KfK 3479
Februar 1983

Generierbare Datenbanksysteme

F. J. Polster
Institut für Datenverarbeitung in der Technik

Kernforschungszentrum Karlsruhe

KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Datenverarbeitung in der Technik

KfK 3479

Generierbare Datenbanksysteme

Franz J. Polster

Als Dissertation genehmigt von der Fakultät für Informatik
der Universität Karlsruhe

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Kernforschungszentrum Karlsruhe GmbH
ISSN 0303-4003

Zusammenfassung

DBMS sei ein allgemeines Datenbanksystem. Es wird untersucht, wie DBMS an die spezifischen Anforderungen einer Anwendung A angepaßt werden kann durch Erzeugen von Versionen, die nur die für A erforderlichen Fähigkeiten von DBMS realisieren.

Die Erzeugung von Versionen von DBMS wird allgemein als das Problem der Erzeugung von Teilsystemen eines vollständigen Programmsystems PS behandelt. Das Programm eines Teilsystems t wird durch Auswahl der für t relevanten Teile des Programms von PS erzeugt. Es wird der Begriff Fragment eingeführt und hiermit ein heuristisches Verfahren zur Zerlegung des Programmtextes von PS in die Teilstrings angegeben, mit denen als Bausteine die Teilsysteme von PS erzeugt werden können. Diese Zerlegung eines Programmtextes wird formal als sog. B-Programm beschrieben: ein geordneter Baum mit den Teilstrings des Programms von PS als Blattknoten und den Fragmenten als innere Knoten. Eine Abbildung ρ: $T \times F \rightarrow \{0,1\}$ (T: Menge der Teilsysteme von PS; F: Menge der Fragmente) gibt zu jedem Fragment f an, ob der Teilbaum mit f als Wurzel für ein Teilsystem t relevant ist oder nicht ($\rho(t,f)=1$ bzw. 0); zu jedem Fragment f wird ein Ersatz $\sigma(f)$ angegeben. Teilsystem-Erzeugung wird als pre-order Durchlauf eines Teilbaumes eines B-Programms eingeführt. Die Menge T ist bestimmt durch Beziehungen zwischen Fragmenten, die sich aus dem Kontrollfluß von PS herleiten; sie werden formal mittels eines sog. Fragmentsystems (gerichteter azyklischer Graph mit den Fragmenten als Knoten) als Eigenschaften von ρ beschrieben. Es wird ein Verfahren angegeben zur Konstruktion einer Teilmenge $CF \subseteq F$, die minimal ist mit der Eigenschaft, daß mit der Zuweisung von Relevanzwerten an die Fragmente von CF die Relevanzwerte aller Fragmente von F festgelegt sind. Jedem $t \in T$ entspricht genau ein $\tau \in B^n$ ($B=\{0,1\}$) mit $n=|CF|$; T wird explizit als Teilmenge von B^n dargestellt. Die Architektur eines allgemeinen Systems zur automatischen Erzeugung von Teilsystemen, insbesondere die Implementierung von B-Programmen als Erweiterungen des Programms von PS, wird angegeben.

V ist mit einer Anwendung verträglich, wenn V alle für das Anwenderprogramm erforderlichen Operationen implementiert und alle von V realisierten Operationen auf die gesamte Datenbank ausführen kann. Die Menge der mit einer Anwendung verträglichen Versionen wird als Teilmenge von B^n charakterisiert.

GENERABLE DATABASE SYSTEMS

Abstract

Let DBMS be a general database management system. The problem of tailoring DBMS to the specific needs of an application A at hand by generating versions of DBMS, which provide only the features of DBMS required by A, is studied.

