

Beschreibung  
und  
Integration  
von  
IC – Technologie –  
spezifikationen

Diplomarbeit von Olav Krämer  
Dortmund im Mai 1989

Betreuer: Herr Dr. rer.nat. Rainer Brück  
Lehrstuhl 1, Abteilung Informatik  
Universität Dortmund

## Gliederung der Arbeit

|  |     |
|--|-----|
| 1. Einleitung  | 1   |
| 1.1 Der IC-Entwurfsprozeß  | 1   |
| 1.2 Die Bestandteile eines CAD-Systems für den physikalischen Entwurf                | 3   |
| 1.2.1 Applikationen  | 4   |
| 1.2.2 Technologiespezifikation   | 4   |
| 1.2.3 Maskendaten  | 5   |
| 1.2.4 Datenhaltung   | 5   |
| 2. Die Technologiespezifikation in der Literatur                                     | 5   |
| 2.1 Bisherige Arbeiten   | 7   |
| 2.2 Bestandteile einer Technologiespezifikation                                      | 7   |
| 2.3 Sprachphilosophie  | 12  |
| 2.4 Integration in ein CAD-System  | 13  |
| 3. Die Sprache DINGO-II zur Spezifikation von Technologien                           | 14  |
| 3.1 Syntaktische Konventionen  | 14  |
| 3.2 Aufbau einer Technologiespezifikation  | 15  |
| 3.2.1 Deklarationsteil   | 16  |
| 3.2.2 Objektdeklarationsteil   | 17  |
| 4. Die Symboltabelle   | 18  |
| 5. Die DROOLY Zwischensprache  | 27  |
| 4.1 Grundlagen   | 28  |
| 4.2 Vererbungsmechanismen  | 29  |
| 4.3 Die Beispieltechnologie in DROOLY  | 30  |
| 4.4 Syntaktische Struktur  | 31  |
| 4.5 Layerdefinitionen  | 33  |
| 4.6 Objektklassendefinitionen  | 33  |
| 4.7 Deklaration von Unterstrukturen  | 34  |
| 4.8 Beschreibungsteil  | 34  |
| 4.9 Wertausdrücke  | 37  |
| 6. Übersetzung DINGO-II nach DROOLY  | 39  |
| 7. Die Tabellen  | 47  |
| 7.1 Objekttabelle  | 47  |
| 7.2 Tabelle der Wertausdrücke  | 58  |
| 8. Der DROOLY-Compiler   | 59  |
| 8.1 Präprozessor   | 59  |
| 8.2 Hilfsmodule für Fehlermeldungen, Dekodierung und Überprüfung der Codeabdeckung   | 61  |
| 8.3 Hauptmodul main.c  | 64  |
| 8.4 Scanner scanner.c  | 66  |
| 8.5 Parser parser.c  | 68  |
| 8.6 Objekte object.c   | 79  |
| 8.7 Wertausdrücke expr.c   | 97  |
| 8.8 Topologievergleich compare.c   | 108 |
| 8.9 Erweiterte semantische Überprüfungen validate.c                                  | 117 |
| 9. Das Technologieinterface für Applikationen  | 119 |
| 10. Beispiel für eine technologieinvariante Applikation:<br>Ein Bauelement-Extraktor | 119 |
| Anhang A: Liste der DROOLY-Token und Schlüsselworte                                  | 127 |
| Anhang B: Fehlermeldungen  | 129 |
| Anhang C: Die vollständigen Tabellen der Beispieltechnologie                         | 130 |
| Literatur  | 133 |

M. Ansblick  
R. Zusammenfassung

## 1. Einleitung

Mit dem Entstehen wirtschaftlich einsetzbarer Computer wurde zu Beginn der 60er Jahre eine rasende Entwicklung der Digitaltechnik eingeleitet. Anfang der 70er Jahre war man so weit, ganze Subsysteme von mehreren tausend Transistoren auf einem Chip unterbringen zu können (Abb. 1.1). Um dem Konflikt aus hohen Entwicklungskosten bei kleinen Produktionsstückzahlen zu entrinnen, wurden vielfach einsetzbare Standard-Bausteine entwickelt, deren bekanntester Vertreter schließlich der Mikroprozessor wurde.

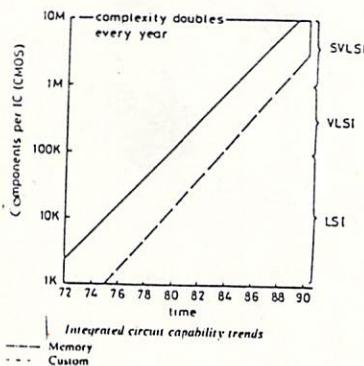


Abb. 1.1: Zahl der Komponenten pro IC gegen Jahr der Einführung [Bro 85]

Steigende Ansprüche an Kompaktheit, Zuverlässigkeit, Leistungsaufnahme und Kostenverringerung führten bei vielen Anwendern zu dem Wunsch, ihre Schaltungen auf einem eigenen Chip zu integrieren. Das ungünstige Verhältnis aus Entwicklungskosten zu Produktionsstückzahlen solcher kundenspezifischer Chips führte allerdings zu unvertretbar hohen Stückkosten.

Aus einer fortschreitenden Formalisierung und damit möglichen Automatisierung des Entwurfsprozesses integrierter Schaltungen entstanden zu Beginn der 80er Jahre IC-Designssysteme. Diese Systeme erlauben durch weitgehende Standardisierung des Entwurfsprozesses den kostengünstigen Entwurf kundenspezifischer Chips mittlerer Größe bei mäßigen Anforderungen an Kompaktheit und Schnelligkeit des Entwurfs [Hör 86].

### 1.1 Der IC-Entwurfsprozeß

Das Ziel des Entwurfs integrierter Schaltungen ist die schrittweise Erstellung eines vollständigen Satzes von Fertigungsunterlagen, die beim nachfolgenden Produktionsprozeß eine korrekte Implementierung des Entwurfs garantieren.

Der Entwurf kundenspezifischer Chips (ASICs) mit mehreren Millionen elektrischer Komponenten bei gleichzeitig hoher Komplexität der implementierten Schaltungen entzieht sich der Verwendung von Standardentwurfssystemen. Um den

Rechnereinsatz trotzdem zu ermöglichen, bedarf es einer geeigneten Modellierung des Entwurfsprozesses. Bewährt hat sich zu diesem Zweck das Y-Modell nach Walker und Thomas [Wal 85]:

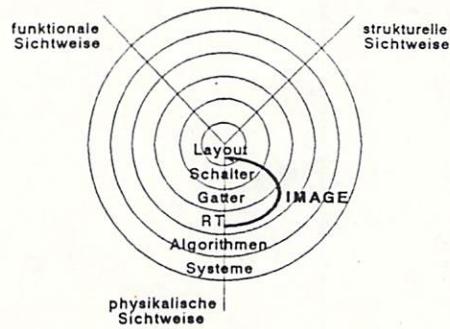


Abb. 1.2: Das Y-Modell

Dieses Modell gliedert den Entwurfsprozeß in eine Hierarchie von Abstraktionsebenen, dargestellt durch die konzentrischen Kreise im Modell:

1. Systemebene
2. Algorithmische Ebene
3. Registertransfer-Ebene
4. Gatterebene
5. Schaltkreisebene
6. Layout

Die auf jeder Ebene erstellten Dokumente sind eine Verfeinerung der Dokumente der darüber liegenden Ebene.

Neben den Abstraktionsebenen beschreibt das Modell drei Sichtweisen des IC-Entwurfs, die sich durch alle Abstraktionsebenen ziehen, auf den unterschiedlichen Ebenen aber verschieden stark ausgeprägt sind:

- Die **funktionale Sichtweise** beschreibt das Systemverhalten auf den verschiedenen Abstraktionsebenen. Auf einer hohen Ebene wird dabei eine funktionale Spezifikation erstellt. Die niedrigeren Abstraktionsebenen beschreiben das Verhalten unter Verwendung geeigneter Modelle, wie z.B. Differentialgleichungen für Transistoren.
- Die **strukturelle Sichtweise** bildet eine Art Brücke zwischen der funktionalen und der physischen Sichtweise. Sie beschreibt die Aufgliederung des Entwurfs in Untereinheiten und die Beziehungen zwischen diesen Untereinheiten. Als Dokumente werden auf den verschiedenen Abstraktionsebenen verschiedene Formen von Netzlisten erstellt. Diese Netzlisten enthalten jedoch keinerlei geometrische Angaben.

- Die physikalische Sichtweise beinhaltet diejenigen Aspekte des IC-Entwurfsprozesses, die speziell auf die Erstellung von Fertigungsvorlagen abzielen. Auf den hohen Abstraktionsebenen wird die Anordnung der funktionalen Unterstrukturen auf der Chipoberfläche festgelegt. Auf den niedrigen Ebenen wird die exakte Auslegung einzelner Schaltungselemente in ihrer geometrischen Struktur, ihren Abmessungen und Positionierung beschrieben.

Das Y-Modell kann dazu benutzt werden, eine Entwurfsmethodik in ihren einzelnen Schritten darzustellen. Eine andere Anwendung ist die Einordnung von Teilaufgaben, die ein CAD-System ausführt.

## 1.2 Die Bestandteile eines CAD-Systems für den physikalischen Entwurf

Allgemein versteht man unter einem physikalischen Entwurf alle Dokumente, die unter der physikalischen Sichtweise erstellt werden. In der vorliegenden Arbeit soll der Begriff des physikalischen Entwurfs enger gefaßt werden. Er umfaßt im wesentlichen die geometrischen Aspekte des Entwurfs, d.h. die erstellten Dokumente beschreiben den Entwurf mithilfe exakter geometrischer Strukturen, die auf verschiedenen Ebenen ("Layern") nach absoluten Koordinaten angeordnet sind. Ziel des physikalischen Entwurfs ist die Erstellung eines geometrischen Layouts. Dieses umfaßt alle Vorlagen für die anschließende Maskenfertigung.

Bei dem im folgenden vorgestellten System für den physikalischen Entwurf handelt es sich entsprechend nur um eine Komponente des CAD-Systems ALICE-1, welches den gesamten IC-Entwurfsprozeß unterstützt und am Lehrstuhl Informatik 1 (Uni Dortmund) entwickelt wird. Seine Aufgaben werden durch den mit IMAGE bezeichneten Pfeil in Abb. 1.2 beschrieben. IMAGE ("Integrated Module Adaption and GEneration system") bildet den technologischen Sockel des ALICE-1-Systems [Brü 88]. Mithilfe der drei Mechanismen Bibliothekszugriff, Modifikation und Neugenerierung werden in IMAGE Realisierungen (d.h. geometrische Layouts) der Module erzeugt, die mithilfe der anderen Komponenten des Systems geplant und entworfen wurden.

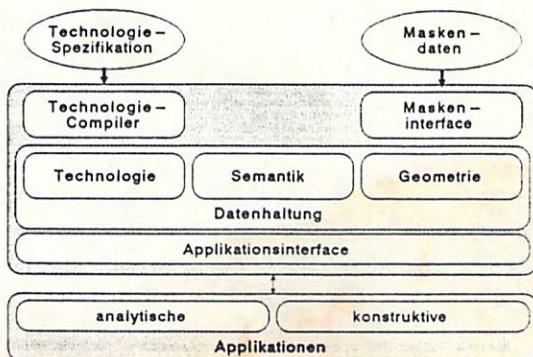


Abb. 1.3: Image

### 1.2.1 Applikationen

Die Applikationen sind die eigentlichen Benutzer des IMAGE-Systems. Entsprechend muß die Datenhaltung auf die Unterstützung der einzelnen Applikationen abgestimmt sein. Man unterscheidet zwei Klassen von Applikationen:

- **analytische Applikationen** bearbeiten vorhandene geometrische Layouts. Ein typisches Beispiel stellen Bauteilextraktoren dar. Mit ihrer Hilfe gewinnt man aus einem geometrischen Layout einen Lageplan der abgefragten semantischen Strukturen. Speichert die Datenhaltung neben den reinen Geometriedaten auch bereits semantische Informationen, so beschränkt sich die Arbeit eines Bauteilextraktors auf den Zugriff auf diese Daten. Weitere Beispiele für analytische Applikationen sind
  - Design Rule Checker (DRC)
  - Electrical Rule Checker (ERC)
  - Parameterextraktoren
- **konstruktive Applikationen** dienen zur gezielten Manipulation von geometrischen Layouts durch Hinzufügen, Löschen oder Verändern geometrischer Strukturen. Die Spannweite solcher Applikationen reicht vom einfachen Maskeneditor, mit dessen Hilfe sich geometrische Strukturen direkt am Bildschirm eingeben lassen, bis hin zu vollständigen Layoutsynthesysystemen. Weitere Beispiele für konstruktive Applikationen sind
  - Kompaktoren
  - Design-Rule-Anpasser
  - Plazierungs- und Verdrahtungssysteme

### 1.2.2 Technologiespezifikation

Ein wesentlicher Bestandteil des IMAGE-Systems und der Angelpunkt für die vorliegende Arbeit ist die explizite Repräsentation von Technologiedaten. Viele heute existierende CAD-Systeme sind für eine bestimmte Technologie entwickelt worden und sind so implementiert, daß die Verwendung von anders strukturierten Technologien nicht ohne große Änderungen am Code des Systems möglich ist. Aus wirtschaftlichen Gründen operieren IC-Hersteller heute jedoch mit verschiedenen Technologien, in zunehmendem Maße sogar gemeinsam in einem Entwurf, so daß die Anwendbarkeit eines Satzes von CAD-Werkzeugen für unterschiedliche Technologien zu einer zentralen Forderung wird [Ehr 83].

Geometrische Layouts werden nach den Regeln einer spezifischen Technologie aufgebaut und manipuliert. Eine Applikation, die auf einem Layout arbeitet, muß die relevanten Aspekte der zugrunde liegende Technologie kennen. Zu Beginn der Automatisierung des Chip-Entwurfs war es üblich, Software-Werkzeuge das benötigte Technologiewissen direkt über die Steuerung ihres Kontrollflusses im Code einzuprogrammieren.

Es könnte beispielsweise ein Switch-Level-Simulator mit der Annahme programmiert worden sein, das eine Schaltung aus einer Hierarchie von Zellen aufgebaut ist, von denen wiederum jede ein Netzwerk von Transistoren mit drei Anschlüssen enthält. In diesem Fall würde der Wechsel auf eine Technologie, die einen zusätzlichen Substratanschluß pro Transistor erfordert, eine umfangreiche und fehleranfällige Änderung am Code des Simulators erforderlich machen (nach [Chu 85]).

Software-Werkzeuge mit solcheart eingebauten Annahmen sind nicht technologieinvariant. Technologieinvarianz bedeutet, daß bei einem Wechsel der Technologie keine softwaretechnologischen Änderungen an der Applikation nötig sind.

Der Begriff der Technologieinvarianz ist nicht synonym zum Begriff der Technologieunabhängigkeit. Physikalische Applikationen arbeiten direkt auf den geometrischen Strukturen eines Layouts und sind damit inhärent technologieabhängig. Der Begriff der Technologieunabhängigkeit ist eher für höhere Abstraktionsstufen adäquat. Für die Arbeit eines Architekturplaners z.B. ist die verwandte Technologie irrelevant.

Im IMAGE-System wird die Technologieinvarianz durch einen datengetriebenen Ansatz realisiert. Technologiedaten werden nicht implizit in den Code der beteiligten Applikationen hineinprogrammiert, sondern werden in einer formalen Sprache beschrieben und von außen dem System zugänglich gemacht.

#### **1.2.3 Maskendaten**

Um Layouts mit der Umwelt austauschen zu können, gibt es im IMAGE-System eine Schnittstelle für Maskendaten. Geometrische Layouts von anderen CAD-Systemen können über diese Schnittstelle eingelesen und in die interne Darstellung umgewandelt werden. Ebenso werden erstellte Layouts über diese Schnittstelle in standardisierten Formaten ausgelesen.

#### **1.2.4 Datenhaltung**

Heute existieren vorwiegend CAD-Werkzeuge, die für eine spezielle Teilaufgabe des Layout-Entwurfsprozesses ausgelegt sind. Diese Systeme sind aufgrund ihrer Datenstrukturen nicht oder nur schwer zu vollständigen Entwurfssystemen integrierbar. Eine abgestimmte, integrierte Implementation von Entwurfswerkzeugen für unterschiedliche Teilaufgaben verspricht jedoch Synergieeffekte, die eine erhöhte Leistungsfähigkeit des Gesamtsystems erwarten lassen. Kern des IMAGE-Systems ist daher die zentrale Datenhaltung mit ihren drei Teilbereichen.

### Geometriedaten

In diesem Teilbereich der zentralen Datenhaltung werden Strukturen zur internen Repräsentation konkreter Layouts während ihrer Bearbeitung durch IMAGE zur Verfügung gestellt. Das Datenmodell fußt auf einer fünfstufigen Hierarchie (Abb. 1.4):

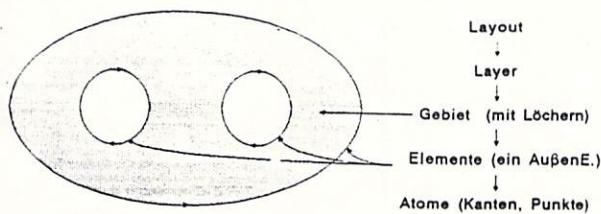


Abb. 1.4: Darstellung geometrischer Strukturen in IMAGE

Die unterste Hierarchiestufe enthält Atome zur Beschreibung von Anfangs- und Endkoordinaten, Funktionsvorschriften und Durchlaufrichtungen.

Die nächste Stufe beschreibt elementare Objekte, jeweils gebildet aus einer sequentiellen Liste atomarer Objekte. Elementare Objekte können äußere oder innere Gebietsabgrenzungen darstellen.

Die dritte Stufe enthält die Gebietsobjekte. Ein Gebietsobjekt besteht aus einem Elemtarobjekt zur Beschreibung seiner Außenabgrenzung und einer (mglw. leeren) Menge von Elementarobjekten zur Beschreibung von inneren Gebietsabgrenzungen.

Die vierte Hierarchiestufe ordnet Gebietsobjekte Layern zu. Ein Layerobjekt beschreibt jeweils eine Maskenebene der zugrunde liegenden Technologie bzw. eine durch Verknüpfung solcher Ebenen entstandene Pseudoebene.

Die oberste Hierarchiestufe faßt die Layerobjekte eines Entwurfs mit allen geometrischen Strukturen der darunterliegenden Hierarchiestufen zu einem Layout zusammen [Wro 87], [Pre 88].

### Semantikdaten

Dieser Teilbereich der IMAGE-Datenhaltung geht über die gewöhnlich in CAD-Systemen für den IC-Entwurf gespeicherten Daten hinaus. Gerade die Speicherung zusätzlicher semantischer Informationen ermöglicht es den Applikationen, die Geometriedaten in "intelligenter" Weise – d.h. in diesem Zusammenhang: abhängig von ihrem sematischen Kontext – zu überprüfen oder zu manipulieren. Semantische Daten können einmal direkt bei der Erstellung eines Layouts (z.B. durch einen technologiegesteuerten Layoutheditor) in IMAGE eingebracht werden. Eine zweite Möglichkeit ist die nachträgliche Extraktion der semantischen Informationen aus einem extern erstellten Layout mithilfe eines Bauteilextraktors. Dem Datenmodell liegen folgende Objekttypen zugrunde [Mör 88]:

- **Relationship Objects** definieren gerichtete Relationen zwischen zwei oder mehreren Objekten. Gegenüber Datenmodellen für konventionelle Datenbanken [Che 76] dürfen Relationship Objects Relationen auf anderen Relationship Objects definieren. Zur Beschreibung von Relationen zwischen (Teil-) Mengen von Objekten stehen verschiedene Stelligkeiten von Relationship Objects zur Verfügung:
  - 1:1 Relationship Objects definieren eine Relation zwischen genau zwei Objekten.
  - 1:n Relationship Objects setzen ein Objekt in Relation zu n anderen Objekten.
  - n:m Relationship Objects setzen n Objekte zu m anderen Objekten in Relation.
- **Composition Objects** definieren neue Objekte, die aus verschiedenen Komponenten bestehen und ermöglichen damit eine Hierarchiebildung innerhalb der Objektstruktur. Ein Composition Object definiert somit eine Komponentenbeziehung zwischen sich und den anhängenden Unterstrukturen.
- **Set Objects** fassen Objekte oder Teilstrukturen zusammen und strukturieren damit die Objektstruktur. Ein Set Object definiert eine Elementbeziehung zwischen sich und den anhängenden Unterstrukturen.
- **Quantity Objects** parametrisieren die Anzahl von Objekten oder Teilstrukturen bei Set Objects.
- **Property Objects** binden Eigenschaften an Objekte.

## Technologiedaten

Wie oben bereits erwähnt, liegen Technologiedaten im IMAGE-System nicht implizit in den Kontrollstrukturen von Applikationen vor, sondern werden explizit in einer formalen Sprache beschrieben. Ein Compiler übersetzt diese Technologiespezifikation in eine interne Repräsentation, welche den Applikationen das benötigte Technologiewissen über einen Satz von Operationen zur Verfügung stellt. Der Compiler und die interne Repräsentation der Technologiedaten wird an späterer Stelle in dieser Arbeit noch detailliert beschrieben.

## 2. Die Technologiespezifikation in der Literatur

### 2.1 Bisherige Arbeiten

Die einfachsten datengetriebenen Ansätze zur Speicherung von Technologiedaten für Layout-Applikationen finden sich in der Literatur in Form von sog. Runsets. Benötigte Steuerinformationen werden in einer externen Datei in einem Format abgelegt, welches direkt vom jeweiligen Programm interpretiert werden kann. Ein typischer Vertreter solcher Runsets liegt z.B. mit der Design-Rule-Tabelle im Design Rule Checker (DRC) MAGIC [Ous 84] vor. In dieser Tabelle werden Design-Rules kantenorientiert aus folgenden Feldern aufgebaut:

|    |                  |   |
|----|------------------|---|
| 1. | Typ1             | Material auf der 1. Seite der Kante                             |
| 2. | Typ2             | Material auf der 2. Seite der Kante                             |
| 3. | d                | Distanz, innerhalb derer die zweite Seite zu überprüfen ist     |
| 4. | erlaubte Layer   | Liste aller Materialien, die im überprüften Gebiet erlaubt sind |
| 5. | "Corner"- Typen  | Liste der Materialien, die eine "Corner Extension" nötig machen |
| 6. | Corner Extension | Größe der Corner Extension                                      |

MAGIC stellt spezielle Makros zur Verfügung (SPACING, WIDTH), die das Erstellen der Design-Rule-Tabelle erleichtern.

Durch Zeile 1 bis 4 werden die Regeln der einfachsten Form beschrieben:

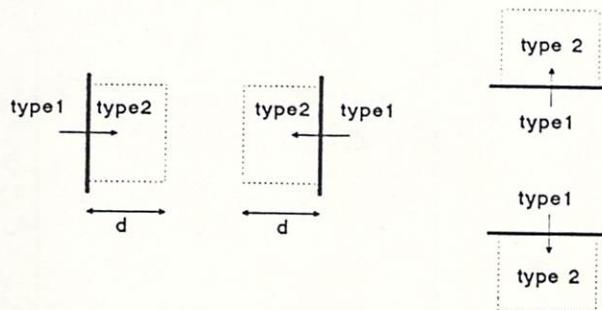


Abb. 2.1: Einfache kantenorientierte Design Rules in MAGIC

Der zu überprüfende Bereich wird evtl. an den Ecken um die Corner Extension erweitert:

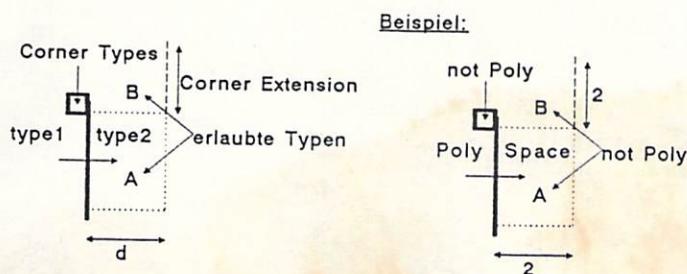


Abb. 2.2: Corner Design Rules

Ein weiterer Vertreter einfacher Runsets wird von W. Schiele in [Schi 85] beschrieben. Schiele benutzt eine Instruktionsliste für die Steuerung eines Programms zur Adaption von Design Rules. Dieses Programm kann ebenso wie MAGIC

nur Manhattan- (d.h. Iso-orientierte) Geometrien verarbeiten. Schiele beschreibt Design Rules durch Ungleichungen vom Typ

$$Z_i - Z_j > c,$$

worin  $Z_i$  und  $Z_j$  x- oder y-Koordinaten der Polygonkanten i und j sind. Im Gegensatz zum Ansatz in MAGIC liegt die Instruktionsliste nicht als Tabelle vor, sondern als Programm in einer formalen Sprache, die durch Syntax-Regeln spezifiziert ist. Die Syntax der Sprache ist einfach gehalten und verwendet ein etwas kryptisches Format. Die einzelnen Instruktionen werden eine nach der anderen eingelesen und direkt interpretiert:

| <u>Instruktionen</u>  |  |
|-----------------------|--|
| M: Melt               | V: Oversize  |
| G: Graph generation   | U: Undersize   |
| A: And                | I: Internal  |
| N: And not            | E: External  |
| O: Or                 | S: Scale   |
|                       | W: Width   |
| M 1                   | I melt Diffusion                                       |
| M 2                   | I melt Buried  |
| M 3                   | I melt Poly  |
| M 5                   | I melt Contact   |
| M 6                   | I melt Metal   |
| M 9                   | I melt Implantation                                    |
| G                     | I graph generation                                     |
| I Minimum layer width | I Transistor and wire Scaling                          |
| I 1; 2                | A 13, 21 I Diff and Poly                               |
| I 2; 4                | N 21 2,22 I And not Buried $\rightarrow$ Transistors   |
| ...                   | S 22; 0.4 I Transistor scaling (5*5 $\rightarrow$ 2*2) |
| I 9; 4                | W 6; 0.4 I Wire width scaling                          |

Abb. 2.3: Beispiel für eine Instruktionsliste zur Design Rule Anpassung

Zusammenfassend kann man festhalten, daß die einfachen Runsets zur Steuerung einer speziellen Applikation entworfen sind und nur die Informationen zur Verfügung stellen, die zur Bearbeitung der jeweiligen Aufgabe erforderlich sind. Beim Wechsel der zugrundeliegenden Technologie muß jedes einzelne Runset gesondert angepaßt werden. Dieser Vorgang ist natürlich zeitintensiv und fehleranfällig.

In integrierten CAD-Systemen wird daher vielfach versucht, eine gemeinsame Basis für technologieabhängige Daten anzulegen. Alle beteiligten Applikationen greifen über ein geeignetes Interface auf die Technologiedaten zu.

Ein CAD-System für den physikalischen IC-Entwurf enthält Applikationen und Tools mit unterschiedlichen Aufgaben, die sie auf unterschiedlichen Abstraktionsebenen erledigen. Demzufolge gibt es verschiedene Arten von Technologieabhängigkeit. Kung-Chao Chu nennt in seinem vielbeachteten Artikel [Chu 85] vier Kategorien von Technologieabhängigkeit:

**Abhängigkeit von Werten** ist die offensichtlichste Art von Technologieabhängigkeit.

In diese Kategorie fallen z.B. eingebaute Annahmen über die maximal erlaubte Anzahl von Maskenlayern, minimale Leitungsweiten und Abstände, Annahmen über die verwendeten Maßeinheiten ebenso wie nicht direkt von einer bestimmten Technologie abhängige feste Voreinstellungen wie maximale Matrixgröße oder die Auflösung numerischer Werte.

**Abhängigkeit von Schlüsselworten** macht bei Technologieänderungen ebenfalls eine Neu-Compilation des betroffenen Programmes erforderlich. Ein Beispiel wäre eine vorgegebene Menge von Layerbezeichnern (z.B. "n-diff", "p-diff", "poly", "alu"), die in einem Vektor im Programmcode definiert sind. Soll bei einem Technologiewechsel ein neuer Layer hinzugenommen werden (z.B. statt "alu" jetzt "met1" und "met1"), muß nicht nur der definierende Vektor neu übersetzt werden, sondern alle Stellen im Programm, die sich auf die alten Schlüsselworte bezogen.

**Abhängigkeit vom Format** liegt z.B. beim oben beschriebenen MAGIC-Extraktor vor. Jedes Feld der Design-Rule-Tabelle hat eine vordefinierte Bedeutung. MAGIC liest die einzelnen Zeilen der Tabelle, zerlegt sie in die einzelnen Felder und speichert sie in einer internen Datenstruktur. Wird die Anordnung der Felder verändert, so ist der Code zum Einlesen der Tabelle unbrauchbar.

**Abhängigkeit von der Modellierung.** Ein physikalisches IC-Entwurfssystem benötigt eine abstraktes Modell der zu bearbeitenden Layoutobjekte. So könnte ein Layouteditor z.B. Kreise und beliebige Winkel im Layout durch Treppenfunktionen modellieren. Die geeignete Wahl eines Modelles ermöglicht eine effiziente Erledigung der gestellten Aufgabe, indem es nur die relevanten Aspekte der Realität beschreibt.

Chu weist in seinem Artikel auf eine weitere Klassifizierungsmöglichkeit von Technologiespezifikationen hin. Wie schon an den oben vorgestellten Runsets des Extraktors und des Design-Rule-Adaptors erkennbar, gibt es die Möglichkeiten, Technologien in einem tabellarischen Format oder in einer syntaktischen Form zu beschreiben. Die tabellarische Form ist besonders leicht direkt von einem Programm zu interpretieren. Es besteht der genannte Nachteil einer Formatabhängigkeit. Dieser kann jedoch durch eine definerte Schnittstelle zwischen Daten und Anwender der Daten gemildert werden. Die Daten bilden zusammen mit den in der Schnittstelle definierten Zugriffsoperationen einen abstrakten Datentyp mit den bekannten Vorteilen.

Die syntaktische Form erlaubt die übersichtliche Formulierung komplexerer Zusammenhänge. Sie muß jedoch allgemein zunächst durch einen Übersetzungsvorgang für die Benutzung durch ein Programm aufbereitet werden. Ein wichtiger Vorteil dieser Form ist ihre Eignung zum Austausch von Technologiespezifikationen z.B. zwischen Prozeßentwickler und Designer. Speziell zu diesem Zweck wurde das EDIF-Format [EDIF 85] entwickelt. In einer LISP-artigen Syntax lassen sich hierin sehr viele Aspekte des IC-Designprozesses beschreiben.

Chu beschreibt eine syntaktische Erweiterung von EDIF. Seine Erweiterung konzentriert sich auf die Unterstützung von Layout-Werkzeugen (Layout-Editoren, Kompaktoren, Router, DRC, Extraktoren, Plotter, Maskengeneratoren). Er erlaubt nur die Beschreibung von Manhattan-Geometrien und gibt drei Objektarten fest vor: (zusammengesetzte) LAYER, VIA und TRANSISTOR. Die Topologie dieser Objektarten ist festgelegt und wird mithilfe der Operatoren minWIDTH, minAREA, minSEPARATION und minOVERHANG plus Maßangabe quantifiziert.

Ausnahmeregelungen von den global geltenden Mindestmaßen innerhalb von VIAs oder TRANSISTOREn sind eingeschränkt möglich. Zusätzlich zu geometrischen Maßangaben können jedoch auch elektrische Größen verwandt werden.

Ph. Smith und S. Daniel verwenden im VIVID-System [Smi 85] zur Technologiebindung eine Reihe von Dateien. Dieses Master-Technologiefile-System (MTF-System) unterstützt Tools wie einen symbolischen Schaltkreiseditor, einen Schaltkreis-Extraktor, einen Simulator, einen Maskenlevel-Übersetzer und einen Kompaktor. Das gesamte System ist auf die Verarbeitung von unterschiedlichen MOS-Technologien (MOSIS3-CMOS, MOSIS4-NMOS) ausgelegt. Die einzelnen Dateien enthalten technologieabhängige Informationen aus verschiedenen Bereichen: Layer, Transistoren und Kontakte, Graphische Attribute und Shapes, Virtuelle Shapes, Menüpunkte, Symbolisch-nach-Maskenlevel-Übersetzung, elektrische und physikalische Konstanten.

Formuliert werden diese Informationen teilweise in einer syntaktisch an die Programmiersprache C angelehnten Daten-Beschreibungssprache. Kompliziertere Sachverhalte werden direkt in C unter Zuhilfenahme von speziellen Prozeduraufufen beschrieben. Beide Formen werden in normale Objektdateien übersetzt. Die Applikationen laden sie allerdings erst zur Laufzeit (sog. dynamisches Binden) und müssen damit bei Technologieänderungen nicht neu compiliert werden.

Einen sehr umfassenden Ansatz zur Speicherung von technologieabhängigen Daten benutzen Aude und Kahn im SIDESMAN System [Aud 86]. Zur Beschreibung von Design Rules dient hier die Sprache DRDL (Design Rule Description Language), die im Gegensatz zu EDIF sehr an das natürliche Englisch angelehnt ist. Design-Rule-Beschreibungen werden mit einem Compiler in eine interne Form übersetzt. Eine Prozedurbibliothek ermöglicht den Applikationen einen kontrollierten Zugriff auf die gespeicherten Informationen. Diese erhalten verschiedene Attribute, um die Zugriffsmöglichkeit von Applikationen zu steuern. Es wird dabei unterschieden zwischen toolbezogenen Attributen, designbereichbezogenen Attributen und abstraktionsebenen-bezogenen Attributen.

Zur weiteren Strukturierung der gespeicherten Informationen sind einzelne Technologiespezifikationen baumartig angeordnet.

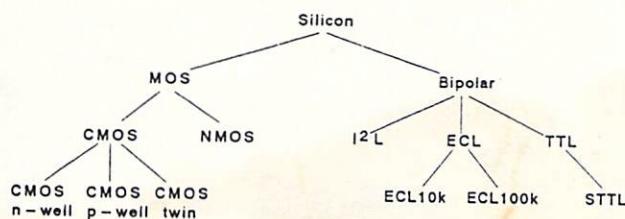


Abb. 2.4: Technologiebaum nach [Aud 86]

Ahnlich der Objekthierarchie in objektorientierten Programmiersprachen wie Smalltalk nimmt die Spezialisierung der Spezifikationen und damit der Grad von Technologieabhängigkeit von der Wurzel zu den Blättern zu. Design Rules werden

von einer Technologie an ihre Söhne vererbt. Es ist jedoch auch möglich, ererbte Design Rules mit eigenen Regeln zu überschreiben.

## 2.2 Bestandteile einer Technologiespezifikation

Im vorigen Abschnitt wurden verschiedene Ansätze für Technologiespezifikationen vorgestellt. Aude und Kahn identifizieren zur Einteilung der verschiedenen Aspekte einer Technologiespezifikation acht Teilbereiche:

1. Zeitverhalten
2. Belastbarkeit
3. Layout
4. Verlustwärme
5. Anschließbarkeit
6. Testbarkeit
7. Interface-Kompatibilität
8. Abschlußstrukturen

Für ein physikalisches Entwurfssystem, wie es in dieser Arbeit betrachtet werden soll, ist besonders der Punkt "3. Layout" von Bedeutung. In einer vollständigen Technologiespezifikation für ein physikalisches Entwurfssystem sollen alle technologieabhängigen Informationen enthalten sein, die die einzelnen Applikationen zur Erledigung ihrer Aufgaben benötigen, bei gleichzeitiger Wahrung ihrer Technologieinvarianz. Die einzelnen Bestandteile einer solchen Spezifikation sind:

**Prozeßbezeichner.** Jede Technologiespezifikation muß sich eindeutig einer bestimmten Technologie zuordnen lassen, um auch die gleichzeitige Verwendung mehrerer Technologien verwalten zu können.

**Maskenbezeichner.** Zur Vermeidung von Schlüsselwortabhängigkeit dürfen die Applikationen keine Annahmen über die Bezeichner erlaubter Maskenlayer machen. Zu jedem Layer werden weiter Informationen über seinen Bezeichner in der externen Maskendatei, über seine Darstellung (Farbe, Schraffur) in einem interaktiven Layoutheditor sowie evtl. sein Ursprung aus einer booleschen Verknüpfung von primitiven Layern benötigt. Der genaue Umfang der Layerattribute ist dabei stark applikationsabhängig.

**Elektrische Konstanten.** Komplexe Design Rules lassen sich manchmal nur in Abhängigkeit von elektrischen Konstanten ausreichend spezifizieren. Zu diesem Zweck müssen die möglichen Werte elektrischer Konstanten mit ihren Einheiten definiert werden.

**Erlaubte Geometrieklassen.** Häufig schränken Layout-Werkzeuge die möglichen Geometrieklassen (z.B. Manhattan, Polygone mit iso-orientierten und 45°-Kanten usw.) in einem Layout ein, um besonders effizient arbeiten zu können. Zur Überprüfung eines Layouts auf die Einhaltung solcher Einschränkungen, u.U. unter Berücksichtigung eines besonderen Bauteil-Kontextes, muß die explizite Formulierung von Geometrieklassen möglich sein.

**Topologie von erlaubten Bauteilen.** Um die Lage eines Bauteils in einem Layout bestimmen zu können, benötigt ein Layout-Extraktor eine Beschreibung der möglichen Ausprägungen dieses Bauteils. Dabei kommt es für die Funktionalität des Bauteils nicht auf die exakt quantifizierten Abmessungen des Bauteils an, sondern nur auf die relative Lage seiner Komponenten zueinander, auf seine Topologie also. Es ist daher sinnvoll, die Topologiebeschreibung eines Bauteils von quantitativen geometrischen Restriktionen zu trennen.

**Geometrische Restriktionen.** Selbst in einfachen Runsets bilden die geometrischen Restriktionen den Schwerpunkt der externen Technologieinformation. Zur Erzielung einer vollständigen Technologiespezifikation ist es allerdings erforderlich, auch kontextabhängige geometrische Restriktionen beschreiben zu können. Der Kontext kann dabei die Topologie eines Bauteils sein oder auch bestimmte elektrische Gegebenheiten auf den betroffenen Objekten.

### 2.3 Sprachphilosophie

Im Abschnitt 2.1 wurde deutlich gemacht, daß es sehr unterschiedliche Ansätze zur Formulierung von Technologiespezifikationen gibt. Besonders geeignet im Sinne von Handhabbarkeit ist eine textuelle Beschreibung, deren Syntax durch eine formale Grammatik definiert wird. Zur einfachen Konstruktion eines Übersetzers sollte dabei beachtet werden, daß die Grammatik der Technologie-Spezifikationssprache vom LALR(1)-Typ ist, da für diesen Typ leistungsfähige Parser-Generatoren existieren [NEMO].

Der Sprachumfang muß groß genug sein, um eine Technologie in allen ihren für den physikalischen Entwurf relevanten Aspekten beschreiben zu können, ohne irgend welche impliziten Annahmen über erlaubte Geometrien zu machen, d.h. die Technologieinvarianz der Sprache muß gewährleistet sein.

Die Syntax soll gut lesbar sein und nur wenige, aber durchgängige Konzepte zur Beschreibung enthalten. Eine deskriptive, an natürlichsprachliche Konstruktionen angelehnte Form bietet sich an. Für die Beschreibung quantitativer Zusammenhänge kann eine an konventionelle Programmiersprachen wie PASCAL oder C angelehnte Syntax verwandt werden. Heute kann bei Entwicklern von Technologien und Layouts die Kenntnis solcher Sprachen vorausgesetzt werden.

Die Beschreibungsmittel der Sprache müssen sich an die Gegebenheiten des zu unterstützenden CAD-Systems anpassen. Da im folgenden das IMAGE-System zugrunde gelegt wird, muß die verwandte Technologie-Spezifikationssprache das semantische Datenmodell in der IMAGE-Datenhaltung unterstützen. Darin wird von einer objektbezogenen Layoutbearbeitung ausgegangen. Das Layout enthält nur erlaubte Instanzen von Bauteil-Prototypen. Die Beschreibung der Topologie solcher Prototypen zusammen mit den auf sie bezogenen Design Rules bilden den Schwerpunkt der Beschreibung einer Technologie für das IMAGE-System.

## 2.4 Integration in ein CAD-System

Technologien werden im IMAGE-System in der speziell dafür entwickelten Beschreibungssprache DINGO-II spezifiziert. DINGO-II erfüllt die genannten Anforderungen an eine solche Sprache. In der folgenden Abbildung wird die Technologieverarbeitung im IMAGE-System noch einmal detaillierter dargestellt:

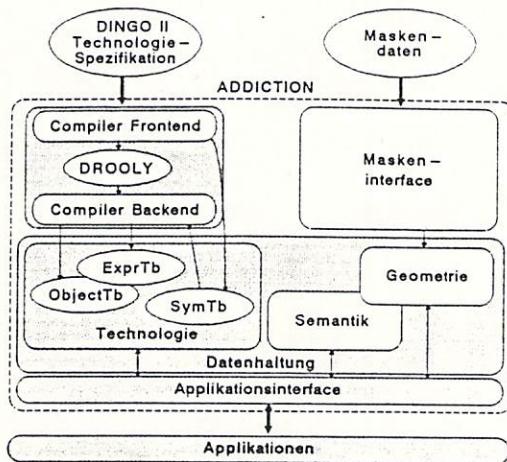


Abb. 2.5: Die Technologieverarbeitung in IMAGE

Die Syntaxdefinition von DINGO-II liegt in Form einer LALR(1)-Grammatik vor, die als Eingabe für den Parser-Generator NEMO dient. Der erzeugte Parser bildet das Frontend eines Technologie-Compilers, der die Technologiespezifikation in zwei Phasen in eine interne Darstellung übersetzt. Das Frontend erzeugt aus dem DINGO-II-Programm eine Symboltabelle und ein textuelles Zwischenformat in der Sprache DROOLY ("Devices Represented in Object Oriented Lyrics"). Die Syntax dieser Zwischensprache ist ebenfalls durch eine LALR(1)-Grammatik definiert. Das Compiler-Backend erzeugt aus dem DROOLY-Programm das gewünschte Zielformat, die interne Form der Technologiespezifikation. Die interne tabellarische Form besteht aus drei Tabellen (Objekttabelle, Ausdruckstabelle und Symboltabelle) und einer Bibliothek von Zugriffsoperationen. Über die Zugriffsoperationen können Applikationen unabhängig von der internen Repräsentation der Daten alle benötigten Technologieinformationen zu der beschriebenen Technologie erhalten.

## 3. Die Sprache DINGO-II zur Spezifikation von Technologien

In den folgenden Abschnitten wird die Sprache DINGO-II anhand der Konstrukte vorgestellt, die sie für eine Technologiespezifikation zur Verfügung stellt. Für eine detailliertere Beschreibung der Syntax und Semantik der Sprache siehe [DINGO].

DINGO-II ist eine descriptive Sprache. Im Zentrum einer Technologiespezifikation steht die Beschreibung erlaubter Objektstrukturen (device-Prototypen) in einem Layout. Eine vollständige Technologiespezifikation besteht aus der Beschreibung aller Objekte, die bei der Erstellung eines Layouts verwendet werden können. Dies beinhaltet sowohl einfache Strukturen wie Leiterbahnen auf

einzelnen Layern als auch komplexe Strukturen, wie Transistoren oder Kontakte. Darüber hinaus können elektrische und geometrische Randbedingungen formuliert werden, die direkt den einzelnen Technologieobjekten zugeordnet sind. Der Abstraktionsgrad der Objektbeschreibungen liegt auf der Ebene der Gebiete im geometrischen Datenmodell von IMAGE (vgl. Abb. 1.4). Zur Einstimmung sei hier eine vereinfachte Technologiebeschreibung in DINGO-II wiedergegeben:

```

declaration
  process CMOS
    layer Metal 1, Poly 2, Contact_Cut 3,
              n_Diff 4, p_Diff 5, n_Well 6
    ...
declaration end

module
  Gate (diffusion)
  using
    p_wire of layer Poly
    diffusion one of (layer n_Diff, layer p_Diff)

  description
    shape of Gate is iso_Polygon
    p_wire cross diffusion

  design rules
    internal
      width of diffusion:
        if diffusion is layer n_Diff then >= 15 else >= 10
module end

device
  n_Channel_Enhancement_Transistor alias n_CE_Trans
  using
    well of layer n_Well
    gt set of module Gate (p_Diff) cardinality range [1,5]

  description
    all gt: gt is gt
    all gt: gt in well

  design rules
    internal
      all gt: indistance of well form gt: >= 20
    external
      distance of n_CE_Trans from n_CE_Trans: >= 30
device end
```

### 3.1 Syntaktische Konventionen

Der Aufbau von DINGO-II Sprachkonstrukten wird in einer BNF-ähnlichen Notation in Form von Produktionsregeln angegeben. Diese ist besonders bei der Beschreibung rekursiver Strukturen prägnanter als die entsprechende NEMO-Eingabe. Es wird dabei die Notation aus dem DINGO-II Handbuch übernommen, welche folgende Konventionen benutzt:

- Metazeichen sind :, !, [ , ], { , }, {+, +}
- Das Metazeichen : trennt die linke von der rechten Seite einer Produktion.
- Das Metazeichen ! trennt alternative rechte Seiten von Produktionen.
- Die Metazeichen [ und ] schließen optionale Anteile auf rechten Seiten von Produktionen ein.
- Die Metazeichen { und } schließen Anteile auf rechten Seiten ein, die beliebig oft (auch kein mal) auftreten dürfen.
- Die Metazeichen {+ und +} schließen Anteile auf rechten Seiten ein, die beliebig oft (mindestens aber ein mal) auftreten dürfen.
- Nichtterminalsymbole werden als Textstrings dargestellt. Sie bestehen aus Kleinbuchstaben, Ziffern und den Trennsymbolen - oder \_.
- Terminalsymbole, die lexikalische Literalkonstanten repräsentieren, werden in " eingeschlossen.
- Übrige Terminalsymbole, die lexikalische Konstanten repräsentieren, werden als Strings bestehend aus Kleinbuchstaben, Ziffern und den Trennsymbolen - und \_ in unterstrichenem Fettdruck dargestellt.
- Terminalsymbole, die lexikalische Variablen repräsentieren, werden als Strings bestehend aus Großbuchstaben und Ziffern in Normaldruck dargestellt (z.B. IDENTIFIER, NUMBER).

### 3.2 Aufbau einer Technologiespezifikation

Eine Technologiespezifikation in DINGO-II gliedert sich in drei Abschnitte:

```
dingo_2_description :
    declarations object_declarations standard_declarations
```

- Im Deklarationsteil werden wichtige globale Parameter der Technologie definiert. Neben dem Prozeßbezeichner werden hier die Layer festgelegt, symbolische Konstanten definiert, erlaubte Geometrieklassen bestimmt und weitere Festlegungen wie Skalierung und verwendete Maßeinheiten getroffen.
- Der Objektdeklarationsteil enthält die Definitionen der Topologie aller erlaubten Technologieobjekte zusammen mit ihren assoziierten Design Rules. Darunter fallen sog. Module, Bauteile und Zellen.
- Der Standardobjekt-Deklarationsteil enthält die Definition von spezialisierten Technologieobjekten. Im Sinne der objektorientierten Programmierung stellen sie Unterklassen von Objekten des Objektdeklarationsteils dar. Durch Angabe einer Oberklasse werden deren Topologiebeschreibungen und Design Rules geerbt. Das Standardobjekt kann aber einzelne Teile davon durch eigene Spezifikationen verschärfen.

Da der Aufbau des Objekt- und des Standardobjekt-Deklarationsteils fast identisch ist, werden sie in der folgenden Beschreibung nicht gesondert betrachtet. Die Syntax einer Standardobjektdeklaration berücksichtigt deren Auffassung als Spezialisierung eines Technologieobjektes, in dem bestimmte Freiheitsgrade der Layoutausprägungen durch feste Wertzuweisungen eingeschränkt sind.

