E_{N2}, E_{\ddot{U}2}, \sigma_2, \xi_2)$ seien die Spezifikationen dieser Produktionen. Seien $|E_{N1}|$, $|E_{N2}|$, $|E_{\ddot{U}1}|$, $|E_{\ddot{U}2}|$ die Kardinalitäten der Mengen E_{N1}, E_{N2}, $E_{\ddot{U}1}$, $E_{\ddot{U}2}$. Sei $H_{s_1} = \{y \,|\, y \in V_1, \sigma(y) = s\}$, $H_{s_2} = \{y \,|\, y \in V_2, \sigma(y) = s\}$. Entsprechend sind $|H_{s_1}|$ und $|H_{s_2}|$ die Kardinalitäten dieser Mengen.

Sei $p_1 \leq_{E'} p_2$ eine Partialordnung auf den Graphen, die durch die Spezifikationen dieser Produktionen festgelegt sind.

Es gilt $p_1 \leq_{E'} p_2$ genau dann, wenn $|E_{N1}| \leq |E_{N2}| \wedge |E_{\ddot{U}1}| \leq |E_{\ddot{U}2}| \wedge \forall s \in (V_1 \cup V_2)$: $|H_{s_1}| \leq |H_{s_2}|$.

Attribute und Constraints finden aus den in 6.3 genannten Gründen keine Berücksichtigung. Kontextknoten werden ebenfalls nicht einbezogen.

Für den Vergleich dieser Definition des $\leq_{E'}$ mit der Definition von $\leq_G$ (Definition 6.1) gilt: Aus $p_1 \leq_G p_2$ folgt $p_1 \leq_{E'} p_2$.

Die Entscheidung für diese Relation fiel aus zwei Gründen.

- Sie ist formal einfacher und klarer, für $\leq_G$ konnte nur eine algorithmische Definition gefunden werden. Sie ist auch durch TAXON effizient auswertbar. Die Berechnung der Relation $\leq_G$ ist aufwendig, zudem wäre sie in dieser Form nicht für TAXON geeignet gewesen.
- Die Zahl der gefundenen Ähnlichkeiten vergrößert gegenüber der ursprünglichen Relation, was uns zumindest in der Erprobungsphase des Systems hilfreicher als eine restriktivere Relation erschien.

Die Einbindung von TAXON in den GGD

Jede Produktion wird durch TAXON klassifiziert. TAXON berechnet entsprechend der oben angegebenen Definition von $\leq_{E'}$ eine Hierarchie der Ähnlichkeiten.

Eine Produktion wird umgesetzt in ein komplexes Konzept. Dieses Konzept besteht aus einer Konjunktion von Prädikaten, jeweils eines für jede in der rechten Seite der Produktion auftauchende Sorte, außerdem für die Nachbarschaften und die Überlappungen. Jedes der Prädikate ist so definiert, daß es genau dann wahr ist, wenn eine Produktion gleichviele oder mehr der jeweiligen Elemente enthält. TAXON ist in der Lage, diese Prädiakte zu vergleichen und so die erwünschte Ordung über die Produktionen zu berechnen.

Beispiel:

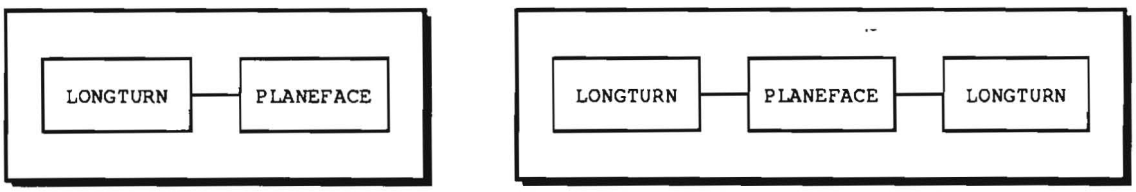


Abbildung 8.4: Zwei einfache rechte Seiten von Produktionen.

Eine Produktion, deren rechte Seite aus einem 'Planeface', einer 'Longturn' und einer Kante besteht (Abb. 8.4 links), würde umgesetzt zu

```
(AND (F-0 OVERLAPS-SORT)
     (F-1 EDGES-SORT)
     (F-1 PLANEFACE-SORT)
     (F-1 LONGTURN-SORT))
```

Eine Produktion, deren rechte Seite aus einem 'Planeface', zwei 'Longturn' und zwei Kanten besteht (Abb. 8.4 rechts), würde umgesetzt zu

```
(AND (F-0 OVERLAPS-SORT)
     (F-2 EDGES-SORT)
     (F-1 PLANEFACE-SORT)
     (F-2 LONGTURN-SORT))
```

In TAXON subsumiert dieses Konzept das oben angegebene.

8.5 Implementierung der Konsistenztests

Die Implementierung der in 7.5.1 beschriebenen Tests bereitete keine größeren Probleme. Die Test werden in der dort eingehaltenen Reihenfolge durchgeführt.

Bei einem Fehler bricht die Testroutine ab und teilt dem Benutzer den gefundenen Fehler mit. Gespeichert wird die Produktion erst, wenn kein Fehler gefunden wird.

Der Test der Regelbasis wird in vier Durchgängen durchgeführt. In jedem Durchgang wird eine Reihe von logisch zueinandergehörigen Tests durchgeführt.

1. **Durchgang** Es werden zwei Tests durchgeführt:

   - Gibt es Zyklen in den Sub– bzw. Supersortenbeziehungen? *Beispiel:* $Sorte_A$ ist Subsorte von $Sorte_B$, gleichzeitig ist $Sorte_B$ Subsorte von $Sorte_A$.
   - Sind alle zu einer Sorte angegebenen Supersorten tatsächlich definiert? Dieser Test muß für die Subsorten einer Sorte nicht durchgeführt werden, da diese nicht explizit gespeichert, sondern aus den Supersorten berechnet werden.

2. **und 3. Durchgang** Es wird geprüft, ob die Graph–Grammatik 'wohlgeformt' (siehe 2.3, und 7.5.2) ist. Es werden die dafür notwendigen Bedingungen getestet:

   - Gibt es mindestens ein Startsymbol, also eine Produktion mit Sorte `Workpiece`. Im Gegensatz zu zahlreichen anderen Definitionen einer Graph–Grammatik kann mehr als nur ein Startsymbol definiert werden.
   - Lässt sich jedes Nonterminal aus den Startsymbolen herleiten? Diese Frage ist gleichbedeutend mit: Gibt es Produktionen, die nie verwendet werden?
     Ausgehend von den Produktionen für das Startsymbol werden rekursiv die Symbole der rechten Seite der Produktionen expandiert.
   - Gibt es für jedes in der rechten Seite einer Produktion verwendete Nonterminal mindestens eine definierende Produktion?
   - Lassen sich alle Nonterminale zu Terminalen ableiten. Das verwendete Verfahren ist hier Bottum–Up. Ausgehend von den Terminalsymbolen wird versucht, Produktionen für alle Nonterminalsymbole zu finden.

4. **Durchgang** Es werden alle Pfade in Constraints auf ihre Zulässigkeit geprüft. Entsprechend dem in 7.4 aufgestellten Sorten– und Attributmodell werden verschiedene Tests durchgeführt. Gleichzeitig wird die Syntax der Constraints geprüft, da dieses nicht während der in 7.5.1 beschriebenen Tests durchgeführt wird.

   Für jede einzelne Produktion wird zunächst überprüft:

   - Wird in keiner Rollenzuweisung (*role-assignment*, siehe A.2.3) eine Sorte spezifiziert, die dem System nicht bekannt ist?
   - Wird kein Name sowohl als Label für einen Knoten als auch für eine Rollenzuweisung verwendet?
   - Wird kein Name sowohl als Attributsname als auch als Label verwendet?

   Für alle Produktionen der einzelnen Sorten folgen diese Tests:

   - Steht kein Label in verschiedenen Produktionen einer Sorte für verschiedene Sorten?
   - Gibt es keinen Sortenkonflikt zwischen den in den verschiedenen Supersorten der Sorte verwendeten Labels.

- Gibt es keinen Sortenkonflikt zwischen den Labels einer Sorte und denen ihrer Supersorten?
- Wird kein Name in einigen Produktionen der Sorte als Attributsname, in anderen als Rollenname verwendet?
- Wird kein Rollenname in einer der Produktionen der Supersorte als Attributname verwendet, und vice versa.

Erst wenn diese Tests abgeschlossen sind, und damit bekannt ist, welche Namen in welchen Produktionen für Attribute oder Rollen bestimmter Sorten stehen, kann die Korrektheit der Constraints überprüft werden.

Es wird für jeden Pfad geprüft, ob er korrekt spezifiziert sind (zur Syntax von Pfaden siehe die Produktion für `<path>` in A.2.3).

- Es darf sich nicht um einen leeren Pfad handeln.
- Jedes Symbol a_i $(i > 1)$ eines Pfades $(\%\ a_1 a_2 \ldots a_n)$ muß in dem durch die Symbole a_1 bis a_{i-1} vorgegebenen Kontext definiert sein. Für a_1 sind die Rollen- und Attributnamen dieser Produktion erlaubt.
  Beispiel: Zu überprüfen sei in einem Constraint einer bestimmten Produktion der Pfad `(% Upper Bottom Radius)`. Dann muß `Upper` ein Rollenlabel dieser Produktion sein, diese Rolle habe die Sorte U. Für die Sorte U muss es dann zumindest eine Produktion geben mit einer Rolle namens `Bottom`, diese habe die Sorte B. In der Sorte B kann `Radius` ein Rollen- oder Attributnamen sein; ist es ein Attributlabel, muß es in mindestens einer der Produktionen von B eine Berechnungsvorschrift für `Radius` geben.
- Ein Pfad darf nach einem Attribut nicht fortgesetzt werden. *Beispiel:* Der Pfad `(% Bottom Radius Middlestep)` ist illegal, wenn es sich bei `Radius` um ein Attribut handelt.
- Wenn ein Pfad das Schlüsselwort `sort` enthält, muß er nach diesem Wort enden.

Die Pfade, die bei Überlappungen und 'tiefen' Nachbarschaften angegeben werden können, müssen in gleiche Weise wie Pfade in Constraints getestet werden. Allerdings muß zusätzlich sichergestellt werden, daß keine Attribute referenziert werden.

8.6 Beispiel für die Erstellung einer Graph–Grammatik mit dem GGD

Es soll eine einfache Grammatik entwickelt werden, mit ihr wird nur ein einfacher Zylinder beschrieben. Die Produktionen dieser Grammatik sind in Abb. 8.5 in der Form abgebildet, wie sie auch auf der Benutzeroberfläche gezeichnet werden. Nicht abgebildet sind die Attribute und die Constraints, die in speziellen Texteditoren definiert werden: Attribut jeder Regel ist 'Radius'. Bei 'Cylinder' und 'Crest' wird getestet, ob die Radien der beiden Elemente gleich sind.

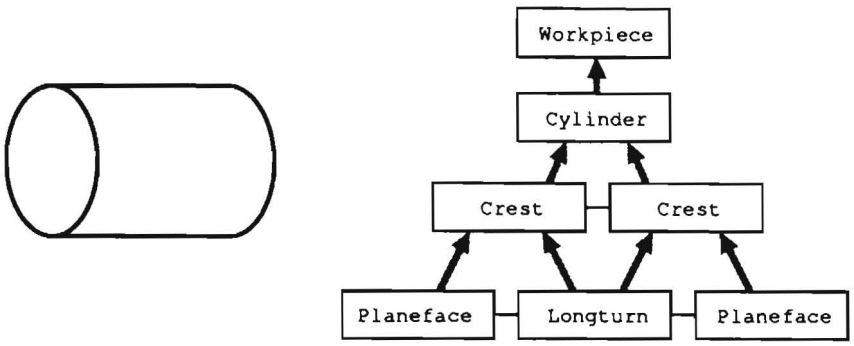


Abbildung 8.5: Ein Zylinder und sein Strukturbaum.