Generation of versions is treated as the general problem of constructing partial systems of a given comprehensive program system PS. The program of a partial system t is composed of the substrings of the program-text of PS relevant to t. The concept of fragment is introduced and a heuristic method is developed for the decomposition of the program of PS into the substrings, which serve as the building blocks for the construction of the partial systems of PS. This decomposition is formally described as a so-called B-program: an ordered tree with the substrings as its leaves and the fragments as its non-leaf vertices. A mapping ρ: T×F --> {0,1} (T: set of partial systems of PS; F: set of fragments) indicates, whether or not the subtree with fragment f as its root is relevant to t ($\rho(t,f)=1$ or 0 resp.); each fragment f is assigned a substitute $\sigma(f)$. Generation of a partial system is formally defined as pre-order traversal of a subtree of a B-program. The set T is characterized by means of interrelationships between fragments reflecting the flow of control of PS. The concept of fragment system (basically a directed acyclic graph) is introduced to describe these relationships as properties of ρ. An algorithm is presented for the construction of a minimal subset CF $\subseteq$ F with the property, that with the assignment of relevance values to the elements of CF the relevance values of all fragments are determined. Thus each t ε T corresponds to exactly one $\tau \varepsilon B^n$ (B={0,1}) with n=|CF|; an explicit representation of T as a subset of B^n is given. The architecture of a program-generator for the automatic generation of partial systems based on these ideas, in particular an implementation of B-programs as an expansion of the program of PS, is delineated.

Version V is compatible with an application, iff V implements (at least) all operations called for by the application program and is able to correctly execute all operations implemented by V on the database. The set of versions compatible with an application is characterized in terms of subsets of B^n.

I N H A L T

B I L D E R

D E F I N I T I O N E N

S Ä T Z E , K O R O L L A R E

V E R F A H R E N

B E Z E I C H N U N G E N

(vgl. auch Anhang 1)

$\neq$	: ungleich
$\equiv$ ($\neg\equiv$)	: Gleichheit (Ungleichheit) von Abbildungen
ε ($\neg\varepsilon$)	: ist enthalten in (ist nicht enthalten in)
$\underline{c}$ ($\neg\underline{c}$)	: ist Teilmenge von (ist keine Teilmenge von)
$<_B$	: preorder-Ordnung eines geordneten Baumes B (Abschnitt 3.3)
$\square$	: Ende eines Beweises
a := b	: a wird definiert als b
a ==> b	: a impliziert b
a <==> b	: a ist äquivalent zu b
$\{x\|a_1,a_2,\ldots$	
$\ldots,a_n\}$	: Menge von Elementen x, die a_1, a_2, ..., a_n genügen
$\emptyset$	: die leere Menge
$\P(S)$	: Potenzmenge der Menge S
$\|S\|$	: Mächtigkeit der Menge S
$S_1 * S_2$	: Durchschnitt der Mengen S_1 und S_2
$S_1 + S_2$	: Vereinigung der Mengen S_1 und S_2
$S_1 - S_2$	: Differenz der Mengen S_1 und S_2
$OR_{i=1}^{m} x_i$	: die OR-Verknüpfung von x_i, $1\leq i\leq m$ (Definition 3.1)
$OR_{x\varepsilon S}\ x$	: die OR-Verknüpfung aller Elemente der Menge S (Def. 3.1)
$l_1 \parallel l_2$	: Verkettung von Listen oder Zeichenketten l_1, l_2
NIL	: der leere string (Abschnitt 3.1)
B	: Menge der Relevanzwerte {0,1} (Definition 3.1)
T	: Menge der Teilsysteme (Abschnitt 3.1)
X(f), O(f)	: Menge der X- bzw. O-Fragmente von f (Definition 4.1, Fußnote 1 von Abschnitt 4)
ρ, ρ_f, $\sigma(f)$	: die Relevanz bzw. der Ersatz von f (Definitionen 3.1, 3.2)
ROOT(x)	: Wurzelfragment einer R1- oder Ω-Menge (Definition 4.5, Verfahren 4.1)
o.B.d.A.	: ohne Beschränkung der Allgemeinheit
DV, EDV	: Datenverarbeitung, Elektronische Datenverarbeitung
DB, DBMS	: Datenbank, Datenbanksystem (database management system)
GDBMS	: Generierbares Datenbanksystem
DML, DDL	: Datenmanipulationssprache, Datendefinitionssprache
JCL	: job control language ("Betriebssprache" /BG 74/)