### 3.2.1 Deklarationsteil

```

declarations :      declaration
                    process IDENTIFIER
                    { declaration } {+ layer_declarator +}
                    declaration_end
declaration :       constant-declaration
                    | atom-declaration
                    | geometry-class-declaration
                    | default-declaration
constant-declaration :
                    const
                    IDENTIFIER arith-expr [ unit ]
                    { "," IDENTIFIER arith-expr [ unit ] }
atom-declaration :
                    atom
                    IDENTIFIER is func-specs
                    { "," IDENTIFIER is func-specs }
func-specs :         func-expr { or func-expr }
geometry-class-declaration:
                    geometry class
                    IDENTIFIER is atom-specs
                    { "," IDENTIFIER is atom-specs }
atom-specs :         atom-spec { or atom-spec }
default-declaration :
                    defaults default { "," default }
default :             function variable IDENTIFIER
                    | geometry class IDENTIFIER
                    | scale [ arith-expr ] scale-unit
                    | figure width [ arith-expr ] scale-unit
scale-unit :          mils | micron | um | nm | lambda
layer-declaration :
                    layer
                    IDENTIFIER layer-attrib
                    { "," IDENTIFIER layer-attrib }
layer-attrib :        CONSTANT | STRING | bool-expr
unit :                V | mV | kV | A | mA | uA | OHM | kOHM
                    | MOHM | nF | pF | fF | W | mW | nW | um | nm

```

Obligatorisch wird im Deklarationsteil als erstes ein Bezeichner für die definierte Technologie festgelegt. Es folgt optional eine Reihe von Deklarationen. Wie in vielen konventionellen Programmiersprachen können in DINGO-II symbolische Konstante definiert werden. Dadurch wird der gesamte Beschreibungstext einfacher lesbar und Änderungen lassen sich einfacher durchführen.

Weiter ist die individuelle Definition von Geometrieklassen möglich. Die hier benutzerdefinierten sowie einige bereits dem System bekannte Bezeichner für Geometrieklassen können in den Objektdeklarationen zur Einschränkung der erlaubten Geometrieklassen benutzt werden. Die Definition von Geometrieklassen fußt auf dem fünfstufigen Geometriemodell in IMAGE. Eine Geometrieklasse ist

definiert durch die Menge unter ihr erlaubter Atome. Atome sind die einzelnen Begrenzungskanten von Gebieten im Layout. Sie werden durch eine eindimensionale Funktionsvorschrift definiert. Dies ist die einzige Stelle in DINGO-II, an der ein Sprachkonstrukt unterhalb der Abstraktionsebene eines Gebietes eingesetzt wird.

Im Default-Deklarationsteil werden Voreinstellungen getroffen, die bestimmte implizite Annahmen des Systems überschreiben. So kann die abhängige Variable in Funktionsdefinitionen für Atome (voreingestellt: "X") ebenso geändert werden wie die Default-Geometrieklasse. Es kann eine globale Skalierung für dimensionslose Konstanten mit optionaler Angaben einer Einheit festgelegt werden und die angenommene Mindestbreite von nicht anderweitig festgelegten Strukturen kann bestimmt werden.

Obligatorisch ist im Deklarationsteil die Deklaration von Masken-Layern. Jeder Layer erhält einen eindeutigen Bezeichner und alternativ ein Attribut zugeordnet. Mit diesem Attribut lässt sich z.B. die Farbe und/oder Schraffur eines Layers für einen Layoutheditor festlegen. Eine weitere Anwendung ist die Zuordnung zwischen DINGO-II Layern und Layern einer externen Maskendatei z.B. im CIF-Format. Eine dritte Möglichkeit ist die Angabe eines booleschen Ausdruckes aus Layern und den Operatoren not, and, andnot, or, eqv und exor erlaubt. Das Attribut wird in DINGO-II nicht weiter ausgewertet, wird aber in der Symboltabelle gespeichert und kann von einer Applikation zur Identifikation der DINGO-II Layer mit den Maskenebenen benutzt werden.

### 3.2.2 Objektdeklarationsteil

```

object-decl-part :
object-decl :      {+ object-decl +}
                  module-declaration
                  | device-declaration
                  | cell-declaration
device-declaration :
                  device
                  object-head
                  { local-module }
                  topology
                  [ design-rules ]
                  device end

```

Innerhalb des Objektdeklarationsteils gibt es die drei Objekttypen Modul, Bauelement und Zelle. DINGO-II nimmt dabei eine Sichtweise im Sinne objektorientierter Konzepte ein. Die definierten Objekte sind keine eigenständigen Layoutstrukturen. Vielmehr stellen sie erlaubte Ausprägungen solcher Layoutstrukturen dar und definieren damit gleichsam Objektklassen. Konkrete geometrische Strukturen im Layout können in diesem Sinn als Instanzen dieser Klassen angesehen werden.

Unter einem Bauelement wird eine semantisch unabhängige Struktur wie etwa ein Transistor oder ein Kontakt verstanden. Ein Modul ist eine semantische Einheit innerhalb eines Bauelementes, die jedoch nicht unabhängig im Layout vorkommen kann. Eine Zelle schließlich ist ein Konglomerat von Bauelementen zu einer funktionalen Einheit wie etwa einem Logikgatter. Ein Bauelement darf Module und Layer als Unterstrukturen haben, eine Zelle darf Bauelemente, Module und Layer als Unterstrukturen haben. Die Unterscheidung ist darüber hinaus jedoch hauptsächlich konzeptionell und hat nur wenige Auswirkungen auf die Syntax der

einzelnen Objektdeklarationen. Daher wird im folgenden nur auf die Struktur einer Bauelement-Deklaration eingegangen.

### Kopf eines Objektes

```

object-head :           IDENTIFIER [ formal-params ]
                        alias IDENTIFIER
                        using
                        using-decl { "," using-decl }
formal-params :         "(" IDENTIFIER { "," IDENTIFIER } ")"
using-decl :            IDENTIFIER { "," IDENTIFIER } type-spec
type-spec :             of nolayer
                        | of any layer
                        | of any module
                        | of some-objects
                        | parameter-type
                        | set-type
some-objects :          local module IDENTIFIER [ actual-params ]
                        | some-global-objects
some-global-objects :   layer IDENTIFIER
                        | module IDENTIFIER [ actual-params ]
                        | "(" IDENTIFIER [ actual-params ]
                        | { "," IDENTIFIER [ actual-params ] } ","
parameter-type :       one of "(" some-global-objects
                        | { "," some-global-objects } ","
set-type :              set of some-global-objects
                        | [ cardinality quantification ]
quantification :       exactly arith-expr
                        | one of
                        | "(" arith-expr { "," arith-expr } ")"
                        | range "[" bound "," bound "]"
bound :                 arith-expr | "?"

```

Kopf und Rumpf eines Bauelementes sind in gewissen Maße mit Kopf und Rumpf einer Prozedur in prozeduralen Sprachen vergleichbar. Der Bauelementkopf definiert einen obligatorischen Bezeichner und optional eine Liste formaler Parameter für das Bauelement. Es folgt die Deklaration aller benutzter Unterstrukturen mitsamt ihrem Objekttyp. In DINGO-II besteht Deklarationspflicht. Jede Deklaration einer Unterstruktur vereinbart einen Bezeichner für diese Unterstruktur, der innerhalb der Bauelementdeklaration bekannt ist. In jedem Fall muß ein Typ angegeben werden. DINGO-II ist daher im Gegensatz zu z.B. Smalltalk eine typisierte Sprache. Die Typisierung ist jedoch nicht streng. Vielmehr kann der Typ einer Unterstruktur auf verschiedene Art festgelegt werden:

- **Unspezifische Typdeklaration** mit einem der Schlüsselworte nolayer, any layer oder any module (Bauelemente dürfen keine Bauelemente oder sogar Zellen als Unterstrukturen haben. Im Kopf einer Zelle sind jedoch auch diese Möglichkeiten erlaubt).

- **Festtypdeklaration** mit oder ohne Angabe aktueller Parameter. Der definierten Unterstruktur wird ein festes Objekt als Typ zugeordnet. Als Objekt kommen im Kopf von Bauelementen nur entweder Layer, zuvor definierte globale Module oder innerhalb des Bauelementes definierte lokale Module in Frage. Im Kopf einer Zelle dürfen dagegen auch Unterstrukturen eines Bauelementtyps oder eines Zelltyps deklariert werden. Durch die Angabe einer Liste aktueller Parameter werden Objekte mit formalen Parametern eindeutig bestimmt (siehe "Parametertypdeklaration").
- **Mengentypdeklaration**. Einer Unterstruktur wird eine Menge von Objekten eines Festtyps (mit oder ohne Parameter) zugeordnet. Durch diese Deklarationsform wird ausgedrückt, daß ein Bauelement mehrere Exemplare einer Unterstruktur verwendet, wobei die (genaue) Kardinalität dieser Menge optional entweder in Form einer exakten Zahl, als Liste von Alternativen oder als Intervall (u.U. mit einer unbestimmten Grenze) angegeben sein kann.
- **Parametertypdeklaration**. Unterstrukturen können mit einer Liste alternativer Typen deklariert werden. Jede solcherart deklarierte Unterstruktur wird in der Liste formaler Parameter des Bauelementes aufgeführt. Die Reihenfolge in der Liste formaler Parameter entspricht der Reihenfolge der Parametertypdeklarationen im using-Teil. Wird ein solcherart definiertes Bauelement (gleiches gilt für Module oder Zellen) als Typ einer Unterstruktur eines zweiten Objektes benutzt, muß diese Typangabe eine Liste aktueller Parameter enthalten. Die Anzahl aktueller Parameter muß mit der Anzahl formaler Parameter übereinstimmen. Jeder aktuelle Parameter wählt einen alternativen Typ aus der one of-Liste der Parametertypdeklaration aus. Eine Bauelementdeklaration mit formalen Parametern lässt sich als abkürzende Schreibweise für die Deklaration einer Menge "ähnlicher" Objektdeklarationen ohne formale Parameter auffassen. Jeder erlaubten individuellen Belegungskombination der formalen Parameter durch aktuelle Parameter entspricht dann ein eigenes parameterloses Objekt.

Beispiel:

```
...
device A (s1, s3)
using
    s1 one of (module c11, module c12)
    s2 module c2
    s3 one of (module c31, module c32)
...
```

äquivalente Menge von Objekten ohne Parameter:

```
device A@c11@c31
using
    s1 module c11
    s2 module c2
    s3 module c31
```

```
device A@c11@c32
using
    s1 module c11
    s2 module c2
    s3 module c32
```

```
device A@c12@c31
using
    s1 module c12
    s2 module c2
    s3 module c31
```

```
device A@c12@c32
using
    s1 module c12
    s2 module c2
    s3 module c32
```

## Lokale Module

```
local-module :      local module
                    object-head
                    { local-module }
                    topology
                    [ design-rules ]
local module end
```

Lokale Module definieren Strukturen, die nur innerhalb des statischen Kontextes eines anderen Objektes bekannt sind. So können Unterstrukturen mit eigener Semantik, die aber nur innerhalb eines Objektes auftreten können, innerhalb dieses Objektes mit einem mnemonischen Bezeichner angesprochen werden. Die Struktur einer Lokalmoduldeklaration entspricht der der Bauelementdeklaration. Innerhalb eines lokalen Moduls dürfen wiederum lokale Module definiert werden, deren Bezeichner nur innerhalb des statischen Kontextes des umschließenden Lokalmoduls bekannt ist. Eine Besonderheit bei lokalen Modulen ist, daß sie erst nach ihrer Benutzung im using-Teil definiert werden.

## Topologiebeschreibungen

```
topology :          description [ quantifiers ] statement
                    { ";" [ quantifiers ] statement }
quantifiers :       { quantifier ":" }
quantifier :        quantor qualified-identifier
quantor :           all | some | any | one
qualified-identifier :
                    IDENTIFIER { "." IDENTIFIER } | self
statement :         topology-description
                    | geometry-description
                    | identification
topology-description :
                    topo-operand { topo-operator topo-operand }
topo-operand :      qualified-identifier
                    | "(" topology-description ")"
topo-operator :     cross | overlap | enclose
                    | in | touch | notouch
geometry-restriction :
                    shape of qualified-id-list
                    is IDENTIFIER { or IDENTIFIER }
qualified-id-list :
                    qualified-identifier
                    | qualified-identifier
                    { "," qualified-identifier }
identification :    qualified-identifier
                    { + is qualified-identifier + }
```

In der Topologiebeschreibung wird die Struktur eines Objektes festgelegt, ohne jedoch auf quantitative Abmessungen der einzelnen Unterstrukturen einzugehen. Da jedoch die Struktur eines Layoutobjektes im wesentlichen seine Funktionalität bestimmt, ist die Topologiebeschreibung der zentrale Teil einer Objektdeklaration. Zur Bildung einer Topologiebeschreibung stehen drei Anweisungsarten zur Verfügung, welche in beliebiger Anzahl und Reihenfolge auftreten dürfen:

- Topologieanweisungen beschreiben die topologischen Beziehungen zwischen deklarierten Unterstrukturen eines Objektes mit Hilfe der Operatoren cross, overlap, enclose, in, touch und notouch. Die Semantik dieser Operatoren soll hier nur anhand einer Graphik veranschaulicht werden. Zur exakten Definition der Semantik siehe [DINGO].

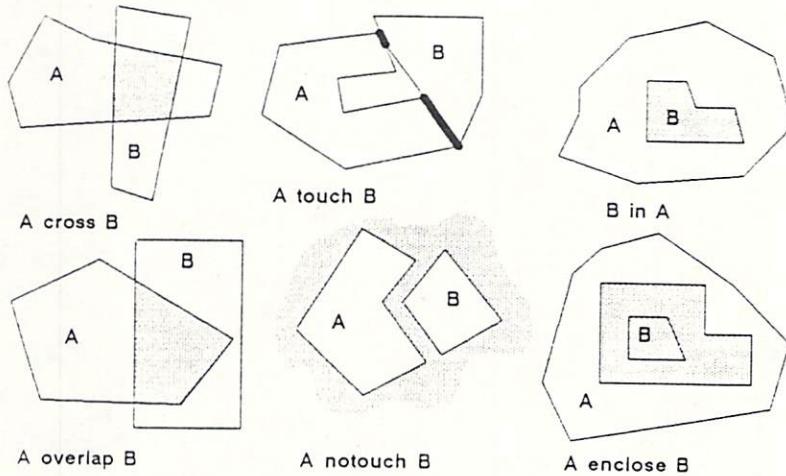


Abb. 3.2: Die Topologieoperatoren in DINGO-II

Als Operanden einer Topologieanweisung dürfen die folgenden Bezeichner benutzt werden:

1. Bezeichner für Strukturen der Typen layer, module, local, module und innerhalb von Zelldeklarationen auch Bezeichner für Strukturen der Typen device, cell, standard device oder standard cell, die im using-Teil der Objektdeklaration deklariert wurden
2. Bezeichner für Unterstrukturen der unter 1. genannten Strukturen. Bezeichner dieser Gruppe werden analog zur Notation für Record-Komponenten in C oder Pascal mithilfe der Punktnotation gebildet. Die erlaubte Länge solcher Pfade ist nur durch die Komplexität der beteiligten Unterstrukturen begrenzt.
3. Schlüsselwort self zur Bezeichnung des Objektes, in dessen statischen Kontext es verwandt wird

Sind einige der Operanden einer Topologieanweisung als Mengentyp deklariert, kann die Topologieanweisung quantifiziert werden. Folgende Quantoren stehen zur Verfügung:

- **all:** Die folgende Anweisung gilt für jedes Element der durch den Quantor gebundenen Struktur.
- **some:** Die folgende Anweisung gilt für mindestens ein Element der durch den Quantor gebundenen Struktur.
- **one:** Die folgende Anweisung gilt für genau ein Element der durch den Quantor gebundenen Struktur.
- **any:** Die folgende Anweisung gilt für ein Element, für einige Elemente oder für alle Elemente der durch den Quantor gebundenen Struktur.

Nicht quantifizierte Topologieanweisungen mit Operanden eines Mengentyps gelten als implizit all-quantifiziert.

- **Geometriebeschränkungen** weisen einem Objekt oder einer Liste von Objekten eine Geometrieklasse zu. Da jedem Objekt in DINGO-II eine Geometrieklasse zugeordnet ist, gilt für Objekte ohne explizite Spezifizierung entweder keine Einschränkung oder die im Deklarationsteil deklarierte Default-Geometrieklasse. Die Zuordnung einer Geometrieklasse gilt nur im lokalen, statischen Kontext. Einem Objekt kann daher in verschiedenen statischen Kontexten verschiedene Geometrieklassen zugeordnet werden.
- **Identifizierungsanweisungen** setzen verschiedene Unterstrukturen eines Objektes zueinander in Beziehung (1). Neben der einfachen Verbindung der einzelnen Operanden wird auch eine Reihenfolge der Anordnung der Unterstrukturen im Layout festgelegt. Typischerweise werden Unterstrukturen unterschiedlicher Hierarchiestufen (durch Angabe qualifizierter Bezeichner) durch eine Identifizierungsanweisung miteinander verbunden. Werden strukturierte Objekte miteinander verbunden, ist damit auch eine Identifizierung korrespondierender Unterstrukturen impliziert. Die Korrespondenz zweier Unterstrukturen kann durch Vergleich der durch ihre *using*-Teile rekursiv definierten part-of-Bäume und anschliessenden Vergleich ihrer Topologie festgestellt werden (siehe Abschnitt 7.8: Topologievergleich).

---

(1) Ein ähnlicher Mechanismus (dort unter dem Namen "merges") ist in einem Artikel über eine spezielle Erweiterung des Smalltalk-Systems zur Bearbeitung von Constraints beschrieben [Bor 81]. Das dort beschriebene System weist auch andere interessante Analogien zu den Beschreibungsmöglichkeiten von DINGO-II auf (z.B. die Referenzierung von Unterstrukturen mittels Pfad-Bezeichner). Der Autor dieses Artikels warnt insbesondere Implementierer von Systemen mit Identifikationsmechanismen vor deren Tücken !

## Design Rule Beschreibungen

```

design-rules :      internal-rules { external-rules }
internal-rules :   internal {+ design-rule +}
external-rules :   external {+ design-rule +}
design-rule :       { quantifier ":" } dr-head dr-body
quantifier :        quantor qualified-identifier
quantor :           all | some | any | one
qualified-identifier :
                    IDENTIFIER { "." IDENTIFIER } | self
dr-head :           dr-property dr-operand ":""
dr-property :       dr-relation dr-operand "," dr-operand ":""
                    width | length | area | in angle
                    | radius | dimension
dr-relation :       overhang | overlap
                    | in distance | distance | out angle
dr-operand :         any layer | any module | any device
                    | any cell | all-objects
                    | qualified-identifier
all-objects :        local module IDENTIFIER [ actual-params ]
                    | all-global-objects
all-global-objects :
                    layer IDENTIFIER
                    | module IDENTIFIER [ actual-params ]
                    | device IDENTIFIER [ actual-params ]
                    | cell IDENTIFIER [ actual-params ]
                    | standard device IDENTIFIER
                    | standard cell IDENTIFIER
actual-params :      "(" IDENTIFIER [ actual-params ]
                    { "," IDENTIFIER [ actual-params ] } ")"
dr-body :            simple-dr-body
                    | conditional-dr-body
simple-dr-body :     { simple-dr-term or } simple-dr-term
simple-dr-term :    { simple-dr-elem and } simple-dr-elem
simple-dr-elem :    [ dr-attribute ] quantification
                    | [ dr-attribute ] dr-expr
                    | asymmetric
                    | "(" dr-value { "," dr-value } ")"
                    | "(" simple-dr-body ")"
dr-attribute :       centered | eccentric | in current direction
                    | against current direction
quantification :    exactly arith-expr
                    | one of
                    | "(" arith-expr { "," arith-expr } ")"
                    | range "[" bound "," bound "]"
bound :              arith-expr | "?"
dr-expr :            comp-op arith-expr
comp-op :             ">=" | ">=" | ">" | "<" | "==" | "!="
dr-value :           dr-expr
                    | "(" dr-expr ")" "*"
                    | "(" dr-expr ")" "+"

```

```

conditional-dr-body :
    if condition then dr-body
        [ else dr-body ]
    | "(" conditional-dr-body ")"
condition :      { cond-term or } cond-term
cond-term :       { cond-elem and } cond-elem
cond-elem :       cond-property dr-operand simple-dr-elem
                  | dr-relation dr-operand "," dr-operand
                  | simple-dr-elem
                  | IDENTIFIER is some-objects
some-objects :   local module IDENTIFIER [ actual-params ]
                  | some-global-objects
some-global-objects :
                  layer IDENTIFIER
                  | module IDENTIFIER [ actual-params ]
cond-property :  dr-property
                  | voltage | current | power | resistance
                  | capacitance | edge length

```

Ebenso wie die Topologie eines Layout-Objektes für seine Funktionalität verantwortlich ist, legen die quantitativen Abmessungen seine elektrischen Parameter fest und sind damit letztendlich ausschlaggebend für das Funktionieren einer implementierten Schaltung. Design Rules können in DINGO-II für jedes Objekt angegeben werden. Es gibt jedoch keine Design Rules, die nicht an irgend ein Objekt gebunden sind. Jedes Objekt kann optional einen Block mit internen und/oder einen Block mit externen Design Rules erhalten. Interne Design Rules legen geometrische Restriktionen für die Unterstrukturen eines Objektes fest. Externe Design Rules beschreiben geometrische Restriktionen für ein Objekt innerhalb seines Layout-Kontextes. Dies beinhaltet geometrische Beziehungen zu anderen Objekten, aber auch zu Layoutinstanzen desselben Objektes.

Jede Design Rule besteht aus den zwei Bestandteilen Kopf und Rumpf. Im Kopf einer Design Rule wird ihr Typ und betroffene Objekte angegeben. Je nach Typ können Eigenschaften eines Objektes oder Beziehungen zwischen Objekten mit einer Design Rule quantifiziert werden. Erlaubte Typen zur Quantifizierung von Eigenschaften sind width, length, area, in angle, radius oder dimension. Erlaubte Typen zur Quantifizierung von Beziehungen sind overhang, overlap, in distance, distance oder out angle. Auch hier sei die Semantik der Operatoren nur anhand einer Graphik illustriert. Die exakte Definition der Semantik findet sich in [DINGO].

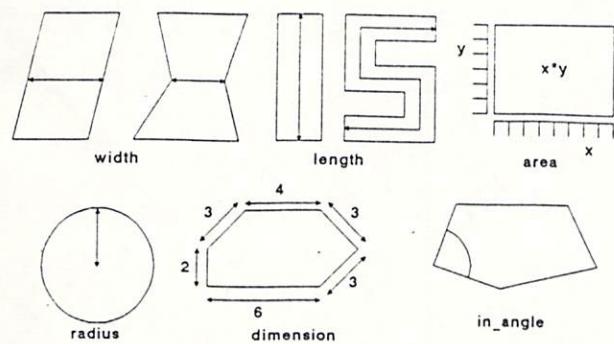


Abb. 3.3a: Design Rule Eigenschaften

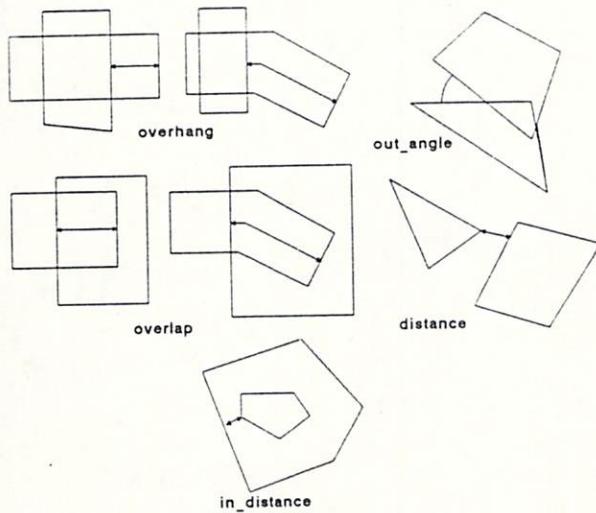


Abb. 3.3b: Design Rule Beziehungen

Ist ein Operand ein Objekt vom SET-Typ, kann die Design Rule mit denselben Quantoren wie in Topologieanweisungen (all, some, any, one) quantifiziert werden. Nicht quantifizierte Design Rules mit Objekten vom SET-Typ gelten auch hier als implizit all-quantifiziert.

Die betroffenen Objekte können auf drei Arten spezifiziert werden:

- einzelne geometrische Strukturen werden durch einen möglicherweise qualifizierten Bezeichner spezifiziert. Dadurch ist auch das Überschreiben von Design Rules für beliebige Unterstrukturen möglich.

- alle Instanzen eines Objektes werden durch Angabe der Objektklasse optional gefolgt von einer Liste aktueller Parameter spezifiziert. Diese Art ist nur in externen Design Rules erlaubt.
- alle Objekte eines Typs werden durch das Schlüsselwort any gefolgt von der Angabe des Objekttyps spezifiziert. Die Design Rule bezieht sich in diesem Fall einfach auf alle Objekte von diesem Typ. Auch diese Art ist nur in externen Design Rules erlaubt.

Der Rumpf einer Design Rule ist vom Kopf durch einen Doppelpunkt getrennt, wodurch der fordernde Charakter einer Design Rule unterstrichen wird. Zur Quantifizierung der geometrischen Restriktion gibt es im Rumpf einer Design Rule syntaktisch zwei Möglichkeiten. Gilt eine Design Rule im gesamten Objektkontext im Layout, reicht eine einfache Quantifizierung. Dazu können in einem Design Rule Wertausdruck Unter- und bei Bedarf auch Obergrenzen eines erlaubten Wertebereiches angegeben werden. Alternativ ist auch die Angabe von Intervallen oder die exakte Festlegung eines Zahlenwertes oder einer konsekutiven Liste von Zahlenwerten möglich. Einfache Quantifizierungen können mit den booleschen Operatoren and und or verknüpft werden.

Reicht es nicht aus, daß eine Design Rule innerhalb des gesamten Objektkontextes gültig ist, kann mit einer Bedingung der lokale Kontext zur Entscheidung über den gültigen Wertebereich herangezogen werden. Die Syntax einer bedingten Design Rule ähnelt der von bedingten Ausdrücken z.B. in der Programmiersprache PASCAL. Als Bedingungen sind boolesche Ausdrücke aus Eigenschaften von und Beziehungen zwischen Objekten erlaubt. Die Syntax ist identisch mit der von Design Rules; Kopf und Rumpf von Bedingungen werden allerdings nicht durch einen ":" getrennt, da es sich nicht um eine Forderung, sondern lediglich um eine Abfrage handelt. Die Liste der erlaubten Schlüsselworte für Eigenschaften in Bedingungen wird um die Schlüsselworte voltage, current, power, resistance, capacitance und edge length erweitert.

Durch die Strukturierungsmöglichkeiten und die Einführung einer Parametrisierbarkeit von Objektbeschreibungen erweist sich noch eine zusätzliche Möglichkeit zur Formulierung von Bedingungen als nützlich: In Design Rule Bedingungen kann die aktuelle Objektklasse eines Operanden abgefragt werden und zur Entscheidung über den erlaubten Wertebereich der geometrischen Einschränkung herangezogen werden.

## 5. Die Symboltabelle

Die DINGO-II Symboltabelle stellt neben der im folgenden Kapitel vorgestellten Zwischensprache DROOLY die zweite Schnittstelle zwischen Frontend und Backend des DINGO-II Compilers dar. Das Compiler-Backend erhält über die Zwischensprache keine Bezeichner für Objekte, sondern nur deren Symboltabellenindizes. Zur Ausgabe sinnvoller Fehlermeldungen benötigt das Backend jedoch auch die zugehörigen Objektbezeichner.

Die Symboltabelle enthält darüber hinaus für eine Applikation u.U. weitere wichtige Informationen wie etwa die im DINGO-II Deklarationsteil definierten Layer-Attribute, Defaultwerte für Geometrieklassen und Maßeinheiten usw. Daher bildet die Symboltabelle neben den vom Backend erzeugten Tabellen für Objektstrukturen und Wertausdrücke den dritten Bestandteil der Technologiedatenhaltung in IMAGE, auf den Applikationen zugreifen können. In dieser Arbeit werden aber nur diejenigen Strukturen und Operationen berücksichtigt, die für das Compiler-Backend von Bedeutung sind.

Aus der Sicht des Compiler-Backends hat die DINGO-II Symboltabelle eine recht einfache Struktur. Sie besteht einfach aus einer Liste von deklarierten, nicht qualifizierten Bezeichnern, in der jeder Bezeichner nur genau einmal auftritt. Durch seine Position in der Liste wird jedem Bezeichner ein eindeutiger Index zugeordnet. Beim Start des Compiler-Backends wird diese Liste in einen Vektor von Zeigern auf Bezeichner eingelesen.

Das Symboltabellenmodul des Compiler-Backends exportiert nur zwei Operationen. Die eine Operation `rdsymtb` hat als einzigen Parameter den File-Descriptor der externen Symboltabelle. `Rdsymtb` läßt beim Start des Compiler-Backends die Symboltabelle ein und alloziert dabei Speicherplatz für die Bezeichner der definierten Objekte. Mit einer zweiten Operation `grepstr` liest das Backend die Bezeichner aus der Symboltabelle aus. Als Parameter wird dazu der Symboltabellenindex übergeben. Funktionsergebnis ist ein Zeiger auf den Bezeichner des gewünschten Objektes.

Zur Veranschaulichung ist hier die resultierende Symboltabelle (aus der naiven Sicht des Compiler-Backends) für die Beispieltechnologie aus Kapitel 3 wiedergegeben:

|     |             |                    |
|-----|-------------|--------------------|
| - 4 | ANYLAYER    |                    |
| - 3 | ANYMODULE   |                    |
| - 2 | ANYDEVICE   |                    |
| - 1 | ANYCELL     |                    |
| 1   | Metal       | 8 diffusion        |
| 2   | Poly        | 9 p_wire           |
| 3   | Contact_Cut | 10 iso_Polygon     |
| 4   | n_Diff      | 11 n_CE_Transistor |
| 5   | p_Diff      | 12 well            |
| 6   | n_Well      | 13 gt              |
| 7   | Gate        |                    |

Abb. 4.1: Symboltabelle der Beispieltechnologie

#### 4. Die DROOLY Zwischensprache

##### 4.1 Grundlagen

Ziel der Definition einer Zwischensprache für den DINGO-II Compiler war es, eine saubere Schnittstelle zwischen Compiler-Frontend (dessen Entwicklung nicht Gegenstand dieser Arbeit ist) und Compiler-Backend zu schaffen. Die hier vorgestellte Zwischensprache DROOLY ("Devices Represented in Object Oriented Lyrics") muß verschiedenen Anforderungen genügen:

- **Vollständige Technologiespezifikation.** Die Struktur von in DINGO-II definierten Technologieobjekten muß sich auch in DROOLY darstellen lassen. Nicht strukturelle Informationen werden dabei nicht nach DROOLY übersetzt, sondern gelangen vom Frontend über die Symboltabelle direkt zu den anfordernden Applikationen.

- **Manuelle Programmerstellung möglich.** Während der Fertigstellung dieser Arbeit stand noch kein voll funktionsfähiges Compiler-Frontend zur Verfügung. Es war daher notwendig, zu Testzwecken DROOLY-Technologiebeschreibungen von Hand zu erstellen. Dies begünstigte die Verwendung eines textuellen Formates. Bei der Integration mit dem Compiler-Frontend sollte jedoch ein Binärformat erwogen werden, da sich ein solches schneller schreiben und lesen lässt.
- **Integrierbarkeit.** Als typische Zwischensprache muß DROOLY einen leichten Übergang von der benutzerfreundlichen, teilweise recht wortreichen Technologiebeschreibungssprache DINGO-II in eine kryptische, dabei aber wohlstrukturierte und möglichst einheitliche interne Darstellung der Technologieinformationen ermöglichen. DINGO-II ist bereits definiert; die Philosophie der objektorientierten Datenhaltung in IMAGE ist ebenso bereits durch mehrere Diplomarbeiten zu diesem Thema [Mör 88], [Pre 88], [Wro 87] weitgehend vorgegeben.

#### 4.2 Vererbungsmechanismen

DINGO-II kennt als objektorientierte Sprache zwei Vererbungsmechanismen, die zwar an unterschiedlichen Stellen bei der Technologiespezifikation auftreten, die jedoch ähnliche Auswirkungen auf die erzeugte Struktur haben.

Objektdeklarationen enthalten in ihrem Kopf die Liste aller Unterstrukturen des Objektes. Jeder deklarierten Unterstruktur sind ein oder mehrere Typen und eine oder mehrere Klassen zugeordnet. Ist die Klasse einer Unterstruktur bei einer Festtypdeklaration oder einer Mengentypdeklaration eindeutig bestimmt, kann im Topologiebeschreibungsteil oder im Design Rule Teil nicht nur auf die Unterstruktur selbst, sondern über einen qualifizierten Bezeichner auch auf deren Unterstrukturen bezug genommen werden. Durch die Verbindung mit einer Klasse erbt die Unterstruktur alle Komponenten ihrer Klasse. Es sind dies nicht nur die deklarierten Unterstrukturen. Auch Topologiebeschreibungen und Resign Rules auf den deklarierten Unterstrukturen werden mitvererbt.

Bei einer Parametertypdeklaration ist keine eindeutige Zuordnung zwischen Unterstruktur und Klasse gegeben. Qualifizierte Bezeichner mit dieser Unterstruktur als Präfix (i.e. als erste Pfadkomponente) sind daher in dieser Objektdeklaration nicht zulässig. Wird aber ein zweites Objekt mit einer Komponente deklariert, die daß erste Objekt als Klasse erhält, so wird über die aktuellen Parameter die Klasse der Unterstruktur des ersten Objektes eindeutig festgelegt. Die Unterstruktur erbt in diesem Moment alle Komponenten ihrer nun eindeutigen Klasse. Qualifizierte Bezeichner werden damit möglich.

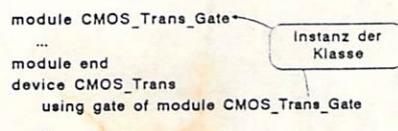


Abb. 5.1: Instanziierung einer Klasse

Der zweite Vererbungsmechanismus in DINGO-II ist offensichtlicher. Durch die Standardobjekt-Deklaration erhält der Benutzer die Möglichkeit, ein Technologieobjekt zu spezifizieren, das in wesentlichen Aspekten seiner

Objektklasse gleicht. Bei der Objektklasse handelt es sich um ein normales Technologieobjekt (device oder cell). Standardobjektbildungen von Standardobjekten sind nicht zulässig. Das Standardobjekt erbt alle deklarierten Unterstrukturen ebenso wie alle topologischen Relationen und Design Rules der Oberklasse auf ihnen. Das Standardobjekt kann einzelne Topologieanweisungen oder Design Rules seiner Oberklasse überschreiben. Jedoch ist es nicht möglich, neue Unterstrukturen zu definieren, da hierdurch die Struktur und damit auch die Funktionalität des Objektes im Layout verändert würde (2).

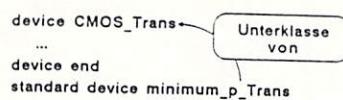


Abb. 5.2: Unterklassenbildung

#### 4.3 Die Beispieltechnologie in DROOLY

Um die folgende Beschreibung der Syntax von DROOLY zu veranschaulichen, wird hier zunächst ein Beispiel für eine Technologiespezifikation in DROOLY gegeben. Das Format entspricht der der Präprozessoreingabe. Der DROOLY-Präprozessor ist kein eigentlicher Bestandteil des Compiler-Backends. Er erlaubt jedoch eine mnemonicische Schreibweise der Schlüsselworte, das Einfügen von Kommentaren sowie die Benutzung von Bezeichnern für Objekte und wurde während der Entwicklung des Compiler-Backends zur Programmierung von Testbeispielen benutzt. Der Präprozessor erzeugt aus seiner Eingabe das eigentliche DROOLY-Format sowie die zugehörige Symboltabelle.

(2) Dies ist ein Unterschied z.B. zu Smalltalk, wenn man auch sonst die DINGO-II Unterstrukturen als Analogon zu den dortigen Instanzvariablen betrachten kann.

```

;
; eine CMOS-Technologie
;

layerdef Metal
layerdef Poly
layerdef Contact_Cut
layerdef n_Diff
layerdef p_Diff
layerdef n_Well

(classdef Gate (fpar diffusion )
 part p_wire (class Poly )
 param diffusion (class n_Diff ) (class p_Diff )

 shape iso_Polygon
 cross [ ] p_wire diffusion
 diffusion width [ if diffusion (classtest n_Diff )
 then >= 15
 else >= 10
 ]
)
(classdef n_Channel_Enhancement_Transistor
 part well (class n_Well )
 set [ and >= 1 <= 5 ] gt (class Gate 2 )

 all gt (merge gt gt )
 all gt indist [ ] gt well
 all gt indist [ >= 20 ] gt well
 dist [ >= 30 ] self self

```

#### 4.4 Syntaktische Struktur

Für die Beschreibung der DROOLY-Syntax wird dasselbe Format wie bei der DINGO-II Beschreibung verwendet. Da DROOLY als Zwischenformat nicht zur direkten Benutzung durch einen menschlichen Benutzer konzipiert ist, kann die lexikalische Struktur der Sprache wesentlich einfacher als die von DINGO-II ausfallen. Eine DROOLY-Technologiespezifikation besteht aus folgenden lexikalischen Primitiven, die formatfrei gemäß der in den folgenden Abschnitten beschriebenen Syntax angeordnet werden können:

- **Trennzeichen** sind Blank, Tabulatorzeichen und Zeilensprung. Die Anzahl und Art von Trennzeichen ist nicht signifikant. In jedem Fall müssen aber Bezeichner, Dezimalkonstanten und Token jeweils durch mindestens ein Trennzeichen voneinander getrennt sein.
- **Kommentare** sind sowohl in der Präprozessoreingabe als auch im DROOLY-Format erlaubt. Ein Kommentar wird durch ein ";" in beliebiger Spalte eingeleitet und reicht bis an das Ende der Zeile. Alle Zeichen zwischen dem Semikolon und dem Zeilenende werden überlesen.

- Lexeme bestehen aus einem, zwei oder maximal drei Sonderzeichen, Ziffern oder Kleinbuchstaben. Die verwendeten Sonderzeichen sind &, (, ), <, >, =, ?, [ , ], :, ! und ~. Zu beachten ist in der Präprozessoreingabe, daß einige Schlüsselworte als erstes Zeichen eine "(" enthalten. Diese ist fester Bestandteil des Schlüsselwortes und darf nicht durch ein Trennzeichen vom Rest des Schlüsselwortes getrennt sein ( z.B.: "classdef" ). Solche Schlüsselworte leiten allgemein ein komplexes Konstrukt (z.B. eine Objektdefinition) oder eine Liste (z.B. die Liste formaler Parameter eines Objektes) ein. Das Konstrukt oder die Liste werden dann durch eine einfache ")" abgeschlossen. Eine vollständige Liste der DROOLY-Lexeme zusammen mit ihren Präprozessor-Ensprachungen findet sich im Anhang A.
- (qualifizierte) Bezeichner haben folgenden Aufbau, geschrieben als regulärer Ausdruck:

@[0-9]\*([0-9]+)\* (3)

Ein Bezeichner besteht also aus einem "@", ohne Trennzeichen gefolgt vom Präfix eines Pfades. Dabei handelt es sich um den Symboltabellenindex eines Objektes. Ist der Bezeichner qualifiziert, folgen jeweils durch einen Punkt, aber ohne Trennzeichen getrennt weitere Symboltabellenindizes. Normalerweise verstehen sich (qualifizierte) Bezeichner relativ zum statischen Kontext ihrer Benutzung. Solche Bezeichner erhalten im Backend automatisch den Bezeichner des aktuellen Objektes als Präfix. Ist jedoch das erste Zeichen nach dem "@" ein Punkt, liegt ein absoluter Pfad vor. Ein besonderer Bezeichner ist das Lexem "%" (das Schlüsselwort self in der Präprozessoreingabe). Dieser wird im Backend durch den Bezeichner des aktuellen Objektes ersetzt.

In der Präprozessoreingabe fehlt das "@"-Zeichen. Die einzelnen Bezeichner für die Pfadkomponenten werden jeweils durch einen Punkt, aber ohne Trennzeichen getrennt. Auch hier können absolute Pfade einfach durch einen Punkt vor dem ersten Bezeichner gebildet werden. Bezeichner werden in der folgenden Syntaxbeschreibung durch IDENTIFIER gekennzeichnet.

- Dezimalkonstanten werden nach folgendem regulären Ausdruck gebildet (Notation wie in der Fußnote (3)):

#[0-9]+

Eine Dezimalkonstante besteht einfach aus einem "#", ohne Trennzeichen gefolgt von einer nichtleeren Ziffernfolge. In der Präprozessoreingabe fehlt das "#".

Im Hinblick auf eine möglichst einheitliche, durchgängige Struktur der zu erzeugenden Tabellen werden schon in DROOLY keine Unterschiede zwischen den einzelnen strukturierten Objekttypen gemacht. Syntaktisch gleichen sich also die DROOLY-Beschreibungen von Modulen, Bauelementen, Zellen, Standardbauelementen und Standardzellen völlig. In Hinblick auf objektorientierte Programmiersprachen definieren alle DROOLY-Objekte Objektklassen. Auch das Konzept der lokalen Module ist syntaktisch in DROOLY nicht mehr zu erkennen. Um trotzdem ein Mittel zur Beschreibung ihrer Lokalität zur Verfügung zu stellen, können Objektklassen mit einem qualifizierten Bezeichner definiert werden. Dieser dient dann auch als

---

(3) [0-9] steht für ein Zeichen (eine Dezimalziffer); runde Klammern ("(", ")") fassen mehrere Zeichen zu einer Kette zusammen; "\*" markiert das kein-, ein- oder mehrmalige Auftreten eines einzelnen Zeichens oder einer geklammerten Zeichenkette; "+" markiert das mindestens einmalige Auftreten eines einzelnen Zeichens oder einer geklammerten Zeichenkette.

impliziter Präfix für alle im statischen Kontext des "lokalen" Moduls benutzten Bezeichner.

#### 4.5 Layerdefinitionen

Alle Informationen aus dem DINGO-II Deklarationsteil werden in der Symboltabelle gespeichert. Da jedoch Layer als unstrukturierte Objektklassen aufgefaßt werden können, werden die Layerdeklarationen in den DROOLY-Text übernommen. Eine Technologiebeschreibung im DROOLY-Zwischenformat besteht somit aus Objektdefinitionen, wobei zwei Objekttypen möglich sind:

```
intermediate : {+ layer_definition +}
                {+ class_definition +}
```

Da Layer unstrukturierte Objekte sind, unterscheidet sich die Syntax ihrer Definition von der Syntax der übrigen Objektdefinitionen:

```
layer_definition : layerdef IDENTIFIER
```

Layer werden abgesehen von ihrer Strukturlosigkeit wie normale Objekte behandelt. Insbesondere dürfen natürlich wie in DINGO-II Unterstrukturen vom Typ Layer deklariert werden. Es ist jedoch wie in DINGO-II nicht möglich, Design Rules direkt an einen Layer zu binden. Sollte so etwas notwendig werden, ist eine eigene Objektklasse mit dem Layer als einziger Unterstruktur zu definieren, innerhalb welcher Design Rules für die Unterstruktur angegeben werden können.

#### 4.6 Objektklassendefinitionen

Bedingt durch die Notwendigkeit, Standardobjekten den Bezeichner ihrer Oberklasse mitzugeben, gibt es zwei Typen von Objektklassendefinitionen:

```
class_definition :
    classdef IDENTIFIER
    [ fpar {+ IDENTIFIER +} ")" ]
    { declaration } { description } ")"
    | stddef IDENTIFIER class_call
        { declaration } { description } ")"
class_call : class IDENTIFIER { INTEGER } ")"
```

Der Kopf eines Objektes definiert einen eindeutigen Bezeichner und optional eine Liste formaler Parameter. Es folgt die Deklaration der Unterstrukturen, wiederum gefolgt von der Beschreibung des Objektes. Standardobjekten wird zusätzlich der Bezeichner ihrer Oberklasse zugewiesen. Gibt die Definition der Oberklasse eine Liste formaler Parameter an, kann das Standardobjekt eine entsprechende Liste aktueller Parameter enthalten. Dadurch können die einzelnen Parametertypdeklarationen von Unterstrukturen der Oberklasse mit eindeutigen Klassen versehen werden. Das Standardobjekt erbt dann Unterstrukturen mit einer eindeutigen Klasse. Werden keine aktuellen Parameter angegeben, werden parametertypdeklarierte Unterstrukturen geerbt. Eine nur teilweise Angabe von aktuellen Parametern ist nicht erlaubt.

Anders als in DINGO-II werden als aktuelle Parameter nicht Bezeichner übergeben, sondern Indizes in die Alternativliste von Klassen bei einer parametertypdeklarierten Unterstruktur. Da die alternativen Klassen wieder aktuelle Parameter tragen können, welche wiederum aktuelle Parameter tragen können,

ergibt sich eine u.U. recht komplizierte Struktur zur genauen Spezifikation einer Klasse (welche allerdings wohl nur in den seltensten Fällen über eine zweifache Schachtelung hinausreichen wird). Durch die Angabe eines einfachen Positionszeigers in die Liste alternativer Klassen wird das sonst nötige Duplizieren dieser komplizierten Struktur vermieden. Schließlich steht bei einer Zwischensprache nicht die einfache Lesbarkeit im Vordergrund, sondern ihre Ausdrucksstärke und einfache Verarbeitbarkeit.

#### 4.7 Deklaration von Unterstrukturen

Wie in DINGO-II ist auch in DROOLY die Deklaration von Unterstrukturen auf verschiedene Art möglich. Die unterschiedlichen Deklarationen werden jeweils durch ein besonderes Schlüsselwort eingeleitet:

```

declaration :      part | param | set
part :            part IDENTIFIER class_call
param :           param IDENTIFIER {+ class_call +}
set :             set IDENTIFIER class_call card_range
card_range :      "[" [ card_body ] "]"
card_body :       eq_spec
                  | or_card
                  | and ">=" INTEGER "<=" INTEGER
or_card :         or or_card eq_spec
                  | or eq_spec eq_spec
eq_spec :         "==" INTEGER

```

Für jede Art von Unterstruktur wird zunächst ein Bezeichner definiert, der innerhalb des statischen Kontextes eindeutig sein muß. Einem PART (Festtyp-deklaration) wird dann eine eindeutige Objektklasse (u.U. mit aktuellen Parametern) zugeordnet. Unspezifische Typdeklarationen werden als Spezialfall der Festtypdeklaration behandelt. Dazu werden vom Compiler-Frontend Symboltabellen-einträge für die speziellen Objekte NOLAYER, ANYLAYER, ANYMODULE, ANYDEVICE und ANYCELL erzeugt und als Objekttypen für unspezifische Typdeklarationen verwandt.

Ein PARAM (Parametertypdeklaration) erhält eine Liste alternativer Objekt-klassen. Ein SET (Mengentypdeklaration) schließlich kann neben der eindeutigen Objektklasse optional noch eine Kardinalitätsangabe erhalten. Die Syntax dieser Kardinalitätsangabe ist eine Untermenge eines einfachen Wertausdruckes, wie er in quantifizierten Geometrieeinschränkungen benutzt wird (s.u). Erlaubt sind an dieser Stelle nur über or verkettete Alternativen oder die Angabe einer unteren und einer oberen Intervallgrenze.

#### 4.8 Beschreibungsteil

Nach der Deklaration aller Unterstrukturen folgt der Beschreibungsteil:

```

description :      merge | forward | relation
merge :            (merge {+ IDENTIFIER +} ")"
forward :          [ [ quantor_spec ] IDENTIFIER ] property
property :         property_type expression
                   | (shape {+ IDENTIFIER +}
property_type :    width | area | dimension | radius | length
quantor_spec :     all | some | any | one

```

Ein Beschreibungs-"Anweisung" besteht in DROOLY alternativ aus einem *merge*, einem *forward* oder einer *relation*. Ein *merge* – der Begriff ist aus dem ThingLab-System entlehnt (siehe Fußnote auf Seite 23) – ist die DROOLY-Entsprechung der Identifikationsanweisung aus DINGO-II. Hinter dem Schlüsselwort (*merge*) kann eine beliebige Liste zumeist qualifizierter Bezeichner angegeben werden. Die Layoutinstanzen der zugehörigen Objekte werden mitsamt ihren Unterstrukturen miteinander identifiziert, wobei durch ihre Reihenfolge in der Liste zusätzlich eine Reihenfolge im Layout induziert wird.

*Forwards* (4) bzw. die zugrundeliegenden *properties* sind die Übersetzungsprodukte von einstelligen DINGO-II Design Rules bzw. von Geometriebeschränkungen aus dem Topologiebeschreibungsteil. Da die beschriebenen Eigenschaften dem DINGO-II Design Rule Teil entstammen, enthalten sie einen Wertausdruck, der die Eigenschaft quantifiziert. Die Struktur von Wertausdrücken wird in Abschnitt 5.9 behandelt.