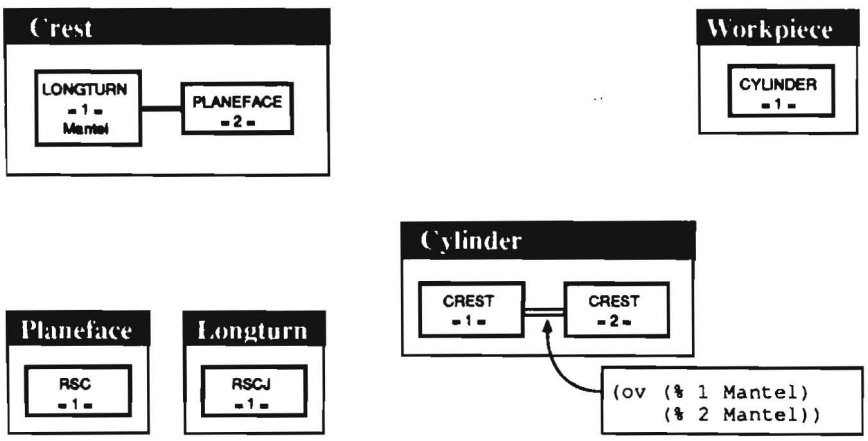


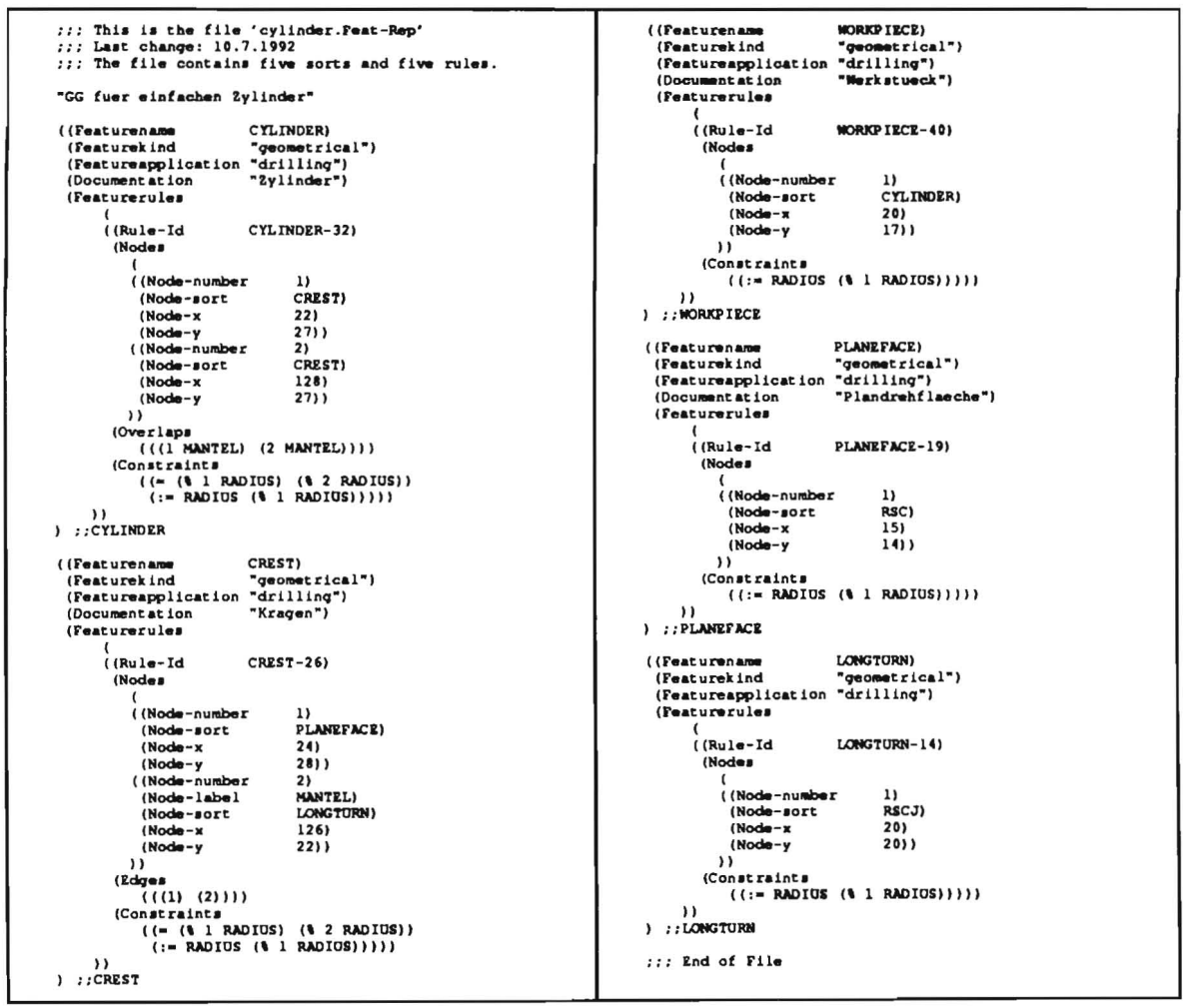
Abbildung 8.6: Produktionen der Beispielgrammatik

Die Darstellung im FEAT-REP-Format

Die Grammatik im FEAT-REP Format zeigt Abb. 8.7.

Die Schnittstelle zu GraPaKL

Die im GGD erstellte Grammatik kann im GraPaKL–Format ausgegeben werden (Abb. 8.8). Diese Datei kann der GraPaKL direkt zum Parsen eines Zylinders verwenden.



```
;;; This is the file 'cylinder.Feat-Rep'
;;; Last change: 10.7.1992
;;; The file contains five sorts and five rules.

"GG fuer einfachen Zylinder"

((Featurename          CYLINDER)
 (Featurekind          "geometrical")
 (Featureapplication   "drilling")
 (Documentation        "Zylinder")
 (Featurerules
     (
     ((Rule-Id         CYLINDER-32)
      (Nodes
        (
        ((Node-number      1)
         (Node-sort        CREST)
         (Node-x           22)
         (Node-y           27))
        ((Node-number      2)
         (Node-sort        CREST)
         (Node-x           128)
         (Node-y           27))
        ))
      (Overlaps
         (((1 MANTEL) (2 MANTEL))))
      (Constraints
         ((= (% 1 RADIUS) (% 2 RADIUS))
          (:= RADIUS (% 1 RADIUS)))))
   ))
) ;;CYLINDER

((Featurename          CREST)
 (Featurekind          "geometrical")
 (Featureapplication   "drilling")
 (Documentation        "Kragen")
 (Featurerules
     (
     ((Rule-Id         CREST-26)
      (Nodes
        (
        ((Node-number      1)
         (Node-sort        PLANEFACE)
         (Node-x           24)
         (Node-y           28))
        ((Node-number      2)
         (Node-label       MANTEL)
         (Node-sort        LONGTURN)
         (Node-x           126)
         (Node-y           22))
        ))
      (Edges
         (((1) (2))))
      (Constraints
         ((= (% 1 RADIUS) (% 2 RADIUS))
          (:= RADIUS (% 1 RADIUS)))))
   ))
) ;;CREST

((Featurename          WORKPIECE)
 (Featurekind          "geometrical")
 (Featureapplication   "drilling")
 (Documentation        "Werkstueck")
 (Featurerules
     (
     ((Rule-Id         WORKPIECE-40)
      (Nodes
        (
        ((Node-number      1)
         (Node-sort        CYLINDER)
         (Node-x           20)
         (Node-y           17))
        ))
      (Constraints
         ((:= RADIUS (% 1 RADIUS)))))
   ))
) ;;WORKPIECE

((Featurename          PLANEFACE)
 (Featurekind          "geometrical")
 (Featureapplication   "drilling")
 (Documentation        "Plandrehflaeche")
 (Featurerules
     (
     ((Rule-Id         PLANEFACE-19)
      (Nodes
        (
        ((Node-number      1)
         (Node-sort        RSC)
         (Node-x           15)
         (Node-y           14))
        ))
      (Constraints
         ((:= RADIUS (% 1 RADIUS)))))
   ))
) ;;PLANEFACE

((Featurename          LONGTURN)
 (Featurekind          "geometrical")
 (Featureapplication   "drilling")
 (Featurerules
     (
     ((Rule-Id         LONGTURN-14)
      (Nodes
        (
        ((Node-number      1)
         (Node-sort        RSCJ)
         (Node-x           20)
         (Node-y           20))
        ))
      (Constraints
         ((:= RADIUS (% 1 RADIUS)))))
   ))
) ;;LONGTURN

;;; End of File
```

Abbildung 8.7: Beispielgrammatik im FEAT-REP Format

```
;;; This is the file 'CYLINDER.gg', a GraPaKL-File.

(make-gg

 :name "CYLINDER"
 :default-graph NIL
 :include "TEC-REP"

;;;-------------------------------------

 :sorts '(

   ((GOAL WORKPIECE )
      (CYLINDER . CYLINDER58)
      (attr . RADIUS)) ;WORKPIECE

   ((NONTERMINAL CYLINDER )
      (CREST . CREST60)
      (CREST . CREST59)
      (attr . RADIUS)) ;CYLINDER

   ((NONTERMINAL CREST )
      (LONGTURN . MANTEL)
      (PLANEFACE . PLANEFACE61)
      (attr . RADIUS)) ;CREST

   ((NONTERMINAL PLANEFACE )
      (RSC . RSC62)
      (attr . RADIUS)) ;PLANEFACE

   ((NONTERMINAL LONGTURN )
      (RSCJ . RSCJ63)
      (attr . RADIUS)) ;LONGTURN

  ) ;sorts

;;;-------------------------------------
```

```
 :rules '(

  ;;; WORKPIECE-40
  ;;;
  (WORKPIECE (
    (CYLINDER CYLINDER58
      ((:= RADIUS (% CYLINDER58 RADIUS))))
   )) ;WORKPIECE-40

  ;;; CYLINDER-32
  ;;;
  (CYLINDER (
    (CREST CREST59 )
    (CREST CREST60
      ((OV (% CREST59 MANTEL) (% CREST60 MANTEL))
       (= (% CREST59 RADIUS) (% CREST60 RADIUS))
       (:= RADIUS (% CREST59 RADIUS))))
   )) ;CYLINDER-32

  ;;; CREST-26
  (CREST (
    (PLANEFACE PLANEFACE61 )
    (LONGTURN MANTEL
      ((NR (% PLANEFACE61) (% MANTEL))
       (= (% PLANEFACE61 RADIUS) (% MANTEL RADIUS))
       (:= RADIUS (% PLANEFACE61 RADIUS))))
   )) ;CREST-26

  ;;; PLANEFACE-19
  ;;;
  (PLANEFACE (
    (RSC RSC62
      ((:= RADIUS (% RSC62 RADIUS))))
   )) ;PLANEFACE-19

  ;;; LONGTURN-14
  (LONGTURN (
    (RSCJ RSCJ63
      ((:= RADIUS (% RSCJ63 RADIUS))))
   )) ;LONGTURN-14
  ) ;rules

) ;make-gg
```

Abbildung 8.8: Beispielgrammatik im GraPaKL Format

Kapitel 9

Vergleich des GGD mit anderen Systemen zur Eingabe von Graph–Grammatiken

Es stellt sich die Frage, ob es nicht besser gewesen wäre, eines der in Kapitel 3 vorgestellten Systeme zur Eingabe von Graph–Grammatiken zu verwenden, anstatt mit GGD ein eigenes System zu entwickeln.

Es sollen hier GraphEd und PAGGED mit dem GGD verglichen werden, und Vor- und Nachteile der einzelnen Systeme herausgestellt werden. Die Diskussion steht vor dem Hintergrund ein System zu finden, das für die Eingabe von auf die Feature-Repräsentation maßgeschneiderten Grammatiken geeignet ist.

Anwendung

GGD wurde von vorneherein als ein spezialisiertes Tool zur Eingabe von Graph–Grammatiken ausgelegt. Dagegen ist GraphEd nicht auf eine spezielle Anwendung festgelegt. PAGGED ist ein Editor für Graph–Grammatiken, die zur Softwareentwicklung eingesetzt werden.

Mächtigkeit der Graph–Grammatik

Sowohl GraphEd als auch PAGGED können die meisten wichtigsten Charakteristika unserer Grammatik wiedergeben: Den Knoten können Sorten zugewiesen werden, Kanten sind in zumindest zwei Typen unterscheidbar (Nachbarschaften und Überlappungen). Folgende Probleme bleiben aber ungelöst:

- Die Notation der Einbettung im PAGGED ist mächtiger, aber auch schwieriger zu spezifizieren als in GDD und GraphEd.
- GraphEd kennt weder Attribute, Berechnungsvorschriften noch Bedingungen. Ihre Repräsentation ist nicht vorgesehen.

- Im GGD können bei Nachbarschaften und Überlappungen sog. Pfade spezifiziert werden. Diese Möglichkeit besteht in GraphEd und PAGGED nicht. Überlappungen können also nur durch einen speziellen Kantentyp spezifiziert werden, die notwendigen Pfade sind nicht vorgesehen
- GraphEd und PAGGED kennen kein Subsortenkonzept wie der GGD. Der Verzicht auf diese Eigenschaft würde ein deutliche Einschränkung bedeuten.
- Nicht möglich ist in GraphEd und PAGGED die Angabe einer Partialordnung auf den Knoten einer Produktion oder den Produktionen, die einem Programm als Heuristik dienen können.

Graphische Oberfläche

Bei GraphEd und PAGGED handelt es sich um Produkte, die in jahrelanger Arbeit entwickelt und verbessert wurden. Es ist nicht überraschend, daß die Oberflächen dieser Systeme ausgereifter sind und mehr Funktionen zur Manipulation eines Graphen anbieten. Allerdings bietet auch GGD alle benötigten Operationen (Erzeugen, Verschieben und Löschen von Kanten und Knoten, Verändern von Knotensorten, Kontrollanweisungen zum Definieren und Löschen von Produktionen). Mit genügend Zeit wäre auch die Oberfläche des GGD entsprechend ausbaubar.

GGD bietet im Gegensatz zu den beiden anderen Programmen einen direkten Zugriff auf Schnittstellen und Tests. Ohne größeren Aufwand können die Daten im FEAT-REP oder GraPaKL Format erzeugt werden, Tests zur semantischen Integrität einer Regel oder der Regelbasis sind integriert und stehen zur direkten Verfügung.

Integration in andere Systemen

PAGGED und GraphEd kommunizieren im wesentlichen über Dateien mit anderen Programmen. GraphEd bietet zudem die Einbindung benutzerdefinierter Module. GGD erlaubt, über dokumentierte Schnittstellen den Zugriff auf alle Datenstrukturen, und ist durch seinen modularen Aufbau leicht erweiterbar. Eine Integration in andere Systeme kann daher ohne größeren Aufwand erfolgen.

Anpassung von PAGGED und GraphEd an unsere Anforderungen

Ohne Frage wäre mit einem entsprechenden Aufwand eine Anpassung des PAGGED und des GraphEd an die in Kapitel 5 definierte Grammatik theoretisch möglich.

Bei beiden Programmen wäre eine Behebung der oben aufgeführten Mängel jedoch schwierig. Der Aufwand wäre wahrscheinlich höher gewesen als der für die eigene Implementierung des GGD. Hätte man eines der Programme so erweitert, daß seine Funktionalität der des GGD entsprochen hätte, wären eine Neuimplementierung größerer Teile notwendig geworden.

- Im GGD können direkt Feat-Rep–Dateien gelesen und erzeugt werden, außerdem können Dateien im GraPaKL-Format erstellt werden. Diese direkte Anbindung fehlt in den beiden anderen Programmen. Notwendig wäre ein Compiler zwischen Feat-Rep und dem jeweiligen Dateiformat anderer Systeme.

- Im GGD ist es möglich, bestimmte Konsistenztests auf der Regelbasis sofort durchzuführen (wie Zusammenhang der rechten Seiten, korrekte Numerierung der Knoten, Verwendung vordefinierter Terminalsymbole). Mit den anderen Programmen solche Tests nur mit entsprechenden Erweiterungen möglich, oder wenn die Datenbasis in Form einer Datei vorliegt.
- Jede neue Regel wird im GGD sofort von TAXON in eine Hierarchie eingeordnet, und gibt dem Benutzer so schon während der Entwicklung wichtige Hinweise. Eine Integration von TAXON in ein anderes System wäre aufwendig, da jeweils die interne Repräsentation der Produktionen verarbeitet werden müßte.
- Im GraphEd ist es nicht möglich, zu jeder Regel Constraints anzugeben. Im GGD und PAGGED steht dafür ein spezieller Editor zur Verfügung.

Ein großes Problem sind auch die in den Programmen verwendete Hard- und Software, die teilweise von der im ARC-TEC Projekt verwendeten abweicht. Eine Benutzung dieser Programme, wenn sie denn innerhalb unserer technischen Möglichkeiten liegt, würde dem Ziel der Integration aller verwendeten Programme daher widersprechen.

Fazit

Angesichts der beschriebenen Probleme bei der Anpassung der von PAGGED und GraphEd wurde mit einer eigenen Implementierung der bessere Weg beschritten. Zudem sind meiner Ansicht nach die Vorteile eines maßgeschneiderten System zur Featurerepräsentation höher zu bewerten als einige zusätzliche Möglichkeiten der Benutzerschnittstelle. Zudem erschwert eine zu große Mächtigkeit eines Systems seine Bedienung, provoziert Eingabefehler und erfordert mehr Wissen des Benutzers.

Kapitel 10

Abschlussbetrachtung und Ausblick

In dieser Arbeit wurde ein maßgeschneidertes System für die Repräsentation von Feature erstellt. Die formale Grundlage lieferten Graph–Grammatiken, mit denen sich die Charakteristika der Feature sehr gut wiedergeben lassen. Ein wichtiger Teil des Systems sind Werkzeuge, die die Entwicklung einer Graph–Grammatik unterstützen.

Vergleicht man die Anforderungen, die in Kapitel 1 aufgestellt wurden mit den erreichten Ergebnissen, so stellt man fest, daß die wesentlichen Ziele erreicht wurden:

- Es wurde mit der ANCEGG eine Datenstruktur entwickelt, die die besonderen Charakteristika von Features berücksichtigt.
- Es wurde mit dem GGD ein System entworfen, mit dem sich Feature–Grammatiken in einfacher Weise entwickeln lassen. Integriert sind Tests zur Syntax von Regeln und zur Wohlgeformtheit der Grammatik.
- Das Programm TAXON wurde integriert, um Ähnlichkeiten zwischen Featuredefinitionen zu entdecken und zu verwalten.
- Durch Schnittstellen wird anderen Programmen der Zugriff auf die repräsentierten Daten erlaubt. Dateien im FEAT-REP Format können gelesen und erzeugt werden. Ein Compiler übersetzt die Definitionen in ein für den Graphparser GraPaKL geeignetes Format.

Leider konnte nicht jedes Konzept so umgesetzt werden, wie dies zu Beginn der Arbeit erhofft wurde. Es handelt sich nicht um prinzipielle Mängel, vielmehr wären mit einem entsprechenden zeitlichen Aufwand bessere Lösungen zu finden. Keines dieser Probleme ist so gravierend, daß es die Verwendung des GGD grundsätzlich einschränken würde.

- Die Partialordnung, nach der TAXON alle Regeln klassifiziert, erfüllt ihre Aufgabe, den Aufbau einer Hierachie ähnlicher Regeln, nicht optimal. Es sind mehr Feature bezüglich dieser Partialordnung vergleichbar als es der Einschätzung

durch einen Experten entspricht. Leider konnte keine bessere Lösung gefunden werden, die auch in TAXON realisierbar ist.

Wünschenswert ist es, auch die Constraints für diese Partialordnung auszuwerten. Es wäre zu untersuchen, wie hier das ARC-TEC–Tool CONTAX[1] eingesetzt werden kann.

- Zu Beginn er Arbeit wurde erhofft, durch 'Structure Sharing' ähnlicher Feature eine kompaktere Repräsentation erreichen zu können. Es wurde aber schnell festgestellt, daß es einen zu großen Aufwand bedeutet hätte, 'Structure Sharing' zu berücksichtigen.

- Die graphische Benutzeroberfläche genügt keinen professionellen Ansprüchen. Verbesserungen wären in der Operationsauswahl, besonders aber beim eingesetzten Texteditor wünschenswert.

Auch wenn die Definition einer ANCEGG aus Kapitel 5 alle wesentlichen Elemente enthält, um die Charakteristika von Feature adäquat zu repräsentieren, ist sie doch nicht ohne Mängel. Man muß sich allerdings fragen, ob diese Mängel wirklich durch eine andere Definition behoben werden sollten, oder es sich nicht vielmehr um Eigenschaften handelt, die erst in einer konkreten Grammatik überprüft werden können.

1. Nicht alle Aspekte des in 7.4 beschriebenen Attributkonzepts werden wiedergegeben. Dazu gehört die Vererbung von Attributen in der Sortenhierarchie, und die Sortenbindung der Label innerhalb einer Sortenhierarchie.
2. Je nach Definition des Einbettungsalgorithmus eines Generators können semantisch nicht sinnvolle Graphen definiert werden (z.B. sind benachbarte Kontextknoten für eine RNLC-Einbettung wie in 2.2 (Seite 13) semantisch nicht sinnvoll).

Der GGD ist zwar ein in sich abgeschlossenes, einsatzfähiges System. Dennoch sind Erweiterungen denkbar, die nützlich wären, und mit vertretbarem Aufwand zu implementieren sein müßten:

- Bei der Übersetzung der eingegebenen Daten in das GraPaKL-Format wird zwar versucht, aus jeder Regel eine für den Parser effizient zu bearbeitende Definition zu erzeugen. Eine Optimierung, die mehr als eine Regel betrachtet, findet aber nicht statt. Es wäre eine interessante Aufgabe, hier nach Verbesserungen zu suchen.

- Skelettpläne sind Teilarbeitspläne für einzelne Feature. Wegen der engen Beziehung zwischen Feature und zugehörigem Arbeitsplan wäre zu überlegen, ob diese nicht gemeinsam repräsentiert werden können. Dieses würde die Arbeit des Ingenieurs erleichtern, der nicht nur die Feauture, sondern auch die zugehörigen Skelettpläne entwickeln muß. Das System könnte den Entwickler unterstützen, z.B. durch Tests, ob im Skelettplan referenzierte Flächen und Attribute überhaupt definiert sind.

[1] Ein Constraint-Solver über hierarchisch strukturierte Domänen.

- Eine Visualisierung der Features als Teile (also nicht nur als abstrakte Graphen) wäre für den Experten von Interesse, ist aber nicht einfach zu implementieren.

Anhang A

Benutzung des GGD

Der GGD ist ein komplexes Werkzeug, um Graph–Grammatiken zu entwickeln. Die Definition dieser Graph–Grammatiken ist speziell dafür entworfen, Features in CIM zu repräsentieren.

Es existiert eine graphische Oberfläche, mit der die Regeln[1] und Sorten der Graph–Grammatik eingegeben werden können. Alternativ kann die Eingabe auch durch den Aufruf der entsprechenden Funktionen im LISP-Reader oder durch andere Programme erfolgen.

In den nächsten Abschnitten wird davon ausgegangen, daß die graphische Oberfläche benutzt wird. Wird diese Benutzeroberfläche nicht benutzt, sind die in Abschnitt B.2 beschriebenen Funktionen zu verwenden.

Die graphische Oberfläche kennt die folgenden Darstellungselemente: **Regeleditor**, **Constrainteditor**, **Menü**, **Textfenster** und **Dialog**.

Der **Regeleditor** ist ein Fenster, in dem interaktiv eine Regel eingegeben und editiert werden kann. Es können mehrere Regeleditoren gleichzeitig geöffnet sein, aber nur einer für jede Regel.

Der **Constrainteditor** ist ein einfacher Texteditor, in dem die Constraints einer Regel editiert werden können. Der Constrainteditor zu einer Regel kann nur aufgerufen werden, wenn auch der Regeleditor dieser Regel geöffnet ist; es kann für jede Regel nur ein Constrainteditor geöffnet werden.

Ein **Menü** gibt dem Benutzer die Auswahl unter Alternativen, die in Zeilen untereinander dargestellt werden. Eine Auswahl wird getroffen, in dem eine der Zeilen angewählt wird. Ohne eine Wahl zu treffen wird das Menü geschlossen, sobald der Mauszeiger das Menüfenster verläßt.

Textfenster sind einfache Fenster, in denen dem Benutzer eine Information angezeigt wird. Beispiele sind Fehlermeldungen und Ergebnisse abgeschlossener Tests.

Ein **Dialog** dient der Eingabe von Werten über die Tastatur. In den meisten Dialogen können Werte in mehrere Felder eingetragen werden. Solange ein Dialog geöffnet ist, kann kein anderes Element der Graphikoberfläche angesprochen werden.

[1] Im Anhang wird der Begriff 'Regel' als Synonym zu 'Produktion' verwendet. Einem Benutzer, der in Graph–Grammatiken weniger bewandert ist, wird das Wort 'Regel' weniger abstrakt erscheinen.

A.1 Entwicklung einer Graph–Grammatik

Eine Graph–Grammatik besteht, einfach gesagt, aus einer Menge von *Regeln*. Jede dieser Regeln ist einer bestimmten *Sorte* zugeordnet.

Im GGD können Sorten und Regeln im Prinzip in beliebiger Reihenfolge eingegeben werden. Sollte eine Regel eingegeben werden, deren Sorte nicht bekannt ist, wird automatisch die Funktion zur Definition einer Sorte aufgerufen. Allerdings ist es sinnvoller, die Daten nicht in beliebiger, sondern einer strukturierten Reihenfolge einzugeben.
Folgende Reihenfolge möchte ich vorschlagen:

1. Zunächst sollten die Sorten aller Nonterminalsymbole eingegeben werden, zusammen mit den Super- und Subsortenbeziehungen. Wird dann direkt die Funktion zum Konsistenztest der Regelbasis aufgerufen, werden zyklische Subsortenbeziehungen entdeckt, allerdings sind Fehlermeldungen über nicht existente Regeln die Folge.

2. Danach sollten für jede einzelne Sorte die Regeln eingegeben werden. Bei ähnlichen Regeln kann die Funktion zur Kopie einer bestehenden Regel ausgenutzt werden. Auch Constraints sollten sofort eingegeben werden. Die meisten Tests zur Korrektheit einer einzelnen Regel führt GGD sofort aus. Falls TAXON benutzt wird, kann die korrekte Klassifizierung direkt überprüft werden.

3. Nachdem alle Regeln eingegeben worden sind, sollte der Konsistenztest aufgerufen werden. In vier Durchgängen wird dabei die semantische Integrität der Regelmenge überprüft. Bei Fehlermeldungen sollte der Mangel unbedingt abgestellt werden. Warnungen bezeichnen Eigenschaften der Regelmenge, die erwünscht sein können, möglicherweise aber auf ein Fehler hindeuten.

4. Schließlich sollten die Daten als FEAT-REP–Datei gespeichert werden. Dieses Dateiformat kann von GGD wieder gelesen werden. Es ist also auch möglich, die Entwicklung einer GG zu unterbrechen, die Daten abzuspeichern, und diese dann später wieder zu laden und weiter zu bearbeiten. Diese Möglichkeit besteht auch dann, wenn der Konsistenztest Fehler meldet.

   Wenn der Konsistenztest ohne Fehlermeldungen ausgeführt wird, kann die Regelmenge auch als GraPaKL-Datei abgespeichert werden. Es ist nicht sinnvoll, eine fehlerhafte Datenbasis als GraPaKL-Datei zu speichern, da diese von GraPaKL nicht zum Parsen von Werkstücken verwendet werden kann.

A.2 Benutzung der graphischen Oberfläche

Alle wichtigen Operationen zur Entwicklung einer Regelbasis können in der graphischen Benutzeroberfläche eingegeben werden. Meistens kann die Maus benutzt werden, zur Eingabe von Namen und Constraints dient die Tastatur.