Fehlt der Bezeichner, so wird die Eigenschaft der aktuellen Klasse zugeordnet. Durch Angabe eines beliebigen, qualifizierten Bezeichners ist es möglich, Eigenschaften von Unterstrukturen und deren Unterstrukturen festzulegen. Beschreibt ein *forward* eine Eigenschaft einer mengentypdeklarierten Unterstruktur, ist die zusätzliche Angabe eines Quantors möglich (nicht quantifizierte *forwards* gelten in diesem Fall als implizit all-quantifiziert).

Bei Geometriebeschränkungen muß die in DINGO-II mögliche Angabe einer Liste von Objekten in mehrere *forwards* aufgelöst werden. Der Bezeichner gehört zu einer im DINGO-II Deklarationsteil definierten Geometrieklasse oder zu einer der dem System direkt bekannten Geometrieklassen, die jedoch ebenfalls eigene Symboltabelleneinträge haben müssen.

```

description :      merge | forward | relation
relation :        quantor_spec IDENTIFIER relation
                  | complex_area
complex_area :    relation_type expression area area
area :            IDENTIFIER | class_call | complex_area
quantor_spec :   all | some | any | one
relation_type :  indist | cross | overlap |
                  touch | dist | outangle
class_call :      (class IDENTIFIER { INTEGER } )"

```

Im Gegensatz zu DINGO-II gibt es in DROOLY keine Unterscheidung zwischen Topologiebeschreibung und Design Rules. Design Rules werden vielmehr als quantitative Topologieausdrücke geschrieben. Zur Rechtfertigung dieser Vorgehensweise ist es notwendig, eine Zuordnung der Operatoren des Topologie-Teils zu den Operatoren des Design Rule Teils zu finden. Eine solche Zuordnung ist in der folgenden Tabelle gegeben. Die erste Spalte enthält den Operator in seiner DINGO-II Topologiebeschreibungs-Form, die zweite Spalte enthält den vergleichbaren Operator aus den Design Rules und die dritte Spalte nennt den entsprechenden Operator in DROOLY-Notation:

(4) In Anlehnung an den Forward-Begriff in einem Artikel über die Erweiterung von Smalltalk um PART-OF-Hierarchien [Bla 87]

| Lage-schlüssel | Design Rule Typ | DROOLY-Operator |
|----------------|-----------------|-----------------|
| in             | in_distance     | indist          |
| cross          | overhang        | cross           |
| overlap        | overlap         | overlap         |
| enclose        | distance        | enclose         |
| touch          | ---             | touch           |
| notouch        | distance        | dist            |
| ---            | out_angle       | outangle        |

Eine weitere Eigenschaft der Design Rule Typen in DINGO-II macht die Zusammenfassung von Topologiebeschreibung und Design Rules sinnvoll: Laut DINGO-II Handbuch ist eine Design Rule vom Typ *overhang* nur definiert, wenn für beide Operanden die Topologiebeziehung *cross* erfüllt ist. Gleiches gilt für die Paare *overlap/overlap*, *indistance/in* und *distance/notouch*. Der Design Rule Typ *out\_angle* ist nur bei einer der Topologiebeziehungen *cross*, *overlap* oder *touch* zwischen den beiden Operanden definiert. Durch diese Zuordnung wird die Verwendung eines einzigen Satzes von Operatoren sowohl für die Topologiebeschreibung als auch für die Design Rules möglich. Topologieausdrücke enthalten dann nur einen leeren Wertausdruck. Als Operanden für eine *relation* kommen verschiedene Objekttypen in Frage:

- innerhalb der statischen Umgebung des aktuellen Objektes deklarierte Unterstrukturen oder deren Unterstrukturen. Sind dem Objekt in einer Unterstrukturdeklaration bereits quantifizierte RELATIONS oder PROPERTIES zugeordnet, werden die neu hinzukommenden Quantifizierungen mit den alten Quantifizierungen and-verknüpft.
- alle Objekte eines Typs. Dieser Fall wird als Sonderfall des ersten Objekttyps unter Verwendung der speziellen "Pseudoobjekte" ANYLAYER, ANYMODULE, ANYDEVICE, ANYCELL behandelt.
- eine Objektklasse, u.U. mit Angabe von aktuellen Parametern. Operanden dieses Typs dürfen in DINGO-II nur in zweistelligen, externen Design Rules verwandt werden. Ist eine Parameterliste spezifiziert, so ist die Design Rule nur auf die Teilmenge von Objektinstanzen anwendbar, die die Parameterbelegung erfüllen, ansonsten gilt sie für Objektinstanzen mit beliebiger Parameterbelegung.
- Operand einer RELATION kann wieder eine RELATION sein. RELATIONS repräsentieren ja selbst Strukturen im Layout (mit Flächenmaß 0 bei *dist* und *touch*) und lassen sich auf natürliche Weise als Operanden weiterer RELATIONS benutzen, was Topologieausdrücke von beliebiger Schachtelungstiefe ermöglicht.

Werden innerhalb einer RELATION Operanden eines Mengentyps benutzt, können RELATIONS von einem oder mehreren Quantoren beherrscht werden. Nicht quantifizierte Operanden eines Mengentyps gelten als implizit all-quantifiziert.

#### 4.9 Wertausdrücke

```

expression :      " [ " [ body ] " ] "
body :           simple_body
                 | expression
simple_body :    if condition then body [ else body ]
                 | or simple_body simple_body
                 | and simple_body simple_body
                 | [ attribute_spec ] value_spec
                 | (asymmetric {+ value_spec +} ")"
value_spec :      ">=" INTEGER | "<=" INTEGER
                 | ">"  INTEGER | "<"   INTEGER
                 | "=="  INTEGER
attribute_spec : centered | eccentric
                  | in current direction
                  | against current direction
condition :       or condition condition
                  | and condition condition
c_forward :       c_forward
relation :        relation
IDENTIFIER :      IDENTIFIER
(classtest IDENTIFIER { INTEGER } ")
[ [ quantor_spec ] IDENTIFIER ] c_property
property :        c_property_type expression
c_property_type : width | area | dimension | radius | length
                  | edge length | voltage | current
                  | power | resistance | capacitance | in angle

```

Wertausdrücke entsprechen in DINGO-II den Design Rule Rümpfen. Syntaktisch sind sie die komplizierteste Struktur der DROOLY-Zwischensprache. Das entspricht aber durchaus den Verhältnissen in DINGO-II. Allerdings sind durch die gewählte Präfixnotation zur Bildung boolescher Ausdrücke zwischen einfachen Wertausdrücken bzw. zwischen Bedingungen zusätzliche Regeln zur Berücksichtigung von Assoziativitäten und Präzedenzen unnötig. Da in DROOLY nicht zwischen Design Rules und Topologiebeschreibungen unterschieden wird, erhält jede RELATION und jede PROPERTY (mit Ausnahme der Geometrie einschränkungen) einen Wertausdruck. Dieser ist bei Übersetzungsprodukten von Topologiebeschreibungen allerdings leer. Wertausdrücke werden außerdem zur Spezifikation einer Kardinalität in Mengentypdeklarationen verwandt. Die dabei zulässige Syntax ist aber wesentlich eingeschränkt (vgl. Seite 33).

Wertausdrücke werden in DROOLY in eckige Klammern ("[", "]") eingeschlossen. Ein Wertausdruck kann entweder aus einem einfachen Rumpf bestehen oder er ist bedingt und besitzt dann einen then- und optional einen else-Zweig die von einer Bedingung beherrscht werden. Ein einfacher Rumpf ist ein Präfixausdruck, bestehend aus den booleschen Operatoren or und and und aus Wertbestimmungen als Operanden. Im einfachsten Fall besteht eine Wertbestimmung aus einem Ordnungssymbol (">=", "<=", "<", ">", "=="") und einem Zahlenwert. Arithmetische Ausdrücke sind anstatt der Zahlenwerte nicht zulässig. Diese dienen in DINGO-II nur der Bequemlichkeit, leichten Lesbar- und Änderbarkeit, werden aber letztendlich vollständig aus Konstanten gebildet und können daher schon im Frontend ausgewertet werden. Quantifikationen in DINGO-II (das sind einfache arithmetische Ausdrücke, Listen von alternativen Ausdrücken oder Intervalle) müssen im Frontend in einfache Rümpfe übersetzt werden. Die Syntax entspricht dabei der einer Kardinalitätsangabe.

Optional kann einer Wertbestimmung noch ein Attribut zugeordnet sein. Dieses besteht aus den Schlüsselworten centered, eccentric, in current direction oder against current direction.

Es gibt eine Sonderform der Wertbestimmung, die im Zusammenhang mit PROPERTIES vom Typ dimension benutzt wird. Der Wertausdruck für eine solche PROPERTY spezifiziert Werte für eine Folge von Polygonkanten. Dabei wird jeder Polygonkante ein Wert zugewiesen. Eine Wertbestimmung dieser Art beginnt mit dem Schlüsselwort (asymmetric, gefolgt von der Liste von Wertspezifikationen, und wird durch eine ")" abgeschlossen.

Mithilfe solcher einfachen Rümpfe lassen sich Werte festlegen, die innerhalb des gesamten statischen Kontextes einer Objektklasse Gültigkeit haben. Muß die Gültigkeit in Abhängigkeit von Bedingungen für den geometrischen Kontext der betroffenen Struktur/Strukturen eingeschränkt werden, bedient man sich bedingter Wertausdrücke. Sie bestehen aus Bedingungen und Wertausdrücken, die bei Erfüllung der Bedingung in Kraft treten.

FORWARD

Eine Bedingung besteht allgemein aus einem booleschen Ausdruck mit forwards, RELATIONS oder CLASSTESTs. Die FORWARD\$ und RELATIONS entsprechen syntaktisch den FORWARDs und RELATIONS, wie sie als Übersetzungprodukt von Design Rule Köpfen entstehen. Der ihnen zugeordnete Wertausdruck stellt in diesem Fall jedoch keine Forderung dar, sondern ist als Test zur Auswahl des then- oder else-Zweiges zu verstehen. In DINGO-II wird dieser semantische Unterschied durch den fehlenden ":" in Bedingungen angezeigt. In DROOLY gibt es keine syntaktische Unterscheidung. Das Konstrukt des CLASSTESTs erlaubt es, Wertausdrücke in Abhängigkeit von der Klasse eines Objektes und speziell von ihrer Belegung mit aktuellen Parametern zu formulieren. Syntaktisch entspricht dieses Konstrukt der Festtypdeklaration einer Unterstruktur, wird jedoch nicht durch das Schlüsselwort (class, sondern durch (classtest eingeleitet.

Als Wertausdrücke im then- und im else-Zweig sind allgemeine (also möglicherweise wieder bedingte) Wertausdrücke zulässig. Treten bedingte Wertausdrücke geschachtelt auf, so wird ein else-Zweig dem jeweils innersten if zugeordnet. Will man von dieser Regelung abweichen, so kann man dies mit Hilfe von Klammerungen erreichen. Im folgenden sind einige Beispiele für DROOLY-Wertausdrücke gezeigt:

```
[ >= 2 ]
"Der Wert soll größer oder gleich 2 sein"

[ and >= 3 <= 6 ]
"Der Wert soll im geschlossenen Intervall [3,6] liegen"

[ if gate.poly width [ == 5 ]
  then < 6
  else > 7
]
"Falls die Breite des Polysilizium-Layers im Gate gleich 5 ist,
  soll der Wert kleiner als 6 sein, andernfalls größer als 7"
```

```
[ if or gate.poly width [ == 5 ]
    gate.diff width [ > 7 ]
then == 15
else if gate.diff width [ < 3 ]
    then or or == 17 == 18 == 19
    else == 16
]
```

"Falls die Breite des Polysilizium-Layers im Gate entweder gleich 5 oder aber größer als 7 ist, soll der Wert gleich 15 sein. Andernfalls, falls der Polysilizium-Layer im Gate schmäler als 3 ist, soll der Wert aus der Menge {17,18,19} sein; sonst wird ein Wert von 16 gefordert."

```
[ if or gate.poly width [ == 5 ]
    gate.diff width [ < 5 ]
then [ if gate.diff width [ < 3 ]
    then or or == 17 == 18 == 19
]
else == 16
]
```

"Falls die Poly-Breite im Gate entweder gleich 5 oder aber kleiner als 5 ist, soll der Wert aus der Menge {17,18,19} sein, falls die Diffusionsbreite auch noch kleiner als 3 ist. Ist weder die Poly-Breite gleich 5 noch die Diff-Breite kleiner als 5, soll der Wert gleich 16 sein"

```
(classdef gate (fpar diff )
    param diff (class n_Diff ) (class p_Diff )
    .
    .
    diff width [ if diff (classtest n_Diff )
        then >= 3
        else >= 5
    ]
}
```

"Falls die Unterstruktur diff durch eine aktuelle Parameterbelegung mit der Klasse n\_Diff deklariert ist, soll die Breite von diff größer als 3 sein, andernfalls größer als 5"

## 6. Übersetzung DINGO-II nach DROOLY

Nachdem die Schnittstellen zwischen Frontend und Backend (Symboltabelle und DROOLY-Zwischenformat) beschrieben sind, soll in diesem Abschnitt eine Anleitung zur Übersetzung einer DINGO-II Datei in die resultierende Symboltabelle und die DROOLY-Datei gegeben werden. Bei der Erzeugung der Symboltabelle werden nur die für das Backend relevanten Teile betrachtet. Teile einer DROOLY-Beschreibung werden in Form der Präprozessoreingabe angegeben. Dazu werden die gleichen Konventionen verwandt wie bei der Beschreibung der DROOLY-Syntax. Das Frontend erzeugt natürlich die im Anhang A aufgelisteten Token für Schlüsselworte, "@" mit Symboltabellenindizes für (qualifizierte) Bezeichner und "#" mit Zahlenwert für Zahlen und ausgewertete arithmetische Ausdrücke.

Eine Technologiespezifikation in DINGO-II beginnt mit dem Deklarationsteil. Die Informationen aus dem Declarationsteil sind zwar für eine Applikation von Bedeutung, nur wenige werden jedoch für die Übersetzung von der DROOLY-Datei in die tabellarische Technologierepräsentation und zur Ausgabe von Fehlermeldungen im Backend benötigt:

- Bezeichner der deklarierten Layer.
- Bezeichner von (dem System implizit bekannten oder benutzerdefinierten) Geometrieklassen.

Zu jedem deklarierten Layer wird eine Zeile der Form

layerdef IDENTIFIER

in die DROOLY-Ausgabe geschrieben. Die Layerattribute erscheinen nur in der Symboltabelle, werden also nicht in die DROOLY-Ausgabe übernommen.

Es folgen die Objektdeklarationen. Für jedes Objekt wird ein neuer Symboltabelleneintrag eingerichtet und Objekttyp, Objektbezeichner und Alias dort eingetragen. Die Deklaration von Modulen, lokalen Modulen, Bauelementen und Zellen beginnt im DROOLY-Format mit einer Zeile der Form

(classdef IDENTIFIER .

Der Objekttyp wird nicht in die DROOLY-Datei übernommen. Zur Deklaration von lokalen Modulen gibt es in DROOLY kein eigenes syntaktisches Konstrukt. Alle Objekte werden global deklariert. Um ein Lokalmodul als solches kenntlich zu machen, wird sein Objektbezeichner qualifiziert, d.h. es erhält die Bezeichner für die umschließenden Objekte als Präfix seines Pfadnamens. Man betrachte dazu folgendes Beispiel:

DINGO-II:

```

module A
  using
    aa of local module AA
  local module AA
    using
      aaa of local module AAA
    local module AAA
    ...
    local module end
    ...
  local module end
  ...
module end

```

DROOLY:

```

(classdef A.AA.AAA
  ...
)

(classdef A.AA
  part aaa (class A.AA.AAA )
)

(classdef A
  part aa (class A.AA )
)

```

Durch die qualifizierten Bezeichner wird in DROOLY dieselbe hierarchische Struktur wie in DINGO-II beschrieben. Zu beachten sind dabei jedoch zwei Punkte:

- Normalerweise werden Bezeichner innerhalb eines Objektes relativ zum Objektbezeichner betrachtet. Die Unterstruktur aaa im lokalen Modul A.AA hat also den absoluten Pfadnamen A.AA.aaa . Klassenbezeichner werden jedoch nicht automatisch mit dem Präfix des Objektes versehen, in dessen Kontext sie benutzt werden. Objektklassen können ja in DROOLY nur global deklariert werden. Hinter dem Schlüsselwort (class müssen also immer absolute Pfadbezeichner stehen. Fehlt der einleitende Punkt, wird er vom Backend automatisch ergänzt.
- Die Bezeichner lokaler Module sind durch die globale Deklaration auch außerhalb ihres statischen DINGO-II Kontextes bekannt. Das Backend würde also auch die Verwendung des Lokalmodulbezeichners außerhalb dieses statischen Kontextes akzeptieren. Bei Übersetzungsprodukten von DINGO-II Beschreibungen kann dieser Fall aber nicht auftreten.

Standardbauelementen und Standardzellen wird bei ihrer Deklaration eine Bauelement- bzw. Zellenzuweisung zur Deklaration der Oberklasse mitgegeben. Die äquivalente Deklaration von Standardobjekten hat in DROOLY folgendes Format:

```
(stddef IDENTIFIER (class IDENTIFIER { INTEGER } ")")
```

Darin ist der erste Bezeichner der Bezeichner des Standardobjektes. Der zweite Bezeichner, zusammen mit der Liste aktueller Parameter, bezeichnet das zugewiesene Bauelement oder die zugewiesene Zelle. Im Gegensatz zu DINGO-II werden in DROOLY aktuelle Parameter nicht durch den jeweiligen Klassenbezeichner u.U. zusammen mit dessen aktuellen Parametern (...) angegeben, sondern durch einen Positionsindex der Liste alternativer Klassen in der Parametertypdeklaration des formalen Parameters. Die alternativen Klassen werden dazu beginnend bei 1 durchnummierter. Man betrachte folgendes Beispiel:

DINGO-II:

```

device CMOS_Transistor ( diffusion, gate )
  using
    diffusion one of ( layer n_Diff, layer p_Diff )
    gate one of (
      module CMOS_Gate ( n_Diff, p_Well )
      module CMOS_Gate ( p_Diff, n_Well )
    )
  ...
device end

standard device Mini_p_Transistor
is
  CMOS_Transistor ( p_Diff, CMOS_Gate ( p_Diff, n_Well ) )
...

```

DROOLY:

```

(classdef CMOS_Transistor (fpar diffusion gate )
  param diffusion (class n_Diff) (class p_Diff)
  param gate (class CMOS_Gate 1 2)
    (class CMOS_Gate 2 1)
  ...
)

(stddef Mini_p_Transistor
  (class CMOS_Transistor 2 2))

```

In diesem Beispiel sieht man auch schon das Übersetzungsprodukt einer Liste formaler Parameter. Ein Objektkopf mit formalen Parametern hat in DROOLY allgemein folgendes Format:

```
classdef IDENTIFIER (fpar { + IDENTIFIER + } ") "
```

Die Bezeichner in der Parameterliste bezeichnen jede parametertypdeklarierte Unterstruktur des Objektes in der Reihenfolge ihrer Deklaration.

Auf die Angabe formaler Parameter folgt in DINGO-II die Deklaration der benutzten Unterstrukturen. Diese Reihenfolge wird auch in DROOLY beibehalten. Unspezifische und Festtypdeklarationen werden in DROOLY durch die Verwendung von Pseudoobjekten zusammengefaßt. Jedes dieser Pseudoobjekte NOLAYER, ANYLAYER, ANYMODULE, ANYDEVICE und ANYCELL hat einen eigenen Symboltabelleneintrag.

Eine Festtypdeklaration sieht in DROOLY folgendermaßen aus:

```
part IDENTIFIER (class IDENTIFIER { INTEGER } ") "
```

Bei einer Deklaration mit unspezifischem Typ wird der Bezeichner des jeweiligen Pseudoobjektes als Klassenbezeichner eingesetzt. Aktuelle Parameter werden auch hier einfach durch Angabe der Positionen in der Liste alternativer Klassen einer parametertypdeklarierten Unterstruktur übergeben.

Parametertypdeklarationen haben in DROOLY folgendes Format:

```
param IDENTIFIER { + class IDENTIFIER { INTEGER } ")" + }
```

Parametertypdeklarationen werden durch ein spezielles Schlüsselwort param eingeleitet. Für die Liste der Klassen werden die einzelnen Klassen u.U. mit aktuellen Parametern aufgelistet, jeweils eingeleitet durch das Schlüsselwort class und beendet durch eine ")".

Mengentypdeklarationen schließlich haben in DROOLY folgendes Format:

```
set "[" card_body "]"
    IDENTIFIER class IDENTIFIER { INTEGER } ")"
```

Wieder gibt es ein spezielles Schlüsselwort set zur Einleitung einer Mengentypdeklaration. In eckigen Klammern erfolgt die Angabe eines erlaubten Kardinalitätsbereiches. Wird der Kardinalitätsbereich nicht weiter eingeschränkt, müssen dennoch die leeren eckigen Klammern in der DROOLY-Datei erscheinen. Syntaktisch entspricht eine Kardinalitätsangabe einem Wertausdruck, wie er in quantitativen Beschreibungen verwandt wird. Dabei sind jedoch nur folgende Konstruktionen zulässig:

- Arithmetische Ausdrücke hinter dem Schlüsselwort exactly werden in einen einfachen Wertausdruck übersetzt:

|                  |   |
|------------------|---|
| <u>DINGO-II:</u> | <u>cardinality</u> <u>exactly</u> 4 + 8 |
| <u>DROOLY:</u>   | [ == 12 ]                               |

- Alternative Listen von Werten werden in einen durch or verknüpften Wertausdruck in Präfixnotation übersetzt:

|                  |  |
|------------------|--|
| <u>DINGO-II:</u> | <u>cardinality</u> <u>one_of</u> ( 2, 4, 6 ) |
| <u>DROOLY:</u>   | [ <u>or</u> <u>or</u> == 2 == 4 == 6 ]       |

- Intervallangaben werden in einen durch and verknüpften Wertausdruck übersetzt:

|                  |   |
|------------------|---|
| <u>DINGO-II:</u> | <u>cardinality</u> <u>range</u> [ 5, 10 ] |
| <u>DROOLY:</u>   | [ <u>and</u> >= 5 <= 10 ]                 |

- Andere Formen werden vom Compiler-Backend syntaktisch erkannt und zurückgewiesen.

Mit der Deklaration aller Unterstrukturen ist der Objektkopf abgeschlossen. In DINGO-II können nun lokale Module definiert werden, deren Bezeichner nur innerhalb des statischen Kontextes einer Objektdefinition bekannt ist. In DROOLY existiert kein äquivalentes syntaktisches Konstrukt. Stattdessen werden alle Objekte global definiert, wie schon weiter oben vorgeführt. Dieser Punkt dürfte bei der Übersetzung von DINGO-II nach DROOLY die größten Schwierigkeiten bereiten. Ordnet man jedoch im Frontend die "Codegenerierungs"-Semantikregeln erst nach Abarbeitung aller Lokalmodule an, kann man unter Ausnutzung des NEMO-Wertestacks auf eine explizite Zwischenspeicherung der Informationen aus dem Objektkopf verzichten. (5)

---

(5) Das Problem der Entflechtung verschachtelter Objektdefinitionen existiert auch bei der Übersetzung konventioneller Programmiersprachen, wie z.B. bei der Übersetzung von PASCAL in das maschinennähere C, welches keine geschachtelten Prozedurdeklarationen kennt.

Es folgt der Beschreibungsteil, der in DINGO-II in die Topologiebeschreibung und die Design Rules unterteilt wird. Topologieanweisungen bestehen entweder aus Geometrieeinschränkungen, Identifikationen oder aus Topologierelationen. Während für die ersten beiden Formen äquivalente Konstrukte in DROOLY existieren, teilen sich die letzteren eine gemeinsame Syntax mit den zweistelligen Design Rules. Geometrieeinschränkungen mit einer Liste von Objekten müssen bei der Übersetzung nach DROOLY in einzelne Anweisungen aufgespalten werden:

|                  |  |
|------------------|--|
| <u>DINGO-II:</u> | <u>shape of polyline, diffline is manhattan or poly_45</u> |
| <u>DROOLY:</u>   | <u>polyline shape manhattan</u>                            |
|                  | <u>polyline shape poly_45</u>                              |
|                  | <u>diffline shape manhattan</u>                            |
|                  | <u>diffline shape poly_45</u>                              |

Mehrfache Geometrieeinschränkungen auf demselben Objekt gelten als implizit or-verknüpft. Identifikationsanweisungen bestehen in DROOLY aus einem Schlüsselwort (merge, gefolgt von der Liste zu identifizierender Objekte, abgeschlossen durch eine ")":

|                  |  |
|------------------|--|
| <u>DINGO-II:</u> | <u>Cont1.diff is Gate1.diff is Gate2.diff</u>  |
|                  | <u>is Cont2.diff is Diff</u>                   |
| <u>DROOLY:</u>   | <u>(merge Cont1.diff Gate1.diff Gate2.diff</u> |
|                  | <u>Cont2.diff Diff )</u>                       |

Topologierelationen werden in DINGO-II als beliebig geklammerter Ausdruck mit qualifizierten Bezeichnern als Operanden und den Operatoren cross, overlap, enclose, in, touch und notouch in Infix-Schreibweise notiert. Durch die Wahl einer Präfix-Notation können die Klammern innerhalb solcher Ausdrücke in DROOLY entfallen. In der Tabelle auf Seite 36 sind die DROOLY-Entsprechungen der DINGO-II Operatoren für Topologie- und Design Rule Beschreibungen aufgeführt. Wie in DINGO-II, so können auch in DROOLY Topologie- und Design Rule Ausdrücke quantifiziert sein, falls Operanden von einem Mengentyp auftreten. Die Quantoren sind wie in DINGO-II all, some, any und one. Das Beispiel gibt einen komplexen Topologieausdruck in DINGO-II und seine Entsprechung in DROOLY an. Die Abbildung zeigt eine mögliche geometrische Ausprägung dazu. Dabei gilt

```
A = { a1, a2, a3 }
B = { b }
C = { c1, c2, c3 } :
```

DINGO-II:

```
some A: all C:
(C in B) overlap ( (A enclose A) in B )
```

DROOLY:

```
some A all C
overlap [ ]
indist [ ] C B
indist [ ]
enclose [ ] A A
B
```

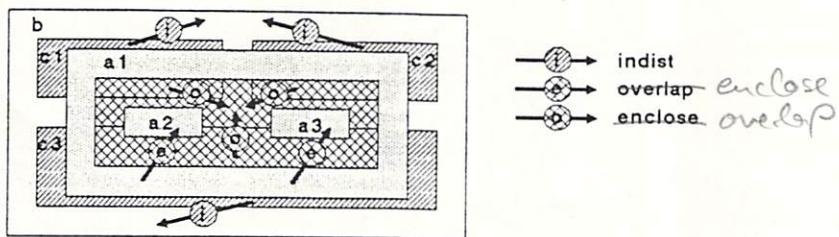


Abb. 6.1: Mögliche Ausprägung eines komplexen Topologieausdruckes

Der letzte Teil einer Objektbeschreibung in DINGO-II umfaßt die internen und externen Design Rules. Deren DINGO-II Syntax ist reichlich umfangreich und unterscheidet sich stark von der Syntax des Topologiebeschreibungsteils. Da aber die Informationen eines Design Rule Kopfes ähnlich zu denen eines Topologieausdruckes sind, wurde in DROOLY die Syntax von Design Rules und Topologiebeschreibungen zusammengefaßt. Haupthindernis dabei ist die teilweise unterschiedliche Bezeichnung der Operatoren. Durch die Zuordnung der Tabelle auf Seite 36 ist jedoch eine Vereinheitlichung möglich. Es bleibt die Syntax für einstellige Design Rules und für den Design Rule Rumpf; für beide Konstrukte gibt es im Topologieteil keine Entsprechung.

Einstellige Design Rules und Geometrieeinschränkungen werden in DROOLY zu *forwards* zusammengefaßt. Beide beschreiben eine Eigenschaft (*property*) eines einzelnen Objektes. Als Objekt wird dabei entweder implizit das den statischen Kontext bestimmende Objekt angenommen (der selbe Effekt wird durch Verwendung des Schlüsselwortes self als Bezeichner erreicht) oder es kann explizit durch einen (qualifizierten) Bezeichner spezifiziert sein. Drei einfache Beispiele für *forwards* in DROOLY sind:

```
Gate.poly width [ < 5 ]
Metal_line shape poly_45
self length [ or == 12 == 10 ]
```

Wertausdrücke bei PROPERTIES, RELATIONS und auch bei einer Kardinalitätsangabe werden in DROOLY grundsätzlich in eckige Klammern eingeschlossen. Wird bei RELATIONS oder Mengentypdeklarationen kein Wertausdruck benötigt, kann diese Klammerpaar auch leer sein (als Lexem müssen sie dann durch mindestens ein Trennzeichen voneinander getrennt sein!). Wertausdrücke sind entweder einfach oder bedingt. Einfache Wertausdrücke bestehen aus einem mit and und/oder or verknüpften booleschen Ausdruck in Präfix-Notation. Die Operanden eines solchen Ausdruckes sind (u.U. attributierte) Wertbestimmungen, bestehend jeweils aus Ordnungssymbol und Zahlenwert. Bei PROPERTIES vom Typ dimension darf auch eine Liste von Wertbestimmungen stehen, eingeleitet durch das Schlüsselwort (asymmetric und abgeschlossen durch eine ")". Als Attribute sind die aus DINGO-II übernommenen Schlüsselworte centered, eccentric, in current direction und against current direction zulässig.

Bedingte Wertausdrücke haben wie in DINGO-II eine Bedingung und zwei Zweige mit u.U. wieder bedingten Wertausdrücken. Die einzelnen Bestandteile werden durch die drei Schlüsselworte if, then und else voneinander getrennt. Zur Umgehung der automatischen Zuordnung eines else-Zweiges zum letzten if können die beiden Zweige optional in eckige Klammern eingeschlossen werden. Eine Bedingung kann ein boolescher Ausdruck mit and und/oder or als Operatoren sein.

Für die Operanden werden gewöhnliche forwards, RELATIONS oder ein spezielles Konstrukt, der CLASSTEST eingesetzt. Die beim Einsatz als Bedingung obligatorischen Wertausdrücke dienen hier nicht zur Formulierung einer Forderung, sondern bilden die Bedingung, nach der entweder der then- oder der else-Zweig ausgewählt werden. Ein CLASSTEST testet die Klasse, speziell die Belegung der Parameter eines Objektes. Die Syntax ist identisch mit einer Festtypdeklaration, nur das statt des Schlüsselwortes class hier classtest steht. Die folgende Abbildung zeigt einige Beispiele für Design Rules in DINGO-II zusammen mit ihren Entsprechungen in DROOLY:

DINGO-II:

```
dimension of Contact.cont_Cut:  
  == 500 or  
  assymmetric ( == 375, == 600, == 375, == 600 )
```

DROOLY:

```
Contact.cont_Cut dimension  
[ or == 500  
  (assymmetric == 375 == 600 == 375 == 600 )  
]
```

DINGO-II:

```
distance of P_A_Cont.cont_Cut from A_Wire:  
  >= 3 or  
  in current direction >= 2
```

DROOLY:

```
dist  
[ or >= 3  
  in current direction >= 2  
] P_A_Cont.cont_Cut A_Wire
```

DINGO-II:

```
if voltage of A_Wire: == Vdd  
then >= 3
```

DROOLY:

```
A_Wire width  
[ if A_Wire voltage [ == 12 ]  
  then >= 3  
]
```

DINGO-II:

```
distance of P_Wire from A_Wire:  
  if width of A_Wire range [5, 10] and  
    width of P_Wire > 10  
  then >= 5  
  else if width of A_Wire > 10  
    then >= 7  
  else >= 3
```

DROOLY:

```

dist
[ if and A_Wire width [ and >= 5 <= 10 ]
    P_Wire width [ > 10 ]
    then >= 5
    else if A_Wire width [ > 10 ]
        then >= 7
        else >= 3
] P_Wire A_Wire

```

## 7. Die Tabellen

Das Compiler-Backend erzeugt aus einer Technologiebeschreibung in der DROOLY-Zwischensprache zwei Tabellen, die Objekttabelle und die Wertausdruckstabelle. Diese Tabellen sind zusammen mit der vom Compiler-Frontend erzeugten Symboltabelle Bestandteil der Technologie-Datenhaltung des IMAGE-Systems. Über eine Funktionsbibliothek erlauben sie Applikationen den Zugriff auf alle Technologieinformationen, die zur technologieinvarianten Bearbeitung der ebenfalls gespeicherten Geomtrie- und Semantikdaten erforderlich sind. Die Aufspaltung in drei Tabellen begründet sich in der unterschiedlichen Struktur der gespeicherten Daten und in deren unterschiedlichen Einsatzbereichen innerhalb der Layoutapplikationen. Manche Applikationen, wie etwa Bauelementextraktoren, werden in der Hauptsache die Informationen der Objekttabelle verwenden, um semantische Einheiten im Layout zu erkennen. Erst bei der Extraktion operationaler Parameter muß auch auf Informationen der Wertausdruckstabelle zugegriffen werden, da diese Parameter im wesentlichen durch quantitative Abmessungen bestimmt werden.

Ein Layoutkompaktor wird dagegen zwar aufgrund der objektorientierten Speicherung der Technologiedaten auf die Hierarchie der deklarierten Unterstrukturen Bezug nehmen (in der Objekttabelle abgelegt), sich aber auf quantitative PROPERTIES und RELATIONS konzentrieren, um seine Aufgabe zu erfüllen. Die zur Verfügung gestellten Zugriffsoperationen erlauben eine solche Auswahl relevanter Daten aus den Tabellen.

Wesentliches Prinzip beim Aufbau der Tabellen ist die Lokalität. Daten, die sich ausschließlich auf ein Objekt beziehen, werden auch lokal bei diesem Objekt und dort wiederum möglichst tief in der Komponentenhierarchie gespeichert. In DINGO-II ist es durchaus möglich, innerhalb einer Objektdefinition Topologieanweisungen oder Design Rules anzugeben, deren Operanden Unterstrukturen ein und derselben Unterstruktur des Objektes sind. Beim Aufbau der Tabellen wird eine solche Regel dem Objekt zugeordnet, dessen Bezeichner sich als längster gemeinsamer Pfadpräfix der beiden Operanden ergibt. Dieses Prinzip erlaubt einerseits die Trennung relevanter von nicht relevanten Daten für eine bestimmte Objektdefinition. Sie erlaubt andererseits den gezielten Zugriff auf alle ein Objekt betreffenden Daten, ohne andere Objektdefinitionen nach verstreuten weiteren Informationen durchsuchen zu müssen.

## 7.1 Objekttabelle

Die Objekttabelle ist als Vektor von Objektstrukturen organisiert. Jedem Objekt ist in eindeutiger Index in diesem Vektor zugeordnet. Jede Objektstruktur bietet Platz für eine Anzahl unterschiedlicher Objekttypen, die im Folgenden

einzel mit ihren Datenfeldern vorgestellt werden. Zunächst gibt es in einer Objektstruktur einige Felder, die bei allen Objekttypen gleich genutzt werden:

**prefix** (6): Verweis auf das Aggregat, von dem das aktuelle Objekt eine Unterstruktur darstellt

**suffix:** Symboltabellenindex dieser Unterstruktur

Diese beiden Felder dienen zur Speicherung des qualifizierten Bezeichners eines Objektes. Es wird also nicht bei jedem Objekt der vollständige Pfad abgelegt, sondern nur der Symboltabellenindex der letzten Pfadkomponente und ein Verweis auf das den statischen Kontext bildende Objekt. Der vollständige Pfad kann durch Verfolgung der **prefix**-Verweise zurückgewonnen werden. Die Objekttabelle übernimmt durch die Aufnahme dieser beiden Felder eine Funktion, die in Compilern für konventionelle Sprachen eigentlich der Symboltabelle zukommt, nämlich die Verwaltung des statischen Kontextes eines Bezeichners. Die Informationen zur Hierarchie der Objekte ist jedoch ein wesentlicher Aspekt einer Technologiebeschreibung in DINGO-II. Durch die Verwaltung der Objekthierarchie innerhalb der Objekttabelle wird die Bearbeitung der Vererbungsmechanismen ohne umfangreiche Annahmen über die Struktur der Symboltabelle möglich.

**left, right:** Zum schnellen Zugriff auf Objekte über ihren qualifizierten Bezeichner wird über diese beiden Felder ein Binärbaum über die Objekttabelle gelegt. Ein INORDER-Durchlauf diesen Binärbaums liefert alle deklarierten Objekte und RELATIONS (welche für diesen Zweck ebenfalls einen Symboltabelleneintrag mit zugeordnetem Index erhalten), sortiert nach Präfix und Suffix ihres qualifizierten Bezeichners.

**user:** Dieses Feld hat keine festgelegte Semantik. Es ist als typloser Zeiger vereinbart und kann durch entsprechende CAST-Operationen auf benutzerdefinierte Datenstrukturen verweisen. Eine Objektstruktur erhält dadurch zusätzliche Flexibilität und kann von einer Applikation ihren Anforderungen gemäß erweitert werden.

**spec:** Dieses Feld unterscheidet die einzelnen Objekttypen, die in einer Objektstruktur gespeichert werden können.

Alle weiteren Felder der Objektstruktur können in einer Variante zusammengefaßt werden. Jedem Objekttyp entspricht dann eine Komponente dieser Variante, wobei der Objekttyp durch das **spec**-Feld bestimmt ist. Die möglichen Objekttypen sind LAYERDEF, CLASSDEF, STDDEF, PART, PARAM, SET, SHAPE, PROPERTY, C\_PROPERTY, RELATION und C\_RELATION. Wie man aber an der folgenden Beschreibung der für diese Objekttypen spezifischen Felder der Objektstruktur erkennt, besteht auch zwischen den einzelnen Objekttypen oft eine große Ähnlichkeit bei der Verwendung der einzelnen Felder. In der Implementierung wird daher die Objektstruktur auch für die typspezifischen Felder nicht als Variante realisiert, sondern alle Felder bilden eine große Verbundstruktur. In der folgenden Beschreibung der Besonderheiten der einzelnen Objekttypen und ihrer

(6) In den folgenden Kapiteln wird auf die verschiedenen Aspekte der Implementierung eingegangen. Dabei werden bestimmte Fonts für folgende Textobjekte benutzt:

#### Schlüsselworte

Variablen- und Feldbezeichner

Funktionsbezeichner

Nonterminalsymbole

Nachrichten-Selektoren

- fett, unterstrichen

- fett

- kursiv

- kursiv

- normal

spezifischen Felder wird allerdings nur auf die jeweils tatsächlich benutzten Felder (in den Graphiken jeweils unterlegt dargestellt) eingegangen. Alle unbenutzten Felder behalten ihren Initialisierungswert (NIL für Verweise in die Objekt- und Wertausdruckstabelle, NULL für Zeiger, 0 für Zähler, UNUSED für Operatoren).

| prefix              | suffix     |
|---------------------|------------|
| left                | right      |
| user                |            |
| spec                | LAYERDEF   |
| compHd              | next_comp  |
| next_same           |            |
| degree              |            |
| fparm_ix            |            |
| next_outermost      |            |
| properties[0..6]    |            |
| c_properties[0..12] |            |
| class               |            |
| class_cnt           | aparm_cnt  |
|                     | aparms     |
|                     | classes[0] |
| lmerges             | rmerges    |
| lmerge_cnt          | rmerge_cnt |
| relationHd          |            |
| next_rel1           | next_rel2  |
| expr                |            |
| quant               |            |
| op1                 | op2        |
| outermost           |            |

Abb. 7.1: Objektstruktur bei LAYERDEF

#### Objekttyp LAYERDEF

keine speziellen Felder

| prefix              | suffix     |
|---------------------|------------|
| left                | right      |
| user                |            |
| spec                | CLASSDEF   |
| compHd              | next_comp  |
| next_same           |            |
| degree              |            |
| fparm_ix            |            |
| next_outermost      |            |
| properties[0..6]    |            |
| c_properties[0..12] |            |
| class               |            |
| class_cnt           | aparm_cnt  |
|                     | aparms     |
|                     | classes[0] |
| lmerges             | rmerges    |
| lmerge_cnt          | rmerge_cnt |
| relationHd          |            |
| next_rel1           | next_rel2  |
| expr                |            |
| quant               |            |
| op1                 | op2        |
| outermost           |            |

Abb. 7.2: Objektstruktur bei CLASSDEF

**Objekttyp CLASSDEF**

**compHd:** Liste der Komponenten. Dieses Feld enthält einen Verweis auf die erste Komponente. Alle weiteren Komponenten sind über ihr **next\_comp**-Feld zu einer linearen Liste verkettet. Mit Hilfe dieser beiden Felder wird die Komponentenhierarchie auf einen Binärbaum abgebildet. Die "linker-Sohn"-Beziehung bindet die Unterstrukturen an ihre Aggregat, die "rechter-Sohn"-Beziehung verbindet Unterstrukturen auf gleichem Level miteinander.

**degree:** Anzahl der deklarierten Unterstrukturen.

**fparm\_ix:** Liste der parametertypdeklarierten Unterstrukturen. Die Reihenfolge der Unterstrukturen in dieser Liste entspricht ihrer Reihenfolge in der Liste formaler Parameter des Objektes.

**next\_outermost:** Dieses Feld verweist auf den Kopf einer Liste von RELATIONS. Diese RELATIONS sind allesamt Komponenten dieses Objektes und bilden jeweils die Wurzeln des Baums einer zusammengesetzten oder quantifizierten RELATION.

**properties:** Vektor von Verweisen auf die möglichen PROPERTIES. Jedem PROPERTY-Typ entspricht ein Vektorelement (SHAPE, WIDTH, AREA, DIMENSION, RADIUS, LENGTH, IN\_ANGLE)

**c\_properties:** Vektor von Verweisen auf die möglichen C\_PROPERTIES. Auch hier entspricht jedem C\_PROPERTY-Typ ein Vektor-Element (SHAPE, ..., IN\_ANGLE, EDGE\_LENGTH, VOLTAGE, CURRENT, POWER, RESISTANCE, CAPACITANCE)

| prefix              | suffix     |
|---------------------|------------|
| left                | right      |
| user                |            |
| spec                | STDDEF     |
| compHd              | next_comp  |
| next_same           |            |
| degree              |            |
| fparm_ix            |            |
| next_outermost      |            |
| properties[0..6]    |            |
| c_properties[0..12] |            |
| class               |            |
| class_cnt           |            |
|                     | aparm_cnt  |
|                     | aparms     |
|                     | classes[0] |
| lmerges             | rmerges    |
| lmerge_cnt          | rmerge_cnt |
|                     | relationHd |
| next_rel1           | next_rel2  |
|                     | expr       |
|                     | quant      |
| op1                 | op2        |
|                     | outermost  |

Abb. 7.3: Objektstruktur bei STDDEF

**Objekttyp STDDEF**

**classes:** Dieses Feld enthält einen Zeiger auf eine spezielle Struktur zur Speicherung der Klasse und evtl. aktueller Parameter. Die einzelnen Felder dieser Struktur sind:

- class:** Verweis auf die Klasse
  - aparm\_cnt:** Anzahl der aktuellen Parameter
  - aparms:** Vektor der aktuellen Parameter. Dieser Vektor speichert die im DROOLY-Format gegebenen Indizes für aktuelle Parameter.
- class\_cnt:** Anzahl der Klassen (immer gleich 1 bei STDDEF)
- compHd:** (wie bei CLASSDEF)
- degree:** Anzahl der deklarierten Unterstrukturen.
- next\_outermost:** (wie bei CLASSDEF)
- properties:** (wie bei CLASSDEF)
- c\_properties:** (wie bei CLASSDEF)

| prefix              | suffix         |
|---------------------|----------------|
| left                | right          |
|                     | user           |
| spec                | PART           |
| compHd              | next_comp      |
|                     | next_same      |
|                     | degree         |
|                     | fperm_ix       |
|                     | next_outermost |
| properties[0..6]    |                |
| c_properties[0..12] |                |
|                     | class          |
| class_cnt           | aparm_cnt      |
|                     | aparms         |
|                     | classes[0]     |
| lmerges             | rmerges        |
| lmerge_cnt          | rmerge_cnt     |
|                     | relationHd     |
| next_rel1           | next_rel2      |
|                     | expr           |
|                     | quant          |
| op1                 | op2            |
|                     | outermost      |

Abb. 7.4: Objektstruktur bei PART

**Objekttyp PART**

**classes:** (wie bei STDDEF)

**class\_cnt:** Anzahl der Klassen (immer gleich 1 bei PART)

**compHd:** Liste der Komponenten. Dieses Feld enthält einen Verweis auf die erste Komponente. Alle weiteren Komponenten sind über ihr **next\_comp**-Feld zu einer linearen Liste verkettet. Ein PART kann auf zwei Arten zu Komponenten kommen. Bei der ersten Möglichkeit erbt es sie von seiner Klasse. Der Unterstrukturbau der Klasse wird dazu kopiert (zum Verfahren der Vererbung siehe Abschnitt 8.6) und durch Austausch der Wurzel an das PART angehängt. Da auch RELATIONS und PROPERTIES als Komponenten eines Objektes aufgefaßt werden, werden auch FORWARDS oder RELATIONS mit nur einer Unterstruktur als Operand dieser Unterstruktur als neue Komponenten hinzugefügt. Diese Vorgehensweise entspricht dem Prinzip der Lokalität.

**next\_comp:** Dieses Feld dient der genannten Verkettung der Unterstrukturen eines Objektes zu einer linearen Liste.

**degree:** Anzahl der aus der Klasse geerbten deklarierten Unterstrukturen

**properties:** (wie bei CLASSDEF)

**c\_properties:** (wie bei CLASSDEF)

**lmerges, rmerges:** Diese Felder enthalten Listen auf alle linksseitigen bzw. rechtsseitigen MERGEs dieses PARTs. Diese entstehen aus DINGO-II Identifikationsanweisungen. Die Reihenfolge der Verkettung entspricht der Reihenfolge in der DINGO-II Identifikationsanweisung. Das PART muß allerdings nicht explizit in einer Identifikationsanweisung auftreten. Durch Identifikation zweier Objekte werden automatisch alle ihre Unterstrukturen identifiziert. Ist eine solche Identifikation aufgrund unterschiedlicher Struktur nicht möglich, wird die Identifikation zurückgewiesen (zum Verfahren der Identifikation siehe Abschnitt 8.8).

**lmerge\_cnt, rmerge\_cnt:** Diese Felder zählen die rechtseitigen bzw. linkseitigen merges.

**relationHd:** Dieses Feld enthält eine Liste von RELATIONS mit diesem PART als Operanden. Diese RELATIONS können, müssen aber nicht Komponenten des PARTs sein.

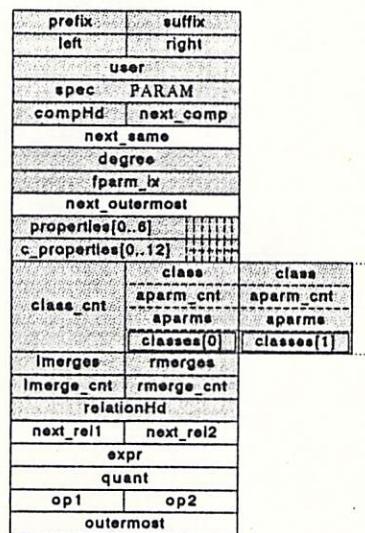


Abb. 7.5: Objektstruktur bei PARAM

**Objekttyp PARAM**

**classes:** Dieses Feld enthält einen Vektor von speziellen Strukturen zur Speicherung der alternativen Klassen und evtl. aktueller Parameter. Eine einzelne Elementstruktur dieses Vektors hat den bei STDDEF beschriebenen Aufbau.

**class\_cnt:** Anzahl der alternativen Klassen. Bei parameterdeklarierten Unterstrukturen ist die Klasse nicht eindeutig bestimmt, sondern wird durch eine Liste alternativer Klassen spezifiziert. Die einzelnen Alternativen werden im classes-Vektor gespeichert; class\_cnt enthält die Anzahl der Elemente in diesem Vektor.