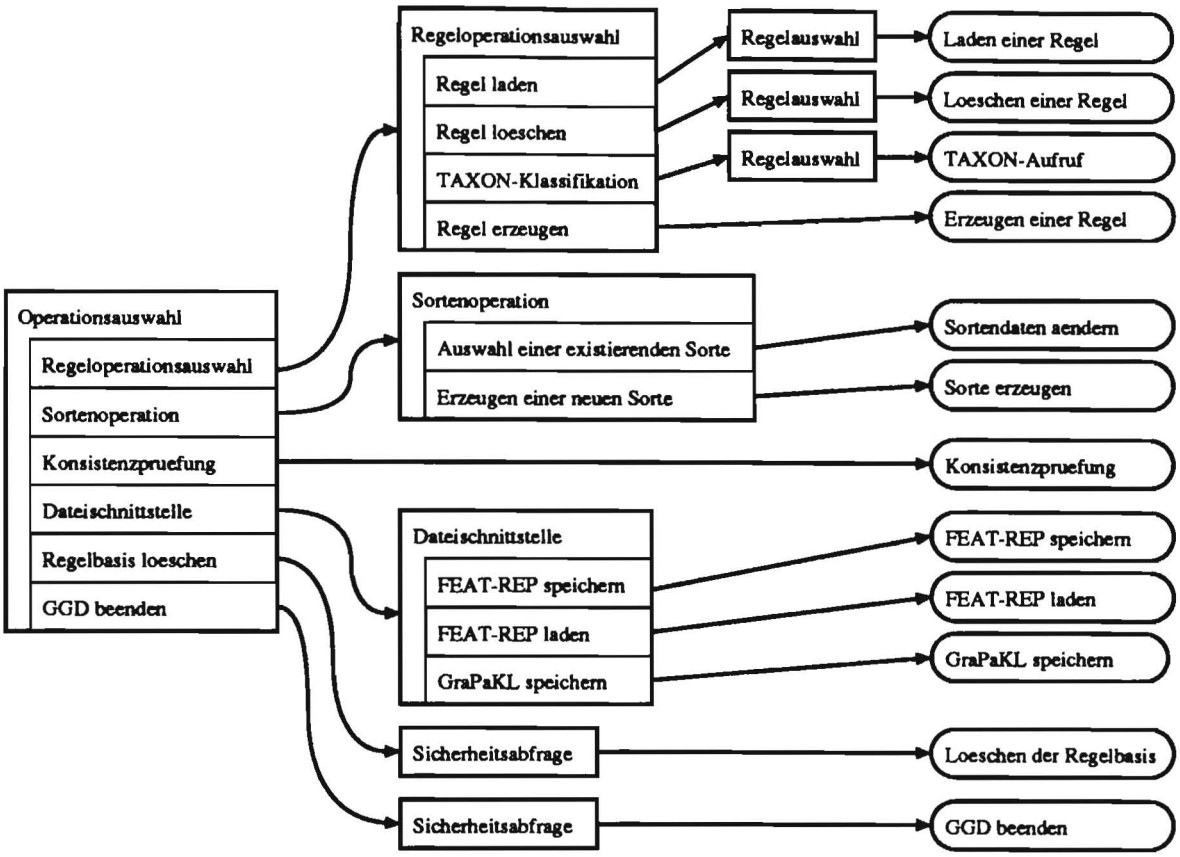


Abbildung A.1: Hierarchische Operationsauswahl

Ist das Fenster der GGD-Oberfläeche (Fenstertitel *LISP-GRAPHICS*) nicht geöffnet, kann dieses mit `(boot-ggd)` aufgerufen werden. Die Arbeit mit dem GGD kann durch `(stop-ggd)` wieder beendet werden.

Die Operationen, die sich nicht auf eine bereits dargestellte Regel beziehen, können über das hierarchische Operationsmenü aufgerufen werden. Das Top–Menü dazu erscheint bei einem Mausclick auf eine beliebige freie Position des Fensters. Die Hierarchie ist Abb. A.1 zu entnehmen.

Im folgenden soll kurz die Aufgabe der verschiedenen 'Blätter' in Abb. A.1, die Funktionsaufrufe symbolisieren, beschrieben werden.

Laden einer Regel: Für die zuvor ausgewählte Regel wird ein Regeleditor geöffnet, dort kann die Regel betrachtet und verändert werden. Die Repräsentation einer Regel und deren graphische Darstellung sind logisch voneinander getrennt, Änderungen werden erst wirksam, wenn die geänderte Regel in die Regelbasis eingefügt ('gespeichert') wird. Es ist nicht möglich (und auch nicht sinnvoll), die gleiche Regel mehrfach graphisch darzustellen. Sollte für die Regel bereits ein offener Editor existieren, bietet das System aber an, eine Kopie dieser Regel zu erzeugen. Auf diese Weise lassen sich schnell ähnliche Regeln eingeben.

Löschen einer Regel Eine Regel wird aus der Repräsentations–Komponente gelöscht. Es gibt kein UNDO, es ist also nicht möglich, eine bereits gelöschte

Regel wieder zu laden. Es ist nicht erlaubt, eine Regel zu löschen, die gerade dargestellt wird.

TAXON-Aufruf Zur ausgewählten Regel wird ihre Position in der von TAXON verwalteten Hierarchie ausgegeben.

Erzeugen einer Regel Öffnet einen leeren Regeleditor für die Sorte `Unknown`. Bevor diese Regel gespeichert wird, sollte die Sorte in eine tatsächlich exisistierende Sorte geändert werden.

In der Regelauswahl vor dem *Laden einer Regel* besteht zusätzlich die Möglichkeit, einen leeren Editor einer bestimmten Sorte zu erzeugen.

Sortendaten ändern Ein Dialog wird geöffnet, in dem die Sortendaten geändert werden können.

Sorte erzeugen Ein Dialog zur Eingabe der Sortendaten erscheint.

Konsistenzprüfung In vier Durchgängen wird die Konsistenz der Regelbasis festgestellt. Nach jedem Durchgang werden die gefundenen Fehler und Warnungen aufgezählt. Bei gefundenen Fehlern ist der Test abzubrechen, und zu wiederholen, nachdem die Fehler eliminiert wurden. Wurden nur Warnungen gefunden, kann der Test fortgesetzt werden.

FEAT-REP speichern Hier muß zunächst der Dateiname eingegeben werden. Ein Mausclick mit der rechten Maustaste auf das Eingabefeld liefert eine Liste der bereits verwendeten Namen. Außerdem kann ein Dokumentarstring eingegeben werden. Dieser sollte zur Beschreibung der Aufgabe dieser Graph–Grammatik dienen. Die Datei erhält das Suffix `.Feat-Rep`.

FEAT-REP laden Eine FEAT-REP–Datei wird wieder geladen. Die Syntax von FEAT-REP ist in A.3 beschrieben. Es ist durchaus möglich, FEAT-REP–Dateien mit einem normalen Texteditor zu verändern, und die geänderte Datei dann wieder in den GGD zu laden. Vor dem Aufruf dieser Funktion müssen alle Fenster im GGD geschlossen werden. Eine alte Regelbasis wird beim Laden einer FEAT-REP–Datei gelöscht.

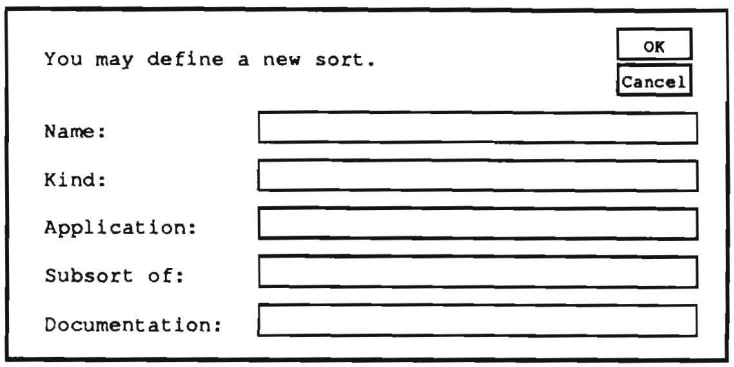
GraPaKL speichern Auch muß zunächst der Dateiname eingegeben werden. Die Datei erhält die Endung **.gg**. Ein Dokumentarstring und der Name eines *Default-Graph* können angegeben werden. Der Default-Graph wird von GraPaKL geparst, wenn kein anderer Graph angegeben wird.

Löschen der Regelbasis Nach einer Sicherheitsabfrage werden alle Regeln und Sorten sowohl in der Repräsentations- als auch der Visualisierungs-Komponente gelöscht. Die Daten der Terminalsymbole bleiben unverändert. Dateien werden nicht gelöscht. Diese Aktion kann nicht rückgängig gemacht werden.

GGD beenden Nach einer Sicherheitsabfrage wird das GGD-Fenster *LISP-GRAPHICS* geschlossen.

A.2.1 Eingabe einer Sorte

Es ist oben beschrieben, wie ein Dialog zur Eingabe oder Änderung der Daten einer Sorte geöffnet wird. Ein solcher Dialog erscheint auch dann, wenn eine Regel einer bisher unbekannten Sorte in die Repräsentationskomponente eingefügt werden soll.



Der Dialog hat etwa die hier abgebildete Form, folgende Daten lassen sich eingeben:

Name: Der *Name* einer Sorte muß von LISP als Symbol repräsentiert werden können. Erlaubt ist also jede Kombination aus Buchstaben, Ziffern und bestimmten anderen Sonderzeichen (siehe auch [Ste 90]). Groß-/Kleinschreibung ist ohne Bedeutung. Nicht erlaubt sind die Namen `T` und `NIL`, oder Namen der TEC-REP-Primitive.

Kind: Die *Art* des Features muß hier eingetragen werden, entsprechend [BKL 91b] sind die Werte `geometrical`, `functional` und `qualitative` erlaubt.

Application: Die Angabe der *Anwendung* des Features ist optional, einige mögliche Werte nach [BKL 91b] sind `design`, `assembling`, `milling`, `drilling` oder `turning`.

Subsort of: Hier können die *Supersorten* dieser Sorte eingetragen werden. Es können auch mehrere Supersorten angegeben werden (getrennt durch Leerzeichen, ohne Klammern um diese Liste). Die Subsorten einer Sorte müssen nicht explizit spezifiziert werden.

Documentation: Ein beliebiger Text (String) kann die Funktion dieser Sorte näher spezifizieren.

Wenn der Wert der globalen Variablen `*sort-order-p*` nicht `NIL` ist, fragt der Dialog auch nach einem Ordnungswert für diese Sorte. Der Default für diese Variable ist `NIL`.

Möglichkeiten, den Namen einer Sorte zu ändern, oder eine Sorte zu löschen, sind noch nicht implementiert. Ersatzweise wird empfohlen, FEAT-REP-Dateien entsprechend zu editieren.

A.2.2 Eingabe einer Produktion

Für jede in der graphischen Oberfläche bearbeiteten Regel (Produktion) wird ein *Regeleditor* geöffnet. Dies ist ein Fenster, in dem die Knoten, Kanten und Überlappungen angezeigt werden.

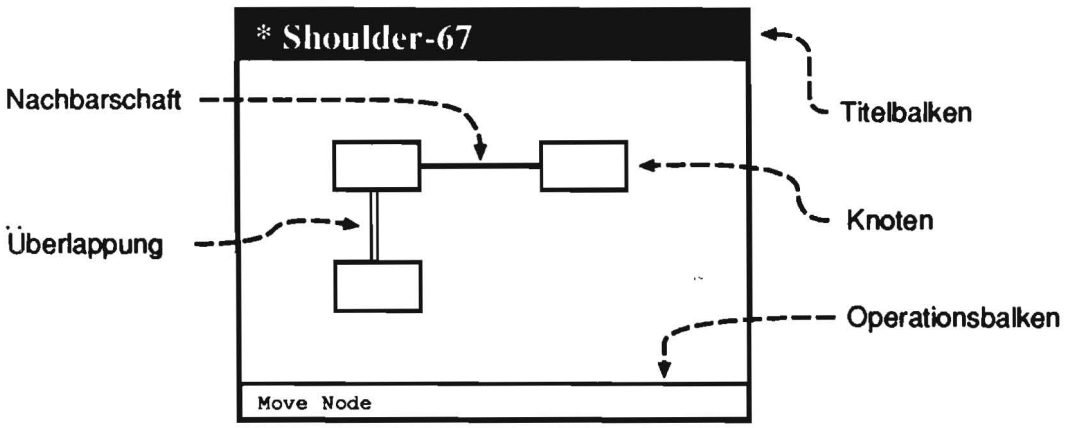


Abbildung A.2: Regeleditor mit beispielhafter Regel

Im letzten Kapitel ist beschrieben, wie ein Regeleditor geöffnet werden kann. Abb. A.2 zeigt ein solches Fenster. Die Bedienung des Regeleditors soll im folgenden anhand der verschiedenen Elemente des Fensters erklärt werden.

Der Titelbalken

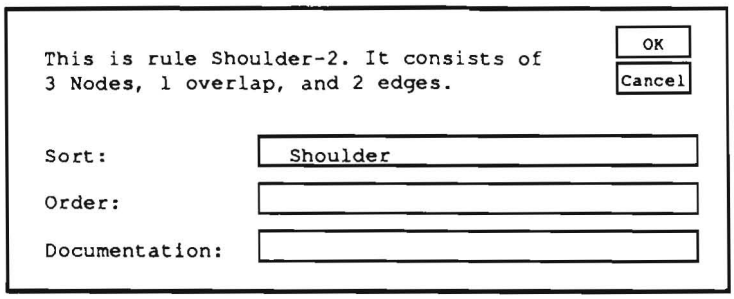
Im Titelbalken wird der aktuelle Name dieser Regel abgebildet. Ist die Regel der Repräsentation-Komponente schon bekannt, ist dies die interne Bezeichnung. Diese hat immer die Syntax **<Sorte>-<Nummer>**.
Wird eine neue Regel eingegeben, die also noch nicht in der Repräsentation-Komponente abgelegt ist, oder wurde die Sorte der Regel geändert, wird im Titelbalken nur die Regelsorte angegeben.

Das Sternchen * erscheint links vom Regelnamen, sobald die Regel geändert wurde. Es ist also sinnvoll, eine solche Regel irgendwann in der Repräsentation-Komponente zu speichern.

Durch Mausclicks auf den Titelbalken werden verschiedene Menüs aufgerufen. Mit der linken Maustauste läßt sich das *Regelmenü*, durch die rechte Maustaste das *Fenstermenü* anwählen.

Im **Regelmenü** können die folgenden Operationen aufgerufen werden:

Show / Change Rule Data Es erscheint ein Dialog, der etwa die folgende Form hat:



Im Feld **Sort** kann eine neue Sorte eingegeben werden. Das Feld darf nicht leer bleiben.

Im Feld **Order** kann ein beliebiger Wert eingegeben werden, solange er von LISP als Symbol interpretiert werden kann. Er wird von einer (Partial-) Ordnung[2] auf der Menge der Regeln verwendet.

In **Documentation** kann ein String eingegeben werden, der die Aufgabe der Regel näher beschreibt.

Show / Edit Constraints Der Constraint-Editor wird geöffnet (wenn er nicht bereits geöffnet ist). Zur Syntax von Constraints siehe A.2.3, zur Bedienung des Editors siehe A.2.3.

Save as changed rule Dieser Menüeintrag erscheint nur, wenn die Regel bereits in der Repräsentations-Komponente gespeichert ist, aber im Regeleditor verändert wurde. Eine Auswahl dieser Operation bewirkt, daß die Regel auch in der Repräsentations-Komponente geändert wird. Zuvor wird die Regel auf ihre Integrität getestet und von TAXON neu klassifiziert.
Ausnahme: Wurde die Sorte der Regel geändert, ist diese Operation nicht möglich.

Save as new Rule Eine Auswahl dieser Operation bewirkt, daß die Regel in der Repräsentations-Komponente gespeichert wird. Ein Aufruf bei einer bereits gespeicherten, unveränderten Regel bewirkt also eine Duplizierung. Dabei wird die Regel auf ihre Integrität getestet und von TAXON in die Regel-Hierarchie eingeordnet.

Quit editing this rule Der Regeleditor wird geschlossen. Ist die Regel geändert worden (zu erkennen am Sternchen im Fensterbalken), erfolgt zuvor eine Sicherheitsabfrage. Ein eventuell geöffneter Constraint-Editor wird ebenfalls geschlossen.

Im **Fenstermenü** können die üblichen Operationen zur Manipulation eines Fensters aufgerufen werden:

Move Verschieben des Fensters. Die linke obere Ecke des Fensters wird neu gesetzt, die Größe des Fensters bleibt unverändert.

Reshape Verändern der Grösse der Fensters. Die rechte untere Ecke des Fensters wird neu festgesetzt.

[2]Die Ordnung selber wird durch die Funktion **extend-order** eingegeben.

Expose Holt das Fenster gegenüber allen anderen Fenstern in der Vordergrund.

Hide Versteckt das Fenster hinter allen anderen Fenstern im Hintergrund.

Der Operationsbalken

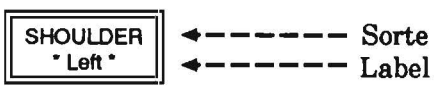
Am unteren Rand eines Regeleditors findet sich der Operationsbalken. Hier kann die aktuelle Operation zur Veränderung der rechten Seite der Regel ausgewählt werden. Der Name der aktuellen Operation wird angezeigt.

Folgende Operationen können ausgewählt werden:

New Node - Show Node Diese Operation erfüllt die zwei Aufgaben, Knoten zu erzeugen, und ihre Daten zu ändern. Es ist die einzige Operation, bei der ein Mausclick in die freie Fensterfläche (außerhalb eines Knotens) eine Funktion hat.

Knoten in einem Regeleditor haben folgende Form:

Darstellung eines Kontextknotens:

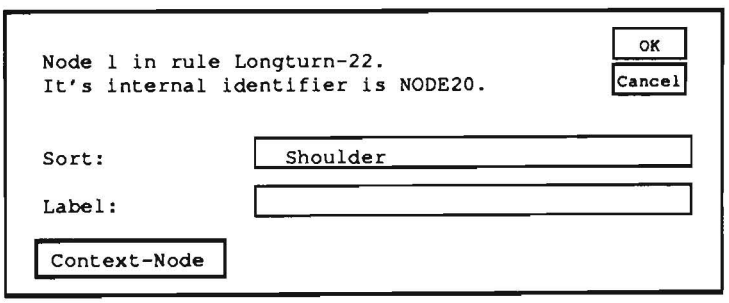


Neben der **Sorte** und der **Nummer** des Knotens wird, falls vorhanden, das **Label**, also ein Bezeichner, dargestellt.

Kontextknoten sind an ihrer besonderen Darstellung zu erkennen. Sie besitzen keine Nummer, können aber mit einem Label versehen werden.

Durch Click auf die freie Fläche des Fensters wird ein neuer Knoten erzeugt. Seine obere linke Ecke wird auf den Ort des Mausclicks gesetzt. Der Benutzer hat selber darüber zu wachen, daß sich der neue Knoten nicht mit bereits vorhandenen Knoten überlappt. Der neue Knoten erhält automatisch eine eindeutige Nummer sowie die Sorte **ANY**.

Durch Anwahl eines Knotens erscheint der folgende Dialog, in dem die Sorte und das Label des Knotens geändert werden können. Durch Anwahl des Feldes **Context-Node** kann ein Knoten zum Kontextknoten erklärt werden. Die interne Bezeichnung ist für den Anwender unerheblich.



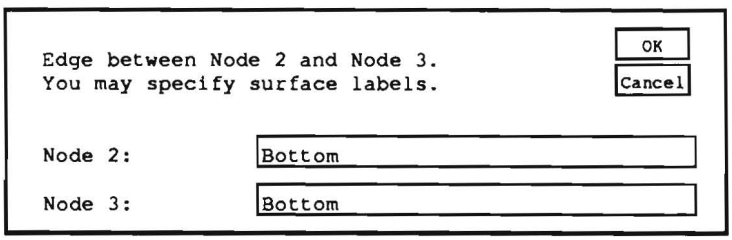
Im Allgemeinen wird man diesen Dialog immer nach Erzeugen eines Knotens aufrufen um die Sorte zu ändern. Als Sorte ist jedes Terminal und Nonterminal erlaubt, mit der Ausnahme von `Workpiece`.

Move Node Verschieben eines Knotens. Dazu wird der Knoten ausgewählt, dann verschoben, und mit einem weiteren Mausclick an seine neue Position gesetzt. Das System wacht darüber, daß Knoten nicht außerhalb des Regelfensters positioniert werden, innerhalb des Fensters ist der Benutzer dafür verantwortlich, daß sich Knoten nicht überlappen.

Delete Node Löscht einen Knoten, sowie alle Kanten und Überlappungen, die mit diesem Knoten inzident sind. Während der Operation nimmt die Maus die Form eines Totenkopfes an.

New Edge Erzeugt eine neue Nachbarschafts-Kante zwischen zwei Knoten. Dazu werden nacheinander beide Knoten angewählt. Zwischen zwei Knoten kann es jeweils nur eine Kante geben.

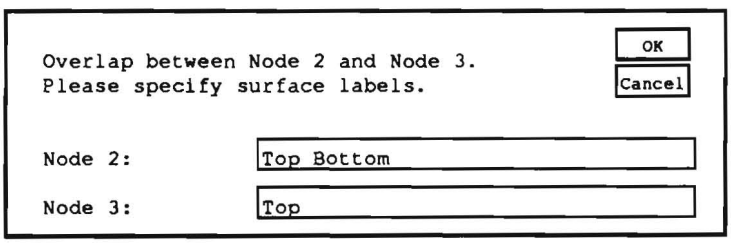
Show Edge Erlaubt die Eingabe der in 4.1 erklärten *tiefen Nachbarschaften*. Nachdem zwei Knoten angewählt wurden, zwischen denen eine Kante existiert, erscheint der folgende Dialog:



Es kann für beide Knoten ein Pfad, also eine Liste von Label eingegeben werden. Dieser Pfad darf aber auch leer sein.

Delete Edge Löscht die Nachbarschaftskante zwischen zwei Knoten. Dazu werden nacheinander die beiden Knoten angewählt.

New Overlap Erzeugt eine Überlappungskante zwischen zwei Knoten. Dazu werden nacheinander beide Knoten angewählt. Es erscheint dann sofort der folgende Dialog:



Bei einer Überlappung muß für jeden der beiden Knoten ein Pfad (eine Liste von Labels) angegeben werden, der nicht leer sein darf. Zwischen zwei Knoten kann es jeweils nur eine Überlappungskante geben.

Show Overlap Erlaubt die Veränderung der bei der Erzeugung der Überlappung eingegebenen Labels. Der gleiche Dialog wie bei **New Overlap** wird dazu geöffnet.

Delete Overlap Löscht die Überlappungskante zwischen zwei Knoten. Dazu werden nacheinander die beiden Knoten angewählt.

Renumber Nodes Vergabe von neuen Nummern an die Knoten einer Regel. Die Nummern der Knoten stellen auch die Reihenfolge dar, in der GraPaKL im Parse nach den Knoten sucht. Eine Umnumerierung der Knoten kann aus diesen Gründen notwendig werden:

- Durch das Löschen von Knoten sind die Knoten nicht mehr durchgehend von 1 bis n (bei n Knoten) numeriert.
- Der Graph gehorcht nicht mehr einer GraPaKL-spezischen Bedingung, nach der jeder Knoten (außer demjenigen mit Nummer 1) zu mindestens einem Knoten mit einer kleineren Nummer durch eine Kante verbunden sein muß.
- Aus Optimierungsgründen wird eine andere Reihenfolge gewünscht.

Nach Auswahl dieser Operation sind die Knoten in der gewünschten Reihenfolge auszuwählen, wobei der erste Knoten die Nummer 1 erhält, der zweite die 2, u.s.w. Zur besseren Übersicht sind die noch nicht neu numerierten Knoten schwarz gefärbt. Kontextknoten werden bei der Umnumerierung nicht berücksichtigt.
Nach Abschluß der Operation werden automatisch, falls notwendig, Referenzen in den Constraints auf die Knotennummern neu gesetzt.

A.2.3 Verwendung von Constraints

Bisher wurde beschrieben, wie Knoten und Kanten einer Regel eingegeben werden. Zusätzlich kann eine Regel mit **Constraints** versehen werden. Constraints dienen zur Formulierung von Bedingungen, die die Regel erfüllen muß, zur Angabe der Berechnungsvorschriften für Attribute, sowie für Rollenzuweisungen.

Die Syntax der Constraints orientiert sich im wesentlichen an der Constraint-Syntax des GraPaKL[3]. Die Constraints zu einer Regel können mit dem später beschriebenen *Constraint-Editor* eingegeben und modifiziert werden.

```
      <constraint> ::= <path-constraint> |
                       <predicate> |
                       (%-role <role-label> <path-constraint> )
                       (%-role <role-label> <predicate> )
<path-constraint> ::= (nr    <path> <path>) |
                       (ov    <path> <path>) |
                       (no-nr <path> <path>)
```

[3]Die folgenden Abschnitte sind teilweise aus [Mau 92] übernommen.

```
                <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-sort> <role-label> <form>)
 <subsort-declaration> ::= (subsort <form>)
```

Dabei bedeuten die Komponenten im einzelnen folgendes:

constraint Kann ein **path-constraint** oder ein **path** sein. Durch Einbettung in das **%-role**–Konstrukt kann ein bestimmter Auswertezeitpunkt in einem GraPaKL-Parse angegeben werden (siehe B.4).

path-constraint Werden im GGD normalerweise nicht benötigt, da diese Informationen graphisch eingegeben werden können. Um aber Fälle wie mehrere Kanten zwischen zwei Knoten behandeln zu können, erlaubt GGD auch **path-constraints**.

path Pfade bezeichnen Knoten und Attribute in Knoten der rechten Regelseite und deren Komponenten. Ein Pfad, der mit dem Schlüsselwort **sort** endet, liefert die Sorte des Knotens 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 Durch Prädikate können Bedingungen formuliert und Attribute berechnet werden. 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 Knoten- und Attributwerten bereits gefundener Knoten berechnet werden, auch mit Hilfe benutzerdefinierter Funktionen.

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. Man beachte den Unterschied zur GraPaKL-Syntax: Um die Sorte des Rollenlabels zu bestimmen, muß diese hier angegeben werden.

subsort-declaration Spezialisiert die Regel-Sorte auf eine berechnete Untersorte. Natürlich muß die Subsorten–Beziehung definiert sein.