**compHd:** Da bei einem PARAM die Klasse nicht eindeutig festliegt, kann der Unterstrukturbau nicht von einer Klasse an das PARAM vererbt werden. Bei den Unterstrukturen eines PARAMs handelt es sich daher ausschließlich um PROPERTIES und RELATIONS mit nur diesem PARAM als Operand.

**next\_comp:** (wie bei PART)

**degree:** Anzahl der deklarierten Unterstrukturen (immer gleich 0 bei PARAM)

**properties:** (wie bei CLASSDEF)

**c\_properties:** (wie bei CLASSDEF)

**fparm\_ix:** Verkettung der parametertypdeklarierten Unterstrukturen eines Objektes. Die Position des PARAMs in dieser Liste entspricht seiner Position in der Liste formaler Parameter des Objektes.

**relationHd:** Dieses Feld enthält eine Liste mit RELATIONS mit diesem PARAM als Operanden. Diese RELATIONS können, müssen aber nicht Komponenten des PARAMs sein.

| prefix              | suffix     |
|---------------------|------------|
| left                | right      |
| user                |            |
| spec                | SET        |
| compHd              | next_comp  |
| next_same           |            |
| degree              |            |
| fparm_ix            |            |
| next_outermost      |            |
| properties[0..6]    |            |
| c_properties[0..12] |            |
| class               |            |
| class_cnt           | aparm_cnt  |
|                     | aparme     |
|                     | classes[0] |
|                     | lmerges    |
| rmerges             |            |
| lmerge_cnt          | rmerge_cnt |
| relationHd          |            |
| next_rel1           | next_rel2  |
| expr                |            |
| quant               |            |
| op1                 | op2        |
| outermost           |            |

Abb. 7.6: Objektstruktur bei SET

## Objekttyp SET

**classes:** (wie bei STDDEF)  
**class\_cnt:** Anzahl der Klassen (immer gleich 1 bei SET)  
**compHd, next\_comp:** (wie bei PART)  
**degree:** (wie bei PART)  
**properties:** (wie bei CLASSDEF)  
**c\_properties:** (wie bei CLASSDEF)  
**lmerges, rmerges:** (wie bei PART)  
**lmerge\_cnt, rmerge\_cnt:** (wie bei PART)

**expr:** Dieses Feld enthält einen Index der Wertausdruckstabelle. In dem zugehörigen Wertausdruck wird die Vorgabe der Kardinalität gespeichert. Wurde keine solche Vorgabe gemacht, enthält dieses Feld den Index NIL.

**relationHd:** Dieses Feld enthält eine Liste von RELATIONS mit diesem SET als Operanden. Diese RELATIONS können, müssen aber nicht Komponenten des SETs sein.

|                     |            |
|---------------------|------------|
| prefix              | suffix     |
| left                | right      |
| user                |            |
| spec                | GEOCLASS   |
| compHd              | next_comp  |
| next_same           |            |
| degree              |            |
| fparm_ix            |            |
| next_outermost      |            |
| properties[0..6]    |            |
| c_properties[0..12] |            |
| class               |            |
| class_cnt           | aparm_cnt  |
|                     | aparms     |
|                     | classes[0] |
| lmerges             | rmerges    |
| lmerge_cnt          | rmerge_cnt |
|                     | relationHd |
| next_re1            | next_re12  |
|                     | expr       |
|                     | quant      |
| op1                 | op2        |
| outermost           |            |

Abb. 7.7a: Objektstruktur bei GEOCLASS

## Objekttyp GEOCLASS

Der spezielle Objekttyp GEOCLASS benötigt keine zusätzlichen Felder. Er wurde auch nur zur Vereinheitlichung der Strukturen in der Wertausdruckstabelle eingeführt. SHAPES verweisen nämlich mit ihrem expr-Feld in die Wertausdruckstabelle. Dort würde außer für diesen Spezialfall kein Feld vom Typ "Symboltabellen-Index" benötigt. Ein Eintrag in der Objekttabelle kann jedoch mit seinem suffix-Feld auf die Symboltabelle verweisen. Da Geometrieklassen in jedem Fall global definiert werden, ist das prefix-Feld einer GEOCLASS immer NIL.

| prefix              | suffix     |
|---------------------|------------|
| left                | right      |
| user                |            |
| spec                | SHAPE      |
| compHd              | next_comp  |
| next_same           |            |
| degree              |            |
| fparm_ix            |            |
| next_outermost      |            |
| properties[0..6]    |            |
| c_properties[0..12] |            |
| class               |            |
| class_cnt           | aparm_cnt  |
|                     | aparms     |
|                     | classes[0] |
| lmerges             | rmerges    |
| lmerge_cnt          | rmerge_cnt |
| relationHd          |            |
| next_rel1           | next_rel2  |
| expr                |            |
| quant               |            |
| op1                 | op2        |
| outermost           |            |

Abb. 7.7b: Objektstruktur bei SHAPE

**Objekttyp SHAPE**

**expr:** Dieses Feld enthält einen Index der Wertausdruckstabelle. Der dort gespeicherte Wertausdruck besteht im einfachsten Fall aus einem Verweis zurück in die Objekttabelle auf den Eintrag für die Geometrieklasse. Es ist aber auch ein durch **or** verknüpfter boolescher Ausdruck mit alternativen Geometrieklassen möglich. In der Wertausdruckstabelle können solche booleschen Ausdrücke gebildet werden.

**next\_comp:** (wie bei PART)

| prefix              | suffix       |
|---------------------|--------------|
| left                | right        |
| user                |              |
| spec                | (C_)PROPERTY |
| compHd              | next_comp    |
| next_same           |              |
| degree              |              |
| fparm_ix            |              |
| next_outermost      |              |
| properties[0..6]    |              |
| c_properties[0..12] |              |
| class               |              |
| class_cnt           | aparm_cnt    |
|                     | aparms       |
|                     | classes[0]   |
| lmerges             | rmerges      |
| lmerge_cnt          | rmerge_cnt   |
| relationHd          |              |
| next_rel1           | next_rel2    |
| expr                |              |
| quant               |              |
| op1                 | op2          |
| outermost           |              |

Abb. 7.8: Objektstruktur bei PROPERTY

### Objekttyp PROPERTY

**expr:** Dieses Feld enthält einen Index der Wertausdruckstabelle. Dort werden die quantitativen Angaben zu einem PROPERTY gespeichert.

**next\_comp:** (wie bei PART)

**next\_same:** Mit diesem Feld werden C\_PROPERTY- und C\_RELATION-Objekte an das zugehörige PROPERTY- bzw. RELATION-Objekt angehängt.

### Objekttyp C\_PROPERTY

**expr:** Dieses Feld enthält einen Index der Wertausdruckstabelle. Dort werden die quantitativen Angaben zu einem C\_PROPERTY gespeichert.

**next\_same:** (wie bei PROPERTY)

Ein C\_PROPERTY ist das Übersetzungsprodukt einer Bedingung in DROOLY, die eine PROPERTY eines Objektes abfragt. C\_PROPERTIES werden nicht in den Komponentenbaum eines Objektes eingehängt, sondern sind nur über ihren Wertausdruck zu erreichen.

| prefix              | suffix       |
|---------------------|--------------|
| left                | right        |
| user                |              |
| spec                | (C_)RELATION |
| compHd              | next_comp    |
| degree              |              |
| parm_ix             |              |
| next_outermost      |              |
| properties[0..6]    |              |
| c_properties[0..12] |              |
| class               |              |
| class_cnt           | aparms_cnt   |
|                     | aparms       |
|                     | classes[0]   |
| lmerges             | rmerges      |
| lmerge_cnt          | rmerge_cnt   |
|                     | relationHd   |
| next_relt           | next_relt2   |
|                     | expr         |
|                     | quant        |
| opt                 | op2          |
|                     | outermost    |

Abb. 7.9: Objektstruktur bei RELATION

**Objekttyp RELATION**

**quant:** Dieses Feld speichert den Operator der RELATION. Als solche sind die Quantoren all, some, any und one zugelassen ebenso wie die DROOLY-Operatoren indist, cross, overlap, touch und dist.

**op1, op2:** Diese Felder enthalten Verweise auf die beiden Operanden der RELATION. Die entsprechenden Objekte können vom Typ PART, PARAM, SET oder RELATION sein. In externen Design Rules können auch Objekte als Operanden auftreten, die nicht als Unterstruktur des aktuellen statischen Kontextes deklariert sind. In diesem Fall werden bei der Übersetzung in die Tabellen neue Unterstrukturen erzeugt und diese als Operanden der betreffenden RELATION eingetragen. Auf diese Weise lassen sich auch Klassen mit aktuellen Parametern in den normalen Rahmen von Unterstrukturen einfügen.

**expr:** Dieses Feld enthält den Verweis auf einen Wertausdruck für den quantitativen Anteil einer RELATION.

**outermost:** RELATIONS werden in der Objekttabelle immer als Tripel aus Operator und zwei Operanden abgelegt. Jedes solche Tripel erhält einen eigenen, automatisch generierten Bezeichner und wird in die Komponentenstruktur des lokalsten Objektes eingehängt (Lokalitätsprinzip). Um komplexe Ausdrücke speichern zu können, ist als Operand ein Objekt vom Typ RELATION erlaubt. Durch die so erzeugte Verweisstruktur wird in der Objekttabelle der Syntaxbaum des komplexen Topologieausdruckes aufgebaut. Von jedem Knoten vom Typ RELATION führt nun über das outermost-Feld ein Verweis zur Wurzel des Syntaxbaumes dieses komplexen Topologieausdruckes.

**next\_outermost:** Dieses Feld verkettet die zur Komponentenstruktur eines Objektes gehörenden Wurzeln der Syntaxbäume komplexer Topologieausdrücke.

**next\_relation1, next\_relation2:** Diese Felder verketten alle RELATIONS auf einem bestimmten PART, PARAM oder SET, und zwar getrennt für rechten und linken Operanden.

**Objekttyp C\_RELATION**

**expr:** Dieses Feld enthält einen Index der Wertausdruckstabelle. Dort werden die quantitativen Angaben zu einer C\_RELATION gespeichert.

Eine C\_RELATION ist das Übersetzungsprodukt einer Bedingung in DROOLY, die eine RELATION zweier Objekte abfragt. C\_RELATIONS werden wie C\_PROPERTIES nicht in den Komponentenbaum eines Objektes engehängt.

Zusammenfassend besteht die Objektstruktur aus folgenden Feldern:

|                     |  |
|---------------------|--|
| prefix              | Verweis auf Aggregat   |
| suffix              | Verweis auf Symboltafel  |
| left                | Sortierung nach prefix, suffix<br>---                            |
| right               |  |
| user                | Zeiger auf benutzerdefinierte Struktur                           |
| spec                | Auswahlfeld für Objekttyp  |
| classes             | Vektor der Klassen mit aktuellen Parametern                      |
| class_cnt           | Anzahl d. Elemente im classes-Vektor                             |
| fparm_ix            | Verkettung der PARAMs eines Objektes                             |
| compHd              | Liste der Komponenten (negativ, wenn INORDER-Fädelung)           |
| next_comp           | Verweis zur nächsten Komponente (negativ, wenn INORDER-Fädelung) |
| next_same           | Verkettung von PROPERTY bzw. RELATION mit zugehörigem C_-Objekt  |
| next_outermost      | Verkettung der Outermost-RELATIONS                               |
| degree              | Anzahl deklarierter Komponenten                                  |
| properties[0..5]    | Zeiger auf die PROPERTY-Objekte                                  |
| c_properties[0..11] | Zeiger auf die C_PROPERTY-Objekte                                |
| lmerges             | linksseitige Merges  |
| lmerge_cnt          | Anzahl der Links-Merges  |
| rmerges             | rechtsseitige Merges   |
| rmerge_cnt          | Anzahl der Rechts-Merges   |
| quant               | Operator bzw. Quantor von RELATIONS                              |
| op1                 | linker Operand von RELATIONS                                     |
| op2                 | rechter Operand von RELATIONS                                    |
| expr                | Verweis in die Wertausdruckstabelle                              |
| relationHd          | Liste der RELATIONS mit diesem Objekt                            |
| next_relation1      | Verkettung der RELATIONS für op1                                 |
| next_relation2      | Verkettung der RELATIONS für op2                                 |
| outermost           | Zeiger auf die Wurzel eines Relationenbaums                      |

## 7.2 Tabelle der Wertausdrücke

Wie die Objekttafel ist auch die Ausdruckstafel als ein Vektor von Strukturen definiert. Eine Struktur zur Speicherung eines "atomaren" Wertausdruckes besteht aus acht Feldern:

**spec:** Dieses Feld legt den Typ des Atomarausdruckes fest. Mögliche Typen sind VALUE, OR, AND, COND\_OR, COND\_AND, C\_PROPERTY und ASYMMETRIC.

**cond:** Dieses Feld enthält alternativ einen Verweis auf einen Atomarausdruck oder einen Verweis in die Objekttafel. Seine Hauptanwendung ist die Speicherung des Verweises auf die Bedingung in einem bedingten Wertausdruck.

**left, right:** Boolesche Ausdrücke werden aus den dyadischen Operatoren or und and bzw. cond or und cond and aufgebaut. Diese beiden Felder verweisen jeweils auf den linken bzw. rechten Operanden eines booleschen Ausdruckes. In bedingten Ausdrücken speichert das left-Feld den Verweis auf den Ausdruck im then-Zweig, das right-Feld speichert den Verweis auf den Ausdruck im else-Zweig.

**attrib:** Dieses Feld speichert ein Attribut. Erlaubte Attributwerte sind CENTERED, ECCENTRIC, IN\_CURRENT\_DIRECTION und AGAINST\_CURRENT\_DIRECTION.

**minval, eqval, maxval:** Diese drei Felder dienen zur Speicherung eines Ordnungszeichens zusammen mit einem numerischen Wert. Dabei wird folgendermaßen zugeordnet ("n" ist der numerische Wert):

| DROOLY   | min | eq | max |
|----------|-----|----|-----|
| > n      | n   |    |     |
| $\geq$ n | n   | n  |     |
| $=$ n    |     | n  |     |
| $\leq$ n |     | n  | n   |
| < n      |     | n  |     |

Die nicht belegten Felder werden durch eine 0 markiert. Auf diese Weise ist es allerdings nicht möglich, einen numerischen Wert von 0 darzustellen.

## 8. Der DROOLY-Compiler

Das eigentliche Backend des DINGO-II Compilers [Krä 89] gliedert sich in neun Untermodule, die in den Abschnitten 8.2 bis 8.10 beschrieben werden. Der Präprozessor simuliert während der Entwicklung das Frontend des DINGO-II Compilers.

### 8.1 Präprozessor

Der Präprozessor ist ein eigenständiges Programm und erfüllt während der Entwicklung des DROOLY-Compilers zwei Aufgaben:

- Er ermöglicht eine mnemonische Eingabe der DROOLY-Lexeme und die Eingabe von symbolischen Bezeichnern.
- Er wandelt die mnemonische Eingabe um in das normale DROOLY-Format und eine externe Symboltabelle.

Der Präprozessors simuliert in seinen Ausgaben ein DINGO-II Compiler-Frontend und ermöglicht so eine unabhängige Entwicklung des Compiler-Backends.

#### Umgebung

Der Aufruf des Präprozessors erfolgt von der Betriebssystemebene mit folgender Syntax:

```
pre <dateiname>.r
```

Der Dateiname trägt eine Erweiterung ".r" (für "readable"). Die Ausgabe des Präprozessors besteht aus der DROOLY-Textdatei mit der Erweiterung ".cod" und der externen Symboltabelle mit der Erweiterung ".sym". Konnte der Präprozessor seine Arbeit erfolgreich beenden, liefert er den Exit-Code 0. Bei einer Fehler-situation (Fehler bei Dateizugriff, Überlauf der internen Tabellen oder Syntaxfehler in der Eingabe) wird mit dem Exit-Code -1 abgebrochen.

## Struktur

Der Präprozessor besteht aus einem kurzen Hauptprogramm *main*, einer Scanner- und Ausgaberoutine *preproc* mit ihren Hilfsroutinen *mystrtok* zum Parsen eines qualifizierten Bezeichners und *preskip\_blank* zum Überlesen von Leerzeichen und Kommentaren und einigen Routinen zur Implementierung einer einfachen Symboltabellenverwaltung (*wsymtb* zum Schreiben der Symboltabelle auf eine Datei; *prelookup* zum Nachschlagen eines Bezeichners in der Symboltabelle; *s\_new* zum Initialisieren eines Symboltabelleneintrages; *new\_string* zum Allozieren von Speicherplatz für einen Bezeichner). Das eigentliche Präprozessormodul wird mit dem Token-Modul des Backends zusammengebunden, um konsistente Kodierungen der DROOLY-Schlüsselworte zu gewährleisten. Aus diesem Modul werden die Funktionen *gtokencode* zur Umwandlung von lesbare Eingabe in einen internen Code und *gtokenstr* zur Umwandlung dieses Codes in die externe DROOLY-Darstellung benutzt. Selbst bei einer völligen Umstellung der lexikalischen Struktur der DROOLY-Zwischensprache beschränken sich Änderungen auf die Tabellen und Zugriffsoperationen im Token-Modul.

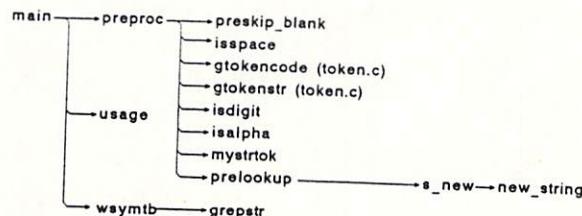


Abb. 8.1: Die Struktur des Präprozessors

## Verfahren

Im Hauptprogramm werden die Ein- und Ausgabedateien geöffnet und anschließend die Routine *preproc* aufgerufen, die das Scannen der Eingabedatei, die Ausgabe der DROOLY-Textdatei und den Aufbau der Symboltabelle steuert.

Die Symboltabelle ist als Vektor von Elementen mit folgender Struktur implementiert:

|              |                           |
|--------------|---------------------------|
| <b>name</b>  | Zeiger auf den Bezeichner |
| <b>left</b>  | linker INORDER-Nachbar    |
| <b>right</b> | rechter INORDER-Nachbar   |

Die Verzeigerung über die Felder *left* und *right* legt einen binären Suchbaum über den Symboltabellenvektor, um die Suche nach Bezeichnern zu beschleunigen. Ein neuer Bezeichner wird von der Funktion *prelookup* in die Symboltabelle eingefügt. Diese sucht mithilfe des Suchbaums den Platz für den gewünschten Bezeichner in der Symboltabelle. Ist der Bezeichner noch nicht eingetragen, wird mit der Routine *s\_new* ein neuer Platz im Symboltabellenvektor angefordert und der neue Bezeichner dort eingetragen. Läuft bei dieser Anforderung die Symboltabelle über, ist zur Zeit keine Reallozierung vorgesehen. Der Präprozessor bricht mit einer Fehlermeldung ab und signalisiert sein Scheitern durch den Exit-Code -1. Trat kein Fehler auf, gibt *prelookup* den Index des alten oder neuen Platzes in der Symboltabelle zurück.

Die Funktion *preproc* liest in einer Endlosschleife den gesamten Eingabetext. Bei jedem Schleifendurchlauf werden zunächst Trennzeichen und Kommentare durch die Funktion *preskip\_blank* überlesen. Wurde dabei das Ende der Eingabedatei noch nicht erreicht, werden nun Zeichen für ein Schlüsselwort, einen Bezeichner oder einen numerischen Wert in einem Zwischenspeicher gesammelt. Die Bedingung, daß einzelne Lexeme in der Eingabe durch mindestens ein Leerzeichen getrennt sein müssen, vereinfacht hier die Abbruchbedingung. Das so gesammelte Lexem wird mit den Funktionen *gtokencode* und *gtokenstr* aus dem Token-Modul in die DROOLY-Darstellung umkodiert. War eine solche Umkodierung nicht möglich, wird durch Überprüfung des ersten Zeichens festgestellt, ob es sich um einen numerischen Wert handelt. Dieser wird mit einem "#" versehen in die DROOLY-Textdatei ausgegeben.

Ist das erste Zeichen ein Buchstabe, "-" oder "\_", wird ein "@" in die DROOLY-Textdatei ausgegeben und es beginnt die Analyse eines u.U. qualifizierten Bezeichners. Die durch "." getrennten Komponenten eines solchen Bezeichners werden durch die Funktion *mystrtok* einzeln abgetrennt und in der Symboltabelle nachgeschlagen. Der gefundene bzw. neue Symboltabellenindex wird jeweils in die DROOLY-Textdatei ausgegeben.

Das beschriebene Verfahren erkennt sicher längst nicht alle illegalen Eingaben in der lesbaren Datei. Da es sich bei dem Präprozessor jedoch nur um eine Test- und Entwicklungsumgebung handelt, wurde auf eine weitere Verfeinerung verzichtet.

Nachdem das Ende der Eingabedatei erreicht ist, wird die Symboltabelle in die externe Datei geschrieben und der Präprozessor beendet seine Arbeit mit dem Exit-Code 0.

## 8.2 Hilfsmodule für Fehlermeldungen, Dekodierung und Überprüfung der Codeabdeckung

Diese drei Module fassen bestimmte Aufgaben zusammen, die für eine leichte Änderbarkeit des Codes in eigenen Modulen untergebracht sind.

### Fehlermeldungen im Modul error.c

Fehlermeldungen werden im Compiler Backend von einer zentralen Funktion *error* aus dem gleichnamigen Modul ausgegeben. Diese Routine akzeptiert eine Fehlernummer und eine variable Anzahl von Parametern. Aus der Fehlernummer wird die Fehlerklasse und der Rumpf der Fehlermeldung mithilfe einer Tabelle ermittelt. Es werden drei Fehlerklassen unterschieden:

- PANIC: Die Übersetzung muß aufgrund von Inkonsistenzen oder Überläufen in internen Tabellen abgebrochen werden.
- ERROR: Die Übersetzung wird aufgrund einer fehlerhaften Benutzereingabe abgebrochen.
- WARNING: Die Übersetzung wird trotz einer fehlerhaften Benutzereingabe weitergeführt. Die erzeugten Tabellen entsprechen aber nicht notwendigerweise der Intention des Benutzers.

Eine vollständige Liste der Fehlermeldungen und der zugehörigen Fehlerklassen findet sich im Anhang B. Die zweite Funktion im *error*-Modul wird nur aufgerufen, falls die Compilation abgebrochen oder regulär beendet werden soll. Sie erhält als Parameter den Exit-Code des Compiler-Backends. Vor Beendigung des Programmlaufs

werden alle Trace-Dateien geschrieben und anschließend alle offenen Dateien geschlossen. Zuletzt wird die *exit*-Funktion mit dem übergebenen Parameter aufgerufen.

#### Dekodierung im Modul token.c

Alle symbolischen Konstanten mit einer lesbaren Entsprechung ebenso wie die externen Darstellungen von mnemonischen und normalen DROOLY-Schlüsselworten zusammen mit ihrer internen Kodierung wird in den Tabellen des Token-Modules zusammengefaßt. Dieses Modul stellt darüberhinaus die Zugriffsoperationen auf diese Tabellen zur Verfügung. Im einzelnen haben die Funktionen folgende Aufgaben:

*gtokenindex* liefert zu einem als Zeichenkette vorliegendem Schlüsselwort (mnemonisch oder normal über einen Ausrufparameter einstellbar) den Index in der jeweiligen Tokentabelle. Zur Unterstützung der binären Suche sind die Tokentabellen alphabetisch sortiert.

*gtokencode* liefert zu einem als Zeichenkette vorliegendem Schlüsselwort (mnemonisch oder normal über einen Ausrufparameter einstellbar) den vom Parser vergebenen Code durch Suchen in der jeweiligen Tokentabelle mithilfe von *gtokenindex*.

*gtokenstr* wird nur zur Unterstützung des Präprozessors benötigt. Diese Funktion liefert zu einem Token-Code die externe DROOLY-Darstellung durch lineare Suche in der entsprechenden Tokentabelle.

*gspecstr* wird hauptsächlich zur Erzeugung lesbbarer Trace-Ausgaben benutzt. Alle symbolischen Konstanten sind in einer speziellen Tabelle zusammen mit ihrer lesbaren Darstellung abgelegt. Die tatsächlichen numerischen Kodierungen der einzelnen symbolischen Konstanten wird aus dieser Tabelle durch ein Awk-Programm erzeugt und in einer Header-Datei abgelegt. Durch Inkludieren dieser Header-Datei zur Compile-Zeit sind die numerischen Kodierungen in der Tabelle automatisch in aufsteigender Reihenfolge sortiert, so daß nach einer Kodierung binär gesucht werden kann. *Gspecstr* liefert die gefundene Zeichenkette als Funktionsergebnis zurück.

```
gtokencode--gtokenindex
gtokenstr
gspestr
```

Abb. 8.2: Die Struktur des token-Moduls

#### Überprüfung der Codeabdeckung im Modul teval.c

Dieses Modul unterstützt die Instrumentierung des Compiler-Backends mit Aufrufen zur Protokollierung der Codeabdeckung bei der Abarbeitung einer Testbibliothek. Protolliert werden soll jeweils der Eintritt und das Verlassen einer Funktion und der Eintritt in einzelne Codeblöcke.

## Umgebung

Zur Ausrüstung des Codes wird ein Satz spezieller Makros definiert, der jeweils die Protokollierung initiiert:

|  |   |
|--|---|
| <code>FKT( &lt;name&gt;, &lt;blockanzahl&gt; );</code> | Eintritt in eine Funktion                               |
| <code>B( &lt;blocknummer&gt; )</code>                  | Eintritt in einen Block                                 |
| <code>END;</code>                                      | Verlassen einer Prozedur                                |
| <code>RETURN( &lt;rückgabewert&gt; );</code>           | Verlassen einer Funktion                                |
| <code>POP;</code>                                      | Verlassen einer semantischen Einheit ohne <u>return</u> |

Die Protokollierungsmakros werden durch die Compileroption -DTEVAL bei der Übersetzung aktiviert.

Das Teval-Modul sammelt die gewonnenen Informationen während des Ablaufs des Compilers in einer Tabelle, welche dann nach abgeschlossener Übersetzung in eine Protokolldatei geschrieben wird. Diese Datei enthält für jede instrumentierte Funktion eine Zeile mit dem Funktionsnamen in der ersten Spalte. Die zweite Spalte gibt die Anzahl der Eintritte in die Funktion, alle weiteren Spalten geben für jeden instrumentierten Block dieser Funktion die Anzahl der Eintritte an. Nicht betretende Funktionen und Blöcke erhalten eine 0 in diesen Spalten.

Die Teval-Tabelle `tv_tab` wird zur Compilezeit mithilfe eines Awk-Skriptes angelegt und zum Compiler-Backend hinzugebunden. Sie ist als Vektor mit Elementen von folgender Struktur organisiert:

**left, right:** Zum schnellen Zugriff auf die gespeicherten Funktionsbezeichner wird durch die Initialisierungsroutine `tv_init` vor Beginn der Protokollierung ein binärer Suchbaum über die Teval-Tabelle gelegt. Die Felder `left` und `right` verweisen jeweils auf den linken bzw. rechten Inorder-Nachbarn.

**name:** Dieses Feld wird zur Compilezeit mit dem Bezeichner einer Funktion initialisiert und dient als Schlüssel für den Zugriff auf die Tabelle.

**fkt\_cnt:** Dieses Feld wird mit 0 initialisiert. Bei jedem Eintritt in die zugehörige Funktion wird es um 1 inkrementiert.

**block\_ctns:** Dieser Vektor von Zählern für die einzelnen instrumentierten Blöcke der zugehörigen Funktion wird vor Beginn der Protokollierung von `tv_init` alloziert und mit Nullen initialisiert. Auf die einzelnen Zähler wird über die Blocknummer des zu protokollierenden Blockes zugegriffen. Bei jedem Eintritt in einen instrumentierten Block wird der zugehörige Zähler um 1 inkrementiert.

**nr\_blocks:** Dieses Feld wird zur Compilezeit mit der Anzahl instrumentierter Blöcke pro Funktion initialisiert. Die gespeicherte Zahl wird zur Allozierung des `block_ctns`-Vektors benötigt.

## Struktur

Das Teval-Modul enthält die Routinen `tv_init` zur Initialisierung der Tevaltabelle zu Beginn der Protokollierung und `tv_write` zum Schreiben der Tabelle in eine Datei nach Abschluß der Protokollierung. Die Protokollierung wird durch die Funktionen `tv_fkt`, `tv_block` und `tv_pop` durchgeführt. Diese werden von einigen lokalen Funktionen unterstützt. Dabei sucht `tv_lookup` einen Funktionsbezeichner

in der Teval-Tabelle, `tv_push` rettet einen Funktionsindex auf den Aufrufstack und `new_int_list` alloziert Speicherplatz für die Blockzähler-Vektoren.

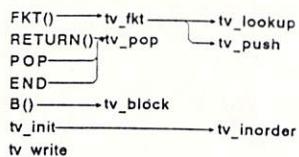


Abb. 8.3: Die Struktur des teval-Moduls

### Verfahren

Der Compilerlauf des Backends verläuft aus der Sicht des Teval-Moduls folgendermaßen: Ganz zu Beginn wird die Routine `tv_init` aufgerufen, um die Informationen in der Teval-Tabelle zu vervollständigen. Dabei werden für jede Zeile in der Tabelle die `block_cnts`-Vektoren angelegt, mit Nullen initialisiert und die Zeile nach ihrem Funktionsbezeichner in den binären Suchbaum eingesortiert.

Während des gesamten Compilerlaufs werden Aufrufe instrumentierter Funktionen über das FKT-Makro an die Funktion `tv_fkt` gemeldet. Diese sucht den übergebenen Funktionsbezeichner in der Teval-Tabelle und merkt sich dessen Index für die folgenden Blockprotokollierungen. Der alte Funktionsindex wird jedoch zuvor auf einen Stack gesichert. Der Funktionseintritt wird durch Inkrementieren des zugehörigen Zählers vermerkt.

Der Eintritt in einen Block der aktuellen Funktion wird über das B-Makro an die Funktion `tv_block` gemeldet. Diese überprüft zunächst, ob der übergebene Blockindex im legalen Bereich für die aktuelle Funktion liegt. Tritt dabei ein Fehler auf, wird der Benutzer gewarnt und nichts protokolliert, der Programmablauf aber fortgesetzt. Ist alles korrekt, wird der zum Block gehörende Zähler um 1 inkrementiert.

Beim Verlassen einer Funktion muß der Tabellenindex der rufenden Funktion wieder vom Aufrufstack geholt werden. Das erledigt die Funktion `tv_pop`, die von einem der Makros `END`, `RETURN` oder `POP` gerufen wird.

Wird die Kontrolle nach Abarbeitung der DROOLY-Datei wieder an das Hauptmodul zurückgegeben, ruft dieses die Routine `tv_write`, welche den Inhalt der Teval-Tabelle in eine Datei mit der Erweiterung ".tv" schreibt.

### 8.3 Hauptmodul main.c

#### Umgebung

Ein legaler Aufruf des Compiler-Backends hat eines der beiden folgenden Formate:

```
drooly [options] <techfile>.r
oder   drooly [options] <techfile>.cod
-v      Erweiterte Semantiküberprüfung
(siehe Abschnitt 8.9)
```

Die folgenden Optionen dienen nur zum Einschalten verschiedener Trace-Ausgaben in der Entwicklungsphase und werden durch Angabe der Compiler-Option `-DTRACE` aktiviert:

```
-x      Trace der Aktivitäten des Token-Moduls
-s      Trace der Scanner-Eingabe
-t<n>  Allgemeiner Trace des Tabellenaufbaus
        mit Detailiertheit <n> (n in [0..9])
```

Sämtliche im Compiler-Backend verwandten Dateien und deren Dateinamen werden im Hauptmodul deklariert, geöffnet und geschlossen. Jede Datei hat eine charakteristische Erweiterung, die nach Löschung der Erweiterung ".r" oder ".cod" an den als Kommandozeilenparameter übergebenen Dateinamen angehängt wird:

|       |               |   |
|-------|---------------|---|
| INOUT | <tecfile>.sym | Symboltabelle   |
| IN    | <tecfile>.cod | DROOLY-Textdatei  |
| IN    | <tecfile>.r   | DROOLY-Textdatei für den Präprozessor   |
| OUT   | <tecfile>.t   | Lesbare Form der erzeugten Tabellen   |
| OUT   | <tecfile>.tec | Erzeugte Tabellen   |
| OUT   | <tecfile>.trf | Testreferenz: Tabellen in einer<br>besonders zum Vergleich in einer<br>Testbibliothek geeigneten Form |
| OUT   | <tecfile>.tv  | Teval-Protokoll   |

#### Struktur

Das Hauptmodul enthält die Funktion `main` und die Hilfsfunktion `usage` zur Ausgabe einer Benutzermeldung bei fehlerhaftem Programmaufruf.

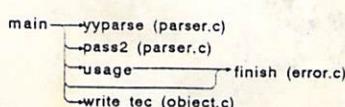


Abb. 8.4: Die Struktur des Hauptmoduls

## Verfahren

Main initialisiert zu allererst die Teval-Tabelle durch einen Aufruf von tv\_init. Anschließend wird die Aufrufzeile auf Optionen und Dateinamen untersucht. Bei Verwendung eines Eingabedateinamens mit der Erweiterung ".r" wird zunächst der Präprozessor mit dieser Datei gerufen. Trat dabei kein Fehler auf, werden die verschiedenen Dateien geöffnet, die Symboltabelle eingelesen und anschließend der DROOLY-Parser gerufen. Konnte der Parser seine Arbeit erfolgreich beenden, wird zur Bearbeitung von merges die Funktion pass2 gerufen. Zuletzt werden die erzeugten Tabellen mithilfe der Funktion write\_tec in eine Datei geschrieben und durch einen Aufruf von finish alle offenen Dateien geschrieben und geschlossen.

### 8.4 Scanner scanner.c

Die einfache lexikalische Struktur von DROOLY erlaubt eine leichte Kodierung des Scanners ohne Verwendung des Scannergenerators Lex. Von Lex generierte Scanner werden im Vergleich zu einem handkodierten Scanner schnell recht umfangreich. So hat der hier beschriebene Scanner nur einen Umfang von vier DIN-A4-Seiten. Die für eine einfache Implementierung des DROOLY-Scanners gemachten Annahmen über die lexikalische Struktur von DROOLY wären allerdings für eine nicht maschinengenerierte Sprache wie etwa DINGO-II nicht akzeptabel. Eine wichtige Voraussetzung für den Scanner in seiner jetzigen Form ist, daß Anfang und Ende eines einzelnen Lexems ohne weiteres zu erkennen sind. In DROOLY müssen daher Schlüsselworte, Bezeichner und numerische Werte durch mindestens ein Leerzeichen voneinander getrennt sein. Darüber hinaus ist die DROOLY-Eingabe aber völlig formatfrei.

## Umgebung

Die einzige exportierte Funktion des Scanner-Moduls ist yylex. Sie wird ohne Parameter vom Parser aufgerufen und liefert als Rückgabewert entweder den Tokencode eines Schlüsselwortes, den Tokencode für IDENTIFIER für einen (qualifizierten) Bezeichner oder den Tokencode für INTEGER für einen numerischen Wert. Darüberhinaus legt sie das gelesene Lexem in der globalen Stringvariable yytext ab. Fester Bestandteil des Kommunikationsprotokolls mit dem vom Parsergenerator Nemo erzeugten Parser ist außerdem die Übergabe des Tokenwertes in der globalen Variable yylval. Ihr Typ ist ebenso wie der Typ des Parser-Wertestacks als Variantentyp YYSTYPE mit folgenden Alternativen in der Headerdatei typedef.h definiert:

#### OBJ:

Für Objekte wird ihr Objekttabellenindex und ein Flag für globale Pfadbezeichner (durch einen "." eingeleitet) in yylval gespeichert.

#### NUM:

Numerische Werte werden direkt in yylval abgelegt.

#### EXPR:

Diese Alternative dient zur Speicherung eines Ausdruckstabellenindizes und wird im Scanner nicht benutzt.

Ebenfalls zur Kommunikation mit dem Parser werden noch folgende globale Variablen im Scanner benutzt:

yyin (IN) ist der File-Deskriptor der DROOLY-Eingabedatei.

`yylineno` (OUT) hält die aktuelle Zeilennummer in der DROOLY-Eingabedatei zur Ausgabe in Fehlermeldungen.

`act_class` (IN) hält den Objekttabellenindex desjenigen Objektes, welches den aktuellen statischen Kontext in der DROOLY-Eingabedatei definiert.

## Struktur

Das Scanner-Modul wird von der Scanner-Routine `yylex` beherrscht. Daneben enthält es noch die beiden Hilfsroutinen `skip_blank` zum Überlesen von Leerzeichen und Kommentaren und `mystrtok` zum Parsen von (qualifizierten) Bezeichnern.



Abb. 8.5: Die Struktur des Scanners

## Verfahren

Die Funktion `yylex` wird vom Parser zum Einlesen des nächsten Lexems aufgerufen. Dazu werden zunächst Leerzeichen und Kommentare durch die Funktion `skip_blank` überlesen. Wurde dabei das Ende der Eingabedatei noch nicht erreicht, werden nun Zeichen für ein Schlüsselwort, einen Bezeichner oder einen numerischen Wert in der Variablen `yytext` gesammelt. Die Bedingung, daß einzelne Lexeme in der Eingabe durch mindestens ein Leerzeichen getrennt sein müssen, vereinfacht hier die Abbruchbedingung. Die Variable `yytext` ist als statischer Zeichenvektor von 80 Zeichen Länge deklariert, so daß es prinzipiell bei sehr langen Pfaden in qualifizierten Bezeichnern zu einem Überlauf kommen könnte. Dabei ist nicht die Länge der einzelnen Pfadkomponenten wichtig (Nur ihre Symboltabellenindizes stehen in der DROOLY-Datei), sondern in der Hauptsache die Anzahl der Pfadkomponenten.

Für das so gesammelte Lexem wird mit der Funktion `gtokencode` aus dem Token-Modul der Tokencode ermittelt. `Gtokencode` liefert 0, wenn es das Lexem nicht gefunden hat. Konnte der Code jedoch ermittelt werden, handelt es sich um eine positive ganze Zahl. Der mögliche Wertebereich wird durch den Code für ein Pseudotoken `BUILTIN` in zwei Bereiche unterteilt. Codes größer als der Code von `BUILTIN` werden als Ergebnis sofort an den Parser zurückgegeben. Die Token aus dem unteren Bereich umfassen die Operatoren für `PROPERTIES` und `C_PROPERTIES`. Sie werden im weiteren Verlauf wie Bezeichner von Objekten ("Pseudoobjekte") behandelt und ihr Tokencode zunächst in einer Variablen `builtin` gespeichert.

Für das spezielle Token `SELF` wird in `yytext` ein Bezeichner generiert, der nur aus einem "@" besteht. Daraus erzeugt `yylex` im weiteren Verlauf eine Referenz auf den durch `act_class` indizierten Objekteintrag. `Builtin` wird auf den Code für `IDENTIFIER` gesetzt.

Als nächstes werden numerische Konstanten an ihrem einleitenden "#" erkannt. Die folgende Ziffernfolge wird in einen numerischen Wert umgewandelt und dieser in der Kommunikationsvariable `yyval.NUM` gespeichert. Der Tokencode für `INTEGER` ist in diesem Fall das an den Parser zurückgelieferte Ergebnis.

Ist das erste Zeichen des Lexems dagegen ein "@" oder ist die Variable `builtin` mit dem Bezeichner eines Pseudoobjektes besetzt, muß der vorliegende Objektbezeichner analysiert werden. Die Objekttabelle übernimmt dabei mit ihrer Funktion `lookup` die Rolle einer Symboltabelle, die Funktion `mystrtok` dient als Parser des Pfades. Die Analyse beginnt mit dem auf den "@" folgenden Zeichen. Handelt es sich um einen ".", liegt ein globaler Pfadbezeichner vor, der statische Kontext wird ignoriert. In Normalfall wird der Objekttabellenindex des statischen Kontextes jedoch als Präfix des analysierten Pfades aufgefaßt. Die Information, daß ein globaler Bezeichner vorliegt, wird im Parser noch benötigt und wird ihm durch Setzen des Flags `yylval.OBJ.is_global` in der Kommunikationsvariablen mitgeteilt.

Bezeichner von Pseudoobjekten bestehen aus dem durch den statischen Kontext bestimmten Präfix und dem negativen Tokencode als Suffix. Zur Erzeugung eines Objekttabelleneintrages wird die Funktion `lookup` mit diesen beiden Pfadkomponenten aufgerufen. Ihr Ergebnis, der alte oder neue Objektindex, wird in der Kommunikationsvariablen `yylval.OBJ.index` gespeichert.

Für normale Bezeichner jedoch muß `lookup` u.U. mehrmals aufgerufen werden, um auch für die Zwischenstufen eines Pfades die Objektindizes zu bestimmen. Die erste Pfadkomponente wird mit `mystrtok` aus `yytext` extrahiert. Ist sowohl sie als auch der Präfix leer, liegt ein Fehler vor und der Scanner bricht mit einer Fehlermeldung ab. Ist nur die erste Pfadkomponente leer, handelt es sich bei dem Bezeichner um eine Referenz (mit `self`) auf den statischen Kontext. Dessen Objektindex wird also in `yylval.OBJ.index` eingetragen.

Der Rest des Pfades wird anschließend in einer Schleife komponentenweise mit `mystrtok` aus `yytext` extrahiert, jeweils als Zwischenstufe mit `lookup` in der Objekttabelle gesucht bzw. neu eingerichtet und als Präfix für den nächsten Schleifendurchlauf verwandt, solange, bis der Pfad in all seine Komponenten zerlegt ist. Der letzte Aufruf von `lookup` liefert dabei den Objektindex des Gesamtpfades. Dieser wird in `yylval.OBJ.index` eingetragen.

Als Rückgabewert liefert der Scanner bei Bezeichnern von Pseudoobjekten den jeweiligen Tokencode, bei `self` und normalen Objektbezeichnern den Code des Tokens IDENTIFIER.

## 8.5 Parser parser.c

Der im Compiler-Backend benutzte Parser zur syntaktischen Analyse einer DROOLY-Datei wird aus einer LALR(1)-Grammatik mit an den Ableitungsregeln hängenden semantischen Aktionen vom Parsergenerator NEMO erzeugt. Der eigentliche generierte Parser besteht aus einer C-Funktion `yparse`, die durch Anwendung der Ableitungsregeln versucht, ihre Eingabe zum Startsymbol `intermediate` zu reduzieren. Im Zuge dieser Reduktion wird nach jeder angewandten Ableitungsregel die zugehörige semantische Aktion ausgeführt. Die semantischen Aktionen rufen geeignete Hilfsfunktionen zum Aufbau der Objekt- und der Wertausdruckstabelle auf. Sie kommunizieren miteinander über den Wertestack des Parsers, dessen Elemente vom gleichen Typ YYSTYPE wie die globale Variable `yylval` sind. Zur Handhabung des Wertestacks stehen in den Semantikroutinen Pseudovariablen zur Verfügung, die bei der Parsergenerierung in Zugriffe auf den Wertestack übersetzt werden.

Der linken Seite einer Regel kann über die Pseudovariable `$$` ein Wert zugewiesen werden. Tritt das Nonterminalsymbol der linken Seite in der rechten Seite einer zweiten Regel auf, kann auf diesen Wert ebenfalls über eine Pseudovariable zugegriffen werden. Dazu sind alle Symbole und semantischen

Aktionen der rechten Seite einer Regel durchnummiertert. `$1` ist dabei die Pseudovariable des am weitesten links stehenden Symbols. Der Wert der Pseudovariablen eines Terminalsymbols ist derjenige Wert, den der Scanner beim Erkennen dieses Terminalsymbols an die Variable `yylval` zugewiesen hat. Der Wert der Pseudovariablen eines Nonterminalsymbols ist der Wert, der `$$` bei seiner Reduktion zugewiesen wurde. Werden für unterschiedliche Regeln unterschiedliche Alternativen der als Varianten deklarierten Pseudovariablen verwandt, kann es durch inkonsistente Benutzung der Alternativen zu Typkonflikten kommen. Sowohl Terminalsymbole als auch Nichtterminalsymbole werden darum gemäß den Alternativen in `YYSTYPE` als von einem bestimmten Typ deklariert. Nemo kann so die konsistente Benutzung der Alternativen überwachen und gibt entsprechende Fehlermeldungen aus.

Der von NEMO erzeugte Parser simuliert einen endlichen Automaten mit einem Stack. Diesem Automaten stehen prinzipiell vier unterschiedliche Aktionen zur Verfügung.

- **ACCEPT:** Der Parser ist dazu ausgelegt, ein spezielles Symbol, das Startsymbol (in DROOLY das Nonterminalsymbol *intermediate*) zu erkennen. Das Ende der Eingabe ist im Fall der DROOLY-Eingabe das EOF-Zeichen. Falls die gelesenen Token bis ausschließlich des EOF-Zeichens eine Struktur bilden, die auf daß Start-Symbol paßt, kehrt die Parser-Funktion zu ihrem Aufrufer zurück, sie akzeptiert die Eingabe.
- **ERROR:** Falls das aktuelle Eingabe-Token zusammen mit dem nächsten Eingabe-Token keine legale Eingabe bildet, kann der Parser nicht mehr gemäß seiner Spezifikation fortfahren. Er meldet einen Fehler und versucht, an einer späteren Stelle in der Eingabe wieder aufzusetzen.
- **SHIFT:** Dies ist die häufigste der Parser-Aktionen. Basierend auf dem nächsten Eingabe-Token wird der aktuelle Zustand auf den Stack geschoben und ein neuer Zustand angesprungen.
- **REDUCE:** Diese Aktion verhindert, daß der Stack über alle Grenzen anwächst. Wenn der Parser die rechte Seite einer Grammatik-Regel gesehen hat, holt er die Anzahl Zustände vom Stack, die den Symbolen der rechten Seite der Grammatik-Regel entsprechen. Mit dem nun freigelegten Zustand und dem Symbol der linken Regelseite wird ein neuer Zustand angesprungen [YACC].

NEMO faßt einige dieser Aktionen zusammen, was in diesem Zusammenhang nicht berücksichtigt zu werden braucht. Bei der Ableitung nach den Regeln der Eingabegrammatik kann es zu zwei Konflikttypen kommen:

- **SHIFT/REDUCE:** YACCs wie NEMOs Strategie ist in diesem Fall das SHIFTen.
- **REDUCE/REDUCE:** YACCs wie NEMOs Strategie ist in diesem Fall, nach der früheren Regel in der Eingabegrammatik abzuleiten.

Die DROOLY-Grammatik enthält einen SHIFT/REDUCE-Konflikt. Dieser wird von der eingebauten Konfliktlösungsstrategie NEMOs derart aufgelöst, daß zunächst geSHIFTet wird. Diese Strategie führt im Zusammenhang mit den verursachenden Regeln genau zu dem gewünschten Verhalten, daß nämlich ein else-Zweig dem letzten if in der Eingabe zugeordnet wird.

## Umgebung

Die Parserroutine `yyparse` wird direkt von `main` ohne Parameter aufgerufen. Bei korrekter Syntax (die Eingabe konnte zum Startsymbol `intermediate` reduziert werden) liefert `yyparse` 0 zurück, sonst -1. `Yyparse` wird von verschiedenen Modulen unterstützt. Grundlegend ist die Zusammenarbeit mit dem Scanner `yylex`, der auf Anforderung ein neues Token aus der DROOLY-Eingabe liefert. Als zentrale Funktion des Compiler-Backends ruft `yyparse` darüber hinaus folgende Routinen aus anderen Modulen:

`link_object` ( `object.c` ) zum Einbinden von neuen Objekten in eine Komponentenhierarchie

`gen_pseudo` ( `object.c` ) zur Erzeugung eines Objektbezeichners für eine RELATION

`msg` ( `object.c` ) für den Aufbau der Objektabelle. Dabei werden die Nachrichtenschlüssel `m_PSPEC`, `m_GSPEC`, `m_GFIRSTCOMP`, `m_GNEXTCOMP`, `m_INSTPART`, `m_INCDEGREE`, `m_PAPARM`, `m_PFPARM`, `m_PEXPR`, `m_PQUANT`, `m_POPS`, `m_PNEXTOUTER`, `m_ADDREL`, `m_GPREFIX`, und `m_GSUFFIX` benutzt.

`is_DECL`, `is_RELATION`, `is_CLASSDEF` ( `object.c` ) zur Überprüfung des `spec`-Feldes eines Objekteintrages

`max_comm_path` ( `object.c` ) zur Ermittlung des längsten gemeinsamen Pfadpräfixes zweier Objektbezeichner

`put_outermost` ( `object.c` ) zur Einrichtung eines Verweises auf die Wurzel eines Relationenbaums

`merge` ( `compare.c` ) zur Initiierung eines Vergleiches zweier Objektstrukturen und Einrichtung von merges

`incr_relat`, `decr_relat` ( `object.c` ) zur Markierung der Wurzel eines Relationenbaums

`incr_expr`, `decr_expr` ( `expr.c` ) zur Markierung der äußersten Ebene eines Wertausdruckes

`gen_expr` ( `expr.c` ) zur Erzeugung und Initialisierung eines neuen atomaren Wertausdruckes

`emsg` ( `expr.c` ) für den Aufbau der Wertausdruckstabelle. Dabei werden die Nachrichtenschlüssel `m_ASSMETRIC`, `m_COND_AND`, `m_COND_OR`, `m_C_PROPERTY`, `m_AND`, `m_OR`, `m_PATTRIB` und `m_VALUE` benutzt.

`error` ( `error.c` ) zur Augabe von Fehlermeldungen und u.U. anschließendem Abbruch der Übersetzung

## Struktur

Das Parser-Modul exportiert als einzige Funktion `yyparse`. Die NEMO-Eingabe zur Erzeugung des Parser-Moduls besteht grob betrachtet aus drei Teilen. Der erste Teil ist reiner C-Programmtext. Hier werden die nötigen Header-Dateien inkludiert und folgende globale Variable zur Kommunikation zwischen den semantischen Aktionen deklariert:

`act_class` ( exportiert nach `scanner.c` ) enthält den Objektindex des statischen Kontextes und wird im Scanner benutzt, um einen dazu relativen Pfad in einen absoluten Pfad umzuwandeln.

`tmp1_class` dient zur Zwischenspeicherung des Wertes von `act_class`. Dies ist an drei Stellen im Parser notwendig, um dem Scanner temporär einen anderen statischen Kontext vorzutäuschen:

- Bei der Ableitung zu `forward` soll das erzeugte PROPERTY-Objekt als Komponente an das bezeichnete Objekt angehängt werden und nicht an die umgebende Objektklasse. Für die Dauer der Ableitung zu `property_type` wird daher `act_class` auf den Index des bezeichneten Objektes gesetzt.
- Innerhalb einer Ableitung zu `property` kann es notwendig sein, `act_class` ein zweites Mal umzusetzen. Handelt es sich nämlich bei der PROPERTY um eine Geometrieeinschränkung (`shape is <Geometrieklasse>`), so soll die Geometrieklasse nicht als Komponente an das bezeichnete Objekt angehängt werden, sondern soll global bleiben. Um diesen Effekt zu erzwingen, wird `act_class` für das Scannen des Geometrieklassenbezeichners auf NIL gesetzt, so daß der Scanner für die Geometrieklasse einen globalen Objekteintrag anlegt.
- Die Klassenbezeichner hinter dem Schlüsselwort `(class` werden implizit als global angenommen. Um dem Scanner diese Sonderbehandlung mitzuteilen, wird ebenfalls für das Scannen des Klassenbezeichners `act_class` auf NIL gesetzt.

`act_object` dient während einer Objektdefinition (Definition von STDDEFS, PARTs, PARAMs, SETs, (C\_)RELATIONS, (C\_)PROPERTIES und CLASSTESTs) zur Speicherung und Übermittlung des Objektindizes zwischen semantischen Aktionen auch über verschiedene Hierarchiestufen von Ableitungen hinweg.

`expr_depth` wird mit 0 initialisiert. Zu Beginn der Ableitung eines Wertausdruckes wird `expr_depth` um 1 inkrementiert, nach erfolgter Ableitung wieder um 1 dekrementiert. Dadurch zeigt ein Wert größer als 0 an, daß gerade ein Wertausdruck abgeleitet wird. Diese Information ist für die Ableitung von `c_properties` und `c_relations` als Bedingungen in Wertausdrücken wichtig, da diese sich syntaktisch nicht von `properties` und `relations` außerhalb von Wertausdrücken unterscheiden.

`relat_depth` wird mit 0 initialisiert. Zu Beginn der Ableitung eines Operanden einer `relation` wird `relat_depth` um 1 inkrementiert, nach erfolgter Ableitung wieder um 1 dekrementiert. Dadurch zeigt ein Wert von 0 die Wurzel eines Relationenbaums an und die nötigen Aktionen können ausgeführt werden.

`global` dient zur Übermittlung des Wertes von `yylval.OBJ.is_global` in die semantischen Aktionen bei der Ableitung zu `property` innerhalb der Ableitung zu `forward`. Dort wird danach entschieden, ob ein C\_PROPERTY den Wert von `main_class` als Klassenindex zugeordnet bekommt. Dieser wiederum wird in der Funktion `copy_expr` bei der Vererbung eines Wertausdrückes bei der Erzeugung neuer Referenzen auf C\_PROPERTIES benutzt.

`yytext` ( exportiert nach `scanner.c`, `main.c` ) dient zur internen Kommunikation des gelesenen Lexems zwischen Scanner und Parser. Im Hauptmodul wird der für `yylex` allozierte Speicherplatz als Puffer zum Aufbau der Funktionsnamen mißbraucht.

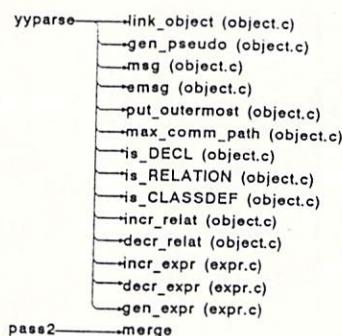


Abb. 8.6: Die Struktur des Parsers

## Verfahren

Die Aufgabe des Parsers besteht im Einrichten der durch die DROOLY-Eingabe spezifizierten Objekte in der Objekttabelle und anhängender Ausdrücke in der Ausdruckstabelle. Daneben leitet der Parser auch die Mechanismen zur Vererbung von Unterstrukturen und die Identifikation von Objekten ein.

Da die einzige Kontextinformation für den Scanner im Wert der globalen Variable `act_class` liegt, generiert dieser für jeden gelesenen Bezeichner mithilfe von `lookup` einen Objekteintrag in der Objekttabelle. Alle typspezifischen Felder bleiben darin unbelegt, da der Scanner den Objekttyp ja nicht kennt. Erst der Parser kann aus seinem Wissen über den Kontext eines Objektbezeichners den Objekttyp bestimmen und die typspezifischen Felder mit Werten füllen. Die einzelnen Objekttypen werden im folgenden einzeln betrachtet und die Zuweisungen an die Felder des Objekteintrages beschrieben.

### Objekttyp LAYERDEF

Bei diesem Objekttyp muß lediglich das `spec`-Feld auf LAYERDEF gesetzt werden.

### Objekttyp CLASSDEF

Bei diesem Objekttyp wird zunächst das `spec`-Feld auf CLASSDEF gesetzt. Der globalen Variable `act_class` wird der Objektindex des Klassenobjektes zugewiesen, um dem Scanner den aktuellen statischen Kontext anzugeben. Hat die Klasse formale Parameter, werden diese in der Ableitung zu `fparm_list` in der Reihenfolge ihres Auftretens in der Parameterliste einzeln erfaßt. Der Zähler `fparm_ix` in der Klassenobjektstruktur wird für jeden formalen Parameter um 1 inkrementiert. Gleichzeitig erhält der gleichnamige Zähler in der Objektstruktur des formalen Parameters den neuen Wert als seinen Positionsindex in der Liste formaler Parameter zugewiesen. Dies erleichtert später einen direkten Zugriff auf den zugehörigen aktuellen Parameter. Bei der Abarbeitung der Liste formaler Parameter legt der Scanner schon jetzt für jeden formalen Parameter einen Objekteintrag an. Die Zuweisung der typspezifischen Werte (ausgenommen `fparm_ix`) erfolgt aber erst bei der Abarbeitung der Deklaration des Objektes als PARAM.

Es folgt der Deklarationsteil für Unterstrukturen. An dieser Stelle sollen nur die für das Klassenobjekt relevanten Punkte der Ableitung zu *declarations* betrachtet werden. Jede vorgenommene Deklaration einer Unterstruktur erhöht den Grad des Klassenobjektes (im Feld *degree* seiner Struktur) um eins. Der Scanner konstruiert für jede deklarierte Unterstruktur einen Bezeichner aus dem Bezeichner der Klasse und dem Bezeichner der Unterstruktur und legt ein Objekt mit diesem Bezeichner in der Objekttabelle an. Erst im Parser jedoch wird dieses Objekt durch einen Aufruf von *link\_object* in die Komponentenstruktur des aktuellen Klassenobjektes eingehängt. Dabei wird die erste Unterstruktur an das *compHd*-Feld des Klassenobjektes angehängt; jede weitere Unterstruktur wird an das *next\_comp*-Feld der zuletzt deklarierten Unterstruktur angehängt. Nach der Deklaration der letzten Unterstruktur werden die PARTs und SETs instanziert, d.h. die Unterstrukturbäume ihrer Klassen werden kopiert und auch an sie angehängt. Das Klassenobjekt merkt davon jedoch nichts, da sich alle Änderungen auf die Komponentenbäume seiner Unterstrukturen beziehen.

Nachdem alle Unterstrukturen deklariert sind, können PROPERTIES, RELATIONS und merges auf den Unterstrukturen des Klassenobjektes definiert werden. Wurden keine Unterstrukturen deklariert, können nur PROPERTIES des Klassenobjektes festgelegt werden. Für eine PROPERTY wird schon im Scanner ein eigener Objekteintrag angelegt, falls ein solcher noch nicht existierte. Der Bezeichner eines PROPERTY-Objektes besitzt als Pfadpräfix den Index des bezeichneten Objektes, als Suffix den jeweiligen PROPERTY-Type. Zur Unterscheidung von Symboltabellenindizes wird der Suffix jedoch negativ gespeichert. Handelt es sich um eine PROPERTY mit einem neuen PROPERTY-Typ, wird das PROPERTY-Objekt mithilfe eines *link\_object*-Aufrufes im Parser in die Komponentenstruktur des durch seinen Pfadpräfix bezeichneten Objektes eingehängt. Dieses Objekt besitzt in seiner Struktur einen Vektor *properties* mit 5 Komponenten, der zu jedem PROPERTY-Type den Objektindex des zugehörigen PROPERTY-Objektes enthält. Für ein neues PROPERTY-Objekt wird dessen Index in diesem Vektor abgelegt, um später ohne Suche darauf zugreifen zu können.

Ist bereits ein Eintrag für einen bestimmten PROPERTY-Typ zu einem Objekt erfolgt, wird kein neuer Eintrag in der Objekttabelle vorgenommen. Stattdessen wird der Wertausdruck für die bisherige PROPERTY mit einem and-Operator mit den Wertausdruck der neuen PROPERTY vernüpft. Der resultierende Wertausdruck überschreibt den Ausdruck im alten PROPERTY-Objekt.

Für eine zweistellige RELATION wird ebenfalls ein eigener Objekteintrag angelegt. Dies kann jedoch erst in Parser selbst geschehen. Um RELATIONS so lokal wie möglich an ihre Operanden zu binden, wird dazu durch Aufruf der Funktion *max\_comm\_path* der längste gemeinsame Pfadpräfix beider Operanden der RELATION ermittelt und als Präfix des Relationsbezeichners benutzt. Für den Suffix wird in der Funktion *gen\_pseudo* eine global eindeutige, negative Kodierung für jede RELATION erzeugt. Mit dem so gewonnenen Präfix und Suffix wird mithilfe von *lookup* ein neues RELATION-Objekt erzeugt und von *link\_objekt* in die Komponentenstruktur seines Klassenobjektes eingehängt.

RELATIONS können nicht nur als einfache zweistellige RELATION auftreten, sondern auch als hierarchischer Relationenbaum. Als Operanden einer RELATION sind neben deklarierten Unterstrukturen also wieder RELATIONS zulässig. Der Parser erkennt, ob er gerade die Wurzel eines Relationenbaums bearbeitet, am Wert einer globalen Variable *relat\_depth*, die mit 0 initialisiert ist, zu Beginn der Ableitung einer zweistelligen RELATION um 1 inkrementiert und am Ende ihrer Ableitung wieder dekrementiert wird. Ein Wert von 1 zeigt damit dem Parser an, daß er die Wurzel eines Relationenbaums bearbeitet. Um von einem Objekt leicht zu den Wurzeln dieser Bäume in der Komponentenstruktur des Objektes zu gelangen,

werden die Wurzelobjekte untereinander und mit ihrem Klassenobjekt über die `next_outermost`-Felder der Objektstruktur verkettet.

*Merges* dürfen erst bearbeitet werden, wenn die vollständige Hierarchie mit allen deklarierten Unterstrukturen und allen `RELATIONS` der betroffenen Objekte aufgebaut ist. Die Objektindizes der zu identifizierenden Objekte werden daher in einem Temporärfile gesammelt und erst nach volliger Abarbeitung der DROOLY-Datei wieder eingelesen, um die Identifikationen zu erzeugen (siehe Abschnitt 8.8). Werden zwei (Klassen-) Objekte identifiziert und ist die Identifikation erfolgreich verlaufen, erhält das linke Objekt einen neuen Eintrag (den Objektindex des zweiten Objektes) in seinem `rmerges`-Vektor und der zugehörige `rmerge_cnt`-Zähler wird um 1 inkrementiert. Entsprechend erhält das rechte Objekt einen neuen Eintrag in seinem `lmerges`-Vektor und sein `lmerge_cnt`-Zähler wird um 1 inkrementiert.

#### Objekttyp STDDEF

Standardobjektdeklarationen unterscheiden sich in DROOLY syntaktisch nur durch ein anderes Schlüsselwort und durch die zusätzliche Angabe einer Objektzuweisung in Form eines Klassenaufrufes von den "normalen" Objekten. Semantisch entspricht die Deklaration eines Standardobjektes der Bildung einer Unterklasse in objektorientierten Sprachkonzepten. Jedes Standardobjekt besitzt in DROOLY eine eindeutig bestimmte Objektzuweisung ("Oberklasse"), multiple inheritance im Sinne objektorientierter Konzepte ist nicht möglich. Innerhalb der Objektstruktur wird die Klasse in einem Vektor `classes` gespeichert. Da die Klasse eines Standardobjektes eindeutig bestimmt ist, enthält dieser Vektor nur ein Element (der zugeordnete Zähler `class_cnt` erhält den Wert 1). Alle übrigen Felder der Objektstruktur werden wie beim Objekttyp CLASSDEF benutzt.

Ein Standardobjekt erbt die Komponentenstruktur seiner Oberklasse. Trägt diese formale Parameter, so muß das Standardobjekt die Klasse jeder parameterdeklarierten Unterstruktur seiner Oberklasse durch Angabe je eines zugehörigen aktuellen Parameters eindeutig festlegen. Ein Standardobjekt kann keine neuen Unterstrukturen deklarieren, wohl kann es aber neue PROPERTIES und `RELATIONS` auf den ererbten Unterstrukturen definieren. Dafür gilt sinngemäß das beim Objekttyp CLASSDEF Gesagte.

#### Objekttyp PART

Objekte vom Typ PART sind festtypdeklarierte Unterstrukturen eines Klassenobjektes. Die Einrichtung eines entsprechenden Eintrags in der Objekttabelle kann drei verschiedene Ursachen haben:

- Deklaration in DROOLY. Eine Unterstruktur vom Typ PART wird im DROOLY-Programmtext deklariert. Dabei wird seine Klasse, falls erforderlich durch Angabe von aktuellen Parametern eindeutig festgelegt. Durch eine solche Deklaration wird eine Instanz der Klasse erzeugt. Das PART erbt die gesamte Komponentenstruktur seiner Klasse.

- **Einfache Vererbung.** Wird ein Objekt vom Typ PART oder SET deklariert, erhält es eine eindeutige Klassenangabe. Das Objekt erbt bei seiner Deklaration die gesamte Komponentenstruktur seiner Klasse, also insbesondere alle PARTs und SETs (PARAMs werden bei der Vererbung immer in PARTs umgewandelt; siehe nächster Punkt). Die Vererbung wird realisiert durch ein rekursives Kopieren aller Objekte, die über die `compHd-` und `next_comp`-Verkettungen am Klassenobjekt hängen. Die neu erzeugten Objekte erhalten jeweils als Pfadpräfix den Pfad des PARTs zugewiesen.
- **Umwandlung eines PARAMS bei der Vererbung.** Die Klasse einer parametertyp-deklarierten Unterstruktur (Objekttyp PARAM, siehe Abschnitt 8.6) wird durch Angabe aktueller Parameter aus ihrer Alternativliste ausgewählt und dadurch eindeutig bestimmt. Dies kann entweder durch Instanziierung bei der Deklaration eines Parts oder durch Unterklassenbildung bei der Deklaration eines Standardobjektes geschehen. Der Objekttyp des PARAMs wird in PART umgewandelt. Die Klasse des Objektes ist eindeutig bestimmt, es erbt damit die gesamte Komponentenstruktur dieser Klasse.

Egal auf welche dieser Arten das Objekt enstanden ist, seinem `spec`-Feld ist PART zugewiesen. Im `classes`-Vektor wird wie bei STDDEFS die einzige Klasse des Objektes eingetragen, der zugehörige Zähler `class_cnt` hat den Wert 1. Durch einen Aufruf von `link_object` wird das PART in die Komponentenstruktur seines Aggregates eingehängt und das `next_comp`-Feld entsprechend gesetzt. Die Unterstruktur seiner Klasse erbt das PART erst am Ende des Deklarationsteils seines Aggregates. Durch diese Verzögerung können CLASSTEST-Bedingungen aufgelöst werden, die die Klasse anderer Unterstrukturen desselben PARTs abfragen (siehe Abschnitt 8.7). Dort erbt jeder Sohn des Aggregates die Komponentenstruktur seiner Klasse durch Aufruf von `m_INSTPART` (siehe Abschnitt 8.6).

### Objekttyp PARAM

Der Wert des `spec`-Feldes für Objekte vom Typ PARAM ist PARAM. Dieser Objekttyp kann nur durch direkte Parametertypdeklaration in DROOLY erzeugt werden. Sein besonderes Kennzeichen ist, daß der Klassenvektor `classes` nicht eine einzige Klasse enthält, sondern mehrere Alternativen. Der Zähler `class_cnt` zeigt die exakte Anzahl gespeicherter Alternativen an. Jede Klasse belegt eine Vektorkomponente. In deren Struktur ist jeweils der Objekttabellenindex des Klassenobjektes, u.U. zusammen mit den erforderlichen aktuellen Parametern abgelegt. Die Anzahl der aktuellen Parameter muß dabei gleich sein mit der Anzahl formaler Parameter der Klasse. Ein Zähler `aparm_cnt` gibt für jede Komponente des Klassenvektors die Anzahl aktueller Parameter an. Die aktuellen Parameter sind als Index in die Alternativklassen-Liste der parametertypdeklarierten Komponente angegeben, die zum korrespondierenden formalen Parameter des Klassenobjektes gehört.

Ein PARAM erbt bei seiner Deklaration keine Unterstrukturen, da seine Klasse nicht eindeutig bestimmt ist. Es können aber natürlich PROPERTIES dieses PARAMs definiert sein oder RELATIONS, die dieses PARAM als Operand enthalten. Die PROPERTIES werden in jedem Fall als Komponenten über die `compHd-/next_comp`-Verkettung an das PARAM angehängt und in seinem `properties`-Vektor referenziert. Es werden dagegen nur solche RELATIONS als Komponenten an das PARAM angehängt, deren beide Operanden gleich dem PARAM sind. Andere RELATIONS werden in die Komponentenstruktur des Aggregates eingehängt.

### Objekttyp SET

Objekte vom Typ SET sind mengentypdeklarierte Unterstrukturen eines Klassenobjektes. Die Einrichtung eines entsprechenden Eintrages in der Objekttabelle kann wie bei einem PART drei verschiedene Ursachen haben:

- Deklaration in DROOLY. Eine Unterstruktur vom Typ SET wird im DROOLY-Programmtext deklariert. Auch bei einem SET wird dabei die Klasse falls erforderlich durch Angabe von aktuellen Parametern eindeutig festgelegt. Das SET erbt die gesamte Komponentenstruktur seiner Klasse. Dabei werden aber alle PARTs in SETs umgewandelt. Die Klassen von PARAMs müssen durch aktuelle Parameter eindeutig bestimmt werden. Bei der Vererbung werden auch PARAMs in SETs umgewandelt.
- Einfache Vererbung. Wird ein Objekt vom Typ PART deklariert, erhält es durch Vererbung die gesamte Komponentenstruktur seiner Klasse, also insbesondere SETs.
- Umwandlung eines PARAMS bei der Vererbung. Die Klasse einer parametertyp-deklarierten Unterstruktur (Objekttyp PARAM, s.u.) wird durch Angabe aktueller Parameter aus ihrer Alternativliste ausgewählt und dadurch eindeutig bestimmt. Durch Instanziierung bei der Deklaration eines SETs werden die ehemaligen PARAMs in SETs umgewandelt.

Bei der Deklaration vom SET-Objekten können neben der Klassenangabe auch Einschränkungen bezüglich der Mengenkardinalität gemacht werden. Die Angabe erfolgt in DROOLY in Form eines Wertausdruckes. In der Tabelle für Wertausdrücke werden hierfür die entsprechenden Einträge gemacht und vom SET-Objekt auf die Wurzel des Ausdrucksbaums mit dem `expr`-Feld verwiesen. Erbt ein SET-Objekt ein SET als Komponente von seiner Klasse, werden die Kardinalitätsausdrücke beider Mengen durch einen and-Operator vernüpft. Die Komponente erhält den resultierenden Kardinalitätsausdruck zugewiesen. SET-Objekten, die durch Vererbung aus PARTs oder PARAMs entstanden sind, wird der Kardinalitätsausdruck ihres neuen Aggregates zugewiesen.

### Objekttyp GEOCLASS

Durch die Einrichtung eines eigenen Objekteintrages mit `spec`-Wert GEOCLASS muß in der Wertausdruckstabelle kein neuer Modus bei der Vernüpfung von Geometrieklassen mit einem or-Operator eingeführt werden. Der Verweis auf eine Geometrieklasse wird wie bei einem Verweis auf eine PROPERTY oder RELATION in Bedingungen (s.u.) einfach durch den entsprechenden Objekttabellenindex realisiert. Objekteinträge vom Typ GEOCLASS werden vom Scanner beim Einlesen einer shape-Anweisung erzeugt. Sie werden allerdings nicht in die Komponentenstruktur irgend eines Objektes eingehängt. Das `prefix`-Feld einer GEOCLASS ist immer NIL.

### Objekttyp SHAPE, Objekttyp PROPERTY

Die Struktur von Objekteinträgen für diese Typen ist völlig identisch. Neben dem Wert des `spec`-Feldes (SHAPE bzw. PROPERTY) enthalten beide in ihrem `expr`-Feld einen Verweis in die Ausdruckstabelle. Beide werden mit einem Aufruf von `link_objekt` in die Komponentenstruktur ihres Aggregates eingehängt und zusätzlich durch die entsprechende Komponente des `property`-Vektors referenziert. Aus dem `suffix`-Feld eines PROPERTIES geht direkt sein PROPERTY-Typ hervor, da der

Scanner die entsprechende Tokenkodierung negativ als Wert des **suffix**-Feldes einsetzt.

#### Objekttyp C\_PROPERTY

C\_PROPERTY-Objekte (Wert des **spec**-Feldes: C\_PROPERTY) dienen zur Speicherung einer Bedingung über eine PROPERTY eines Objektes. Sie werden wie PROPERTIES in die Komponentenstruktur dieses Objektes eingehängt. Aus Kompatibilität zu Objekten vom Typ CLASSTEST erhalten sie den aktuellen statischen Kontext (das durch die globale Variable **act\_class** referenzierte Objekt) als Klasse in ihrem **classes**-Vektor eingetragen.

#### Objekttyp RELATION, Objekttyp SET\_RELATION

Komplexe Topologieausdrücke aus DROOLY werden bei der Übersetzung in die Tabellen in atomare Ausdrücke zerlegt.

Ein Teilausdruck vom Typ RELATION beschreibt unmittelbar ein mögliche Relation von Layoutobjekten. Ihm ist im allgemeinen eine Flächeninterpretation im Layout zugeordnet. Davon ausgenommen sind Teilausdrücke mit dem Operator **touch**, denen im Layout nur die Berührungsstrecke (oder die Berührungspunkte) der beteiligten Layoutinstanzen zugeordnet sind sowie Ausdrücke mit dem Operator **out angle**, die überhaupt keine Topologieeigenschaft beschreiben. Ein Teilausdruck vom Typ SET\_RELATION hat nur im Zusammenwirken mit seinen abhängigen RELATION-Objekten eine Flächeninterpretation im Layout. Aufgrund ihrer syntaktischen Ähnlichkeit werden jedoch beide Objekttypen in der Objekttabelle fast gleich behandelt.

Jeder atomare Ausdruck erhält einen eindeutigen Bezeichner und einen eigenen Eintrag in der Objekttabelle. Der Bezeichner setzt sich aus einem Pfadpräfix, der aus den Operanden des Ausdruckes ermittelt wird, und einer generierten Kennung zur global eindeutigen Identifizierung zusammen. Diese Kennung wird in der Funktion **gen\_pseudo** durch Weiterzählen eines statischen Zählers bei jedem Aufruf generiert. Die Kennung wird durch Voranstellen der Buchstaben "ps" zu einem gültigen Bezeichner erweitert und als solcher in der Symboltabelle abgelegt.

Jeder atomare Topologieausdruck besitzt einen Operator und zwei Operanden. Bei RELATION-Objekten (Wert des **spec**-Feldes: RELATION) dürfen beide Operanden sowohl Objekttabellenindizes von (SET\_)RELATION-Objekten als auch von deklarierten Unterstrukturen der aktuellen Klasse (erlaubte Objekttypen sind PART, PARAM, SET) sein. Bei SET\_RELATION-Objekten (Wert des **spec**-Feldes: SET\_RELATION) muß der erste Operand der Objekttabellenindex einer deklarierten Unterstruktur der aktuellen Klasse sein. Der zweite Operand kann entweder der Index eines weiteren SET\_RELATION-Objektes sein, falls im Quellausdruck aus DROOLY mehrere Quantoren geschachtelt auftreten oder er kann der Index eines RELATION-Objektes sein.

Wird eine deklarierte Struktur als Operand einer (SET\_)RELATION in die op1- und/oder op2-Felder eingetragen, wird gleichzeitig die RELATION in eine Liste eingehängt, über die man leicht alle RELATIONS finden kann, die eine bestimmte Struktur als Operand besitzen. Die erste solche RELATION wird von der deklarierten Struktur über ihr **relationHd**-Feld referenziert. Die Verkettung zwischen den RELATIONS findet für den ersten Operanden über das **next\_relation1**-Feld, für den zweiten Operanden über das **next\_relation2**-Feld der RELATIONS statt.

Zu beachten ist, daß sowohl in DINGO-II als auch in DROOLY Quantoren immer ganz links im Ausdruck stehen müssen. Im DROOLY-Parser wird dies durch eine entsprechende Anordnung der Regeln erzwungen. In der Objekttabelle kann aber die Reihenfolge bei einer Optimierung durch Herausziehen von Schleifeninvarianten (siehe Kapitel 10.) durchaus auch so sein, daß ein RELATION-Objekt ein SET\_RELATION-Objekt als Operanden hat.

Jedes (SET\_)RELATION-Objekt wird in der Objekttabelle durch Aufruf der Funktion *link\_objekt* in die Komponentenstruktur eines deklarierten Objektes eingehängt. Dieses muß aber nicht mit dem statischen DINGO-II- oder DROOLY-Kontext übereinstimmen. Das passende Aggregat für eine (SET\_)RELATION wird vielmehr von der Funktion *max\_comm\_path* aus den Pfaden ihrer Operanden ermittelt. Der Pfadpräfix für das (SET\_)RELATION-Objekt ergibt sich als der maximale, gemeinsame Pfadpräfix seiner Operanden. Auf diese Weise wird eine (SET\_)RELATION immer tiefstmöglich in einer Komponentenhierarchie eingehängt (Prinzip der Lokalität).

Einzelne atomare Ausdrücke eines zusammengesetzten Ausdruckes können damit durchaus auch an verschiedenen Aggregaten hängen. Um auf einfache Weise aus einer Referenz auf einen atomaren Teilausdruck den gesamten Topologieausdruck rekonstruieren zu können, trägt jeder Teilausdruck in der Objekttabelle einen Verweis auf die Wurzel seines Relationenbaums im *outermost*-Feld. Die einzelnen Wurzeln werden von ihrem Aggregat über die *next\_outermost*-Felder miteinander verkettet.

SET\_RELATIONS stellen nur ein syntaktisches Hilfskonstrukt zur Integration von quantifizierten Ausdrücken zur Verfügung. Sie beschreiben für sich genommen keine flächigen Layoutobjekte. Daher kann einer SET\_RELATION kein Wertausdruck zugeordnet werden. Einer RELATION dagegen kann, wenn ihr Ursprung eine zweistellige Design Rule ist, ein Wertausdruck zugeordnet werden. Dies geschieht durch einen Verweis über das *expr*-Feld in die Ausdruckstabelle.

#### Objekttypen C\_RELATION, C\_SET\_RELATION

Objekte der Typen C\_RELATION und C\_SET\_RELATION unterscheiden sich von RELATION-Objekten bzw. von SET\_RELATION nur durch die Belegung ihrer **spec**-Felder (RELATION bzw. C\_SET\_RELATION). Sie werden über die selben Grammatikregeln erzeugt, wobei jedoch der Wertausdruck bei C\_RELATION-Objekten nicht leer sein darf. Durch den Wert der globalen Variable *expr\_level* kann der Parser entscheiden, welcher Objekttyp im einzelnen vorliegt. Semantisch stellen diese Objekttypen Bedingungen dar, die in bedingten Design Rules benutzt werden können. Als Bedingungen haben sie keine Flächeninterpretation im Layout.

#### Objekttyp CLASSTEST

Objekte vom Typ CLASSTEST (Wert des **spec**-Feldes: CLASSTEST) sind neben C\_PROPETRIES, C\_RELATIONS und C\_SET\_RELATIONS eine weitere Möglichkeit, Bedingungen in bedingten Design Rules zu konstruieren. Mit ihrer Hilfe kann die Klassenzugehörigkeit und insbesondere deren Belegung mit aktuellen Parametern überprüft werden. Bis auf den unterschiedlichen Wert des **spec**-Feldes entspricht die Belegung des Objektabelleneintrages der einer deklarierten Unterstruktur, so daß bei der Auswertung eines CLASSTESTs ein einfacher Vergleich möglich ist.

Objekte dieses Typs werden nicht in die Komponentenstruktur eines Aggregates eingehängt. Stattdessen werden sie ausschließlich über den zugehörigen Wertausdruck referenziert. Objekte vom Typ CLASSTEST stellen nur eine Hilfskonstruktion während der Übersetzung dar. Wird eine Klasse durch Deklaration einer Unterstruktur unter Angabe der erforderlichen aktuellen Parameter instanziert, werden alle PARAMs dieser Klasse in PARTs umgewandelt, da ihre Klasse durch die aktuellen Parameter eindeutig bestimmt wird. Ein CLASSTEST, welches die Klasse eines solchen ehemaligen PARAMs abfragt, kann ausgewertet und durch seinen then- bzw. else-Zweig ersetzt werden.

## 8.6 Objekte object.c

Das Modul object.c des Compiler-Backends stellt die Objekttabelle als abstrakten Datentyp zur Verfügung. Darin definiert ist die Datenstruktur der Objekttabelle als Vektor von Elementen des oben beschriebenen Typs objectT. Jedes Vektorelement speichert einen Eintrag in der Objekttabelle. Die Vektorgröße ist statisch deklariert. Da jedoch jede Anforderung und "Inbetriebnahme" eines neuen Objekteintrages ausschließlich über einen Aufruf der Funktion *gen\_obj\_entry* abgewickelt wird, ist für eine Erweiterung auf dynamische Reallozierung bei drohendem Tabellenüberlauf nur eine entsprechende Anpassung dieser einen Funktion notwendig.

### Umgebung

Neben der Kommunikation mit dem Parser über die exportierten Funktionen werden die drei Dateideskriptoren *tabout* zur Ausgabe einer lesbaren Form der Objekttabelle, *trfout* zur Ausgabe einer DROOLY-ähnlichen Form der Objekttabelle (7) und *tecout* zur Ausgabe der Technologiedatei aus dem Hauptmodul importiert. Aus dem Parser importiert wird der globale Zähler *relat\_depth*, der jedoch nur in den Funktionen *incr\_relat* und *decr\_relat* manipuliert wird. Es werden explizit keine Datenstrukturen aus diesem Modul exportiert. Auf die Objekttabelle kann außerhalb des Moduls object.c nur über die exportierten Funktionen zugegriffen werden. Als Parameter zur Referenz auf Einträge in die Objekttabelle erwarten alle diese Funktionen einen positiven, ganzzahligen Wert. Da der nulle Objekttabelleneintrag nicht benutzt wird, dient der Index 0 als unspezifizierter (NIL-) Index. Eine entsprechende symbolische Konstante ist in der Kopfdatei const.h definiert.

### Struktur

Der direkte Zugriff auf die Felder eines Objekteintrages erfolgt innerhalb des Moduls ausschließlich über einen Satz von Makros, wobei für jedes Feld in der Struktur objectT ein entsprechendes Makro (*o\_...*) definiert ist. Außerhalb des Moduls object.c sind diese Makros nicht zugänglich. Als zentrale Schnittstelle zu den in der Objekttabelle gespeicherten Informationen dient die Funktion *msg*, die gleichsam eine Nachricht an ein bestimmtes Objekt in der Objekttabelle absetzt. Diese Funktion erwartet als ersten Parameter den Index des betroffenen Objektes (der Empfänger der Nachricht) und als zweiten Parameter einen Schlüssel, der die Art der Aktion auf dem Objekt spezifiziert (der Selektor). Die erlaubten Schlüssel sind in dem Modul token.c definiert und werden bei der Übersetzung des Compiler-Backends automatisch (mittels eines Awk-Skriptes) in die Kopfdatei specs.h

(7) Die DROOLY-ähnliche Ausgabeform eignet sich deshalb besser zu Vergleichszwecken bei der systematischen Überprüfung in einer Testbibliothek, weil referenzierte Objekte ohne Angabe des Tabellenindizes protokolliert werden. Damit wird diese Ausgabeform in gewissem Umfang formatunabhängig im Sinne Chus.

übernommen. Die *msg*-Funktion überprüft nur, ob der übergebene Objektindex und der Schlüssel im erlaubten Bereich liegen und ruft dann eine Hilfsfunktionen auf, die die eigentlichen Manipulationen auf der Objekttabelle ausführt. Zu deren Aufruf wird aus dem übergebenen Schlüssel ein Index in einen Funktionenvektor berechnet, dem dann die Adresse der aufzurufenden Hilfsfunktion entnommen wird. Die Aufruffolge *msg* → Hilfsfunktion wird in der folgenden Beschreibung zusammenfassend als Operation mit dem Schlüssel als Operationsname bezeichnet.

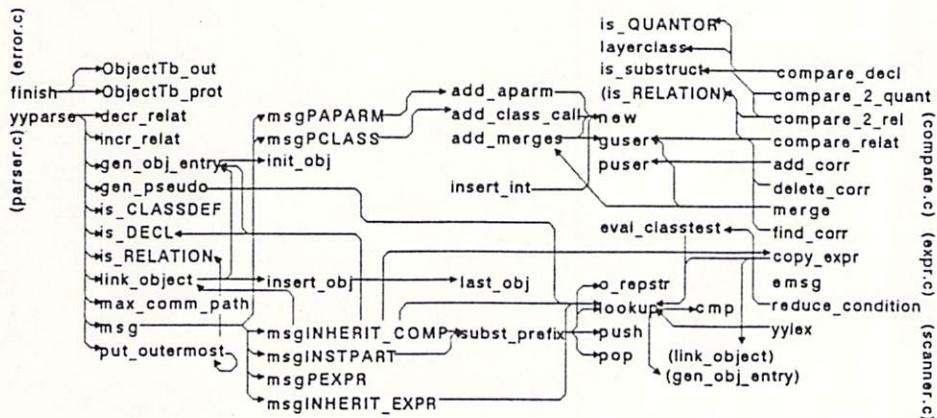


Abb. 8.7: Die Struktur des object-Moduls

Neben den genannten obligatorischen zwei Parametern akzeptiert *msg* einen, zwei oder drei weitere Parameter, die an die jeweilige Hilfsfunktion weitergereicht werden. Das angebotene Aktionsspektrum umfaßt folgende Operationsklassen:

#### Elementare Leseoperationen (m\_G...)

Elementare Leseoperationen benötigen keinen zusätzlichen Parameter. Sie ermöglichen entweder den direkten Zugriff auf die Felder der Struktur *objectT* eines ausgewählten Objekteintrages oder liefern Informationen, die zwar nicht direkt in einem Feld enthalten sind, die sich aber aus einem einzelnen Feld ableiten lassen. Die angebotenen Schlüsse für diese Operationen sind im einzelnen:

|              |  |
|--------------|--|
| m_GOUTERMOST | Lesen des Feldes outermost                               |
| m_GOP1       | Lesen des Feldes op1                                     |
| m_GOP2       | Lesen des Feldes op2                                     |
| m_GQUANT     | Lesen des Feldes quant                                   |
| m_GDEGREE    | Lesen des Feldes degreee                                 |
| m_GEXPR      | Lesen des Feldes expr                                    |
| m_GCLASS     | Lesen des Objektindizes der ersten eingetragenen Klasse  |
| m_GCLASS2    | Lesen des Objektindizes der zweiten eingetragenen Klasse |
| m_GSPEC      | Lesen des Feldes spec                                    |
| m_GPREFIX    | Lesen des Feldes prefix                                  |
| m_GSUFFIX    | Lesen des Feldes suffix                                  |
| m_GNEXTOUTER | Lesen des Feldes next_outermost                          |

Msg mit dem Schlüssel

m\_GXOP

benötigt einen weiteren Parameter, in dem der Index eines deklarierten Objektes übergeben wird. Daraus ermittelt es, ob dieses Objekt erster (Rückgabewert 1), zweiter (Rückgabewert 2) oder doppelter (Rückgabewert 3) Operand der Empfänger-RELATION ist. Ist das übergebene Objekt überhaupt nicht Operand der RELATION, ist 0 der zugehörige Rückgabewert.

#### Generatoren (m\_GFIRST..., m\_GNEXT...)

Diese Operationen generieren Folgen von Objekten, die in Listen- oder Baumstruktur in der Objekttabelle verkettet sind. Allgemein betrachtet liefert ein Generator alle Objekte einer Menge. Ein Generator besteht allgemein aus einer Kommunikationsstruktur INFO und zwei Operationen FIRST und NEXT. FIRST initialisiert die Kommunikationsstruktur und liefert das erste Element der Menge. NEXT erhält jeweils als Parameter die INFO-Struktur, berechnet daraus das nächste Element und aktualisiert die INFO-Struktur. Das Ende der generierten Folge wird durch Rückgabe eines speziellen Wertes signalisiert. Es existieren Programmiersprachen, die das Generatorkonzept als eingebautes Sprachkonstrukt enthalten [ICON]. In der Objekttabelle werden Generatoren durch ein Paar von Operationen FIRST und NEXT implementiert und die INFO-Struktur durch einen oder zwei zusätzliche Aufrufparameter der NEXT-Operation realisiert. Der erste Parameter übergibt dabei immer das zuletzt generierte Objekt.

Die beiden Operationen

|              |                            |
|--------------|----------------------------|
| m_GFIRSTCOMP | Lesen des Feldes compHd    |
| m_GNEXTCOMP  | Lesen des Feldes next_comp |

bilden einen Generator für die unmittelbaren Nachfahren eines Objektes. Der erste solche Nachfahre wird mittels m\_GFIRSTCOMP ermittelt, alle folgenden mit m\_GNEXTCOMP. Die folgenden zwei Operatoren generieren von einem Wurzelobjekt rekursiv alle seine Komponenten und deren Komponenten in Preorderanordnung:

|             |                                  |
|-------------|----------------------------------|
| m_GPREFIRST | Lesen der ersten Unterstruktur   |
| m_GPRENEXT  | Lesen der nächsten Unterstruktur |

Die Operation m\_GPREFIRST liefert immer das Empfänger-Objekt zurück und ist damit eigentlich eine Null-Funktion. Sie wurde zur Konsistenz mit dem allgemeinen Generatorkonzept jedoch aufgenommen. M\_GPRENEXT bestimmt durch Vergleich des generierten Objektindizes mit dem Wert ihres zusätzlichen Parameters das Ende der Folge und gibt bei Übereinstimmung den Wert NIL zurück. Würde in einem alternativen Konzept schon die Operation m\_GREFIRST die Wurzel des zu generierenden Baumes als zusätzlicher Parameter übergeben und von dieser in einer statischen Variablen zur Benutzung durch m\_PRENEXT abgelegt, könnte immer nur ein Generator mit diesen beiden Operationen gleichzeitig benutzt werden, da ein zweiter Aufruf von m\_PREFIRST die alte Wurzel überschreiben würde.

### Die Operationen

|                          |  |
|--------------------------|--|
| <code>m_GFIRSTREL</code> | Lesen des Feldes <code>relationHd</code>     |
| <code>m_GNEXTREL1</code> | Lesen des Feldes <code>next_relation1</code> |
| <code>m_GNEXTREL2</code> | Lesen des Feldes <code>next_relation2</code> |

bilden zusammen mit `m_XOP` ebenfalls einen Generator, der zu einer deklarierten Struktur alle RELATIONS liefert, die diese Struktur als Operand tragen. In diesem Fall gibt es zwei NEXT-Operationen, da für jeden Operanden der RELATION eine entsprechende Verkettung vorhanden sein muß. Welche von beiden Operationen aufgerufen werden muß, entscheidet man für einen bestimmten Operanden durch Aufruf von `m_XOP`.

### Elementare Schreiboperationen (`m_P...`, `m_INC...`)

Elementare Schreiboperationen schreiben den als Parameter übergebenen Wert (u.U. erst nach einer Plausibilitätsprüfung) direkt in die Felder der Struktur `objektT` eines Objekteintrages. Es werden folgende Operationen zur Verfügung gestellt:

|                           |   |
|---------------------------|---|
| <code>m_POUTERMOST</code> | Schreiben des Feldes <code>outermost</code>   |
| <code>m_POPS</code>       | Schreiben der Felder <code>opl</code> und <code>op2</code>  |
| <code>m_PQUANT</code>     | Schreiben des Feldes <code>quant</code>   |
| <code>m_PEXPR</code>      | Schreiben des Feldes <code>expr</code>  |
| <code>m_PSPEC</code>      | Schreiben des Feldes <code>spec</code>  |
| <code>m_PNEXTOUTER</code> | Schreiben des Feldes <code>next_outermost</code>  |
| <code>m_INCDEGREE</code>  | Inkrementieren des Feldes <code>degree</code> um 1  |
| <code>m_INCFPARAM</code>  | Inkrementieren des Feldes <code>fparam_ix</code> im Empfänger und Schreiben des resultierenden Wertes im Parameter-Objekt |

### Operationen zum Einfügen in Listen (`m_ADD...`)

Die Objekttabelle enthält einige Listen, die explizit vom Parser durch Anfügen der einzelnen Objekte aufgebaut werden müssen:

|                         |   |
|-------------------------|---|
| <code>m_ADDREL</code>   | Einfügen am Kopf der Liste von RELATIONS mit dem Empfänger als Operand                  |
| <code>m_ADDCLASS</code> | Hinzufügen einer neuen Klasse (noch ohne aktuelle Parameter)                            |
| <code>m_ADDAPARM</code> | Hinzufügen eines aktuellen Parameters an die zuletzt hinzugefügte Klasse des Empfängers |

### Operationen zur Behandlung der Vererbungsmechanismen

Die Instanziierung eines PARTs oder SETs bzw. die Bildung von Unterklassen für Standardobjekte wird ebenfalls durch eine Nachricht an das zu instanzierende Objekt bzw. an die Unterklasse ausgelöst:

|                             |  |
|-----------------------------|--|
| <code>m_INSTPART</code>     | Instanziierung eines PARTs oder SETs                 |
| <code>m_INHERIT_COMP</code> | Vererbung aller Komponenten der Klasse               |
| <code>m_INHERIT_EXPR</code> | Vererbung aller Wertausdrücke der Klasse             |
| <code>m_COPYCLASS</code>    | Kopieren einer Klasse mit ihren aktuellen Parametern |

Die Vorgehensweise dieser Operationen wird unten noch eingehend beschrieben.

Alle über *msg* erreichbare Operationen liefern den Wert des gelesen Feldes (ein Objekttabellenindex oder eine ganze Zahl) zurück. Neben diesen Operationen gibt es im Modul object.c noch eine Reihe weiterer Funktionen, die sich entweder nicht auf nur ein Objekt beziehen, andere Rückgabetypen liefern oder einfach zur weiteren Strukturierung dienen. Die Anwendung dieser Funktionen wird im Punkt "Verfahren" an der jeweiligen Stelle beschrieben. Sie lassen sich in folgende Klassen einteilen:

#### Funktionen für allgemeine Verwaltungsaufgaben

|                      |  |
|----------------------|--|
| <i>new</i>           | Allgemeine Allozierung von Speicherplatz                       |
| <i>renew</i>         | Neuallozierung bei verändertem Speicherbedarf                  |
| <i>gen_obj_entry</i> | Bereitstellung und Initialisierung eines neuen Objekteintrages |
| <i>init_object</i>   | Initialisierung eines Objekteintrages                          |
| <i>obj_push</i>      | Verwaltung eines Stacks von Symboltabellenindizes              |
| <i>_obj_pop</i>      | - " -  |

#### Funktionen zur Verwaltung der Komponentenhierarchie

|                      |   |
|----------------------|---|
| <i>link_object</i>   | Einbinden einer neuen Komponente                                  |
| <i>insert_obj</i>    | Einfügen in die Komponentenliste                                  |
| <i>last_obj</i>      | Aufsuchen das letzten Objektes in der Komponentenliste            |
| <i>max_comm_path</i> | Ermittlung des längsten gemeinsamen Pfadpräfix                    |
| <i>subst_prefix</i>  | Auswechseln einer Anzahl Komponenten am Anfang eines Objektpfades |

#### Funktionen zum symbolischen Zugriff auf Objekte

|                   |  |
|-------------------|--|
| <i>cmp</i>        | Vergleich von Präfix und Suffix zweier Objekte   |
| <i>lookup</i>     | Zugriff auf ein Objekt über seinen Präfix und Suffix. Neueinrichtung bei Bedarf            |
| <i>gen_pseudo</i> | Generierung eines Relationsbezeichners und Einrichten eines Eintrages in der Objekttabelle |

#### Funktionen für Ja/Nein-Anfragen

|                     |   |
|---------------------|---|
| <i>is_RELATION</i>  | Anfrage an das spec-Feld  |
| <i>is_DECL</i>      | - " -   |
| <i>is_CLASSDEF</i>  | - " -   |
| <i>is_QUANTOR</i>   | - " -   |
| <i>is_substruct</i> | Vergleich des Pfades eines Objektes mit dem Pfadpräfix eines zweiten Objektes |

### Funktionen zur Bearbeitung semantischer Aufgaben

|                       |  |
|-----------------------|--|
| <i>eval_classtest</i> | Vergleich zweier Klassen   |
| <i>add_merges</i>     | Anfügen von Identifikationen nach ihrer Ermittlung im Modul compare.c                      |
| <i>put_outermost</i>  | Rekursives Anfügen einer Referenz auf die Wurzel eines Relationenbaums in allen Baumknoten |

### Sonstige Funktionen

|                       |   |
|-----------------------|---|
| <i>add_aparm</i>      | Hilfsfunktion für m_ADDAPARM              |
| <i>add_class_call</i> | Hilfsfunktion für m_ADDCLASS              |
| <i>puser</i>          | Schreiben des Feldes user                 |
| <i>guser</i>          | Lesen des Feldes user                     |
| <i>o_reptr</i>        | Ermittlung des textuellen Pfadbezeichners |

### Verfahren

Die Abläufe zur Verwaltung der Objekttabelle bilden keinen auf das Objektmodul beschränkten Kontrollfluß, sondern werden beim Parsen des DROOLY-Textes vom Parser angestoßen. Es erscheint aus diesem Grunde sinnvoll, die einzelnen Abläufe jeweils anhand eines auslösenden DROOLY-Beispiels darzustellen.