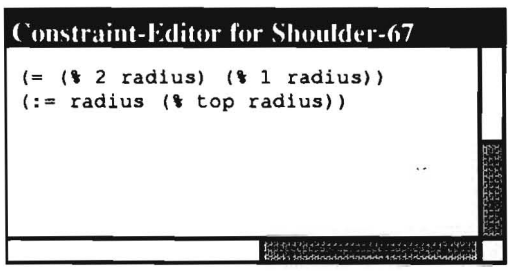
lisp-form Ein beliebiger Lisp-Ausdruck. Neben allen Standard Common Lisp Funktionen sind folgende Prädikate für Vergleiche in der Sortenhierarchie verwendbar:

(**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 $s2 \leq_S s1$.

Der Constraint-Editor

Dieser Texteditor wird durch Auswahl des entsprechenden Menüpunktes im Regelmenü einer Regel geöffnet. Er hat etwa folgendes Aussehen:



Durch die Maus kann der Cursor (ein senkrechter Strich) positioniert werden. Neben den üblichen Tasten zur Eingabe von Zahlen und Buchstaben sind folgende Tasten belegt (auf einer SUN-Workstation):

BackSpace oder Delete	Löscht das Zeichen links des Cursors
← oder Control-b	Setzt den Cursor um eine Position nach links
→ oder Control-f	Setzt den Cursor um eine Position nach rechts
↑ oder Control-p	Setzt den Cursor um eine Zeile nach oben
↓ oder Control-n	Setzt den Cursor um eine Zeile nach unten
Control-a	Setzt den Cursor an den Zeilenanfang
Control-e	Setzt den Cursor an das Zeilenende
Home	Setzt den Cursor an den Zeilenanfang der ersten Zeile
End	Setzt den Cursor an das Zeilenende der letzten Zeile
PgDn oder Control-v	Blättert eine Seite nach unten
PgUp oder Meta-v	Blättert eine Seite nach oben
Control-k	Löscht die aktuelle Zeile
Control-y	Fügt eine gelöschte Zeile vor der aktuellen Zeile ein.

Steht der Cursor hinter einer schließenden Klammer), wird die entsprechende öffnende Klammer (invertiert dargestellt.

Durch Click auf den Fensterbalken des Constraint-Editors erscheint ein Menü, welches folgende Operationen anbietet:

Save 'Speichert' den Fensterinhalt, d.h. der Text wird als neue Constraintmenge der Regel interpretiert. Dabei wird getestet, ob der Text ebensoviele öffnende wie schließende Klammern enthält.

Move, Reshape, Expose, Hide Die üblichen Funktionen, um ein Fenster zu verschieben, seine Größe zu ändern und um es vor bzw. hinter andere Fenster zu bringen.

Exit Beendet den Constraint-Editor. Zuvor wird der Text wie bei **Save** 'gespeichert'.

A.2.4 Benutzung von TAXON

Alle Regeln, die mit Hilfe des GGD eingegeben werden, werden durch TAXON klassifiziert. Die Klassifizierung wurde in 6 näher erläutert.

Durch den Befehl (`setf *use-taxon* NIL`) kann die Verwendung von TAXON abgeschaltet werden. Dies kann beispielsweise aus Effizienzgründen sinnvoll sein. Es ist aber nicht möglich, TAXON dann nachträglich wieder zu benutzen (neue Regeln sind nicht klassifiziert worden). Soll die Taxonomie wieder korrekt aufgebaut werden, ist folgendes Verfahren empfehlenswert:

1. Speichern der Regeln als FEAT-REP–Datei.
2. Eingabe von (`setf *use-taxon* T`).
3. Laden der gespeicherten FEAT-REP–Datei.

Die von TAXON errechnete Klassifikation einer Regel durch die folgenden Operationen ausgegeben:

1. Nach dem Einfügen einer neuen Regel die Repräsentations–Komponente, z.B. beim 'Speichern' einer in der Visualisierungs–Komponente eingegebenen Regel. Eine Ausnahme ist das Laden einer FEAT-REP–Datei, aus einsichtigen Gründen wird dabei nicht für jede Regel die errechnete Klassifikation ausgegeben.
2. Aufruf der Operation *'Show TAXON–Classification'* in der Regeloperationsauswahl

Es wird dann ein Fenster geöffnet, das diese Daten zur Klassifikation ausgibt:

- Die Namen der Regeln der gleichen Sorte, die von TAXON als äquivalent, als direkt unter, bzw. als direkt über der Regel klassifiziert werden.
- Die Namen der Regeln anderer Sorten, die von TAXON als äquivalent, als direkt unter, bzw. als direkt über der Regel klassifiziert werden.

A.3 Das Dateiformat FEAT-REP

Die Syntax für FEAT-REP–Dateien, wie sie vom GGD erzeugt und gelesen wird, ist die folgende:

```
<FEAT-REP-file> ::= { ( Documentation <documentation> ) }
                    { ( Order <orderset> ) }
                    <feature>*

<orderset>      ::= ( <order>* )
<order>         ::= ( <symbol> <symbol>+ )
```

```
<feature>       ::= (   ( Featurename <sort-name> )
                        ( Featurekind <featurekind> )
                      { ( Featureapplication <string> ) }
                      { ( Documentation <documentation> ) }
                      { ( Supersorts ( <sort-name>+ ) ) }
                      { ( Subsorts ( <sort-name>+ ) ) }
                      { ( Featureorder <symbol> ) }
                      { ( Featurerules ( <rule>* ) ) } )

<rule>          ::= (   ( Rule-Id <rule-name> )
                      { ( Documentation <documentation> ) }
                      { ( Order <symbol> ) }
                      { ( Nodes ( <node>* ) ) }
                      { ( Kontext-Nodes ( <node>* ) ) }
                      { ( Edges ( (path path)+ ) ) }
                      { ( Overlaps ( (path path)+ ) ) }
                      { ( Constraints ( <constraint>+ ) ) } )

<node>          ::= (   ( Node-number <node-number> )
                      { ( Node-label <symbol> ) }
                      { ( Node-x <integer> ) }
                      { ( Node-y <integer> ) } )

<featurekind>   ::= geometrical | functional | qualitative
<path>          ::= ( <node-number> <node-label>+ )
<documentation> ::= <string>
<node-number>   ::= <integer>
```

Die Syntax eines `<constraint>` wurde bereits in A.2.3 aufgeführt. `<string>`, `<symbol>` und `<integer>` stehen für die Definition eines Strings, eines Symbols bzw. eines Integerwertes in LISP.

`<orderset>` gibt die Partialordung an, die auf den Symbolen der Regel definiert wurde. Jedes Element von `<orderset>` ist eine Liste von Symbolen $(s_0\, s_1, \ldots s_n)$ mit (s_0, s_i) Element der Ordnung für alle $i \in [1, \ldots, n]$.

Die Klassifizierung der Regeln durch TAXON wird nicht in der FEAT-REP–Datei gespeichert, die Hierarchie wird immer wieder neu berechnet.

Man beachte, daß diese Syntax von der in [BKL 91b] vorgeschlagenen teilweise abweicht. Die wichtigsten Unterschiede:

- Die Ausdrücke für die drei Featurearten *Qualitative-Feature*, *Funktional-Feauture* und *Geometrical-Feature* werden hier zu einem Ausdruck zusammengefaßt; dieser eine Audruck hat einen Slot `<featurekind>`.
- Die Slots für *Spezialize-Feature* und *Subsumes-Features* wurden durch Slots für *Supersorts* und *Subsorts* ersetzt. Die Slots für *has-parts* und *is-part-of*, die nur redundante Informationen enthielten, wurden gestrichen.

- *Rule-attributes* werden jetzt explizit den verschiedenen Regeln zugeordnet, und nicht, wie in der alten Syntax, zusätzlich zu den Regeln in einem eigenen Slot angegeben.
- Die alte Syntax für Regeln erlaubte nicht, die Struktur einer Regel adequat auszudrücken. Die Syntax wurde daher geändert.
- Neu ist die Möglichkeit, zu jeder Produktion (Regel) einen Ordnungswert anzugeben, und eine Ordnung auf diesen Symbolen zu spezifizieren.
- Die Option, einen Dokumentationsstrings anzugeben, wurde ergänzt.

Anhang B

Implementierung des GGD

GGD ist implementiert in COMMON-LISP ohne Verwendung von CLOS. Es umfasst mehr als 250 Funktionen und Macros in über 15 Dateien, die Größe aller Dateien beträgt knapp 300 KB. Das System ist sowohl unter Symbolics Genera wie auch unter Kyoto Common Lisp lauffähig.

Im folgenden sollen nicht alle Einzelheiten der Implementierung erläutert werden. Es soll aber gezeigt werden, wie der GGD aufgebaut ist, welches die wesentlichen Dateien, Datenstrukturen und Funktionen sind, und wie GGD auch ohne Graphik-Oberfläche benutzt werden kann. Damit haben sowohl der Autor eines Programmes, das auf GGD-Funktionen zugreift, als auch ein Programmierer, der den GGD erweitern will, einen Ansatzpunkt für ihre Arbeit.

B.1 Zugriff auf DAG's

Der DAG als zentrale Datenstruktur des GGD wurde in 8.3 vorgestellt. Hier soll seine Verwendung aufgezeigt werden.

Jeder DAG wird in einer (oft globalen) Variablen gespeichert. So gibt es u.a. eine Variable `*rules*` für die repräsentierten Regeln, und eine Variable `*sorts*` für die Sorten.

Zur Verwaltung von DAG's werden nur wenige Funktionen benötigt. Diese reichen aus, um Attribute und Werte zu setzen und abzufragen. Realisiert sind DAG's im GGD über A–Listen. Dadurch erreicht man eine relativ effiziente, einfache Implementierung von DAG's.

`(dag-value dag path)` ist die wichtigste der Funktionen. `dag` ist der Name einer bereits gebundenen Variablen, `path` der Pfad, über den zugegriffen werden soll.

Mit einem Aufruf von `(dag-value)` wird der Wert des durch den Pfad spezifizierten Attributs ausgegeben. Existiert der Pfad in diesem DAG nicht, wird `NIL`, aber kein Fehler ausgegeben. Ist der Wert des spezifizierten Attributs wiederum ein DAG, wird dieser DAG ausgegeben.

Außerdem kann `(dag-value)` zusammen mit `setf` zum Setzen eines Wertes eines Attributs benutzt werden. Existiert ein Pfad nicht, wird er bei dieser Operation

generiert.
Um ein Attribut zu löschen, wird der Wert des Attributs auf `NIL` gesetzt.

Beispiel:
`(dag-value *sorts* '(shaft application)`
ermittelt den Wert des Attributs `application` für die Sorte `shaft`.
`(setf (dag-value *rules* '(shoulder-64 nodes)) NIL)`
löscht alle Knoten in der Regel `shoulder-64`.

B.2 Funktionen und Macros des GGD

GGD kann auch ohne Graphik-Oberfläche benutzt werden. Das bietet sich an beispielsweise für Programme, die auf die GGD–Wissensbasis zugreifen möchten. Es folgt daher ein Überblick über die wichigsten GGD-Funktionen. Gleichzeitig soll er als Information für jeden dienen, der das Programm GGD verändern will.

Das erste Wort innerhalb der Klammern ist jeweils der Funktionsname, die anderen Wörter sind Namen für die Argumente.

Zugriffsmacros

`(sort-kind sortname)` Die Art des Features, das durch diese Sorte repräsentiert wird.

`(sort-application sortname)` Die Art des Features, das durch diese Sorte repräsentiert wird.

`(sort-documentation sortname)` Der String zur Dokumentierung dieser Sorte.

`(sort-rule-ids)` Eine Liste der Namen aller Regeln, deren linke Seite diese Sorte hat.

`(rule-sort rule-id)` Die Sorte der linken Seite der Regel, kurz die Sorte der Regel.

`(rule-nodes rule-id)` Die Knoten der Regel. Für den Zugriff auf die Knoten-Slots gibt es eigene Zugriffsmacros (s.u.).

`(rule-context-nodes rule-id)` Die Kontextknoten der Regel. Auf ihre Slots kann mit den gleichen Macros wie für die regulären Knoten zugegriffen werden.

`(rule-constraints rule-id)` Liefert eine Liste der aktuellen Constraints.

`(rule-edges rule-id)` Die Kanten der Regel. Die Syntax entspricht der in A.3 für `Edges` angegebenen.

`(rule-overlap rule-id)` Die Überlappungen (Feature-Interactions) der Regel. Die Syntax entspricht der in A.3 für `Overlaps` angegebenen.

`(rule-order rule-id)` Das Symbol, daß zum Aufbau einer Partialordnung auf der Regelbasis genutzt werden kann.

`(rule-documentation rule-id)` String, der nähere Informationen zur Regel liefern kann.

`(node-number node)` Die Nummer des Knotens.

`(node-x node)` Die x-Koordinate des Knotens in einem Regeleditor.

`(node-y node)` Die y-Koordinate des Knotens in einem Regeleditor.

`(node-sort node)` Die Sorte des Knotens.

`(node-label node)` Das Label des Knotens.

Diese Macros können auch mit dem LISP-Befehl `setf` benutzt werden.
Beispiel: `(setf (rule-order 'band-34) 3)` setzt den Ordnungswert der Regel `Band-34` auf 3.

Die `(sort-...`-Macros brauchen als Argument den Sortennamen, die `(rule-...`-Macros den Regelnamen, die `(node-...`-Macros benötigen einen DAG, der einen Knoten repräsentiert. Das Macro `rule-nodes` liefert dementsprechend eine Liste von DAGs, deren Elemente für die `(node-...`-Macros geeignet sind.