Man betrachte folgenden Ausschnitt aus einer CMOS-Technologiebeschreibung in DROOLY (Die Nummerierung der Zeilen hier wie in den folgenden Beispielen ist nicht Bestandteil der Technologiebeschreibung, sondern dient der Erleichterung der Bezugnahme im Text):

```

1 layerdef Metal
2 layerdef Poly
3 layerdef Contact_Cut
4 layerdef n_Diff
5 layerdef p_Diff
6 layerdef n_Well

```

Der Ablauf zum Einrichten der entsprechenden Objekte in der Objekttabelle ist folgender: Der Parser fordert vom Scanner das Einlesen des ersten Lexems an. An dem zurückgelieferten Wert des Tokens LAYERDEF erkennt er die Layerdefinition und fordert ein weiteres Lexem an. Der Scanner erkennt einen Bezeichner und ruft *lookup* zur Einrichtung eines Eintrages in der Objekttabelle auf. *Lookup* wird vom Scanner derart aufgerufen, daß für eine neue Präfix/Suffix-Kombination in jedem Fall ein neuer Objekteintrag angelegt wird.

*Lookup* durchsucht die Objekttabelle binär, beginnend mit dem durch die statische Variable *s\_root* bezeichneten aktuellen Objekt, durch wiederholten Aufruf von *cmp* zum Vergleich des übergebenen Präfix/Suffix-Paars mit dem Präfix/Suffix-Paar des aktuellen Objekttabelleneintrages. *Cmp* liefert drei verschiedene Rückgabewerte für "kleiner", "gleich" oder "größer". Bei "kleiner" oder "größer" verfolgt *lookup* das left- bzw. right-Feld des aktuellen Objekteintrages so lange, bis es auf den Wert NIL in dem jeweiligen Feld trifft. Ein Aufruf von *gen\_obj\_entry* erzeugt daraufhin einen mit dem übergebenen Präfix und Suffix initialisierten neuen Objekteintrag. Die einzelnen Einträge der Objekttabelle werden dabei einer nach dem anderen mit aufsteigendem Index vergeben. Der neue Objekteintrag wird in den durch die left/right-Felder definierten binären Suchbaum

eingehängt und sein Index als Ergebnis von *lookup* zurückgegeben. Lieferte *cmp* jedoch "gleich" zurück, war das an *lookup* übergebene Präfix/Suffix-Paar schon in einem Objekteintrag enthalten. Der Index dieses Objekteintrages wird als Ergebnis von *lookup* zurückgegeben.

Im konkreten Beispiel der ersten Layerdefinition ist der Präfix NIL, als Suffix wird der Symboltabellenindex des Bezeichners "Metal" übergeben. Die Objekttabelle ist noch unbesetzt, d.h. *s\_root* hat den Wert NIL. *Lookup* fordert also durch Aufruf von *gen\_obj\_entry* einen neuen Objekteintrag an, der mit dem übergebenen Präfix und Suffix initialisiert wird. *Lookup* gibt den Objektindex 1 als Ergebnis an den Scanner zurück. Dieser liefert an den Parser das Token IDENTIFIER und speichert den Objektindex 1 in der Kommunikationsvariablen *yylval.OBJ.index*. Der Parser kann daraufhin die Ableitung

```
layer_definition : layerdef IDENTIFIER
```

durchführen. Die an der zugehörigen Regel hängende Semantik setzt über einen *m\_PSPEC*-Aufruf das *spec*-Feld des neuen Objekteintrags auf LAYERDEF und weist es damit als Layerdefinition aus. Nach demselben Schema werden auch Objekteinträge für alle weiteren Layerdefinitionen des obigen Beispiels eingerichtet, so daß die Objekttabelle nach Ableitung der letzten Layerdefinition folgende Einträge enthält:

|                   | pref  | suf        | cls   | cls   | cmp    | nxt     | nxt                             | lmg                | rmg   | quantor | /     |       |       |       | rel   | nxt   | nxt   | /     | nxt   | fpa   |
|-------------------|-------|------------|-------|-------|--------|---------|---------------------------------|--------------------|-------|---------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|                   | fix   | fix spec   |       |       | cnt Hd | cmp san | cnt operator op1 op2 val deg Hd | rel re2 out out ix |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ===== | =====      | ===== | ===== | =====  | =====   | =====                           | =====              | ===== | =====   | ===== | ===== | ===== | ===== | ===== | ===== | ===== | ===== | ===== | ===== |
| [1] {Metal}       |       |            |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   |       | 1 LAYERDEF |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ----- | -----      | ----- | ----- | -----  | -----   | -----                           | -----              | ----- | -----   | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| [2] {Poly}        |       |            |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   |       | 2 LAYERDEF |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ----- | -----      | ----- | ----- | -----  | -----   | -----                           | -----              | ----- | -----   | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| [3] {Contact_Cut} |       |            |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   |       | 3 LAYERDEF |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ----- | -----      | ----- | ----- | -----  | -----   | -----                           | -----              | ----- | -----   | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| [4] {n_Diff}      |       |            |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   |       | 4 LAYERDEF |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ----- | -----      | ----- | ----- | -----  | -----   | -----                           | -----              | ----- | -----   | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| [5] {p_Diff}      |       |            |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   |       | 5 LAYERDEF |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ----- | -----      | ----- | ----- | -----  | -----   | -----                           | -----              | ----- | -----   | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| [6] {n_Well}      |       |            |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   |       | 6 LAYERDEF |       |       |        |         |                                 |                    |       |         |       |       |       |       |       |       |       |       |       |       |
|                   | ----- | -----      | ----- | ----- | -----  | -----   | -----                           | -----              | ----- | -----   | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |

Diese Tabelle ist ein direkter Auszug aus der Ausgabe, wie sie das Compiler-Backend zum Trace der Objekttabelle in der Datei mit der Erweiterung ".t" ausgibt. Zum Format dieser Tabelle sind einige Erläuterungen notwendig:

Die Tabellierung eines Objekteintrages beginnt jeweils mit der Angabe des Objekttabellindizes in eckigen Klammern. Ihm folgt der vollständige Pfadbezeichner in geschweiften Klammern. Diese Ausgabe erfolgt nur zur leichteren Lesbarkeit der Tabelle. Der Objektbezeichner wird dazu aus dem Symboltabelleneintrag des jeweiligen Objektes und aller seiner Pfadvorgänger ermittelt. Bis auf die Felder *left*, *right*, *lmerges*, *rmerges* und *user* gibt es für jedes Feld der *objectT*-Struktur eine Spalte in der Tabelle. Um die Tabelle übersichtlich zu halten, wurden für die Spaltenüberschriften folgende Abkürzungen gewählt:

|                |                  |
|----------------|------------------|
| prefix         | -                |
| suffix         | -                |
| spec           | -                |
| classes[0]     | cls              |
| class_cnt      | cls.cnt          |
| compHd         | cmp.Hd           |
| next_comp      | nxt.cmp          |
| next_same      | nxt.sam          |
| lmerge_cnt     | lmg.cnt          |
| rmerge_cnt     | rmg.cmt          |
| quant          | quantor/operator |
| op1            | -                |
| op2            | -                |
| expr           | val              |
| degree         | deg              |
| relationHd     | rel.Hd           |
| next_relation1 | nxt.re1          |
| next_relation2 | nxt.re2          |
| outermost      | out              |
| next_outermost | nxt.out          |
| fparm_ix       | fpa.ix           |

Mit Ausnahme von **spec** und **quant** werden alle Felder als Zahlenwerte tabelliert. Ist der jeweilige Wert gleich NIL, bleibt die zugehörige Spalte leer. Für die Felder **spec** und **quant** wird die symbolische Entsprechung des Feldinhaltes angegeben. Bei der Tabellierung der Klassen und merges ist eine Sonderbehandlung vorgesehen. Der Klassenindex des ersten Elementes im **classes**-Vektor wird als normaler Feldinhalt angegeben. Enthält das Objekt entweder mehrere Klassenaufrufe (z.B. bei PARAMs) oder enthält ein Klassenaufruf aktuelle Parameter, so wird für jedes Element des **classes**-Vektor eine zusätzliche Zeile der Form

```
| CLASS <Klassenindex> (aktuelle_Parameter)
```

ausgegeben. Enthält der lmerges- oder rmerges-Vektor Elemente, so werden diese ebenfalls einzeln in zusätzlichen Zeilen der folgenden Formen tabelliert:

```
| LMERGE <Objektindex>
| RMERGE <Objektindex>
```

Wie man der Tabelle entnimmt, ist für die einzelnen Layerdefinitionen jeweils ein eigener Objekteintrag angelegt worden, der eindeutig durch einen Objekttabellenindex identifiziert werden kann. Die Felder **prefix** und **suffix** enthalten die Verweise auf den Pfadpräfix bzw. den Symboltabellenindex des jeweiligen Objektbezeichners.

In der Beispiel-Technologiebeschreibung folgt auf die Layerdefinitionen die Definition einer Klasse mit einem formalen Parameter:

```

7 classdef Gate fpar diffusion )
8   part p_wire class Poly )
9     param diffusion class n_Diff ) class p_Diff )
10
11   shape iso_Polygon
12   cross [ ] p_wire diffusion
13   diffusion width [ if diffusion classtest n_Diff )
14           then >= 15
15           else >= 10
16       ]
17 )

```

Auf die Anforderung eines neuen Tokens liefert der Scanner das Token CLASSDEF an den Parser. Beim Einlesen des folgenden Lexems erkennt der Scanner einen Bezeichner und ruft *lookup* zur Einrichtung eines Objekteintrages. *Lookup* wird dazu der Präfix NIL und der Suffix 7 (Symboltabellenindex von "Gate") übergeben. *Lookup* findet zu diesem Paar keinen Objekteintrag und richtet einen neuen Eintrag ein, dessen Index es an den Scanner zurückliefert. Dieser meldet das Token IDENTIFIER an den Parser. Im Parser kann an dieser Stelle das spec-Feld des neuen Objekteintrages (dessen Index der Parser aus der Kommunikationsvariablen *yylval.OBJ.index* ausliest) auf CLASSDEF gesetzt werden. Das so erzeugte Klassenobjekt definiert bis zur Ableitung zu *class\_definition* den statischen Kontext. Sein Index wird in der globalen Variable *act\_class* gespeichert und dient im Scanner als Pfadpräfix bei der Erzeugung neuer Objekteinträge innerhalb dieser statischen Umgebung.

Es folgt das Parsen des formalen Parameters durch Übergabe des Tokens FPARAM nach dem Einlesen des nächsten Lexems. Für jeden der folgenden Bezeichner wird beim Einlesen durch den Scanner ein Objekteintrag angelegt. Der Scanner übergibt dazu den Wert von *act\_class* als Präfix und den jeweiligen Symboltabellenindex als Suffix an *lookup*. Als Folge dieser Konstellation werden die Bezeichner von formalen Parametern als relativ zur aktuellen Klasse angesehen. Im Beispiel wird also ein Objekt mit dem absoluten Bezeichner Gate.diffusion vom Scanner angelegt.

Sowohl das Klassenobjekt als auch das Objekt zu jedem formalen Parameter haben ein Feld *fpParm\_ix*. Bei einem Aufruf der Operation *m\_ADDFPARM* mit dem Klassenobjekt als Empfänger wird dieses Feld im Klassenobjekt für jeden formalen Parameter um 1 inkrementiert. Der selbe Aufruf speichert den inkrementierten Wert im *fpParm\_ix*-Feld des Parameterobjektes als dessen Index in der Parameterliste.

Mit dem Schlüsselwort part beginnt im DROOLY-Text der Deklarationsteil der Klasse. Der Scanner liefert nach dem Token PART den Objektindex der neu erzeugten Komponente Gate.p\_wire an den Parser. Der Parser speichert diesen Objektindex in der globalen Variable *act\_object* zwischen, um bei der folgenden Ableitung zu *class\_call* darauf zugreifen zu können.

Als PART erhält die Komponente p\_wire eine eindeutig bestimmte Klasse zugewiesen. Zu beachten ist, daß Klassenbezeichner implizit als global angesehen werden. Um diesen Sachverhalt dem Scanner bekannt zu machen, wird nach Erkennen des Tokens CLASS die Kommunikationsvariable *act\_class* auf den Wert NIL gesetzt. Der folgende Klassenbezeichner wird dann mit eben diesem Wert als Präfix in der Objekttabelle gesucht. Der Parser erhält den Index des gefundenen Objektes zurück. Handelt es sich bei diesem Objekt um keine Layer- oder Klassendeklaration

(erkennbar an Wert des `spec`-Feldes), wird die Übersetzung mit einer entsprechenden Fehlermeldung abgebrochen. Andernfalls wird die Klassenzuweisung dem aktuellen Objekt (im Beispiel also die Klasse Poly dem PART `p_wire`) über einen Aufruf von `m_ADDCLASS` zugewiesen. Diese Operation erledigt ihre Aufgabe durch Aufruf der Funktion `add_class_call`, der dazu die Indizes des aktuellen Objektes und des Klassenobjektes als Parameter mitgegeben werden.

Die Klassenzuweisungen einer Komponente sind im Objekteintrag als Vektor `classes` von Klasseneinträgen organisiert. Beim ersten Eintrag einer Klassenzuweisung (bei PARTs bleibt es die Einzige) wird ein neuer Vektor mit CLASSINC Elementen angelegt. Jedes dieser Elemente kann den Index eines Klassenobjektes und einen Vektor aktueller Parameter speichern. Werden bei der Deklaration eines PARAMs mehr als CLASSINC Klassenzuweisungen benötigt, realloziert `add_class_call` den Klassenvektor mit zusätzlich CLASSINC Elementen. Der Wert der symbolischen Konstante CLASSINC ist mit 3 am Anfang des Objekt-Moduls definiert.

Eine Klassenzuweisung kann mit der Angabe aktueller Parameter verbunden sein. Diese werden im DROOLY-Text als Liste von Indizes in die Klassenzuweisungslisten der PARAM-Komponenten des jeweiligen Klassenobjektes dargestellt. In der Objekttabelle wird eine solche Liste als `aparms`-Vektor in der Struktur einer Klassenzuweisung gespeichert. Die aktuellen Parameter werden einzeln bei Abarbeitung der aktuellen Parameterliste von der Operation `m_PAPARM` in die Objekttabelle eingetragen. Die eigentliche Arbeit erledigt auch hier eine Dienstfunktion `add_aparm`. Beim ersten aktuellen Parameter der zuletzt eingetragenen Klassenzuweisung wird ein Parametervektor von PARINC Elementen alloziert. Sollten mehr als PARINC aktuelle Parameter benötigt werden, realloziert `add_aparm` den Parametervektor mit zusätzlich PARINC Elementen. Der Wert der symbolischen Konstante PARINC ist ebenfalls mit 3 am Anfang des Objekt-Moduls definiert.

Nachdem die Klassenzuweisung u.U. mit aktuellen Parametern eingelesen ist, wird das neue PART in die Komponentenhierarchie der aktuellen Klasse eingehängt. Dies geschieht im Parser bei der Ableitung zu `part` durch einen Aufruf von `link_object`. `Link_object` wird benötigt, da der Scanner pauschal für jeden eingelesenen, neuen Bezeichner über `lookup` einen Objekteintrag anlegt. Beim Einlesen eines qualifizierten Bezeichners legt er alle Komponenten des Pfades neu an, falls sie noch nicht in der Objekttabelle existierten. Aufgrund der ihm zur Verfügung stehenden semantischen Informationen kann er jedoch nicht die Funktion des erzeugten Objektes in einer Komponentenstruktur erkennen. Dies ist Aufgabe des Parsers, welcher mit einem Aufruf von `link_object` die notwendigen Verkettungen in der Objekttabelle herstellt.

`Link_object` hat zwei Parameter. Der Erste (`obj`) übergibt den Index des einzuhängenden Objektes, der zweite Parameter (`force_entry`) ist ein Flag, mit welchem die Neueinrichtung eines Objekteintrages erzwungen werden kann, auch wenn ein Eintrag mit dem selben Präfix und Suffix bereits in der Objekttabelle vorhanden ist. `Link_object` erhält alle benötigten Informationen aus dem übergebenen Objekteintrag. Wichtige Bedingung dabei ist, daß im Parser `link_object` vor der Zuweisung eines Wertes an das `spec`-Feld aufgerufen wird. An dem unbelegten `spec`-Feld erkennt `link_object`, daß es sich bei dem Objekt um einen neuen Eintrag in die Objekttabelle handelt. Ist der Präfix des Objektes belegt, handelt es sich also um kein globales Objekt, wird zum eigentlichen Verketten `insert_obj` mit dem Objektindex und dem Index des Vaterobjektes (gleich dem Wert des `prefix`-Feldes des Objektes) aufgerufen. War `force_entry` gesetzt oder handelt es sich bei dem Objekt um ein C\_PROPERTY- oder C\_RELATION-Objekt, wird ein

neuer Objekteintrag mit identischem Präfix und Suffix erzeugt, dieser aber in keine Komponentenhierarchie eingehängt.

*Insert\_object* hängt den übergebenen Objekteintrag als letzte Komponente an die Liste der unmittelbaren Söhne des übergebenen Vaterobjektes ein. Dabei wird auch die Inorder-Fädelung des Komponentenbaums des Vaterobjektes aktualisiert, mit deren Hilfe man ohne rekursive Funktionen oder einen zusätzlichen Stack jede Komponente und Unterkomponente des Komponentenbaums des Vaterobjektes in Inorderreihenfolge bzw. in Preorderreihenfolge erreichen kann (dies wird vom Generatorpaar *m\_PREFIRST*, *m\_PRENEXT* ausgenutzt). *Insert\_object* bedient sich der Funktion *last\_object*, um daß letzte Objekt der Komponentenliste des Vaterobjektes aufzufinden.

Im Beispiel wird nach dem *PART* *p\_wire* noch ein *PARAM* diffusion deklariert. Dessen Objekteintrag wurde schon beim Einlesen der Liste formaler Parameter des aktuellen Klassenobjektes angelegt. Der Index dieses Objektes wird vom Scanner an den Parser übergeben und von diesem in der Variable *act\_object* gespeichert. Es folgt das Einlesen der einzelnen Klassenzuweisungen, deren jede ein Element des *classes*-Vektors des *PARAMs* belegt. Mit der Deklaration des *PARAMs* ist der Deklarationsteil der Klasse Gate abgeschlossen. Die Objekttabelle enthält folgende neuen Objekteinträge (8):

```

pre| suf|      |cls|cls|cmp|nxt|nxt|lmg|rng|quantor|/|  |  |  |  |rel|nxt|nxt|  |nxt|fpa
fix| fix|spec  |  |cnt|Hd|cmp|sam|cnt|cnt|operator|op1|op2|val|deg|Hd|rel1|re2|out|out|ix
=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====
[7] {Gate}
    | 7|CLASSDEF|  |  | 9|  |  |  |  |NONE  |  |  |  | 2|  |  |  |  |  |  |  |  | 1
-----
[8] {Gate.diffusion}
    | CLASS 4 ( )
    | CLASS 5 ( )
    7| 8|PARAM  |  | 2|  |  |  |  |NONE  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1
-----
[9] {Gate.p_wire}
    7| 9|PART  | 2| 1|  | 8|  |  |  |NONE  |  |  |  |  |  |  |  |  |  |  |  |
-----
```

Mit der Ableitung zu *declarations* sind alle Unterstrukturen der aktuellen Klasse deklariert. Etwas anschaulicher als durch die Verkettungen in der Objekttabelle lässt sich die Komponentenhierarchie durch folgenden Baum darstellen:

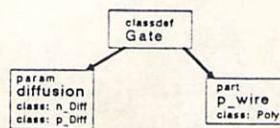


Abb. 8.1: Tabellen der Beispieltechnologie (I)

(8) Negative Werte in den prefix- oder suffix-Feldern gehören zu einer Fädelung, die zusätzlich zur normalen Komponentenverkettung in die Objektabelle eingebracht wird und zur Generierung aller Unterstrukturen eines Objektes verwandt werden kann, ohne dazu rekursive Funktionen verwenden zu müssen.

Es folgt das Einlesen des Beschreibungsteils. Eine Beschreibungsanweisung kann in DROOLY eine von drei Formen *forward*, *merge* oder *relation* annehmen, welche in beliebiger Reihenfolge aufgeführt sein können. Der Beschreibungsteil der Klasse Gate beginnt mit einem *forward*. Forwards genügen der allgemeinen Syntax

```
forward : [ [ quantor_spec ] IDENTIFIER ] property
property :
property_type expression | shape IDENTIFIER
property_type : width | area | dimension | radius | length
quantor_spec : all | some | any | one
```

Ein *forward* spezifiziert eine Eigenschaft eines Objektes. Im allgemeinen geht der Beschreibung der Eigenschaft der Bezeichner des betroffenen Objektes voran. Fehlt dieser oder ist er durch das Schlüsselwort *self* ersetzt, wird eine Eigenschaft der aktuellen Klasse spezifiziert. Die Zeile 11 im Beispiel legt also eine erlaubte Geometrieklasse für das Klassenobjekt Gate fest, in diesem Fall *iso\_Polygon*. Der Scanner erzeugt beim Einlesen dieser Zeile zwei Objekte, ein PROPERTY-Objekt mit dem Bezeichner Gate.shape sowie ein zunächst völlig unspezifiziertes Objekt mit dem globalen Bezeichner *iso\_Polygon*. Der Parser hängt das PROPERTY-Objekt durch Aufruf von *link\_object* in die Komponentenhierarchie der Klasse ein und weist dem spec-Feld mit *m\_PSPEC* den Wert SHAPE zu. Das zweite Objekt wird in keine Komponentenhierarchie eingehängt. Lediglich seinem spec-Feld wird der Wert GEOCLASS zugewiesen.

In der Zeile 12 folgt eine *relation* zwischen den Objekten Gate.p\_wire und Gate.Diffusion, der in diesem Fall kein Wertausdruck zugeordnet ist. Für die RELATION wird vom Parser durch Aufruf von *gen\_pseudo* ein Objekteintrag mit einem eindeutigen Bezeichner erzeugt. Der Pfadpräfix dieses Bezeichners wird dazu zunächst durch die Funktion *max\_comm\_path* aus den Pfadbezeichnern der beiden Relationsoperanden bestimmt. Durch diese Maßnahme wird erreicht, daß eine RELATION möglichst tief in der Komponentenhierarchie der aktuellen Klasse eingehängt wird (Lokalitätsprinzip). Im Beispiel sind die Pfade der beiden Operanden gegeben durch

```
Gate.p_wire und
Gate.diffusion .
```

Als längster gemeinsamer Pfadpräfix ergibt sich daraus Gate. *Gen\_pseudo* erzeugt für eine RELATION jeweils einen neuen Symboltabelleneintrag und trägt dort einen Bezeichner der Form psXXXX ein (XXXX ist eine fortlaufende Nummerierung, beginnend bei 5000). Das RELATION-Objekt erhält somit den Bezeichner Gate.ps5000 und wird als direkter Sohn an das Gate-Objekt angehängt. Seinem spec-Feld wird RELATION, seinem quant-Feld der Operator CROSS und seinen Feldern op1 und op2 werden die Indizes der Operandenobjekte Gate.p\_wire und Gate.diffusion zugewiesen. Um von den Operanden leicht alle betreffenden RELATIONS erreichen zu können, werden von deklarierten Komponenten beginnend alle RELATIONS miteinander verkettet, die diese Komponente als einen ihrer Operanden haben. Der Parser ruft dazu die Operation *m\_ADDREL* mit einer zusätzlichen Kennung. Je nach dem Wert dieser Kennung erfolgt die Verkettung der RELATION über ihr *next\_relation1*- oder ihr *next\_relation2*-Feld.

Handelt es sich bei der RELATION um die Wurzel eines Relationenbaums, werden noch zwei weitere Verkettungen vorgenommen. Alle Wurzeln von unter einer Klasse definierten Relationenbäumen werden als Liste an das Klassenobjekt durch die Operation *m\_ADDNEXTOUTER* angehängt. Diese Operation erhält den Objektindex der aktuellen Klasse (aus *act\_class*) und den Index der RELATION als Parameter. Die zweite Verkettung wird durch einen Aufruf von *put\_outermost* mit dem Objektindex der RELATION als Parameter eingeleitet. Diese Funktion steigt rekursiv

in den Relationenbaum hinab und weist dem **outermost**-Feld jeder besuchten RELATION den Index der Baumwurzel zu.

Die Zeilen 13-16 enthalten noch ein weiteres *forward*. Dieses bezieht sich jedoch auf die Unterstruktur *diffusion*. Dem Scanner wird durch temporäres Umsetzen der Kommunikationsvariable *act\_class* mitgeteilt, daß er ein Objekt mit dem Bezeichner *Gate.diffusion.width* zu erzeugen hat. Dieses Objekt benötigt einen Wertausdruck, der in DROOLY von eckigen Klammern eingeschlossen wird. Der Parser erzeugt durch Aufrufe verschiedener Funktionen aus dem Wertausdrucksmodul (siehe Abschnitt 8.7) entsprechende Wertausdrücke. Im Beispiel enthält der Wertausdruck eine Bedingung

```
diffusion (classtest n_Diff ) .
```

Für diese Bedingung wird durch Aufruf von `gen_obj_entry` ein neuer Eintrag mit dem Bezeichner `Gate.diffusion` in der Objekttabelle angelegt, dieser jedoch nicht in die Komponentenhierarchie von `Gate` eingehängt. Diesem neuen Objekt wird die Klasse `n_Diff` zugewiesen und es erhält als Wert seines `spec`-Feldes `CLASSTEST`. Dieses Objekt wird noch bei der Vererbung der Komponenten der Klasse `Gate` eine Rolle spielen.

Mit dem Beschreibungsteil ist die Definition der Klasse Gate abgeschlossen. Die Objekttafel hat sich inzwischen folgendermaßen entwickelt:

In der graphischen Darstellung ergibt sich folgendes Bild:

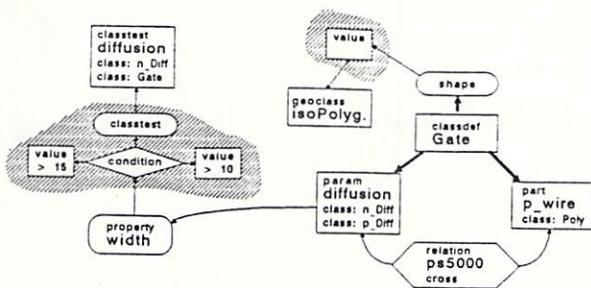


Abb. 8. 9: Tabelle der Beispieltechnologie (II)

Das Beispiel wird mit der Definition einer weiteren Klasse fortgesetzt:

```

18 classdef n_CE_Trans
19   part well (class n_Well )
20   set [ and >= 1 <= 5 ] gt (class Gate 2 )
21
22   all gt (merge gt gt )
23   all gt indist [ ] gt well
24   all gt indist [ >= 20 ] gt well
25   dist [ >= 30 ] self self
26 )

```

Diese Klasse deklariert keine PARAMs als Unterstrukturen, demzufolge besitzt sie keine Liste formaler Parameter. Der Deklarationsteil deklariert zwei Komponenten. Die erste, ein PART, erhält als Klassenzuweisung den Layer n\_Well. Das folgende SET erhält die Klassenzuweisung Gate mit Auswahl der zweiten alternativen Klasse für den ersten (und einzigen) formalen Parameter von Gate. Wie bei parameterlosen Klassenzuweisungen wird dem classes-Vektor des neuerzeugten SET-Objektes zunächst der Index des Klassenobjektes Gate mithilfe der Operation m\_ADDCLASS hinzugefügt. Der aktuelle Parameter wird anschließend dieser Klassenzuweisung mithilfe von m\_PAPARM hinzugefügt. Diese Operation ermittelt aus dem class\_cnt-Feld (bei SETs und PARTs immer gleich 1) des SET-Objektes die letzte Klassenzuweisung und fügt den neuen, aktuellen Parameter in deren über das aparm\_cnt-Feld der aktuellen Klassenzuweisung indizierten Vektor aktueller Parameter ein.

Die Deklaration des SET-Objektes enthält im Beispiel noch eine Einschränkung des erlaubten Kardinalitätsbereiches in Form eines Wertausdruckes. Die an dieser Stelle erlaubte Syntax für den Wertausdruck ist eine echte Untermenge der Syntax von Wertausdrücken im Beschreibungsteil. In DINGO-II sind zur Festlegung eines erlaubten Kardinalitätsbereiches exakte Werte (Schlüsselwort exactly), Alternativlisten von Werten (Schlüsselwort one of) oder Intervalle (Schlüsselwort range) erlaubt. In DROOLY werden diese Beschreibungarten durch mit and und or verknüpfte boolesche Ausdrücke dargestellt und auch so in die Ausdruckstabelle übernommen. Auf die Übersetzung von DROOLY-Wertausdrücken in ihre tabellarische Darstellung wird bei der Beschreibung des Ausdrucks-Moduls noch eingegangen. An dieser Stelle ist nur wichtig, daß im expr-Feld eines SET-Objektes mit eingeschränktem Kardinalitätsbereich ein Ausdruckstabellenindex eingetragen wird.

Mit der Deklaration des SET-Objektes sind alle Unterstrukturen der Klasse n\_CE\_Trans deklariert. Für das Klassenobjekt n\_CE\_Trans und seine Unterstrukturen sind in der Objekttabelle folgende Einträge angelegt worden:

```

pre| suf|      |cls|cls|cmp|nxt|nxt|lmg|rng|quantor|| | | | |rel|nxt|nxt| |nxt|fpa
fix| fix|spec | |cnt|Ed|cmp|sam|cnt|cnt|operator|op1|op2|val|deg|Ed|rel|re2|out|out|ix
=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====
[15] {n_CE_Trans}
    | 11|CLASSDEF| | |16| | | | |NONE | | | | |2|28| | | |25|
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[16] {n_CE_Trans.well}
    15| 12|PART | 6| 1| 17| | | |NONE | | | | |26| | | |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[17] {n_CE_Trans.gt}
    | CLASS 7 ( 2 )
    15| 13|SET | 7| 1| 18|24| | | |NONE | | | |6| 2|26| | | |22|
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----

```

Zu beachten ist an dieser Stelle besonders die Klasse des Objektes n\_CE\_Trans.gt. Bei ihr handelt es sich um ein strukturiertes Objekt, d.h. sie selbst enthält wiederum Unterstrukturen. Die Semantik eines Objektes in DINGO-II wie in DROOLY schreibt nun vor, daß ein Objekt rekursiv alle Unterstrukturen seiner Klasse erbt. Im Beispiel heißt das, daß der Vererbungsmechanismus noch folgende Objekte in der Objekttabelle erzeugen muß:

```

n_CE_Trans.gt.diffusion
n_CE_Trans.gt.p_wire
n_CE_Trans.gt.shape
n_CE_Trans.gt.diffusion.width

```

Es muß also der gesamte am Objekt Gate hängende Teilbaum kopiert und an das Objekt n\_CE\_Trans.gt angehängt werden. Im Anschluß an die letzte Deklaration wird daher im Parser jede deklarierte Unterstruktur der aktuellen Klasse zunächst überprüft, ob ihre Klasse eindeutig bestimmt ist (Dies trifft genau bei PART- und SET-Objekten zu). Nur dann macht das Kopieren des Klassenbaums Sinn. Für eine Unterstruktur mit eindeutig bestimmter Klasse ruft der Parser jeweils die Operation m\_INSTPART.

M\_INSTPART ist bei weitem die komplexeste Operation des Objektmoduls. Beim Aufruf werden ihr der Index des zu instanzierenden PART- bzw. SET-Objektes sowohl als Empfängerobjekt als auch als zusätzlicher Parameter übergeben. Während bei rekursiven Aufrufen in diesem zusätzlichen Parameter immer der Index des ursprünglichen Objektes und damit die Wurzel des neu zu erzeugenden Komponentenbaumes für die Anpassung der Klassen in bedingten Wertausdrücken übergeben wird, bestimmt das Empfängerobjekt die jeweilig bearbeitete Unterstruktur des Klassenobjektes. Aufgabe von m\_INSTPART ist es, dem Empfängerobjekt Kopien aller Komponenten seiner Klasse zuzuordnen. Ist die Klasse parametrisiert, werden Komponenten mit den aktuellen Parametern als Klassen instanziert, nachdem die Zahl der aktuellen Parameter mit der Zahl der formalen Parameter verglichen wurde. Stimmen die Parameteranzahlen nicht überein, wird der Instanziierungsprozeß abgebrochen und eine Fehlermeldung ausgegeben. Die eigentliche Arbeit des Kopierens wird von m\_INSTPART auf zwei Hilfsoperationen m\_INHERIT\_COMP und m\_INHERIT\_EXPR aufgeteilt. Diese werden in zwei getrennten Durchläufen über alle direkten Nachfahren der Klasse aufgerufen.

`M_INHERIT_COMP` erhält neben dem Empfänger zwei weitere Parameter: Im ersten Parameter wird die gerade zu ererbende Unterstruktur des Klassenobjektes übergeben, der zweite Parameter übernimmt den Parameter `act_object` aus `m_INSTPART` unverändert. Im ersten Schritt wird der Objekttyp der zu ererbenden Komponente aus ihrem `spec`-Feld bestimmt. Handelt es sich um ein `C_PROPERTY`- oder `C_RELATION`-Objekt aus einem bedingten Wertausdruck, wird es übersprungen. Für alle anderen Objekttypen legt `lookup` ein neues Objekt mit dem Empfängerindex als Präfix und dem Suffix der Klassenkomponente als Suffix an. Der Typ dieses Objektes hängt sowohl vom Typ der Instanz als auch vom Typ der Klassenkomponente ab. Ist mindestens eines dieser beiden Objekte ein SET, ist auch das neu erzeugte Objekt vom Typ SET und sein `spec`-Feld wird entsprechend gesetzt. War nur eines der beiden Objekte ein SET-Objekt, wird dessen Kardinalitätseinschränkung einfach an das neue Objekt vererbt. Waren jedoch beides SET-Objekte, wird dem neuen Objekt die AND-Verknüpfung der beiden Kardinalitätseinschränkung als eigene Einschränkung zugewiesen.

Handelte es sich bei der Klassenkomponente um ein PARAM-Objekt, wird seine Klasse durch einen aktuellen Parameter der Instanzklasse eindeutig bestimmt und das neu erzeugte Objekt wird ein PART oder SET je nach dem Typ des Instanzobjektes. Der Vektor der aktuellen Parameter des zu instanzierenden PARTs oder SETs enthält an einer bestimmten Stelle denjenigen aktuellen Parameter, der für die momentan bearbeitete Klassenkomponente (ein PARAM) zuständig ist. Der Index des aktuellen Parameters im Vektor ergibt sich aus dem `fparm_ix`-Feld des PARAMs. Über den aktuellen Parameter wird der gewünschte Klassenaufruf ausgewählt. Die aktuellen Parameter dieses Klassenaufrufs werden zum Klassenaufruf des oben mit `lookup` neu erzeugten Objektes hinzukopiert. Mit der derart ausgerüsteten PART-Instanz wird nun rekursiv `m_INSTPART` gerufen, da die Komponenten eines PARAMs ja erst bei der gerade erfolgten Festlegung seiner exakten Klasse bestimmt sind.

Bei der Behandlung von RELATION- oder SET\_RELATION-Objekten müssen auch die Operanden an den neuen statischen Kontext angepaßt werden. Nachdem das `spec`-Feld und der Operator gesetzt sind, muß für jeden der beiden Operanden der neue Pfadpräfix bestimmt werden. Die Hilfsfunktion `subst_prefix` erhält zu Erledigung dieser Aufgabe drei Parameter: Im ersten Parameter wird der Objektindex des jeweiligen Operanden übergeben, der zweite Parameter übernimmt den Präfix des neu erzeugten Objektes und der dritte Parameter übernimmt den Präfix der aktuellen Klassenkomponente. `Subst_präfix` verfolgt den Pfad des als ersten Parameter übergebenen Objektes zurück, bis es auf den als zweiten Parameter übergebenen Objektindex trifft. Die Suffizes der dabei durchlaufenen Objekte werden auf einem Stack gesichert. Zur Erzeugung des neuen Pfades wird der dritte Parameter als Präfix benutzt, und iterativ zusammen mit dem nächsten vom Stack zurückgeholten Suffix mittels `lookup` das zugehörige Objekt in der Objekttabelle aufgesucht. Sind alle Suffizes vom Stack geholt, wird der resultierende Pfad an `m_INHERIT_COMP` zurückgegeben und dort als Operand in das neue Objekt eingetragen. Da jedes Objekt eine Liste aus Verweisen auf alle RELATIONS trägt, in denen es als Operand auftritt, wird bei einem neu erzeugten RELATION-Objekt dieses nun noch durch `m_ADDREL` in die entsprechenden Listen seiner Operanden eingefügt.

Damit ist die Kopie der Klassenkomponente auf dieser Hierarchiestufe abgeschlossen. In einer Schleife ähnlich der in `m_INSTPART` ruft sich `m_INHERIT_COMP` mit dem gerade erzeugten neuen Objekt und jeder einzelnen der Unterstrukturen der aktuellen Klassenkomponente rekursiv auf, um so den gesamten Unterstrukturnbaum der Klasse auf das aktuelle PART bzw. SET zu kopieren.

Wieder zurück in `m_INSTPART` werden noch einmal alle direkten Nachfahren des Klassenobjektes durchlaufen, wobei aber statt `m_INHERIT_COMP` jetzt `m_INHERIT_EXPR` mit derselben Parameterbelegung gerufen wird. Diese Operation

vererbt jeweils den Wertausdruck der aktuellen Klassenkomponente an die entsprechende Komponente des aktuellen Objektes und ruft sich anschließend rekursiv für jede Unterstruktur der Klassenkomponente auf.

Die Kopie eines Wertausdruckes erfolgt über einen Aufruf der Operation m\_INHERIT aus dem Wertausdrucksmodul und wird bei dessen Beschreibung erläutert.

In der Objekttabelle wurden durch die Vererbungsmechanismen folgende neue Objekte erzeugt:

```

pre| suf|      |cls|cls|cmp|nxt|nxt|lmg|rng|quantor|| | | | |rel|nxt|nxt| |nxt|fpa
fix| fix|spec | |cnt|Hd|cmp|sam|cnt|cnt|operator|op1|op2|val|deg|Hd|rel|re2|out|out|ix
=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====|=====
[17] {n_CE_Trans.gt}
     | CLASS 7 ( 2 )
15| 13|SET   | 7| 1| 18| 24| | | |NONE | | | 6| 2| 26| | | | 22|
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[18] {n_CE_Trans.gt.p_wire}
17| 9|SET   | 2| 1|-16| 19| | | |NONE | | | 8| 1| 22| | | | |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[19] {n_CE_Trans.gt.diffusion}
17| 8|SET   | 5| 1| 20| 21| | | |NONE | | | 10| 1| 22| | | | |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[20] {n_CE_Trans.gt.diffusion.width}
19|-1036|PROPERTY| | -18|-19| | | |NONE | | | 14| | | | | | | |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[21] {n_CE_Trans.gt.shape}
17|-1034|SHAPE | | -19| 22| | | |NONE | | | 15| | | | | | | |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
[22] {n_CE_Trans.gt.ps5000}
17|-5000|RELATION| | -21|-17| | | |CROSS | 18| 19| | | | | | | |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----

```

Damit ist auch der Deklarationsteil der Klasse n\_CE\_Trans vollständig bearbeitet und der Parser beginnt mit dem Einlesen des Beschreibungsteils. In Zeile 22 werden alle Elemente der Menge n\_CE\_Trans.gt miteinander identifiziert. Im Layout bedeutet das, daß alle Gates des beschriebenen n-Kanal-Transistors miteinander verbunden sind. Da bei der Identifikation auch alle RELATIONS miteinander verglichen werden müssen, werden merge-Anweisungen zunächst in einer temporären Datei gesammelt und erst bei der Ableitung zu *class\_definition* wieder eingelesen und bearbeitet. Das Verfahren zur Bearbeitung einer Identifikationsanweisung wird bei der Beschreibung des compare-Moduls im Abschnitt 8.8 erläutert.

Die Anweisungen der Zeilen 23 und 24 sind quantifizierte RELATIONS. Dabei entstand die Zeile 23 aus einer entsprechenden Anweisung im DINGO-II - Topologieteil, Zeile 24 dagegen spezifiziert einen Wertausdruck und entstammt damit dem Design Rule Teil der entsprechenden DINGO-II Objektbeschreibung. Wie schon bei der Bearbeitung der Klasse Gate beschrieben, erzeugt der Parser aus jeder dieser zwei Anweisungen zwei RELATION-Objekte in der Objekttabelle. Die Anweisung der Zeile 25 entstand aus einer Design Rule des extern-Blockes des DINGO-II - Design Rule Teils. Die Operanden self werden als Referenzen auf das

aktuelle Klassenobjekt in die Objekttabelle eingetragen. Damit sind alle Objekte dieses Beispiels in der Objekttabelle eingetragen: (9)

Die vollständige Verkettung aller Objekte sei noch einmal in einer Graphik dargestellt:

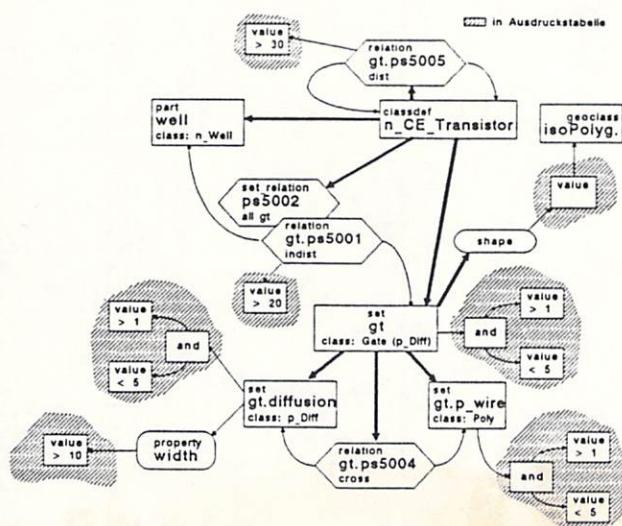


Abb. 8.10: Tabelle der Beispieltechnologie (III)

(9) Im Anhang C ist noch einmal die gesamte Tabelle nach der vollständigen Compilierung zusammen mit der resultierenden Wertausdruckstabelle abgedruckt.

## 8.7 Wertausdrücke expr.c

Das Modul expr.c des Compiler-Backends stellt die Ausdruckstabelle als abstrakten Datentyp zur Verfügung. Darin definiert ist die Datenstruktur der Ausdruckstabelle als Vektor von Elementen des oben beschriebenen Typs exprT. Jedes Vektorelement speichert einen atomaren Wertausdruck in der Ausdruckstabelle. Die Vektorgröße ist wie die der Objekttabelle statisch deklariert. Da jedoch auch hier jede Anforderung und "Inbetriebnahme" eines neuen Atomarausdruckes ausschließlich über einen Aufruf einer Funktion `gen_expr_entry` abgewickelt wird, ist für eine Erweiterung auf dynamische Reallozierung bei drohendem Tabellenüberlauf nur eine entsprechende Anpassung dieser einen Funktion nötig.

### Umgebung

Neben der Kommunikation mit dem Parser bei der Neuerzeugung von Wertausdrücken und mit dem Objektmodul bei der Vererbung von Ausdrücken werden zwei Dateideskriptoren `tabout` zur Ausgabe einer lesbaren Form der Ausdruckstabelle (nach Protokollierung der Objekttabelle), `trfout` zur Ausgabe einer DROOLY-ähnlichen Form der Wertausdrücke und `tecout` zur Ausgabe der Technologiedatei aus dem Hauptmodul importiert (siehe Fußnote auf Seite 79). Aus dem Parser importiert wird der globale Zähler `expr_depth`, der jedoch nur in den Funktionen `incr_expr` und `decr_expr` manipuliert wird. Es werden explizit keine Datenstrukturen aus diesem Modul exportiert. Auf die Ausdruckstabelle kann außerhalb des Moduls expr.c nur über die exportierten Funktionen zugegriffen werden. Als Parameter zur Referenz auf Einträge in der Ausdruckstabelle erwarten all diese Funktionen einen positiven, ganzzahligen Wert. Da der nullte Ausdruckstabelleneintrag nicht benutzt wird, dient der Index 0 auch hier als unspezifizierter (NIL-) Index.

### Struktur

Der direkte Zugriff auf die Felder eines atomaren Ausdrucks erfolgt innerhalb des Moduls ausschließlich über einen Satz von Makros, wobei für jedes Feld in der Struktur exprT ein entsprechendes Makro (`e_...`) definiert ist. Außerhalb des Moduls expr.c sind diese Makros nicht zugänglich. Als zentrale Schnittstelle zu den in der Ausdruckstabelle gespeicherten Informationen dient die Funktion `emsg`, die gleichsam eine Nachricht an einen bestimmten Atomarausdruck in der Ausdruckstabelle absetzt. Diese Funktion erwartet als ersten Parameter den Index des betroffenen Atomarausdruckes (der Empfänger der Nachricht) und als zweiten Parameter einen Schlüssel, der die Art der Aktion auf dem Atomarausdruck spezifiziert (der Selektor). Die erlaubten Schlüssel sind in dem Modul token.c definiert und werden bei der Übersetzung des Compiler-Backends automatisch zusammen mit den Schlüsseln der Funktion `msg` aus dem Objektmodul in die Kopfdatei specs.h übernommen. Die `emsg`-Funktion ruft jeweils eine Hilfsfunktion auf, die dann die eigentliche Manipulationen auf der Ausdruckstabelle ausführt. Die Aufruffolge `emsg -> Hilfsfunktion` wird in der folgenden Beschreibung zusammenfassend als Operation mit dem Schlüssel als Operationsname bezeichnet.

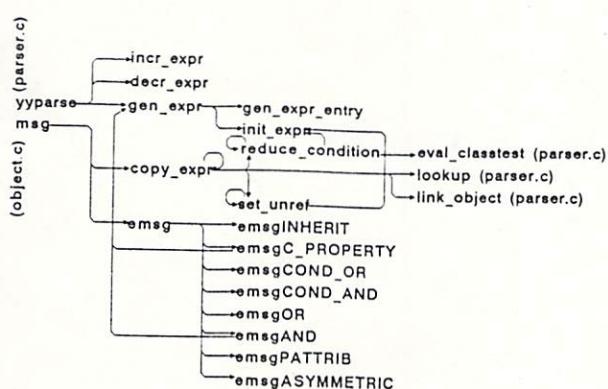


Abb. 8.11: Die Struktur des expr-Moduls

Neben den genannten obligatorischen zwei Parametern akzeptiert *emsg* keinen, einen oder zwei weitere Parameter, die an die jeweilige Hilfsfunktion weitergereicht werden. Das angebotene Aktionsspektrum umfaßt folgende Operationsklassen.

#### Elementare Leseoperationen

|                  |                         |
|------------------|-------------------------|
| <i>m_GCOND</i>   | Lesen des Feldes cond   |
| <i>m_GSPEC</i>   | Lesen des Feldes spec   |
| <i>m_GLEFT</i>   | Lesen des Feldes left   |
| <i>m_GRIGHT</i>  | Lesen des Feldes right  |
| <i>m_GATTRIB</i> | Lesen des Feldes attrib |
| <i>m_GMINVAL</i> | Lesen des Feldes minval |
| <i>m_GEQVAL</i>  | Lesen des Feldes eval   |
| <i>m_GMAXVAL</i> | Lesen des Feldes maxval |

Im Compiler-Backend wird zwar keine dieser Operationen benötigt, im Hinblick auf eine universelle Einsetzbarkeit der Tabellen als abstrakter Datentyp sind sie jedoch aufgenommen worden.