Funktionen zur Regel– und Sortenmanipulation

`(new-sort &key name kind application order supersort docu)`
Definition einer neuen Sorte. Alle Angaben außer *name* und *kind* sind optional.

name Der Name der Sorte. Muß ein LISP–Symbol sein.

kind Art des Features. Erlaubte Werte sind die Strings 'geometrical', 'functional' und 'qualitative'.

application Ein String zur Anwendung des Features.

order Symbol, kann zum Aufbau einer Ordnung auf den Sorten genutzt werden.

supersort Liste der Supersorten dieser Sorte.

docu Ein String, der Informationen zur Sorte gibt. Um den String in GGD-Dialogen darstellen zu können, sollte er kein *NewLine*-Zeichen enthalten.

`(list-all-sorts)` Liefert eine Liste der Namen aller Sorten.

`(list-all-sorts-with-rules)` Liefert eine Liste der Namen aller Sorten, für die jeweils mindestens eine Regel definiert wurde.

`(direct-sub-sorts sortname)` Gibt eine Liste der Namen aller direkten Subsorten zurück.

`(direct-super-sorts sortname)` Gibt eine Liste der Namen aller direkten Supersorten zurück.

`(sub-sorts sortname)` Gibt eine Liste der Namen **aller** Subsorten, also auch transitiv, zurück.

`(super-sorts sortname)` Gibt eine Liste der Namen **aller** Supersorten, also auch transitiv, zurück.

`(direct-sub-sort-p sort1 sort2)` Liefert genau dann `T`, wenn sort1 eine direkte Subsorte von sort2 ist.

`(direct-super-sort-p sort1 sort2)` Liefert genau dann `T`, wenn sort1 eine direkte Supersorte von sort2 ist.

`(sub-sort-p sort1 sort2)` Liefert genau dann `T`, wenn sort1 eine Subsorte (auch transitiv) von sort2 ist.

`(super-sort-p sort1 sort2)` Liefert genau dann `T`, wenn sort1 eine Supersorte (auch transitiv) von sort2 ist.

`(set-rule &key` *siehe Aufzählung*) Definition einer neuen Regel bzw. Änderung einer bestehenden. Folgende Schlüsselwörter sind erlaubt:

`sort` Die Sorte der Regel.

`order` Ein Symbol, das zum Aufbau einer Partialordnung auf allen Regeln benutzt werden kann.

`nodes` Liste der Knoten dieser Regel. Die Spezifizierung der Knoten hat folgende Syntax:

```
( :number <integer>
  :sort   <sortname>
  :label  <symbol>
  :x      <integer>
  :y      <integer> )
```

`context-nodes` Liste der Kontext-Knoten dieser Regel. Die Syntax entspricht der für dir `nodes`.

`edges` Eine Liste der Kanten. Die Syntax entspricht der in A.3 für `Edges` angegebenen.

`overlaps` Die Überlappungen (Feature-Interactions) der Regel. Die Syntax entspricht der in A.3 für `Overlaps` angegebenen.

`documentation` Ein Dokumentationsstring. Er sollte keine *NewLine*-Zeichen enthalten.

`feat-rep-id` Normalerweise wird für eine neue Regel von dieser Funktion ein neuer Name erzeugt. Durch `feat-rep-id` kann eine neue Regel aber auch einen bestimmten Namen erhalten. Dies wird derzeit benutzt, wenn Regeln aus einer FEAT-REP–Datei gelesen werden.

`old-rule-id` Der Name einer bestehenden Regel, falls diese geändert werden soll. Am Vorhandensein dieses Argument erkennt GGD, daß keine neue Regel erzeugt werden soll.

Vor dem Einfügen einer Regel in die Repräsentationskomponente wird `(check-rule)` zur Überprüfung der Konsistenz der Regel aufgerufen, danach `(tx-insert-rule)`.

`(delete-rule rule-id)` Löscht die Regel mit Namen `rule-id`.

`(list-all-rules)` Liefert eine Liste der Namen aller Regeln in der Repräsentations-Komponente.

`(extend-order symbol-1 symbol-2)` Dient zur Definition einer Ordung auf den entsprechenden Symbolen, die zu jeder Regel angegeben werden können. Das geordnete Paar `(symbol-1 symbol-2)` wird als zusätzliches Element in die Ordnung aufgenommen.

`(check-rulebase)` Testet die Konsistenz der Regelmenge (in vier Durchgängen).

(help) Gibt die Namen der wichtigsten Funktionen zur Bedienung des GGD ohne Graphik-Oberfläche aus.

(init-ggd) Initialisiert die Regelbasis und alle globalen Variablen, außerdem TAXON. Die Graphikoberfläche wird nicht geöffnet oder geschlossen.

(boot-ggd) Öffnet die Graphikoberfläche, falls sie noch nicht geöffnet ist. (init-ggd) wird aufgerufen.

(stop-ggd) Schließt die Graphikoberfläche. Diese Funktion oder die gleichnamige Operation im Operationsauswahlmenü sollte auf jeden Fall zum Abschluß der Benutzung der Graphikoberfläche aufgerufen werden.

Funktionen zum Zugriff auf TAXON

Da einige Tricks notwendig waren, um gewisse Mängel von TAXON zu überwinden (siehe B.3), mußten zusätzliche Funktionen zum Zugriff auf TAXON definiert werden.

(tx-insert-rule rule-id &key query) Klassifizierung einer Regel durch TAXON. Wenn query ungleich NIL ist, wird ein Fenster mit Angaben zur Klassifizierung dieser Regel geöffnet.

(tx-show-and-edit-classification rule-id) Öffnet ein Fenster mit Angaben zur Klassifizierung dieser Regel. Eine Veränderung der Klassifikation (z.B. durch Disjunktionen) ist bisher nicht möglich.

(tx-upper-same-sort-rules rule-id) Namen der Regeln der gleichen Sorte, die TAXON als direkt über rule-id stehend klassifiziert.

(tx-lower-same-sort-rules rule-id) Namen der Regeln der gleichen Sorte, die TAXON als direkt unter rule-id stehend klassifiziert.

(tx-equivalent-same-sort-rules rule-id) Namen der Regeln der gleichen Sorte, die TAXON als äquivalent zu rule-id klassifiziert.

(tx-upper-struct-rules) Namen der Regeln beliebiger Sorte, die TAXON als direkt über rule-id stehend klassifiziert.

(tx-lower-struct-rules) Namen der Regeln beliebiger Sorte, die TAXON als direkt unter rule-id stehend klassifiziert.

(tx-equivalent-struct-rules) Namen der Regeln beliebiger Sorte, die TAXON als äquivalent zu rule-id klassifiziert.

Ein-/Ausgabe von Dateien

(load-feat-rep path) Lädt eine FEAT-REP-Datei. Die Syntax dieser Datei ist in A.3 beschrieben. path ist ein Pfad, der die zu ladende Datei spezifiziert.

(save-feat-rep path &optional docu) Speichert eine Datei im FEAT-REP-Format. docu ist ein optionaler String, der die Grammatik dokumentiert.

`(save-grapakl path &optional docu default)`
Speichert eine Datei im GraPaKL–Format. `docu` ist ein optionaler Dokumentations–String; `default` ist der Name einer Datei, die GraPaKL als Beschreibung eines Default–Graphen verwenden soll.

B.3 Einbindung von TAXON

Im GGD wird eine Hierarchie der Regeln berechnet: Jede einzelne Regel wird in ein TAXON-Konzept umgesetzt (wie in Kapitel 6 beschrieben), dieses Konzept wird in der T–BOX von TAXON klassifiziert, und diese Klassifizierung wird als Hierarchie auf den Regeln interpretiert. Die Möglichkeiten, die die A–BOX von TAXON bietet, werden nicht ausgenutzt.

Bei der Einbindung von TAXON tauchten aber einige Probleme auf, die sich aus verschiedenen Eigenschaften und Fehlern von TAXON ergeben. Im folgenden wird geschildert, welche Anforderungen an ein System zur Klassifizierung von Regeln gestellt wurden, wo sich dabei Probleme mit TAXON ergaben, und wie diese, falls möglich, gelöst wurden.

- Es ist einem Benutzer des GGD erlaubt, Regeln zu ändern und zu löschen. Jede Regeländerung hat i.allg. auch eine Änderung des entsprechenden TAXON–Konzeptes zur Folge, dieses müßte dann also geändert oder gelöscht werden. Es besteht aber in TAXON keine Möglichkeit ein einmal definiertes Konzept zu ändern oder zu löschen.

- Es wäre schön, könnte man das Konzept zu einer Regel 'probeweise' in eine Hierarchie eingeordnen. So könnte der Benutzer testen, ob eine Regel wunschgemäß klassifiziert wird, um dann nötigenfalls die Regel zu ändern. Leider ist das entsprechende TAXON–Konstrukt fehlerhaft. Diese Möglichkeit besteht also nicht.

- Es ist im GGD sinnvoll, mehrere Hierarchien zu verwalten. Man könnte sowohl Hierarchien für die Regeln einzelner Sorten als auch eine Hierarchie für die Regeln aller Sorten aufbauen. TAXON ordnet aber alle Konzepte immer in genau eine Hierarchie ein.

- Es gibt keine Möglichkeiten, *Disjunktionen* zu erklären.

Die ersten drei Probleme konnten mit der im Folgenden erklärten Einbindung gelöst werden, Disjunktionen bleiben zukünftigen Erweiterungen vorbehalten.

- Das eine Regel beschreibende Konzept wird nicht unter dem Namen der Regel in TAXON eingefügt, sondern unter generisch erzeugten Namen. Funktionen im GGD übernehmen die Abbildung von den Regelnamen zu generierten Namen.

- Jede Regel wird zweimal (unter verschiedenen) Namen in TAXON eingefügt. Einmal als Konzept, daß keine Information über die Sorte der linken Seite der Regel enthält, einmal mit dieser Information. Auf diese Weise lassen sich mehrere Hierarchien aufbauen, eine für jede Sorte, eine weitere für alle Regeln aller Sorten.

  Beispiel: Die Regel `Schulter-23` wird in ein TAXON–Konzept umgesetzt, dieses Konzept wird ohne Sortenname der linken Regelseite als `I0028`, mit Sortennamen als `S0029` in TAXON eingefügt. Um die in der von TAXON berechneten Hierarchie über `Schulter-23` stehenden Schultern zu ermitteln, werden die Konzepte über `S0029` abgefragt.

- Wird eine Regel gelöscht, nimmt GGD die für diese Regel generierten Konzeptnamen in eine dafür vorgesehene Liste auf. TAXON kennt diese Konzepte dann immer noch. GGD–Funktionen müssen daher zusätzliche Berechnungen anstellen, wenn TAXON–Anfragen den Namen einer gelöschten Regel zurückliefern.

  Beispiel: Die Regel `Schulter-23` soll gelöscht werden. Neben den Änderungen in den GGD–Datenstrukturen werden `S0029` und `I0028` als gelöscht markiert. Liefert nun eine Anfrage zu einer anderen Regel z.B. `S0029` als höherstehenden Konzept, müssen für eine korrekte Antwort die Konzepte über `S0029` ermittelt werden.

Die oben erklärte Einbindung hat auch gewisse Nachteile, und zwar ein geringer Performanceverlust sowie eine Reduzierung der Lesbarkeit der TAXON-Hierarchie, wenn man nicht die in B.2 aufgeführten Funktionen benutzt. Dies wirkt sich um so mehr aus, je mehr Regeln während des Benutzung des GGD gelöscht bzw. geändert werden.

B.4 Generierung von GraPaKL-Dateien

Um eine GGD–Wissensbasis in eine GraPaKL–Datei auszugeben, sind einige zusätzliche Berechungen notwendig. Voraussetzung ist der bestandene Konsistenz–Check, u.a. weil in der aktuellen Implementierung während des Konsistenz–Checks einige Werte gesetzt werden, die für die GraPaKL–Ausgabe benötigt werden[1].

Die GraPaKL–Syntax verlangt, daß alle Attributnamen und Knotendaten einer Sorte getrennt von den eigentlichen Regeln angegeben werden. Für jeden Knoten muß Name und Sorte angegeben werden. Namenskonflikte dürfen nicht auftreten. Knoten mit gleichem Namen müssen in allen Regeln eines Features der gleichen Sorte angehören. Die Menge der Namen der Attribute darf sich nicht mit der Menge der Namen der Knoten überschneiden.

Im GGD haben zwar alle Knoten eine *Nummer*, oft aber kein *Label*, also einen expliziten Namen. Die Label dienen dazu, bestimmte Knoten in Überlappungen und

[1] Außerdem läßt sich eine GraPaKL-Datei aus einer fehlerhaften Regelbasis nicht sinnvoll verwenden.

Pfaden anzusprechen. Weil i.allg. nicht alle Knoten einer Regel auf diese Weise genutzt werden, ist es nicht sinnvoll, für jeden Knoten ein Label zu verlangen. *Attribute* werden durch Constraints eingeführt.

Für jede Sorte werden bei der GraPaKL–Ausgabe zunächst die Namen der Attribute und Label aus allen Regeln gesammelt. Es wird getestet, ob sich Namenskonflikte zwischen Attributen und Label ergeben, und ob gleiche Label immer Knoten gleicher Sorte bezeichnen (siehe auch 7.4).