#### Elementare Schreiboperationen (*m\_P...*)

Zum Aufbau der Wertausdrücke müssen nur zwei Felder eines Atomarausdruckes gesondert geschrieben werden. Alle übrigen Felder werden entweder direkt bei der Allozierung eines neuen Atomarausdruckes oder implizit bei einer der komplexeren Operationen und Funktionen gesetzt:

|                  |                             |
|------------------|-----------------------------|
| <i>m_PREF</i>    | Schreiben des Feldes cond   |
| <i>m_PATTRIB</i> | Schreiben des Feldes attrib |

#### Operationen zur Erzeugung von Verknüpfungen

Mithilfe dieser Klasse von Operationen können Ausdrucksbäume erzeugt werden. Dabei erhält ein Atomarausdruck mit einem dyadischen Operator (OR, AND, COND\_OR, COND\_AND) zwei Söhne, die er über seine left- und right-Felder referenziert. Bedingte Ausdrücke werden mit dem triadischen Operator C\_PROPERTY erzeugt. Auf den *then*-Zweig wird mit dem left-Feld verwiesen, auf den *else*-Zweig

mit dem `right`-Feld und auf die Bedingung wird mit dem `cond`-Feld verwiesen. Der Operator `ASYMMETRIC` ist ebenfalls dyadisch und wird anstelle des monadischen Operators `VALUE` in den wertspeichernden Atomarausdrücken einer `asymmetric`-Wertliste verwandt.

|                           |  |
|---------------------------|--|
| <code>m_OR</code>         | jeweils Eintragen des entsprechenden   |
| <code>m_AND</code>        | Operators in das <code>spec</code> -Feld und Neuerzeugung  |
| <code>m_COND_OR</code>    | eines Sohn-Ausdrückes, falls vorher schon  |
| <code>m_COND_AND</code>   | ein Operator ungleich <code>VALUE</code> eingetragen   |
|                           | war  |
| <code>m_C_PROPERTY</code> | Erzeugung eines Bedingungs-Ausdrückes mit<br><code>then</code> - und <code>else</code> -Zweig als Söhne      |
| <code>m_ASYMMETRIC</code> | Eintragen von <code>ASYMMETRIC</code> in das <code>spec</code> -Feld<br>und Verweis auf die anhängende Liste |

### Operationen und Funktionen zur Behandlung der Vererbungsmechanismen

Auch im Ausdrucksmodul gibt es eine Operation und eine Funktion, die Vererbungsmechanismen bereitstellen. Bei der Instanziierung einer Klasse durch einen PART oder SET wird ja nicht nur die Komponentenhierarchie von der Klasse auf das PART oder SET übertragen, sondern es müssen auch die in der Komponentenhierarchie eingebundenen Wertausdrücke mit Verweisen auf `C_PROPERTIES` und `C_RELATIONS` kopiert werden.

|                        |   |
|------------------------|---|
| <code>m_INHERIT</code> | Aufruf von <code>copy_expr</code>       |
| <code>copy_expr</code> | Kopieren der gesamten Ausdrucksstruktur |

### Funktionen zur Behandlung bedingter Ausdrücke

Enthält die Klasse eines PARTs oder SETs Objekte vom PARAM-Typ und darüber hinaus bedingte Wertausdrücke mit CLASSTESTs auf diesen PARAMs, kann durch die aktuellen Parameter des Klassenaufrufs bei der Instanziierung der Klasse durch ein PART oder SET die Klasse der PARAMs eindeutig festgelegt werden. Sie verwandeln sich damit in PARTs bzw. SETs. Die darauf bezogenen CLASSTESTs können aufgelöst und durch den zutreffenden Zweig ersetzt werden. Die CLASSTEST-Objekte in der Objekttabelle werden nicht mehr benötigt und können freigegeben werden.

|                               |  |
|-------------------------------|--|
| <code>reduce_condition</code> | Reduzieren von CLASSTESTs, falls die Klasse inzwischen eindeutig bestimmt ist      |
| <code>set_unref</code>        | Rekursives Freigeben des Parameters und aller referenzierter Ausdrücke und Objekte |

### Funktionen für allgemeine Verwaltungsaufgaben

Jede Allozierung eines atomaren Wertausdruckes wird über die Funktion `gen_obj_entry` abgewickelt. Es gibt noch eine komfortablere Variante `gen_expr`, mit deren Hilfe der neu allozierte Wertausdruck gleich initialisiert werden kann. Dies ist die Hauptfunktion zur Erzeugung neuer Atomarausdrücke vom Parser aus. Da die Syntax von `properties` und `c_properties` bzw. von `relations` und `c_relations` identisch ist, wird zu ihrer Unterscheidung die Ausdrucks-Schachtelungstiefe benötigt. Befindet der Parser sich innerhalb der Ableitung eines Wertausdruckes, kann es sich nur um den jeweiligen `c...-Typ` handeln.

|                       |  |
|-----------------------|--|
| <i>gen_expr_entry</i> | Allozieren eines neuen Atomarausdruckes                    |
| <i>gen_expr</i>       | Allozieren und initialisieren eines neuen Atomarausdruckes |
| <i>init_expr</i>      | initialisieren eines Atomarausdruckes                      |
| <i>incr_expr</i>      | Verfolgung der Schachtlungstiefe                           |
| <i>decr_expr</i>      | von DROOLY-Wertausdrücken                                  |

### Verfahren

Da auch die Erzeugung von Wertausdrücken in erster Linie syntaxgetrieben beim Parsen einer DROOLY-Technologiebeschreibung erfolgt, soll sie anhand einiger Beispiele erläutert werden. Diese Beispiele sind der ursprünglichen DINGO-Diplomarbeit [Wic 85] entnommen und von Hand nach DROOLY übersetzt.

Anhand des ersten, einfachen Beispiels soll der prinzipielle Aufbau von Wertausdrücken durch den Parser demonstriert werden:

```
1. S.65 unten
; einfache unbedingte Design Rule
;
dist [ >= 10 ] PA_Contact PA_Contact
```

Der Wertausdruck besteht hier nur aus einer einzelnen Wertbestimmung, die der Parser über die Zwischenschritte *term*, *simple\_body* und *body* zu *expression* ableitet.

Im ersten Ableitungsschritt legt der Parser durch Aufruf von *gen\_expr* einen neuen Atomarausdruck an und initialisiert dessen Felder folgendermaßen:

|       | cond   |             | minval |        |
|-------|--------|-------------|--------|--------|
|       | left   | right       | eqval  | maxval |
| spec  | attrib |             |        |        |
| VALUE | NIL    | NIL NIL NIL | 10 10  | NIL    |

Einfache Wertbestimmungen erhalten also den Typ VALUE (In den folgenden Beispielen werden die NIL-Belegungen nicht mehr aufgeführt). Die folgende Abbildung stellt den Syntaxbaum des Wertausdrückes graphisch dar:

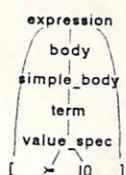


Abb. 8.12: Syntaxbaum eines einfachen Wertausdrückes

Das folgende Beispiel zeigt eine zwar unbedingte, aber aus zwei OR-verknüpften Wertbestimmungen bestehende Design Rule:

```
1. S.62 unten
; OR-verknüpfte einfache und ASYMMETRIC-Wertbestimmung
;
Contact.Cont_Cut dimension
[ or == 500 asymmetric == 375 == 600 == 375 == 600 ) ]
```

Die folgende Abbildung zeigt den zugehörigen Syntaxbaum.

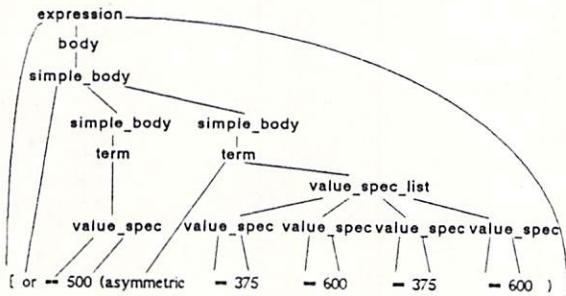


Abb. 8.13a: Syntaxbaum einer OR-Verknüpfung

In den ersten Ableitungsschritten werden die "=="-Operatoren mit den ihnen jeweils folgenden Zahlenwerten zu *value\_desc* abgeleitet. Ein Aufruf von *gen\_expr* erzeugt für jede dieser Wertbestimmungen einen Atomarausdruck und initialisiert ihn wie im ersten Beispiel gezeigt. Die Wertbestimmungen hinter dem Schlüsselwort (asymmetric werden iterativ zu einer Liste zusammengefaßt. Die Verkettung der Listenelemente in der Ausdruckstabelle nimmt die Operation *m\_ASYMMETRIC* vor, welche jeweils die alte Liste und das neue Listenelement als Parameter erhält. Im nächsten Ableitungsschritt werden die aus der ersten Wertbestimmung und aus der Wertbestimmungsliste abgeleiteten Terme mit dem *or*-Operator zu *simple\_body* zusammengefaßt. Die Verknüpfung in der Ausdruckstabelle wird von der Operation *m\_OR* erzeugt. Diese erhält die beiden Terme als Parameter und liefert den Index des resultierenden Ausdruckes zurück. Bei den noch verbleibenden Ableitungsschritten *simple\_body* -> *body* -> *expression* werden keine Manipulationen auf der Ausdruckstabelle mehr vorgenommen. Die Ausdruckstabelle enthält danach folgende Elemente (Die Wurzel des Ausdrucksbaumes ist der Atomarausdruck mit dem Index 1):

| # | spec       | attrib | cond | right | minval | maxval |
|---|------------|--------|------|-------|--------|--------|
|   |            |        | left |       | eqval  |        |
| 1 | OR         |        |      | 2     | 500    |        |
| 2 | ASYMMETRIC |        |      | 3     | 375    |        |
| 3 | ASYMMETRIC |        |      | 4     | 600    |        |
| 4 | ASYMMETRIC |        |      | 5     | 375    |        |
| 5 | VALUE      |        |      |       | 600    |        |

An diesem Beispiel wird ein wichtiges Prinzip beim Aufbau der Wertausdrücke deutlich. Der Wert VALUE im spec-Feld eines Atomarausdrückes wird wenn möglich durch eine zusätzliche Funktion überschrieben. Dies spart Atomarausdrücke, weil jeweils sowohl die Referenzfelder (cond, left, right) als auch die Wertfelder (minval, eqval, maxval) benutzt werden. Eine graphische Darstellung des resultierenden "Baums" verdeutlicht dies:

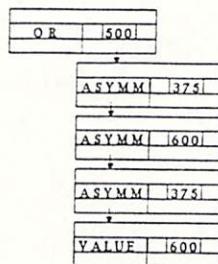


Abb. 8.13b: Gaphische Darstellung der Atome

Das folgende Beispiel zeigt eine bedingte Design Rule, deren Bedingung noch aus einer Verknüpfung zweier Terme besteht. Jeder dieser Terme entspricht syntaktisch einem *forward* und wird auch über dieselben Regeln abgeleitet. Der Parser erkennt am Wert der globalen Variable `expr_depth`, ob es sich um ein einfaches *forward* (gleiches gilt für eine *relation*) oder um Bedingungen handelt:

```

1. S.73 unten
; bedingte Design Rule mit OR-verknüpften Bedingungen
;
PC_Trans.Diff width
[ if or PC_Trans.Diff area [ < 28 ]
  Gate.Poly length [ > 3 ]
  then >= 3
]
  
```

Der Parser arbeitet hier folgende Reduktionsschritte ab (gestrichelte Linien markieren nicht vollständig aufgetragene Pfade):

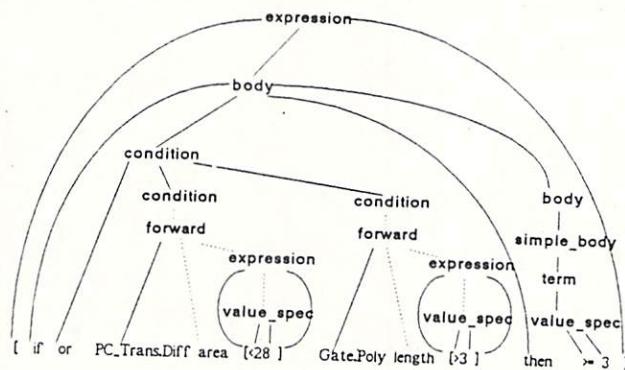


Abb. 8.14a: Syntaxbaum einer bedingten Design Rule

Wieder werden zunächst relationale Operatoren und Zahlenwerte zu Wertbestimmungen und diese schließlich zu `body` bzw. `expression` abgeleitet. Von den für die Bedingungen erzeugten Einträgen in der Objekttabelle werden die Ausdrücke "`[<28]`" und "`[>3]`" über das `expr`-Feld referenziert. In der Wertetabelle wird je ein Atomarausdruck mit `VALUE` und dem Objektindex im `cond`-Feld über einen Aufruf von `gen_expr` erzeugt. Diese beiden Atomarausdrücke werden bei der Ableitung zu `condition` über einen Atomarausdruck mit `COND_OR` im `spec`-Feld verknüpft. Dieser Ausdruck wird von einem Aufruf der Operation `m_COND_OR` erzeugt, die analog zu `m_OR` arbeitet. Der Satz `if condition then body` wird dann zu `body` und mit den umschließenden eckigen Klammern zu `expression` abgeleitet. Die Operation `m_C_PROPERTY` erzeugt den zugehörigen Atomarausdruck. Sie erhält die Ausdruckindizes der Bedingung, des `then`-Zweiges und bei Bedarf eines `else`-Zweiges und generiert über `gen_expr` einen neuen Atomarausdruck mit `C_PROPERTY` im `spec`-Feld, dem Index der Bedingung im `cond`-Feld, dem Index des `then`-Zweiges im `left`-Feld und u.U. dem `else`-Zweig im `right`-Feld. Es resultieren folgende Einträge in der Ausdruckstabelle sind (Die Wurzeln der Teilbäume sind jeweils mit "\*" gekennzeichnet; die mit "~" markierten Felder enthalten Indizes in die Objekttabelle):

| # | spec         | attrib | cond |       | minval |        |
|---|--------------|--------|------|-------|--------|--------|
|   |              |        | left | right | eqval  | maxval |
| * | 1 VALUE      |        |      |       |        | 28     |
| * | 2 VALUE      |        |      |       | 3      |        |
| 3 | VALUE        |        |      |       | 3      |        |
| 4 | VALUE        |        |      |       |        |        |
| 5 | VALUE        |        |      |       |        |        |
| 6 | COND_OR      |        | 4    | 5     |        |        |
| * | 7 C_PROPERTY |        | 6    | 3     |        |        |

Zur Verdeutlichung der Verkettungen auch hier die zugehörige graphische Darstellung:

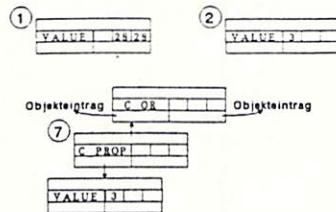


Abb. 8.14b: Darstellung der Atome

Das letzte Beispiel enthält zwei geschachtelte, bedingte Design Rules. Die innere Design Rule enthält eine spezielle Form von Bedingung, ein *classtest*. Mit diesem Konstrukt ist die Formulierung von Design Rules in Abhängigkeit von einer bestimmten Parameterbelegung einer Objektklasse möglich. Wie im vorherigen Beispiel wird auch für diese Form der Bedingung ein Eintrag in der Objektabelle angelegt. Seine Feldbelegung entspricht der eines deklarierten PARTs, nur enthält das spec-Feld den speziellen Wert CLASSTEST.

```

1. S.75 oben
; geschachtelt bedingte Design Rule mit AND-verknüpfter
; Bedingung und Classtest
;
dist
[ if and A_Wire width [ and >= 5 <= 10 ]
  P_Wire width [ > 10 ]
  then >= 5
  else if A_Wire (classtest Metall1 )
    then >= 7 else >= 3
] P_Wire A_Wire

```

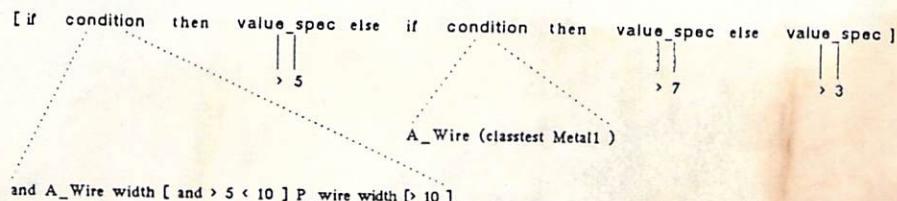


Abb. 8.15a: verkürzter Syntaxbaum einer geschachtelt bedingten Design Rule

Es resultieren folgende Einträge in der Wertetabelle (Die Wurzeln von Teilbäumen sind auch hier mit '\*' markiert):

| # spec         | attrib | cond |       | minval |        |
|----------------|--------|------|-------|--------|--------|
|                |        | left | right | eqval  | maxval |
| * 1 AND        |        |      | 2     | 5      | 5      |
| 2 VALUE        |        |      |       | 10     | 10     |
| * 3 COND_VALUE |        | 5    | 5     | 10     |        |
| 4 VALUE        |        |      |       | 10     |        |
| 5 VALUE        |        | 6    |       |        |        |
| 6 VALUE        |        |      |       | 5      | 5      |
| 7 CLASSTEST    |        | 7    |       |        |        |
| 8 VALUE        |        |      |       | 7      | 7      |
| 9 VALUE        |        |      |       | 3      | 3      |
| 10 C_PROPERTY  |        | 7    | 8     | 9      |        |
| 11 C_PROPERTY  |        | 3    | 6     | 10     |        |
| * 7 C_PROPERTY |        | 6    | 3     |        |        |

Die graphische Darstellung macht die Verkettungen wesentlich klarer:

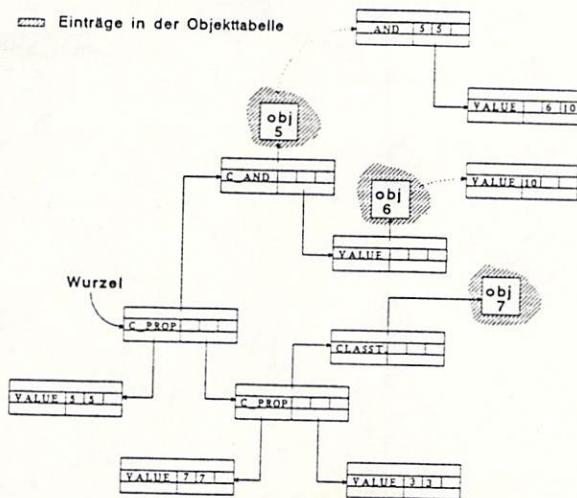


Abb. 8.15b: Darstellung der Atome

Wertausdrücke können nicht nur direkt beim Einlesen des DROOLY-Textes erzeugt werden. Erbt eine Komponente die Unterstrukturen seiner Klasse, so müssen zwischen den neu erzeugten Unterstrukturen dieselben geometrischen Einschränkungen wie in der Klasse generiert werden. Bei der Instanzierung einer Klasse werden also auch neue Wertausdrücke angelegt. Im einfachsten Fall handelt es sich um ein direktes Kopieren der einzelnen Atomarausdrücke, natürlich unter Beachtung der neuen Indizes. Bedingte Wertausdrücke enthalten jedoch zusätzlich Referenzen auf Objekte in der Objekttabelle. Auch diese Referenzen müssen

angepaßt werden. Schließlich werden bei der Instanziierung eines Klassenobjektes mit formalen Parametern die Klassen aller enthaltener PARAMs eindeutig festgelegt. Als Bedingung eines Wertausdruckes auftretende CLASSTESTs können bei der Übertragung in die Instanz aufgelöst und durch den jeweils zutreffenden Zweig ersetzt werden.

Die Vererbung eines Wertausdruckes wird im Objektmodul von der Operation `m_INHERIT_EXPR` angestoßen. Diese ruft die Operation `m_INHERIT` aus dem Ausdrucksmodul mit zwei Parametern. Als Empfänger wird der zu vererbende Ausdruck übergeben. Der zusätzliche Parameter übergibt die Objektinstanz, deren Unterstruktur den aktuellen Ausdruck erben soll. `M_INHERIT` gibt den Ausdrucksindex der erzeugten Kopie an `m_INHERIT_EXPR` zurück.

Die eigentliche Arbeit wird von der Funktion `copy_expr` erledigt, an welche `m_INHERIT` den alten Ausdruck und den Index der erbenden Instanz weiterreicht. `Copy_expr` erzeugt zunächst einen neuen Atomarausdruck, dessen Index am Ende als Wurzel der Kopie an die rufende Funktion zurückgegeben wird. Die Wertfelder (`minval`, `eqval`, `maxval`) des neuen Atomarausdruckes werden mit den entsprechenden Werten aus dem alten Ausdruck belegt, ebenso wie das `attrib`-Feld. Die Werte dieser Felder sind ja vom Index innerhalb der Ausdruckstabelle unabhängig. Für jedes der drei Referenzfelder (`cond`, `left`, `right`) wird anschließend ein eigener Zweig durchlaufen, falls der Wert des jeweiligen Feldes nicht NIL ist. In jedem dieser Zweige ruft sich `copy_expr` rekursiv mit dem Wert des Feldes als erstem und mit dem einfach weitergereichten Objektinstanz-Index als zweitem Parameter auf, um von den referenzierten Atomarausdrücken ebenfalls Kopien anzulegen. Die Indizes dieser Kopien werden als neue Referenzen in das `left`-, `right`- bzw. `cond`-Feld eingetragen. Für das `cond`-Feld ist bei einem `spec`-Wert von CLASSTEST, VALUE, COND\_OR oder COND\_AND allerdings eine Sonderbehandlung nötig, da in diesen Fällen das `cond`-Feld auch einen Objektindex enthält (CLASSTEST) bzw. enthalten kann.

In der Ausdruckskopie müssen alle Objektreferenzen nicht mehr auf Unterstrukturen des Klassenobjektes verweisen, sondern auf Unterstrukturen der Objektinstanz. Ein Teil des Pfadpräfixes der Klassenkomponenten ist dazu abzutrennen und durch den der Instanzindex zu ersetzen. Eine Schwierigkeit ergibt sich daraus, daß das Klassenobjekt nicht unbedingt immer ausschließlich von der ersten Pfadkomponente bestimmt wird. Handelt es sich bei der zu instanzierenden Klasse nämlich um ein lokales Modul, ist sein Pfad zwei- oder sogar mehrelementig und entsprechend müssen in einem solchen Fall die ersten beiden oder mehr Pfadkomponenten des Bedingungsobjektes ausgetauscht werden. Um entscheiden zu können, welche Pfadkomponenten ersetzt werden müssen, ordnet der Parser Objekten vom Typ C\_PROPERTY, C\_RELATION und CLASSTEST das den statischen Kontext bestimmende Objekt (aus der globalen Variable `act_class`) als Klasse zu. `Copy_expr` liest die Klasse aus dem betroffenen Objekt aus und stapelt die Suffizes des Pfades solange auf einem Stack, bis es auf diese Klasse trifft. Die alte Klasse wird durch den Index der aktuellen Objektinstanz ersetzt und der Pfad schrittweise mit `lookup` wieder aufgebaut.

Der letzte Aufruf von `lookup` liefert den Index derjenigen Komponente der Objektinstanz, deren Klassenzuweisung durch den bedingten Wertausdruck überprüft wird. Zur Erzeugung eines neuen CLASSTEST-Objektes wird `gen_obj_entry` mit Präfix und Suffix der Komponente aufgerufen. Dem dabei erzeugten neuen Objekt wird der `spec`-Wert CLASSTEST und der erste und zweite Klasseneintrag des ursprünglichen CLASSTEST-Objektes zugewiesen. CLASSTEST-Objekte werden nicht in die Komponentenhierarchie ihrer Klasse eingehängt.

Bei cond-Verweisen auf C\_PROPERTY- oder C\_RELATION-Objekte jedoch wird das vom letzten *lookup*-Aufruf gelieferte Objekt über *link\_object* in die Komponentenhierarchie eingehängt. Dort existiert allerdings schon das Objekt mit gleichem Präfix und Suffix wie das C\_PROPERTY- bzw. C\_RELATION-Objekt. Um ein individuelles, neues Objekt für die Bedingung zu erzeugen, wird daher *link\_object* derart aufgerufen (durch Setzen des Parameters *force\_entry*), daß es in jedem Fall einen neuen Objektindex zurückliefert und beide Objekte über das *next\_same*-Feld miteinander verkettet. Der Wertausdruck, den das C\_PROPERTY- bzw. C\_RELATION-Objekt trägt, wird zuletzt durch einen rekursiven Aufruf von *copy\_expr* kopiert.

Damit ist die Bearbeitung des Wertausdruckes aber noch nicht abgeschlossen. Handelt es sich nämlich um einen bedingten Wertausdruck, so können in der Bedingung enthaltene CLASSTESTs u.U. ausgewertet und dadurch die gesamte Bedingung zu TRUE oder FALSE reduziert werden. Die Reduktion einer Bedingung wird von der Funktion *reduce\_condition* vorgenommen. Diese erhält als einzigen Parameter einen Verweis auf die Wurzel der zu reduzierenden Bedingung. Ihr Rückgabewert ist der Index der (reduzierten) Bedingung. Konnte die Bedingung durch Auswertung der enthaltenen CLASSTESTs vollständig berechnet werden, hat der zurückgegebene Atomarausdruck einen spec-Wert von CONST und einen eqval-Wert von entweder TRUE oder FALSE. In *copy\_expr* werden diese Felder nach der Rückkehr von *reduce\_condition* überprüft. Fand sich tatsächlich ein konstanter TRUE- oder FALSE-Wert, kann die Wurzel des bedingten Wertausdruckes durch *set\_unref* freigegeben werden und statt ihrer mit dem then-Zweig (bei TRUE) bzw. mit dem else-Zweig (bei FALSE) fortgefahren werden.

*Reduce\_condition* liefert die übergebene Bedingung unverändert zurück, falls sie keine CLASSTESTs enthält. Eine Bedingung kann folgende spec-Werte haben, für die jeweils unterschiedliche Aktionen durchzuführen sind:

- **VALUE.** Die Bedingung besteht aus einem einzelnen Verweis auf ein C\_PROPERTY oder C\_RELATION-Objekt. *Reduce\_condition* gibt die Bedingung unverändert zurück.
- **C\_PROPERTY.** Eine Bedingung ist als Teil einer Bedingung nicht zulässig. Dieser Fall wird mit einer Fehlermeldung quittiert und die Übersetzung abgebrochen.
- **CLASSTEST.** Dies ist der elementar auswertbare Fall. Zur eigentlichen Überprüfung der Klassenzuweisung des betroffenen PARAMs wird die Funktion *eval\_classtest* aus dem Objektmodul gerufen. Diese kann die Klasse des überprüften Objekteintrages in jedem Fall eindeutig bestimmen, weil bei der Instanziierung einer Klasse alle formalen Parameter durch entsprechende aktuelle Parameter belegt sein müssen und daher jedes PARAM in der Klasse eine eindeutig bestimmte Klassenzweisung besitzt. *Eval\_classtest* liefert je nach Ergebnis des Klassenvergleichs entweder TRUE oder FALSE. Die ursprüngliche Bedingung wird über *set\_unref* freigegeben und stattdessen ein neuer Atomarausdruck mit dem spec-Wert CONST und dem eqval-Wert TRUE bzw. FALSE erzeugt.
- **COND\_AND / COND\_OR.** Besteht die Bedingung aus einem booleschen Ausdruck, bestehen folgende Möglichkeiten:  
*somit*
  - Das cond-Feld ist unbelegt. Am betrachteten Atomarausdruck hängen somit zwei Zweige, die über das left- bzw. right-Feld referenziert sind.
  - Das cond-Feld verweist auf einen Objekteintrag vom Typ C\_PROPERTY oder C\_RELATION. Der andere Operand des COND\_AND bzw. COND\_OR hängt dann am right-Feld.

Entsprechend dieser möglichen Belegungen werden nacheinander die über die `left`- und `right`-Felder referenzierten Teilausdrücke untersucht und das jeweilige Ergebnis in zwei Variablen `left_result` und `right_result` gespeichert. Diese Variablen können einen der drei Werte TRUE, FALSE oder FAILED annehmen, wobei sie mit FAILED initialisiert werden. Hat der jeweilige Zweig den `spec`-Wert CONST, so wird der zugehörigen Variablen der Wert des `eqval`-Feldes (TRUE oder FALSE) zugewiesen. Handelt es sich bei dem Zweig um einen CLASSTEST, wird `eval_classtest` aufgerufen und dessen Rückgabewert (TRUE oder FALSE) der Variablen `right_result` zugewiesen.

Es folgt die Auswertung von `left_result` und `right_result`. Haben beide den Wert TRUE, ist unabhängig von `COND_AND` oder `COND_OR` der Wert der gesamten Bedingung ebenfalls TRUE und ein entsprechend initialisierter Atomarausdruck wird für die Rückgabe dieses Ergebnis initialisiert. Ähnliches gilt, falls beide Variablen den Wert FALSE haben. In jedem Fall ist dann das Ergebnis der gesamten Bedingung ebenfalls FALSE und der Atomarausdruck für die Rückgabe wird entsprechend initialisiert. Die Kombination TRUE/FALSE bzw FALSE/TRUE führt bei `COND_OR` und `COND_AND` zu unterschiedlichen Ergebnissen.

Bei `COND_OR` wird ein Atomarausdruck mit CONST TRUE als Ergebnis von `reduce_condition` zurückgegeben.

`COND_AND` dagegen liefert für die Kombinationen TRUE/FALSE- bzw. FALSE/TRUE einen Atomarausdruck, der mit CONST FALSE belegt ist. Ist einer der Zweige TRUE, der andere dagegen noch nicht eindeutig ausgewertet, wird der `COND_AND`-Atomarausdruck gelöscht und das Ergebnis eines rekursiven Aufrufs von `reduce_condition` mit dem jeweils unaufgelöstem Zweig als Ergebnis zurückgegeben. Sind schließlich beide Zweige nicht eindeutig auswertbar, bleibt der ursprüngliche Wurzel-Atomarausdruck erhalten, seine Zweige sind die Ergebnisse von rekursiven `reduce_condition`-Aufrufen.

Im Beispiel aus Abschnitt 8.7 enthält das Gate-Modul eine einfache CLASSTEST-Bedingung (vgl. auch Tabellen in Anhang C), welche durch Instanziierung mit aktuellen Parametern aufgelöst wird.

## 8.8 Topologievergleich compare.c

Die Operationen, die im Modul `compare.c` zusammengefaßt sind, arbeiten zusammen zur Erledigung einer einzigen Aufgabe: dem Vergleich zweier Objektstrukturen. Eigentliches Ziel dabei ist, korrespondierende Elemente der Komponentenhierarchien zu identifizieren und miteinander in der Objekttabelle zu verketten.

### Umgebung

Die Operationen zum Topologievergleich werden bei der Ableitung einer `merge`-Anweisung, der DROOLY-Entsprechung der Identifikation in DINGO-II nach folgenden Regeln angestoßen:

```
merge :          (merge merge_list
merge_list :      merge_inner_list ")
merge_inner_list : IDENTIFIER IDENTIFIER
                  | merge_inner_list IDENTIFIER
```

Wichtige Voraussetzung für den Vergleich der beiden Objektstrukturen ist, daß sie bereits vollständig mit allen Deklarationen und RELATIONS aufgebaut sind. In DINGO-II und deshalb auch in DROOLY darf eine Identifikationsanweisung jedoch an beliebiger Stelle im Beschreibungsteil eines Objektes stehen, um zwei seiner Unterstrukturen miteinander zu identifizieren. Würde diese Anweisung sofort an dieser Stelle zum Vergleich der beiden Topologien führen, könnten später noch eingeführte RELATIONS nicht mehr berücksichtigt werden. Um dies zu vermeiden, werden die Indizes von an einer Identifikationsanweisung beteiligten Objekten zunächst paarweise auf eine temporäre Datei gesichert. Ist die Bearbeitung des DROOLY-Textes abgeschlossen, wird für jedes gespeicherte Objektpaar die Funktion *merge* mit den Indizes der beiden Objekte als Parameter aufgerufen.

### Datenstrukturen

Um die gewonnenen (Zwischen-) Ergebnisse den beiden Objekten und ihren Unterstrukturen einfach zuordnen zu können, bedienen sich die Funktionen dieses Moduls des Zeigerfeldes *user* in der Objektstruktur. Das Objektmodul stellt zwei Zugriffsoperationen *puser* und *guser* bereit, um diesen Zeiger zu schreiben bzw. zu lesen. Zur Speicherung der Korrespondenzen wird ein Listenknoten *corrT* verwendet, der ein Feld *value* und einen Zeiger *next* auf eine *corrT*-Struktur enthält. Variable von diesem Typ werden im compare-Modul beim Auffinden einer neuen Korrespondenz zwischen zwei Objekten angelegt, und bilden für jede Unterstruktur des ersten Objektes eine Liste von korrespondierenden Objekten in der Komponentenstruktur des zweiten Objektes.

Die Vektoren *lmerges* und *rmerges* in der Objektstruktur sowie ihre zugehörigen Zähler *lmerge\_cnt* und *rmerge\_cnt* werden erst nach Abschluß der eigentlichen Vergleiche in *merge* durch Aufruf der Funktion *add\_merges* aus dem Objektmodul in beiden Objekten gesetzt. Die Korrespondenzlisten werden anschließend nicht mehr benötigt und können wieder freigegeben werden.

### Struktur

Bei der Abarbeitung der Identifikationsanweisungen durch Einlesen des Temporärfils wird jeweils die Funktion *merge* aufgerufen. Nacheinander werden von ihr die Funktionen *compare\_decl* zur Herstellung von Korrespondenzen zwischen deklarierten Unterstrukturen und anschließend *compare\_relat* zur Verifizierung dieser Korrespondenzen durch Vergleich der auf den deklarierten Komponenten definierten RELATIONS aufgerufen. *Compare\_relat* wird allerdings nur gerufen, falls *compare\_decl* überhaupt irgend welche Korrespondenzen gefunden hat. *Compare\_decl* liefert true als Ergebnis, falls es zu jeder Unterstruktur des ersten Objektes mindestens eine korrespondierende Unterstruktur im zweiten Objektes gefunden hat, anderenfalls liefert es false. Konnte auch *compare\_relat* seine Arbeit erfolgreich beenden, liefert es ebenfalls den Wert true als Erbenis zurück. *Merge* durchläuft daraufhin mithilfe des Generators (*m\_GPREFIRST*, *m\_GPRENEXT*) alle Komponenten des ersten Objektes und ordnet durch Aufruf von *add\_merges* der jeweiligen Komponente im ersten Objekt ihre Identifikation im zweiten Objekt zu und umgekehrt.

*Compare\_decl* und *compare\_relat* selbst rufen eine Reihe weiterer Funktionen zur Erledigung bestimmter Teilaufgaben auf. Die gesamte Aufrufstruktur lässt sich am besten mithilfe einer Graphik darstellen:

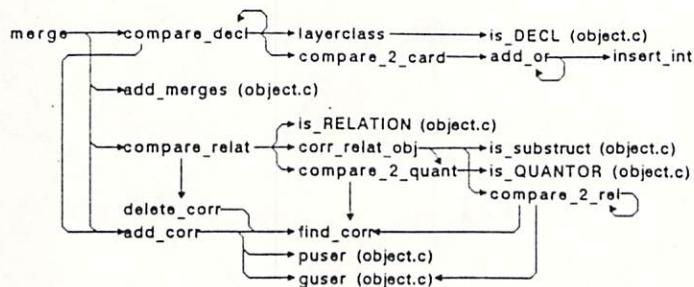


Abb. 8.16: Die Struktur des *compare*

### Verfahren

Der erste Schritt beim Aufbau der Identifikationen zwischen den einzelnen Unterstrukturen ist der Vergleich der deklarierten Komponenten beider Objekte durch die Funktion *compare\_decl*. Ziel dabei ist der Aufbau von 1-zu-n-Korrespondenzen zwischen je einer Komponente des ersten Objektes und einer Anzahl möglicherweise korrespondierender Komponenten des zweiten Objektes (10). Der zweite Schritt, der Vergleich der RELATIONS in beiden Objekten, löscht diejenigen Korrespondenzen, deren RELATIONS "nicht zueinander passen", so daß bei erfolgreicher Identifikation nur noch 1-zu-1-Korrespondenzen übrigbleiben.

Der Vergleich der deklarierten Unterstrukturen durch *compare\_decl* beginnt mit der Überprüfung, ob es sich bei beiden Objekten um Layer handelt. Ist nur eines der beiden Objekte ein Layer, kann ihre Struktur nicht übereinstimmen, *compare\_decl* liefert sofort false zurück. Sind beide Objekte Instanzen der gleichen Layerklasse, werden bei SET-Deklarationen die erlaubten Kardinalitätsbereiche durch die Funktion *compare\_2\_card* überprüft. Obwohl diese Funktion nur Warnungen ausgibt und die Identifikation nur in dem Fall abbricht, daß sie auf eine nicht korrekt aufgebaute Kardinalitätsangabe stößt, muß doch einiger Aufwand getrieben werden, um alle in DINGO-II gegebenen Möglichkeiten der Kardinalitätseinschränkung zu berücksichtigen. Aus den Möglichkeiten in DINGO-II ergibt sich in DROOLY folgende Syntax für Kardinalitätseinschränkungen:

```

card_range :      '[' card_body ']'
card_body :       eq_spec
                | or_card
                | and ">=" INTEGER "<=" INTEGER
or_card :        or or_card eq_spec
                | or eq_spec
eq_spec :         "==" INTEGER
  
```

(10) Der Begriff *Korrespondenz* wird im folgenden anstelle von *Relation* benutzt, um eine Verwechslung mit den Topologierelationen zu vermeiden.

Für die Formulierung einer Kardinalitätseinschränkung bestehen in DROOLY folgende Möglichkeiten:

- Einzelne Wertbestimmung: [ == 5 ]
- Alternativliste: [ or or == 3 == 5 == 7 ]
- Halboffenes Intervall: [ >= 5 ]
- Geschlossenes Intervall: [ and >= 2 <= 5 ]

*Compare\_2\_card* beginnt mit der Analyse der Kardinalitätseinschränkung des ersten übergebenen Objektes. Anhand ihres **spec**-Feldes lassen sich drei Fälle unterscheiden, die später beim Vergleich mit der Kardinalitätseinschränkung des zweiten Objektes anhand der Belegung eines **Selektors** unterschieden werden können:

- **VALUE**: Entweder handelt es sich um eine einzelne Wertbestimmung (nur das **eqval**-Feld ist belegt) oder um ein halboffenes Intervall (eins der Felder **minval** bzw. **maxval** und das **eqval**-Feld sind belegt). Der Fall einer einzelnen Wertbestimmung soll im folgenden wie eine Alternativliste behandelt werden (**Selektor = 0**), der Fall eines halboffenen Intervalls bildet einen Spezialfall der Intervallbehandlung (**Selektor = 1**).
- **OR**: Es liegt eine Liste alternativer Kardinalitätseinschränkungen in Form eines ausschließlich mit or verknüpften Wertausdruckes vor. Die einzelnen Alternativen werden von einer speziellen Funktion **add\_or** aus dem Wertausdruck extrahiert und in einen Vektor in aufsteigender Reihenfolge eingesortiert. Der **Selektor** wird auf 0 gesetzt.
- **AND**: In diesem Fall liegt ein geschlossenes Intervall mit Untergrenze und Obergrenze vor. Dabei ist die eine Grenze direkt im **minval**- bzw. **maxval**-Feld des AND-Eintrages vermerkt, auf die andere Grenze wird mit dem **right**-Feld verwiesen. Die beiden Grenzen werden extrahiert und in den Variablen **lb1** (untere Grenze) und **ub1** (obere Grenze) gemerkt. Der **Selektor** wird auf 1 gesetzt.

Bei der Analyse der zweiten Kardinalitätseinschränkung werden wieder diese drei **spec**-Werte unterschieden. Alternativlisten werden in einen zweiten Vektor eingesortiert und Intervallgrenzen in zwei Variablen **lb2** und **ub2** gemerkt. Im Fall eines Intervalls wird dem alten Selektowert 2 hinzugefügt.

Der folgende Vergleich der so aufbereiteten Kardinalitätseinschränkungen wird vom resultierenden Wert des **Selektors** gesteuert:

**Selektor=0** (Einzelne(r) Wert(e) - Einzelne(r) Wert(e)). Die beiden Vektoren werden verglichen. Nur bei identischer Anzahl und gleicher Werte resultiert der Rückgabewert **true**. Alle anderen Fälle werden mit einer Warnung und dem Rückgabewert **false** quittiert.

**Selektor=1** (Intervall - Einzelne(r) Wert(e)).

**Selektor=2** (Einzelne(r) Wert(e) - Intervall). Diese beiden Fälle resultieren in jedem Fall in einem Rückgabewert **false**. Je nachdem, ob alle Alternativen im Intervall liegen oder nicht wird jedoch eine unterschiedliche Warnung ausgegeben.

**Selektor=3** (Intervall - Intervall). Die jeweiligen Intervallgrenzen werden verglichen (**lb1** mit **lb2**, **ub1** mit **ub2**). Sind die Intervalle identisch, ist der Rückgabewert **true**, sonst **false** bei entsprechender Warnung.

In der Funktion *compare\_decl* wird der Rückgabewert von *compare\_2\_card* ignoriert, d.h. mit Ausnahme der ausgegebenen Warnungen haben unterschiedliche Kardinalitätsvereinbarungen keine Auswirkungen auf den weiteren Verlauf der Identifikation. *Compare\_decl* meldet bei zwei Layern einfach aufgrund gleicher oder ungleicher Layerzugehörigkeit Erfolg (true) bzw. Nichterfolg (false) an *merge* zurück.

Handelt es sich bei den übergebenen Objekten nicht um Layer, werden ihre Unterstrukturvereinbarungen miteinander in zwei ineinander geschachtelten Schleifen miteinander verglichen. Aus dieser Struktur resultiert ein Laufzeitverhalten von Komponentenzahl (obj1) mal Komponentenzahl (obj2). Der Vergleich verläuft nach folgendem Schema:

```

while (noch weitere Komponenten compl des ersten Objektes
      and wenigstens eine Korrespondenz für die aktuelle
          Komponente gefunden)
{
    while (noch weitere Komponenten comp2 des zweiten Objektes)
    {
        rufe rekursiv compare_decl mit den aktuellen Komponenten
        if (erfolgreich)
        {
            merke "mindestens eine Korrespondenz gefunden"
            füge comp2 zur Korrespondenzliste von compl hinzu
        }
        nächste Komponente comp2 des zweiten Objektes
    }
    nächste Komponente compl des ersten Objektes
}
Ergebnis von compare_decl ist true, falls zu jeder Komponente
compl mindestens eine Komponente comp2 mit gleicher
Unterkomponentenstruktur gefunden wurde; sonst ist das
Ergebnis false

```

Falls *compare\_decl* erfolgreich eine 1-zu-n-Korrespondenz zu jeder deklarierten Unterstruktur des ersten Objektes herstellen konnte, fügt *merge* das zweite Objekt der Korrespondenzliste des ersten Objektes hinzu und ruft *compare\_relat*, um durch Vergleich der RELATIONS in beiden Objekten die Korrespondenzen zu 1-zu-1-Korrespondenzen zu reduzieren und dadurch eine eindeutige Identifikation herbeizuführen.

*Compare\_relat* vergleicht die RELATIONS zwischen den mit *compare\_decl* gewonnenen Korrespondenzen, und streicht dabei Korrespondenzen aus der Korrespondenzliste des ersten Objektes. Im Pseudocode für *compare\_relat* werden folgende Variablen verwandt:

c11 wird nacheinander mit allen Komponenten des 1. Objektes belegt  
c21 alle zu c11 korrespondierenden Komponenten des 2. Objektes  
r1 eine RELATION im 1. Objekt mit c11 und einem weiteren Operanden  
r2 eine RELATION im 2. Objekt mit c21 und einem weiteren Operanden:

```

for (alle Unterstrukturen und
      Unterunterstrukturen c11 des 1. Objektes)
{
    if (c11 ist RELATION)
    {
        rufe corr_relat_obj zur Sonderbehandlung von RELATIONS
        weiter mit nächstem c11 (äußerste for-Schleife)
    }
    while (alle Korrespondenzen c21 von c11)
    {
        setze "mindestens ein r2 gefunden"
        while (alle RELATIONS r1, die sich auf c11 beziehen
                 and mindestens ein r2 gefunden)
        {
            setze "noch nicht mindestens ein r2 gefunden"
            while (alle RELATIONS r2, die sich auf c21 beziehen)
            {
                if (die zur Ermittlung der Korrespondenz der RELATIONS
                      r1 und r2 gerufene Funktion compare_2_relat
                      liefert true)
                {
                    if (die r1 und r2 beherrschenden Quantoren
                          werden von der Funktion compare_2_quant
                          als übereinstimmend markiert)
                    {
                        setze "mindestens ein r2 gefunden"
                    }
                    else
                        Fehler: Quantorketten nicht identisch
                    }
                    nächste RELATION r2
                }
                nächste RELATION r1
            }
            if (nicht mindestens ein r2 gefunden)
            {
                da keine korrespondierende RELATION r2 zu r1 gefunden,
                lösche Korrespondenz zwischen c11 und c21
            }
            nächste Korrespondenz c21
        } /* c21 */
    } /* c11 */

/* Feststellung der Eindeutigkeit: */
setze Rückgabewert auf true
for (alle Unterstrukturen und Unterunterstrukturen c11
      des 1. Objektes)
{
    if (c11 ist SET_RELATION)
    {
        überspringe dieses c11 (SET_RELATIONS haben keine
        Korrespondenzen)
    }
}

```

```
if (Korrespondenzliste von c11 ist leer (1-zu-0-Relation)
    or
        --"-
            enthält mehr als ein
                Element (1-zu-n-Relation))
{
    Fehler: Korrespondenzen zu c11 nicht eindeutig
    setze Rückgabewert auf false
}
}

Funktionsergebnis von compare_relat ist
der resultierende Rückgabewert
```

Wie man dem Pseudocode entnehmen kann, ruft *compare\_relat* (neben einigen nicht erwähnten Hilfsfunktionen) die drei Funktionen *corr\_relat\_obj*, *compare\_2\_rel*, und *compare\_2\_quant* auf.

*Corr\_relat\_obj* erledigt die Sonderbehandlung für RELATIONS. Eine übergebene RELATION *r1* aus dem ersten Objekt wird mithilfe der Funktionen *compare\_2\_rel* und *compare\_2\_quant* mit allen RELATIONS aus dem zweiten Objekt verglichen.

*Compare\_2\_rel* überprüft dabei zunächst, ob zwischen den beiden übergebenen RELATIONS *r1* aus dem ersten Objekt und *r2* aus dem zweiten Objekt eine Korrespondenz besteht. In diesem Fall ist *compare\_2\_rel* fertig und liefert TRUE zurück (11). Es folgt ein etwas länglicher, rekursiver Vergleich der beiden RELATIONS anhand ihrer Operanden und Operatoren, der sich auch hier am klarsten durch seinen Pseudocode beschreiben lässt (es werden Variablenbezeichner mit denselben Bedeutungen wie bei *compare\_relat* benutzt):

(11) Zur Unterscheidung vom booleschen Wert true ist dieser Rückgabewert Element der Wertemenge { TRUE, FALSE, FAILED }.

```
/* Überprüfung der jeweils ersten Operanden c11 von r1 und c21
   von r2: */
if (c11 ist RELATION)
{
    if (c21 ist keine RELATION)
    {
        Funktionsergebnis ist FALSE
        return
    }
    /* sowohl c11 als auch c21 sind RELATIONS: */
    if (c11 und c21 korrespondieren nicht (ermittelt durch
        rekursiven Aufruf von compare_2_rel))
    {
        Funktionsergebnis ist FALSE bzw. FAILED, je nach Ergebnis
        des rekursiven Aufrufes
        return
    }
    /* c11 und c21 sind korrespondierende RELATIONS: */
    füge c21 zur Korrespondenzliste von c11 hinzu
} /* c11 ist RELATION */
else
{
    /* c11 ist deklariert: */
    if (c11 ist nicht Unterstruktur des ersten Objektes)
    {
        Funktionsergebnis ist FALSE bzw. FAILED, je nachdem
        ob c21 Unterstruktur des zweiten Objektes ist oder nicht
        return
    }
    if (zwischen c11 und c21 besteht keine Korrespondenz)
    {
        Funktionsergebnis ist FALSE
        return
    }
} /* c11 ist deklariert */

/* Überprüfung der jeweils zweiten Operanden c12 von r1 und
   c22 von r2: */
if (c12 ist RELATION)
{
    if (c22 ist keine RELATION)
    {
        Funktionsergebnis ist FALSE
        return
    }
    /* sowohl c12 als auch c22 sind RELATIONS: */
    if (c12 und c22 korrespondieren nicht (ermittelt durch
        rekursiven Aufruf von compare_2_rel))
    {
        Funktionsergebnis ist FALSE bzw. FAILED, je nach Ergebnis
        des rekursiven Aufrufes
        return
    }
    /* c12 und c22 sind korrespondierende RELATIONS: */
    füge c22 zur Korrespondenzliste von c12 hinzu
} /* c12 ist RELATION */
```

```

else
{
    /* c12 ist deklariert: */
    if (c12 ist nicht Unterstruktur des ersten Objektes)
    {
        Funktionsergebnis ist FALSE bzw. FAILED, je nachdem
        ob c22 Unterstruktur des zweiten Objektes ist oder nicht
        return
    }
    if (zwischen c12 und c22 besteht keine Korrespondenz)
    {
        Funktionsergebnis ist FALSE
        return
    }
} /* c12 ist deklariert */

if (Operatoren ungleich)
{
    Funktionsergebnis ist FALSE
    return
}
/* da wir bis hierher gekommen sind, hat sowohl c11
   als auch c12 aus dem ersten Objekt eine Korrespondenz
   im zweiten Objekt: */
füge r2 zur Korrespondenzliste von r1 hinzu
Funktionsergebnis ist TRUE

```

Die dritte große, von *compare\_relat* gerufene Funktion ist *compare\_2\_quant*. Sie vergleicht zu zwei gegebenen Outermost-RELATIONS *r1* und *r2* die Quantorketten. Beim Vergleich verläßt sich die Funktion darauf, daß die Quantoren in einer RELATION die äußersten Objekte darstellen. Die DINGO-II-Syntax ebenso wie die DROOLY-Syntax läßt jedoch keine anderen Konstruktionen zu, so daß dies keine Einschränkung darstellt. Der Vergleich der beiden Quantorketten kann also abgebrochen werden, sobald es sich bei der jeweils am rechten Operanden hängenden RELATION nicht mehr um eine SET\_RELATION handelt. *Compare\_2\_quant* liefert true, falls beide Quantorketten gleich sind, sonst liefert sie false.

Das Modul compare.c enthält noch eine Reihe weiterer Funktionen, die zum Aufbau, Löschen oder Suchen von Korrespondenzen dienen oder die Listen von alternativen Kardinalitätseinschränkungen verwalten. Die Aufrufe dieser Funktionen sind in der Aufrufstruktur zu Beginn dieses Abschnittes dargestellt. Ihre Funktion soll jedoch nicht weiter beschrieben werden, da es sich durchweg um die einfache Verwaltung linearer Listen handelt.

Nach erfolgreicher Rückkehr aus der Funktion *compare\_relat* bleibt für *merge* noch als Aufgabe, die gewonnenen und durch Überprüfung in *compare\_relat* garantiert eindeutigen Korrespondenzen aus den Verweisen über die *user*-Felder des ersten Objektes in die *rmerges*-Vektoren des ersten Objektes und seiner Unterstrukturen sowie in die *lmerges*-Vektoren des zweiten Objektes und seiner Unterstrukturen zu übertragen und die *user*-Felder anschließend wieder freizugeben. *Merge* ruft dazu für das erste Objekt, für alle seine Unterstrukturen und deren Unterunterstrukturen die Funktion *add\_merges* aus dem Objektmodul auf. *Add\_merges* erledigt die genannten Aktualisierungen der *rmerges*- bzw. *lmerges*-Vektoren. In einem zweiten Durchlauf über alle Unter- und Unterunterstrukturen des ersten Objektes werden die *user*-Felder wieder freigegeben.

### 8.9 Erweiterte semantische Überprüfungen validate.c

Im normalen Übersetzungsvorgang wird die Semantik der Topologieoperatoren nicht zu Überprüfungen herangezogen. Durch Setzen der Option -v beim Aufruf des Compiler-Backends ist es jedoch möglich, auch derartige Überprüfungen durchzuführen. Bei jedem Eintrag einer neuen Topologierelation in die Objekttabelle wird dann überprüft, ob die gewünschte Relation zwischen den Operanden nicht einer bereits in der Tabelle enthaltenen Relation widerspricht. Widersprüchliche Relationen werden mit einer Fehlermeldung zurückgewiesen. Zusätzlich werden neben der einen geforderten auch alle solchen Topologierelationen in die Objekttabelle eingetragen, die automatisch aus der geforderten Relation folgen.

Die Überprüfung findet im Modul validate.c statt. Sie beruht auf eingebauten Tabellen, die resultierende bzw. verbotene Relationen zwischen den Operanden einer Topologierelation verzeichnen. Der erste Schritt legt eine einfache Topologierelation zugrunde, die aus zwei Flächenelementen a und b und einem Operator besteht. Dabei spielt es keine Rolle, ob die Flächenelemente Instanzen von deklarierten Strukturen oder Resultate anderer Topologierelationen sind. Die folgende Tabelle benennt jeweils zu einer gegebenen Relation diejenigen Relationen,

- die nicht erlaubt sind (" "),
- die erlaubt aber nicht unbedingt gültig sind ("e") bzw.
- die in jedem Fall gültig sind ("x"):

|         | indist | cross | overlap | enclose | touch | dist |
|---------|--------|-------|---------|---------|-------|------|
| indist  | x      | e     |         |         |       |      |
| cross   |        |       | x       |         |       |      |
| overlap |        |       |         | x       |       |      |
| enclose |        |       |         |         | e     | e    |
| touch   |        |       |         | e       | e     | x    |
| dist    |        |       |         | e       | e     | x    |

Abb. 8.17: Valitabl: Relationen zwischen zwei Flächenelementen

In den Spalten für INDIST und ENCLOSE muß zusätzlich berücksichtigt werden, daß die Relationen nicht symmetrisch sind, so daß hier unterschiedliche Fälle für a\*b und b\*a zu berücksichtigen sind. Beispielsweise ist bei einer gegebenen Relation "a INDIST b" die Umkehrung "b INDIST a" nur möglich, wenn a und b die gleiche Flächenrepräsentation haben.

In einem zweiten Schritt werden die Verhältnisse zwischen den Flächenelementen und der resultierenden Fläche x untersucht. Die Relationen  $a^*x$  bzw.  $x^*a$  und  $b^*x$  bzw.  $x^*b$  werden in zwei getrennten Tabellen untersucht:

| neu →   | indist | cross | overlap | enclose | touch | dist |
|---------|--------|-------|---------|---------|-------|------|
| $a^*b$  | $a^*a$ |       |         | $a^*e$  |       |      |
| indist  | x      |       |         |         |       |      |
| cross   |        | x     |         |         |       |      |
| overlap |        | x     |         |         |       |      |
| enclose |        |       | x       | e       |       |      |
| touch   |        |       | e       | e       | x     |      |
| dist    |        |       |         | e       |       |      |

| neu →   | indist | cross | overlap | enclose | touch | dist |
|---------|--------|-------|---------|---------|-------|------|
| $a^*b$  | $b^*b$ |       |         | $b^*b$  |       |      |
| indist  | e      | x     |         |         |       |      |
| cross   |        | x     |         |         |       |      |
| overlap |        | x     |         |         |       |      |
| enclose | x      |       |         | x       | e     |      |
| touch   |        |       |         | e       | e     | x    |
| dist    |        |       |         | e       |       |      |

Abb 8.18: Valitab2: Relationen zwischen Flächenelementen und Resultatflächen

In einem dritten Schritt werden Topologierelationen aus drei Flächenelementen a, b und c und zwei (verschiedenen) Operatoren x und y untersucht. In der Tabelle werden bei einer Klammerung nach  $(axb)yc$  Relationen zwischen den Flächenelementen a und c bzw. b und c aufgeführt, bei einer Klammerung nach  $ax(byc)$  werden Relationen zwischen a und b bzw. zwischen a und c aufgeführt. Auch in diesem Schritt lassen sich wiederum die Relationen zwischen den Flächenelementen und den Resultatflächen untersuchen.

Würde die Untersuchung in dieser Weise fortgesetzt, müßten immer umfangreichere Tabellen konsultiert werden. Erst ein rekursives Verfahren zur gleichmäßigen Behandlung beliebiger Schachtelungstiefen kann dieses Problem lösen. Im Rahmen dieser Arbeit soll dieser Weg jedoch nicht weiter verfolgt werden.

Implementiert ist lediglich eine Überprüfung nach der ersten Stufe. Ist beim Aufruf des Compiler-Backends die -v - Option gesetzt, ruft der Parser nach dem Eintragen einer neuen Relation die Funktion `valid_topo` mit dem neu eingetragenen Relationsobjekt als Parameter auf. `Valid_topo` extrahiert daraus den Operator und die beiden Operanden. Unter Verwendung des Generatorpaars (`m_GFIRSTREL`, `m_GNEXTREL2`) werden zum ersten Operanden alle Relationen zum zweiten Operanden überprüft. Durch Konsultierung einer Tabelle gemäß Abb. 8.17 wird jeweils diese Relation und die als Parameter übergebene Relation überprüft. Gemäß dem Tabelleneintrag können drei Fälle auftreten:

1. Die Relation widerspricht der neuen Relation. Eine Warnung wird ausgegeben.
2. Die Relationen können nebeneinander bestehen. Keine Aktion ist notwendig.
3. Die Relation gehört zu der Klasse von Relationen, die in jedem Fall zusammen mit der neuen Relation gelten. Ihre Existenz wird beim anschließenden "Auffüllen" dieser Klasse berücksichtigt.

Im folgenden werden diejenigen Relationen erzeugt, die in jedem Fall zusammen mit der neuen Relation gelten. Für jeden der Operatoren INDIST, CROSS, OVERLAP, ENCLOSURE, TOUCH und DIST wird eine Relation erzeugt, falls nicht eine solche schon besteht (Fall 3). Wird die neue Relation von Quantoren beherrscht, werden die zugehörigen SET\_RELATIONS aus der neuen Relation übernommen. Die neu erzeugte

Relation wird in diesem Fall an das op2-Feld der innersten SET\_RELATION angehängt.

## 9. Das Technologieinterface für Applikationen

Nach Beendigung des Übersetzungsvorganges liegt die Technologiebeschreibung in einer Binärdatei vor. Um von einer Applikation auf diese zugreifen zu können, benötigt sie Operationen zum Einlesen der Datei in interne Strukturen und zum anschließenden Zugriff auf diese Strukturen. Object- und Wertausdrucksmodul stellen derartige Operationen zur Verfügung. Werden bei der Erstellung einer Applikation also die Module object.c und expr.c zur ausführbaren Datei hinzugebunden, kann die Technologiedatei durch einen Aufruf der Funktion `read_tec` eingelesen werden. Diese Operation liest die gespeicherten Daten zu den einzelnen Technologieobjekten und ihren anhängenden Wertausdrücken ein und rekonstruiert die Objekt- und Ausdruckstabellen in der Form, wie sie vom Compiler-Backend angelegt wurden. Der Applikation stehen dann alle Operationen des Objekt- und des Ausdrucksmoduls zur Verfügung, um auf diese Tabellen lesend und auch manipulierend zuzugreifen.

Beim Compilieren der Module object.c und expr.c zur Einbindung in eine Applikation müssen bestimmte Schalter gesetzt sein. Ist nämlich die `-DTRACE`-Option gesetzt, werden Objektreferenzen in symbolischer Form ausgegeben. Dazu wird allerdings eine Funktion aus dem Symboltabellen-Modul benötigt, welches bei gesetzter `-DTRACE`-Option also ebenfalls an die Applikation angebunden werden muß. Ist weiterhin die Option `-DTEVAL` zur Erzeugung einer Codeabdeckungsstatistik gesetzt, werden Funktionen aus dem teval.c-Modul benötigt, welches dann zusammen mit den Funktionstabellen im Modul tv\_tab.c ebenfalls an die Applikation angebunden sein muß. Die Ausgabe von Fehlermeldungen schließlich erfolgt in allen Modulen ausschließlich über die Funktion `error` aus dem Modul error.c. Soll dieses Modul nicht ebenfalls an die Applikation angebunden werden müssen, muß in jedem Fall eine `error`-Funktion zur Verfügung gestellt werden. Die den Fehlernummern entsprechenden symbolischen Konstanten finden sich in der Kopfdatei const.h, die zugehörigen Texte der Fehlermeldungen stehen in der Datei error.h (siehe Anhang B).

## 10. Beispiel für eine technologieinvariante Applikation: Ein Bauelement-Extraktor

In diesem Abschnitt soll anhand eines Beispiels die Anwendung der tabellarischen Technologiebeschreibung verdeutlicht werden. Bei der gewählten Applikation handelt es sich um einen technologieinvarianten Bauelementextraktor ähnlich dem Extraktor MONDRIAN, der im Rahmen einer Projektgruppe an der Universität Dortmund entwickelt wurde [MONDRIAN].

MONDRIAN erhält seine Technologieinformationen aus einer Datei in der Sprache DINGO-I. Die zu extrahierenden Objekte werden in einer speziell für diesen Anwendungszweck entwickelten Abfragesprache SALOMON beschrieben. Das SALOMON-Programm wird in einen Extraktionsbaum übersetzt. Die Abarbeitung dieses Baums initiiert die einzelnen Extraktionsschritte, die zur Extraktion aller Objekte einer Anfrage erforderlich sind. Die Technologiebeschreibung wird ebenfalls in eine interne Darstellung, das Extraktions-Schema übersetzt, mit welchen für jedes Technologieobjekt die Aufrufreihenfolge der Extraktionsalgorithmen festgelegt wird. Die Abarbeitung eines Extraktions-Schemas und der Aufruf der Algorithmen ist Aufgabe des Technologieprozessors in MONDRIAN, des sog. DINGO-Managers.

Zentrale Steuerungskomponente MONDRIANS ist jedoch der SALOMON-Manager. Dieser veranlaßt zunächst das Einlesen des Layouts aus einer Datei im CIF-Format in die MONDRIAN-Datenstruktur. Anschließend wird der DINGO-Manager aufgerufen, um legale Technologieobjekte in dieser Datenstruktur zu finden und zu markieren. Der SALOMON-Manager extrahiert dann daraus unter Abarbeitung des Extraktionsbaums die geforderten Daten und liefert sie an den Ausgabeprozessor zur Formatierung und Ausgabe.

Es soll plausibel gemacht werden, wie die Objekttabelle und die Wertausdruckstabelle in die beschriebenen Abläufe einer Applikation wie MONDRIAN zu integrieren sind. In MONDRIAN ergäbe sich für die Extraktion eines Bauelementes unter Verwendung der in dieser Arbeit entwickelten Tabellen folgender Ablauf:

EINGABE: 1. Die Technologiespezifikation in Form von SymTb,  
ObjektTb und ExprTb  
2. Ein Layout und die Möglichkeit, neue Pseudolayer einzuführen

AUSGABE: Extrahierte Bauelemente in einem Pseudolayer

```
"hole Zeiger auf Objektbeschreibung in ObjectTb durch Zugriff
über den Symboltabellindex mithilfe von lookup"
"setze im user-Feld des Objekteintrages und aller seiner
Unterstrukturen (Generator m_PREFIRST, m_PRENEXT) eine
Markierung, daß diese Struktur noch nicht extrahiert ist"
"hole erste deklarierte Komponente mithilfe der
Generatoroperation m_GFIRSTCOMP"
while "Komponente nicht leer" do
begin
    if "Komponente ist PARAM"
        then "FEHLER: bei Bauelementen müssen alle Komponentenklassen
               eindeutig bestimmt sein"
    else
    begin
        if "Komponente noch nicht extrahiert (Markierung im user-
              Feld)"
            then "extrahiere diese Komponente durch rekursiven Aufruf
                  dieses Algorithmus"
            while "es gibt noch nicht extrahierte RELATIONS (über
                  m_NEXT_RELATION und die user-Markierung)" do
                "extrahiere die äußerste zu dieser RELATION gehörige
                 RELATION (m_GOUTERMOST) und markiere alle davon
                 abhängigen RELATIONS als extrahiert"
            end { PART oder SET }
            "hole mithilfe von m_NEXTCOMP nächste deklarierte
             Komponente"
        end { alle deklarierten Komponenten }
        "überprüfe alle PROPERTIES"
        "überprüfe alle Design Rules, die sich auf keine Unterstruktur
         beziehen"
```

In diesem Verfahren werden folgende Möglichkeiten beim Zugriff auf die Tabellen besonders genutzt:

- Generatoren für
  - alle deklarierten Komponenten
  - alle RELATIONS auf einer gegebenen Komponente
  - alle objektglobalen PROPERTIES
  - alle Unterstrukturen
- Möglichkeit, an jedes Objekt eine benutzerdefinierte Informationen anzuhängen, diese abzufragen und zu ändern (z.B. "ist bereits extrahiert", "Design Rule verletzt")

All diese Zugriffsarten werden in den Tabellenmodulen durch entsprechende Zugriffsoperationen ermöglicht. Besonders die Möglichkeit, beliebige eigene Strukturen über das user-Feld in die Objekttabelle einzuhängen, vereinfacht die Zuordnung applikationsspezifischer Informationen zu einzelnen Objekteinträgen.

## 11. Ausblick

In diesem Kapitel sollen drei Aspekte betrachtet werden, die in der Implementierung des DROOLY-Compilers nicht oder nur ansatzweise berücksichtigt wurden, welche aber für den Komfort des Technologie-Compilers bzw. für die Optimierung des Einsatzes der Technologiebeschreibung von Bedeutung sind und somit sinnvolle Erweiterungen darstellen.

Trotz des bereits jetzt erreichten Umfanges des implementierten Verfahrens zur Identifikation zweier Objekthierarchien sind noch Steigerungen im Komfort für den Benutzer möglich und sogar wünschenswert. Bisher bleiben die speziellen semantischen Eigenschaften der einzelnen Operatoren bei der Identifikation völlig unberücksichtigt. Eine Identifikation kann also nur dann erfolgreich verlaufen, wenn neben einer bis auf die Reihenfolge gleichen Struktur der Hierarchie deklarierter Unterstrukturen die Relationen auf diesen Unterstrukturen in beiden Objekten in gleichem Umfang und mit gleichen Operatoren vorhanden sind. Bei der Untersuchung der einzelnen Operationen fällt jedoch auf, daß sich einzelne Topologierelationen durch unterschiedliche Operatoren beschreiben lassen. Solcherart äquivalente Relationen könnten aber bei der Identifikation ebenfalls berücksichtigt werden.

Mit diesem Punkt im Zusammenhang steht auch die Suche nach einem allgemein anwendbaren Verfahren zur Validitätsüberprüfung topologischer Relationen. Das zur Zeit implementierte Verfahren eignet sich nur zur Überprüfung gering geschachtelter Topologierelationen. Darüber hinausgehende Überprüfungen erfordern eine intelligenter Tabellensstruktur, um noch handhabbar zu sein.

Als letzter wichter Punkt sei auf eine Optimierungsmöglichkeit hingewiesen. Bei der Erstellung der Tabellen werden quantifizierte Topologierelationen nicht optimiert. Zwei Ansatzpunkte für derartige Optimierungen aus dem klassischen Compilerbau lassen sich jedoch auch auf die Übersetzung von komplexen Topologieausdrücken anwenden [Aho 78]:

- Vereinigung gemeinsamer Teilausdrücke
- Herausziehen von Schleifeninvarianten

Zur Anwendung des zweiten Punktes faßt man einen quantifizierten Topologieausdruck als eine Schleife über alle Elemente der abhängigen Menge auf. Das Herausziehen von Invarianten vereinfacht sich durch den einfachen Aufbau dieser Schleifen und ihre Abhängigkeit von jeweils nur einer Variablen. Dem Optimierungsverfahren liegt der Aufbau eines gerichteten, azyklischen Graphen (DAG) zugrunde.

Als Vorbereitung ist es erforderlich, zunächst mehrfach auftretende, quantifizierte Mengenbezeichner durch neue, eindeutige Bezeichner zu ersetzen und die Quantoren entsprechend zu vervielfachen. Dieser Schritt ist erforderlich, damit jede Instanz einer Menge mit jeder anderen in Relation gesetzt werden kann. Dies ist gerade die Semantik einer Regel wie

some A  
enclose [ ] A A .

Da eine solche Regel eine Beziehung zwischen Elementen einer Menge A und nicht zwischen zwei Mengen A beschreibt, würde man die Regel mithilfe von Prädikaten folgendermaßen formulieren:

$$\exists a_1, a_2 \in A: \text{enclose}(a_1, a_2)$$

Nach einer derartigen Umformung ergibt sich z.B. für den komplexen Topologieausdruck von Seite 43 folgender neuer Ausdruck:

```

1 some A1
2 some A2
3   all C
4     overlap [ ]
5     indist [ ] C B
6     indist [ ]
7       enclose [ ] A1 A2
8       B

```

Mit dem in [Aho 78] auf den Seiten 418 ff. angegebenen Algorithmus zum Aufbau eines DAGs aus einem Dreiaadress-Code ergibt sich für diesen umgeformten Topologieausdruck der folgende DAG, in dem gemeinsame Teilausdrücke bereits zusammengefaßt sind (Die einzelnen Teilausdrücke sind durch ihre Zeilennummer identifiziert):

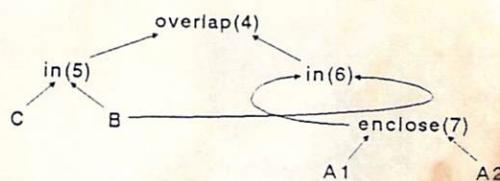


Abb. 11.1: Der DAG eines komplexen Topologieausdruckes

Zur Verlagerung von Schleifeninvarianten lässt sich ebenfalls dieser DAG verwenden, wenn die benutzte Datenstruktur Durchläufe von den Blättern zur Wurzel erlaubt. Für jeden Quantor wird ein neuer DAG aufgebaut, der die auf der Schachtelungstiefe der zugehörigen Schleife berechenbaren Teilausdrücke aufnimmt. Die Bearbeitung beginnt mit mit der abhängigen Variablen des innersten Quantors, im Beispiel ist das "all C". Der neu erzeugte DAG erhält als Namen diesen Quantor. Im ursprünglichen DAG werden von C aus alle Pfade zur Wurzel der Reihe nach durchlaufen. jeder besuchte Operator wird mit seinen direkten Söhnen in einen neuen DAG aufgenommen und aus dem alten gelöscht. Nach dem Durchlaufen des Pfades für "all C" ergibt sich folgendes Bild:

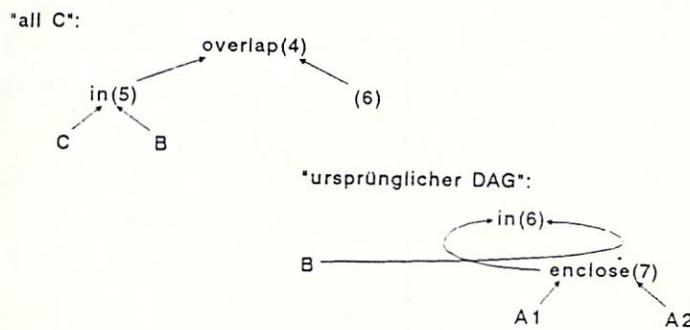


Abb. 11.2: Der DAG nach Durchlaufen des Pfades für "all C"

Das Durchlaufen des Pfades ab der abhängigen Variable des nächsten Quantors "some A2" liefert den gesamten DAG als neuen DAG mit der Bezeichnung "some A2". Für den Quantor "some A1" bleibt kein Operator mehr übrig. Im letzten Schritt der Übersetzung wird aus den erzeugten DAGs eine Tabellendarstellung erzeugt, wie sie bei direkter Übersetzung des folgenden DROOLY-Topologieausdruckes entstanden wären (die resultierende Zusammenfassung gemeinsamer Teilausdrücke ist durch Angabe der Zeilenummern als Operanden angedeutet):

```

1 some A1
2   some A2
3     indist [ ]
4       enclose [ ] A1 A2
5       B
6   all C
7     indist [ ] C B
8       overlap [ ] (7) (3)

```

## 12. Zusammenfassung

Ziel der vorliegenden Arbeit war die Realisierung eines Verfahrens zur Erzielung von Technologieinvarianz und dessen Integration in die Datenhaltung des Layout-CAD Systems IMAGE. Nach einer einleitenden Beschreibung der Aufgaben eines Layout-CAD Systems und seiner Bestandteile werden im 2. Kapitel verschiedene Ansätze der Technologieverarbeitung in der Literatur untersucht. Aus den vorhandenen Ansätzen lässt sich eine Hierarchie von möglichen Technologie-Spezifikationsmechanismen nach dem Grad ihrer Eignung für ein integriertes Layout-CAD-System ableiten:

1. **Formale Sprachen** zur Beschreibung aller relevanten Aspekte einer Technologie.  
Dieser Ansatz zeichnet sich durch folgende Merkmale aus:
  - a) **Flexibel.** Für verschiedene Applikationen relevante Technologieinformationen lassen sich gemeinsam in einer Spezifikation unterbringen und bei Bedarf erweitern, ohne das eine Anpassung bereits unterstützter Applikationen notwendig ist.
  - b) **Komplexe Zusammenhänge beschreibbar.** Durch die Verwendung von Schlüsselworten, Formatfreiheit und geeigneter syntaktischer Regeln können auch komplexe Abhängigkeiten übersichtlich beschrieben werden. Eine weitgehend dem Englischen angenäherten Syntax ist einem kryptischen Format vorzuziehen, da dadurch eine Technologiespezifikation sowohl vom Technologen als auch vom Designer leicht nachvollziehbar ist.
  - c) **Überprüfbar.** Da die Technologiespezifikation in einem textuellen Format vorliegt, ist sie einer manuellen Überprüfung unmittelbar zugänglich.
  - d) **Kommunizierbar.** Ein textuelles Format eignet sich besonders zur Übermittlung von Technologiedaten vom Chiphersteller zum Designer. Mißverständnisse lassen sich durch eine formale Beschreibung vermeiden; technologieabhängige (aber durch Verwendung einer expliziten Technologiebeschreibung technologieinvariante) Applikationen beim Designer lassen sich direkt durch eine vom Chiphersteller gelieferte, maschinenlesbare Technologiespezifikation mit Technologieinformationen versorgen.
2. Verwendung einer **Datenbank** zu Speicherung von Technologieinformationen. Gemessen an den Merkmalen der Verwendung einer formalen Sprache sind sicher die Punkte (a) und (b) erfüllt. Eine Datenbank ist aber weder einfach manuell zu Überprüfen noch einfach kommunizierbar.
3. **Tabellarisches Format.** Dies ist der ursprüngliche datengetriebene Ansatz zur Spezifikation von Technologieinformationen für einzelne Applikationen. Gegenüber dem Vorteil einer direkten Les- und Auswertbarkeit für eine Applikation lassen sich in einem manuell erstellten tabellarischen Format nur wenig komplexe Zusammenhänge darstellen. Sollen Abhängigkeiten vom Kontext einer Entwurfsregel formuliert werden, wird die Tabelle aufgrund von erforderlichen Querverweisen schnell so komplex, daß sie nicht mehr manuell wartbar ist. Geht man allerdings von einer automatischen Erstellung einer tabellarischen Technologiebeschreibung aus einem textuellen Format aus, scheint dieser Ansatz aufgrund seiner leichten Handhabbarkeit auf der Applikationsseite zur Speicherung von Technologieinformationen in einem Layout-CAD System geeignet zu sein.

Im Rahmen dieser Arbeit wurde ein Compiler für eine Technologiebeschreibungssprache DROOLY entwickelt, welcher als Ausgabe eine tabellarische Format erzeugt. Zum Zugriff auf diese Tabellen wird einer Layoutapplikation ein umfangreicher Satz auch komplexer Zugriffsoperationen zur Verfügung gestellt, um eine Formatabhängigkeit zu vermeiden. Obwohl DROOLY durch den zu Testzwecken implementierten Präprozessor ein durchaus lesbares Format erhält, ist diese Sprache nur als Zwischensprache in einem Compiler für die Technologiebeschreibungssprache DINGO-II konzipiert. DINGO-II erfüllt durch seine dem Englischen angenäherte Syntax die Anforderungen an eine Technologiebeschreibungssprache. Es stellt einen umfangreichen Satz unterschiedlicher Objekttypen und eine komfortable Syntax zur Beschreibung sowohl allgemeiner Technologieaspekte als auch qualitativer wie quantitativer Eigenschaften legaler Technologieobjekte zur Verfügung.

DROOLY kennt demgegenüber nur zwei unterschiedliche Objekttypen. Lokale Module werden durch die Möglichkeit vermieden, Objekte mit qualifizierten Bezeichnern zu definieren. Der Parametermechanismus ist gegenüber DINGO-II durch die Übergabe von Indizes in Parameterlisten vereinfacht. Topologiebeschreibung und Design Rules sind durch eine gemeinsame Syntax zusammengefaßt.

Durch diese Vereinfachungen liegt der Abstraktionsgrad einer DROOLY-Beschreibung zwischen dem einer DINGO-II Beschreibung und dem tabellarischen Format. Über das Compiler-Frontend werden dabei möglichst wenige Annahmen gemacht, da ein solches noch nicht zur Verfügung stand. Im 5. Kapitel wird die Struktur der erwarteten Symboltabelle beschrieben; in 6. Kapitel wird eine Anleitung zur Übersetzung von DINGO-II nach DROOLY gegeben.

Das tabellarische Format setzt sich aus den drei Tabellen Symboltabelle, Objekttabelle und Wertausdruckstabelle zusammen. Die Aufteilung in diese drei Tabellen ergibt sich aus der unterschiedlichen Struktur der zu speichernden Daten:

In der Symboltabelle werden alle nicht strukturellen Informationen über die Technologieobjekte gespeichert. Im Compiler-Backend werden daraus nur die Objektbezeichner benötigt.

Die Objekttabelle speichert qualitative Informationen über die Struktur der Technologieobjekte. Einzelne Tabellenelemente sind die Technologieobjekte mit ihren Unterstrukturen sowie Relationen und Eigenschaften dieser Objekte. Wichtiges Prinzip ist dabei die Lokalität der Speicherung von auf ein Objekt bezogenen Informationen. Ein Objekt erhält Verweise auf alle Unterstrukturen. Die Vererbung von Unterstrukturen aus dem Klassenobjekt trägt viel zur Komplexität des DROOLY-Compilers bei.

Von der Objekttabelle wird auf quantitative Informationen in der Wertausdruckstabelle verwiesen. Aus Atomarausdrücken aufgebaut geben die hier gespeicherten Baumstrukturen die Syntaxbäume der DROOLY-Wertausdrücke wieder.

Der Compiler zum Aufbau der Tabellen aus einer DROOLY-Beschreibung arbeitet im wesentlichen syntaxgetrieben. Aus einer Syntaxbeschreibung der DROOLY-Sprache zusammen mit der Angabe semantischer Aktionen erzeugt der Parsergenerator NEMO einen Parser für die DROOLY-Sprache. Von den semantischen Aktionen werden beim Erkennen der jeweiligen syntaktischen Konstrukte in der DROOLY-Eingabe Operationen aus den object- und expr-Modulen zum Aufbau der Tabellenstrukturen aufgerufen. Einen besonders komplexen Punkt bei der Erstellung der Tabellen stellt die Übersetzung einer Identifikationsanweisung dar, welche von den Operationen des compare-Moduls behandelt wird. Ein Modul mit Operationen zur Validitätsüberprüfung von Topologierelationen, ein Scanner, ein Modul zur Fehlerbehandlung und ein solches zur Protokollierung des Compilerlaufs vervollständigen den implementierten DROOLY-Compiler. Sein Aufbau wird im 8. Kapitel beschrieben.

Die Zusammenarbeit mit Layoutapplikationen wird im 9. Kapitel allgemein und im 10. Kapitel anhand eines konkreten Beispiels beschrieben. Als Beispiel wurde der Baulementextraktor MONDRIAN gewählt. MONDRIAN erwartet seine Technologie-Informationen in der DINGO Design Rule Beschreibungssprache. Die Interpretation der so erhaltenen Technologiedaten erfolgt in einem speziellen Modul, dem DINGO-Manager. Durch Anpassung dieses Moduls ließe sich MONDRIAN mit relativ geringem Aufwand zur Benutzung der hier entwickelten Tabellen modifizieren.

Im 11. Kapitel schließlich werden drei Aspekte für mögliche Erweiterungen der Operationen auf den Tabellen aufgezeigt, die in bestimmten Applikationen erforderlich sein könnten bzw. die den Komfort des Technologie-Compilers erhöhen würden.

## Anhang A: Liste der DROOLY-Token und Schlüsselworte

| Präprozessor              | DROOLY | Symbolische Konstante     |
|---------------------------|--------|---------------------------|
| (assymmetric              | (a     | ASSYMETRIC                |
| (class                    | (c     | CLASS                     |
| (classdef                 | (cd    | CLASSDEF                  |
| (classtest                | (c?    | CLASSTEST                 |
| (fpar                     | (p     | FPAR                      |
| (merge                    | (m     | MERGE                     |
| (stddef                   | (sd    | STDDEF                    |
| )                         | )      | CLOSEPAR                  |
| <                         | <      | LSS                       |
| <=                        | <=     | LEQ                       |
| ==                        | ==     | EQU                       |
| >                         | >      | GTR                       |
| >=                        | >=     | GEQ                       |
| [                         | [      | OPENBRAC                  |
| ]                         | ]      | CLOSEBRAC                 |
| against_current_direction | s1     | AGAINST_CURRENT_DIRECTION |
| all                       | q1     | ALL                       |
| and                       | &      | AND                       |
| any                       | q2     | ANY                       |
| area                      | t1     | AREA                      |
| capacitance               | t2     | CAPACITANCE               |
| centered                  | s2     | CENTERED                  |
| cross                     | t3     | CROSS                     |
| current                   | t4     | CURRENT                   |
| dimension                 | t5     | DIMENSION                 |
| dist                      | t6     | DIST                      |
| eccentric                 | s3     | ECCENTRIC                 |
| edge_length               | t7     | EDGE_LENGTH               |
| else                      | ?2     | ELSE                      |
| enclose                   | t19    | ENCLOSE                   |
| if                        | ?      | IF                        |
| in_angle                  | t17    | IN_ANGLE                  |
| in_current_direction      | s4     | IN_CURRENT_DIRECTION      |
| indist                    | t8     | INDIST                    |
| layerdef                  | l      | LAYERDEF                  |
| length                    | t9     | LENGTH                    |
| one                       | q3     | ONE                       |
| or                        | !      | OR                        |
| out_angle                 | t18    | OUT_ANGLE                 |
| overlap                   | t10    | OVERLAP                   |
| param                     | p:     | PARAM                     |
| part                      | p      | PART                      |
| power                     | t11    | POWER                     |
| radius                    | t12    | RADIUS                    |
| resistance                | t13    | RESISTANCE                |
| self                      | %      | SELF                      |
| set                       | s      | SET                       |
| shape                     | -      | SHAPE                     |
| some                      | q4     | SOME                      |

|         |     |         |
|---------|-----|---------|
| then    | ?1  | THEN    |
| touch   | t14 | TOUCH   |
| voltage | t15 | VOLTAGE |
| width   | t16 | WIDTH   |

---

## Anhang B: Fehlermeldungen

| Nummer | Klasse  | Meldung   |
|--------|---------|---|
| 0      | ERROR   | Unknown message   |
| 1      | WARNING | Relations of %s and %s have different quantifiers                 |
| 2      | WARNING | Component %s has %d correspondences                               |
| 3      | PANIC   | ExprTb overflow   |
| 4      | ERROR   | Condition not allowed as part of a condition                      |
| 5      | PANIC   | Illegal Spec in condition   |
| 6      | PANIC   | Main_class of condition %d not found                              |
| 7      | PANIC   | Illegal message selector  |
| 8      | PANIC   | Can't open input file %s  |
| 9      | PANIC   | Error in call to Preprocessor                                     |
| 10     | PANIC   | Can't open Intermediate File %s                                   |
| 11     | PANIC   | Can't open TestEvaluation Output File %s                          |
| 12     | PANIC   | Can't open TestReference Output File %s                           |
| 13     | PANIC   | Can't open Symbol-Table %s  |
| 14     | PANIC   | Can't open output file %s   |
| 15     | PANIC   | SuffixStack overflow  |
| 16     | PANIC   | SuffixStack underflow   |
| 17     | PANIC   | ObjectStack overflow  |
| 18     | PANIC   | ObjectStack underflow   |
| 19     | PANIC   | No more space   |
| 20     | PANIC   | ObjTb overflow  |
| 21     | PANIC   | Illegal call of insert_obj (obj=%d father=%d)                     |
| 22     | PANIC   | Illegal call of link_object with NIL-Object                       |
| 23     | PANIC   | Illegal ClassTest on classless object                             |
| 24     | PANIC   | ClassCall.aparm_cnt=%d not equal classtest_classCall.aparm_cnt=%d |
| 25     | PANIC   | Illegal req_spec %d   |
| 26     | PANIC   | Illegal class-index   |
| 27     | ERROR   | Value %d of actual parameter not in legal range [1..%d]           |
| 28     | ERROR   | Number of actual parameters =%d, should be =%d                    |
| 29     | ERROR   | In Property: Object not defined                                   |
| 30     | ERROR   | In Relation: Quantifier not defined                               |
| 31     | ERROR   | In Relation: First object not defined                             |
| 32     | ERROR   | In Relation: Second object not defined                            |
| 33     | ERROR   | Class not defined   |
| 34     | ERROR   | In ClassTest: Object not defined                                  |
| 35     | ERROR   | In ClassTest: Class not defined                                   |
| 36     | PANIC   | No more space for string  |
| 37     | PANIC   | SymTb overflow  |
| 38     | WARNING | Identifier truncated  |
| 39     | PANIC   | No more space for int_list  |
| 40     | WARNING | Teval - Function %s not found                                     |
| 41     | PANIC   | Teval - TevalStack overflow                                       |
| 42     | PANIC   | Teval - TevalStack underflow                                      |
| 43     | WARNING | Teval - Unknown function name                                     |
| 44     | WARNING | Teval - Block %d not declared for function %s (nr_blocks=%d)      |
| 45     | WARNING | Teval - Function %s defined twice                                 |
| 46     | WARNING | Illegal Token %s  |
| 47     | PANIC   | Compare_2_card: expr %d is no cardinality spec                    |
| 48     | PANIC   | Insert_int: first call must say \"new_table=true\"                |
| 49     | PANIC   | Compare_2_card: both Min and Max set in interval bound            |
| 50     | WARNING | Cardinality declarations of %s and %s: %s                         |
| 51     | ERROR   | Declarations of %s and %s not compatible                          |

X Eicherfassig

## Anhang B: Fehlermeldungen

| Nummer | Klasse  | Meldung  |
|--------|---------|--|
| 0      | ERROR   | Unknown message  |
| 1      | WARNING | Relations of %s and %s have different quantifiers                    |
| 2      | WARNING | Component %s has %d correspondences                                  |
| 3      | PANIC   | ExprTb overflow  |
| 4      | ERROR   | Condition not allowed as part of a condition                         |
| 5      | PANIC   | Illegal Spec in condition  |
| 6      | PANIC   | Main_class of condition %d not found                                 |
| 7      | PANIC   | Illegal message selector   |
| 8      | PANIC   | Can't open input file %s   |
| 9      | PANIC   | Error in call to Preprocessor  |
| 10     | PANIC   | Can't open Intermediate File %s                                      |
| 11     | PANIC   | Can't open TestEvaluation Output File %s                             |
| 12     | PANIC   | Can't open TestReference Output File %s                              |
| 13     | PANIC   | Can't open Symbol-Table %s   |
| 14     | PANIC   | Can't open output file %s  |
| 15     | PANIC   | SuffixStack overflow   |
| 16     | PANIC   | SuffixStack underflow  |
| 17     | PANIC   | ObjectStack overflow   |
| 18     | PANIC   | ObjectStack underflow  |
| 19     | PANIC   | No more space  |
| 20     | PANIC   | ObjTb overflow   |
| 21     | PANIC   | Illegal call of insert_obj (obj=%d father=%d)                        |
| 22     | PANIC   | Illegal call of link_object with NIL-Object                          |
| 23     | PANIC   | Illegal ClassTest on classless object                                |
| 24     | PANIC   | ClassCall.aparm_cnt=%d not equal<br>classtest_classCall.aparm_cnt=%d |
| 25     | PANIC   | Illegal req_spec %d  |
| 26     | PANIC   | Illegal class-index  |
| 27     | ERROR   | Value %d of actual parameter not in legal range [1..%d]              |
| 28     | ERROR   | Number of actual parameters =%d, should be =%d                       |
| 29     | ERROR   | In Property: Object not defined                                      |
| 30     | ERROR   | In Relation: Quantifier not defined                                  |
| 31     | ERROR   | In Relation: First object not defined                                |
| 32     | ERROR   | In Relation: Second object not defined                               |
| 33     | ERROR   | Class not defined  |
| 34     | ERROR   | In ClassTest: Object not defined                                     |
| 35     | ERROR   | In ClassTest: Class not defined                                      |
| 36     | PANIC   | No more space for string   |
| 37     | PANIC   | SymTb overflow   |
| 38     | WARNING | Identifier truncated   |
| 39     | PANIC   | No more space for int_list   |
| 40     | WARNING | Teval - Function %s not found  |
| 41     | PANIC   | Teval - TevalStack overflow  |
| 42     | PANIC   | Teval - TevalStack underflow   |
| 43     | WARNING | Teval - Unknown function name  |
| 44     | WARNING | Teval - Block %d not declared for function %s (nr_blocks=%d)         |
| 45     | WARNING | Teval - Function %s defined twice                                    |
| 46     | WARNING | Illegal Token %s   |
| 47     | PANIC   | Compare_2_card: expr %d is no cardinality spec                       |
| 48     | PANIC   | Insert_int: first call must say \"new_table=true\"                   |
| 49     | PANIC   | Compare_2_card: both Min and Max set in interval bound               |
| 50     | WARNING | Cardinality declarations of %s and %s: %s                            |
| 51     | ERROR   | Declarations of %s and %s not compatible                             |

Anhang C: Die vollständigen Tabellen der Beispieltechnologie

### Symbolabelle:

```
1 Metal  
2 Poly  
3 Contact_Cut  
4 n_Diff  
5 p_Diff  
6 n_Well  
7 Gate  
8 diffusion  
9 p_wire  
10 iso_Polygon  
11 n_OE_Trans  
12 well  
13 gt
```

### Objekttabelle:



### Wertausdruckstabelle: (12)

| #   | ?spec    | <- | -> | attr | min  | eq | max |
|-----|----------|----|----|------|------|----|-----|
| 11  | VALUE    |    |    | NONE |      |    |     |
| 14  | CLASSTST |    |    | NONE |      |    |     |
| 31  | VALUE    |    |    | NONE | 15   | 15 |     |
| 41  | VALUE    |    |    | NONE | 10   | 10 |     |
| 51  | LOC_PROP | 21 | 3  | NONE |      |    |     |
| 61  | AND      |    |    | 6    | NONE | 11 | 11  |
| 71  | VALUE    |    |    | NONE |      | 51 | 5   |
| 81  | AND      |    |    | 8    | NONE | 11 | 11  |
| 91  | VALUE    |    |    | NONE |      | 51 | 5   |
| 101 | AND      |    |    | 10   | NONE | 11 | 11  |
| 111 | VALUE    |    |    | NONE |      | 51 | 5   |
| 141 | VALUE    |    |    | NONE | 10   | 10 |     |
| 151 | 11 VALUE |    |    | NONE |      |    |     |
| 161 | VALUE    |    |    | NONE | 20   | 20 |     |
| 171 | VALUE    |    |    | NONE | 30   | 30 |     |

(12) Für die Spaltenüberschriften der Wertausdruckstabelle wurden in der „.at“-Datei folgende Abkürzungen gewählt:

|        |     |
|--------|-----|
| cond   | ?   |
| spec   | -   |
| left   | <-  |
| right  | ->  |
| minval | min |
| eqval  | eq  |
| maxval | max |

## Literatur

- [Aho 78] A.V. Aho, J.D. Ullman: Principles of Compiler Design  
Addison-Wesley Publishing Company (1978)
- [Aud 86] J.S. Aude, H.J. Kahn: A Design Rule Database System to Support Technology-Adaptable Applications  
Proc. 23rd Design Automation Conference (1986), S.510
- [Bar 83] E. Barke: A Layout Verification System for Analog Bipolar Integrated Circuits  
Proc. 20th Design Automation Conference (1983), S.353
- [Bla 87] E. Blake, S. Cook: On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk  
Proc. ECOOP (European Conference on Object-Oriented Programming) '87, Springer Verlag (1987), S.41
- [Bor 81] A. Borning: The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory  
ACM Trans. on Programming Languages and Systems, Vol.3, No.4, (1981), S.353
- [Bro 85] J.S. Brothers et.al.: The Megacell concept: an approach to painless custom design  
IEE Proc. Solid State & Electronic Devices (1985), S.91
- [Brü 88] R. Brück: IMAGE - A Technology Invariant CAD System for IC Layout  
eingereicht zu ICCD 1988
- [Che 76] P. Pin-Shan Chen: The Entity-Relationship Model - Toward a Unified View of Data  
ACM Trans. on Database Systems Vol.1, No.1, (1976), S.9
- [Chu 85] K.-C. Chu, Y.E. Lien: Technology Tracking for VLSI Layout Design Tools  
Proc. 22nd Design Automation Conference (1985), S.279
- [DINGO] Rainer Brück: DINGO-II Handbuch  
Abteilung Informatik, Universität Dortmund (1988)
- [EDIF 85] EDIF Steering Committee: EDIF Electronic Design Interchange Format Version 1 0 0 (1985)
- [Hör 86] E. Hörbst, C. Müller-Schloer, H. Schwärtzel: Standardentwurfsverfahren für integrierte Schaltungen  
Spektrum der Informatik (1986), S.215
- [ICON] R.E. Griswold, M.T. Griswold: The Icon Programming Language  
Prentice-Hall (1983)
- [Krä 89] Olav Krämer: DROOLY-Compiler  
Installiert auf dem Sun-Rechner *Charon* der Abteilung Informatik, Universität Dortmund im Verzeichnis *~kraemer/DROOLY*. Das ausführbare Programm steht in diesem Verzeichnis unter dem Namen *drooly* und wird mit dem dortigen *makefile* durch Aufruf von *make* aus dem Programmquellen im Unterverzeichnis *DROOLY.src* erzeugt. Testbeispiele finden sich im Unterverzeichnis *DROOLY.bsp*.
- [Lee 87] W. Lee, G. Liu, K. Peterson: TED: A Graphical Technology Description Editor  
Proc. 24th Design Automation Conference (1987), S.423
- [MONDRIAN] Projektgruppe MONDRIAN: Endbericht  
Abteilung Informatik, Universität Dortmund (1988)
- [Mör 88] B. Mörschen: Diplomarbeit  
Abteilung Informatik, Universität Dortmund (1988)
- [NEMO] Christoph Ohsendoth: NEMO: Nicely Modified YACC  
Abteilung Informatik, Universität Dormund (1986)

- [Ous 84] J.K. Ousterhout, W.S. Scott, G. Hamachi, R.N. Mayo: MAGIC: A VLSI Layout System  
Proc. 21nd Design Automation Conference (1984), Kap. 4 bis 6
- [Pre 88] H. Preuß: Diplomarbeit  
Abteilung Informatik, Universität Dortmund (1988)
- [Schi 85] W. Schiele: Automatic Design Rule Adaption of Leaf Cell Layouts  
INTEGRATION, the VLSI journal 3 (1985); North-Holland, S.93
- [Smi 85] Ph. Smith, S. Daniel: The VIVID System Approach to Technology Independence: The Master Technology File System  
Proc 22nd Design Automation Conference (1985), S.76
- [vEh 83] G.J. von Ehr: Position Paper; Role of Technology Design Rules in Design Automation  
Proc. 20th Design Automation Conference (1983), S.395
- [Wal 85] R.A. Walker, D.E. Thomas: A Model of Design and Representation  
Proc. 22nd Design Automation Conference (1985), S.453
- [Wic 85] R. Wicke: Diplomarbeit  
Abteilung Informatik, Universität Dortmund (1985)
- [Wro 87] H. Wronn: Diplomarbeit  
Abteilung Informatik, Universität Dortmund (1987)
- [YACC] S.C. Johnson: YACC: Yet Another Compiler-Compiler  
Bell Laboratories, New Jersey 07974 (1978)