Aus Gründen der Homogenität der Regelbasis werden die Supersorten der Sorte mit einbezogen: kein Name eines Attributs einer Supersorte darf in der Sorte als Label verwendet werden, entsprechendes gilt für Label einer Supersorte und Attribute der Sorte sowie für Sortenkonflikte zwischen Label gleichen Namens.

Da im GraPaKL jeder Knoten einen Namen braucht, werden für GGD–Knoten ohne Label sog. *Surrogatlabel* generiert. Die Nummern der Knoten können nicht verwendet werden, da gleiche Nummern i.allg. in verschiedenen Regeln für Knoten unterschiedlicher Sorten stehen. Die Surrogatlabel werden so 'sparsam' wie möglich generiert, d.h. jedes Surrogatlabel wird, wenn möglich, in mehreren Regeln verwendet.

Die Reihenfolge, in der die Knoten in einer GraPaKL–Regel angegeben werden, ist dem GGD durch die Knotennummern vorgegeben. Zusätzlich müssen die Constraints sowie die Nachbarschaften und Überlappungen (die im GraPaKL als Constraints repräsentiert werden) angegeben werden.

Die einfachste Lösung wäre, alle Constraints nach der Liste der Knoten auszugeben. GraPaKL würde dann während des Parse-Vorgangs zunächst alle Knoten der Regel suchen, und dann die Constraints auswerten.
Es ist in GraPaKL aber möglich, ein Constraint sofort anzugeben, wenn alle in den Pfaden des Constraints referenzierten Knoten angegeben wurden. (**Beispiel:** Ein Constraint, in dem nur Knoten Nr. 1 referenziert wird, kann direkt nach Knoten 1 ausgegeben werden.) Aus Effizienzgründen ist dies oft sinnvoll.

In der aktuellen Implementierung des GGD wurde folgende Taktik gewählt: Ein Constraint, das sich aus einer Nachbarschaft oder Überlappung ergibt, wird sofort nach den beteiligten Knoten ausgegeben (**Beispiel:** Das Constraint für eine Kante von Knoten 1 nach Knoten 2 wird sofort nach den Daten für Knoten 2 ausgegeben). Nach den Daten eines Knotens werden ausserdem die vom Benutzer definierten Constraints ausgegeben, die das `%-role`–Konstrukt benutzen und deren `<role-label>` dem Label oder der Nummer dieses Knotens entspricht. Alle weiteren vom Benutzer eingeführten Constraints ohne `%-role` werden nach dem letzten Knoten ausgegeben.

B.5 Dateien

Das System GGD ist zweckmäßigerweise auf verschiedene Dateien aufgeteilt. Die folgenden Dateien gehören (in alphabetischer Reihenfolge) dazu, und haben folgende Aufgaben:

check.lisp Hat die Aufgabe, alle notwendigen Tests zur Konsistenz der Regelbasis durchzuführen.

dag.lisp DAGs sind die zentrale Datenstruktur im GGD. In dieser Datei werden sie definiert und verwaltet.

editor.lisp Ein einfacher Editor, der speziell für die Eingabe von Constraints programmiert wurde. Die Benutzung als normaler Texteditor ist möglich.

feat-rep.lisp Lesen und Generieren von FEAT-REP-Dateien. Die Einlese-Routinen sind darauf ausgelegt, Fehler in von Hand erstellten oder geänderten FEAT-REP-Dateien abfangen zu können.

grapakl-out.lisp Ausgabe der Wissensbasis in eine Datei im GraPaKL-Format (Suffix .gg). Vorraussetzung für den Aufruf der Ausgabefunktion ist der vorherige Konsistenztest.

init-ggd.lisp Einige kleinere Funktionen zur Initialisierung der DAGs und von globalen Variablen

interface-v-r.lisp Das Interface zwischen der Visualisierungskomponente und der Repräsentationskomponente.

laden.lisp Lädt alle Programm-Dateien in das jeweilige LISP-System.

macros.lisp Verschiedene Zugriffsmacros.

rule-editor-dags.lisp Die Verwaltung der Regeldaten, wenn die entsprechenden Regeln in der Visualisierungskomponente bearbeitet werden.

rule-editor.lisp Der wesentliche Teil der Visualisierungskomponente, die interaktive, mausgesteuerte Eingabe und Modifizierung von Regeln.

rules.lisp Verwaltung der Regeldaten in der Repräsentationskomponente. Außerdem Tests der einzelnen Regeln auf semantische Korrektheit (z.B. zusammenhängende rechte Regelseite).

sorts.lisp Verwaltung der Daten der Sorten, außerdem Test der Sorten auf Korrektheit (z.B. gültiger Name, erlaubte Feature-Application).

startup.lisp Startet und beendet die Benutzung der graphischen Obverfläche.

taxon.lisp Anbindung des Wissensrepräsentationssytems TAXON. Außerdem einige Funktionen zur (teilweisen) Überwindung der TAXON-Mängel.

terminals.lisp Ein-/Ausgabe und Verwaltung der Terminals (also der TEC-REP Primitive).

Weitere Dateien sind **graphtools.lisp**, **easy-load.lisp**, **lispm-init.lisp** und **patrrc.lisp**. Diese Dateien entstanden nicht im Rahmen der Implementierung des GGD, stellen aber einige Systemfunktionen zur Verfügung. **graphtools.lisp** ist eine für den GGD angepasste Version der gleichnamigen Datei von Harald Aust und ergänzt die graphische Oberfläche `X-Window-Toolkit` um zusätzliche Dialoge.

Literaturverzeichnis

[AlMe 91] H. Albas, B. Melichar (Hrgb.): *Attribute Grammars, Applications and Systems.* Proceedings of the International Summer School SAGA, Prague, 1991.

[ASU 86] A. Aho, R. Sehti, J. Ullmann: Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing, 1986.

[Bec 91] A. Becker: *Analyse der Planungsverfahren der KI im Hinblick auf ihre Eignung für die Arbeitsplanung.* Document D-91-17, DFKI 1991.

[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.* Research Report RR-91-20, DFKI 1991.

[BKL 91c] A. Bernardi, C. Klauck, R. Legleitner: *PIM: Planning in Manufacturing using Skeletal Plans and Features.* Research Report RR-92-19, DFKI 1992.

[BKLSS 92] A. Bernardi, C. Klauck, R. Legleitner, M. Schulte, R. Stark: *Feature based Integration of CAD and CAPP.* Research Report RR-92-05, DFKI 1992.

[ChHe 91] S. Chuang, M. Henderson: *Compound Feature Recognition by Web Grammar Parsing.* Research in Engineering Design 1991-2, pp. 147 - 158, Springer Verlag Berlin, Heidelberg, New York 1991.

[Dro 89] V. Drobot: *Formal languages and automata theory.* Computer Science Press, Freeman & Co., New York, 1989.

[Ehr 78] H. Ehrig: *Introduction to the algebraic theory of graph grammars.* in: [EKG 79-91], LNCS 73, pp 1 - 69.

[EKG 79-91] H. Ehrig et.al.: *Graph Grammars and their Application to Computer Science.* 1th - 4th International Workshop, LNCS 73,153,291,532. Springer Verlag Berlin, Heidelberg, New York 1979-90.

[EMR 84] A. Ehrenfeucht, M. Main, G. Rozenberg: *Restrictions on NLC Graph Grammars.* In: Theoretical Computer Science 31, pp. 211 - 223, North–Holland, 1984.

[EnRo 91] J. Engelfriet, G. Rozenberg: *Graph grammars based on node rewriting: an introduction to NLC graph grammars* in: [EKG 79-91], LNCS 532, pp 12 - 23, 1991.

[Eve 89] W. Eversheim: *Organisation in der Produktionstechnik.* VDI Verlag Düsseldorf 1989.

[FFPR 90] S. Finger, M. Fox, F. Prinz, J. Rinderle: *Concurrent Design.* Applied Artificial Intelligence Special Issue on AI in Manufacturing, 1990.

[Göt 88] H. Göttler: *Graphgrammatiken in der Softwaretechnik.* Informatik Fachberichte 178, Springer Verlag Berlin, Heidelberg, New York 1988.

[Him 89] M. Himsolt: *GraphEd: An Interactive Graph Editor*, in: *Stacs 89, Lecture Notes in Computer Science 349.* Springer Verlag Berlin, Heidelberg, New York 1989

[Him 91] M. Himsolt: *GraphEd User Manual.* Universität Passau 1991

[HaLä 90] D. Hahn, G. Laßmann: *Produktionswirtschaft – Controlling Industrieller Produktion*, Band 1. Physica-Verlag Heidelberg 1990.

[JaRo 80] D. Janssens, G. Rozenberg: *Restrictions, Extensions, and Variations on NLC Grammars.* In: Information Science 20, pp. 217 - 244, North-Holland 1980.

[JaRo 83] D. Janssens, G. Rozenberg: *A Survey of NLC Grammars* In: Trees in Algebra and Programming, 8th Colloquium, 1983.

[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.

[Karg 90] H. Kargl: *Industrielle Datenverarbeitung*, in: *Industriebetriebslehre*, hrsg. v. M. Schweitzer, Vahlen Verlag München 1990.

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

[Knu 68] D. E. Knuth: *Semantics of context-free languages* in: Mathematical Systems Theory, pp. 127 - 145, 1968.

[Mah 88] U. Mahn: *Attributierte Grammatiken und Attributierungsalgorithmen.* Informatik Fachberichte 157, Springer Verlag Berlin, Heidelberg, New York 1988.

[Mau 92] J. Mauss: *Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken.* Document D-92-10, DFKI 1992.

[Nag 79] M. Nagl: *Graph-Grammatiken (Theorie, Implementierung, Anwendungen).* Vieweg Verlag, Braunschweig 1979.

[New 88] F. Newbery: *EDGE: An Extendible Directed Graph Editor.* Technical Report 8/88, Department of Informatics, University of Karlsruhe 1988.

[UnRa 88] M. Unger, S. Ray: *Feature-Based Process Planning at the AMRF.* In: Computers in Engineering 1988, American Society of Mechanical Engineers, San Francisco 1988.

[Roz 86] G. Rozenberg: *An Introduction to the NLC Way of rewriting Graphs* in: [EKG 79-91], LNCS 291, pp 55 - 66, 1986.

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

[Ste 90] G. Steele: *Common Lisp - The Language.* Digital Press, Bedford 1990.

[Zäp 89] G. Zäpfel: *Strategisches Produktions-Management.* de Gruyter-Verlag, Berlin 1989.

[Zim 82] H. Zima: *Compilerbau, Band 1: Analyse.* Bibliographisches Institut, Zürich 1982.

[ZiNa 79] H. Zischler, M. Nagl: *A Dialog System for the Graphical Representation of Graphs.* In: *Graphs, Data Structures, Algorithms.* Hrsg. v. M. Nagl und H. J. Schneider, Applied Computer Science 13, Hanser Verlag München, 1979.

Index

RR-92-25
Franz Schmalhofer, Ralf Bergmann, Otto Kühn, Gabriele Schmidt: Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations
12 pages

RR-92-26
Franz Schmalhofer, Thomas Reinartz, Bidjan Tschaitschian: Intelligent documentation as a catalyst for developing cooperative knowledge-based systems
16 pages

RR-92-27
Franz Schmalhofer, Jörg Thoben: The model-based construction of a case-oriented expert system
18 pages

RR-92-29
Zhaohur Wu, Ansgar Bernardi, Christoph Klauck: Skeletel Plans Reuse: A Restricted Conceptual Graph Classification Approach
13 pages

RR-92-33
Franz Baader
Unification Theory
22 pages

RR-92-34
Philipp Hanschke
Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35
Manfred Meyer
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke
Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer
An Alternative to Θ-Subsumption Based on Terminological Reasoning
9 pages

DFKI Technical Memos

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

TM-92-03
Mona Singh
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

DFKI Documents

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-92-04
Judith Klein, Ludwig Dickmann: DiTo-Datenbank - Datendokumentation zu Verbrektion und Koordination
55 Seiten

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

D-92-07
Susanne Biundo, Franz Schmalhofer (Eds.): Proceedings of the DFKI Workshop on Planning
65 pages

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 Chart-Parser für attributierte Graph-Grammatiken
87 Seiten

D-92-11
Kerstin Becker: Möglichkeiten der Wissensmodellierung für technische Diagnose-Expertensysteme
92 Seiten

D-92-12
Otto Kühn, Franz Schmalhofer, Gabriele Schmidt: Integrated Knowledge Acquisition for Lathe Production Planning: a Picture Gallery (Integrierte Wissensakquisition zur Fertigungsplanung für Drehteile: eine Bildergalerie)
27 pages

D-92-13
Holger Peine: An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis
55 pages

D-92-14
Johannes Schwagereit: Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM
98 Seiten

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

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.): UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

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

D-92-26
Enno Tolzmann:
Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

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-91-24
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 of 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-92-10
M. Bauer: An Interval-based Temporal Logic in a Multivalued Setting
17 pages

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-92-16
Jochen Heinsohn, Daniel Kudenko, Berhard Nebel, Hans-Jürgen Profitlich: An Empirical Analysis of Terminological Representation Systems
38 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-92-21
Jörg-Peter Mohren, Jürgen Müller
Representing Spatial Relations (Part II) -The Geometrical Approach
25 pages

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

RR-92-23
Gert Smolka, Ralf Treinen:
Records for Logic Programming
38 pages

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

Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM

Johannes Schwagereit

D-92-14
Document