1. Einleitung

1.1. Datenhaltung auf Kleinrechnern

Der technologische Fortschritt der letzten Jahre sowohl auf dem Gebiet der Hardware wie auch Software hat zu einem vielseitigen Angebot kostengünstiger und zugleich leistungsfähiger Kleinrechnersysteme geführt. Die auf diesen Rechnern nunmehr verfügbaren Hilfsmittel zur Programmerstellung (insbes. Compiler für höhere Programmiersprachen) unterscheiden sich in ihrer Leistungsfähigkeit nicht mehr von denen auf Großrechnern, es ist somit möglich und ökonomisch geworden, auch an sich komplexe Anwendungssysteme auf Kleinrechnern zu implementieren. Als Folge ist eine zunehmende Verbreitung von Datenverarbeitungskapazität in Form von dezentralen Kleinrechnern zu beobachten:
Kleinrechner wurden als "Prozeßrechner" zunächst überwiegend für technische Anwendungen (meßtechnische Aufgaben, Prozeßüberwachung und -steuerung) eingesetzt. Solche Systeme übernehmen jetzt auch weitere Aufgaben, die die Verwaltung relativ umfangreicher Datenbestände erfordern /De 77/. Hierzu einige Beispiele:

- In /UB 77/ wird der Einsatz eines Prozeßrechners zur Meßdatenerfassung an Motorprüfständen dargestellt: neben der Automatisierung der Versuchsdurchführung selbst, einschließlich Meßdatenerfassung, dient dieser Rechner auch zur anschließenden Auswertung der Meßdaten; die hierzu erforderliche Datenbasis enthält neben den eigentlichen Meßdaten u.a. "Versuchsbeschreibungen", "Vorschriften" verschiedener Staaten, etc. /Ur 80/.
  Mit dieser Anlage werden auch Steuerprogramme für NC-Werkzeugmaschinen erstellt; angestrebt wird ein System zur Verwaltung und Wartung von NC-Programmen, was ebenfalls die Haltung von Daten auf Externspeichern erfordert /Ur 80/.
- Zur Steuerung und Überwachung der Produktion in einem Walzwerk wird ein DV-System mit zwei Prozeßrechnern entwickelt /Sc 81/: Ein "Steuerrechner" übernimmt die zeitkritischen Aufgaben der Steuerung des technischen Prozesses (Walzstraße) und der Datenerfassung; für die Auswertung der anfallenden "Prozeßdaten" zum Zweck der Qualitätskontrolle ist anstelle eines zentralen Großrechners ein eigener "Auskunftsrechner" vorgesehen, ebenfalls ein Prozeßrechner.

Vollkommen neue Einsatzmöglichkeiten ergeben sich daraus, daß nun auch dem EDV-Laien, etwa in Kleinbetrieben, die Möglichkeiten der DV-Technologie zugänglich sind. Für diesen Benutzerkreis, "der einerseits zwar auch Probleme im Rahmen der Führung von Massendaten hat, die am einfachsten mit Hilfe einer EDV-Anlage zu bewältigen wären, ... für den andererseits die Selbstprogrammierung oder gar die Bearbeitung durch einen Anwendungsprogrammierer keine ökonomisch vertretbare Lösung darstellt" /BW 80/, sind vorgefertigte Programmpakete anzubieten. Wie die Vielfalt der am Markt erhältlichen "Branchenpakete" zur Finanzbuchhaltung, Lohnabrechnung, Auftragsabwicklung, etc. /CM 82/ zeigt, sind dies herkömmliche, "kommerzielle" Anwendungen.

Es ist also festzustellen, daß aufwendige und anspruchsvolle DV-Systeme zur Datenhaltung auf Kleinrechnern implementiert werden bzw. Bedarf hierfür besteht. Was die Art der Daten und die zu ihrer Verwaltung erforderlichen Funktionen angeht, so unterscheiden sie sich prinziell nicht von Datenbankanwendungen auf zentralen Großrechnern, lediglich das Datenvolumen erreicht nicht die dort üblichen Größenordnungen (10^8 Bytes und mehr /TF 76, BW 80/). Der für das Folgende wesentliche Unterschied besteht darin, daß Kleinrechnersysteme d e d i z i e r t eingesetzt werden: es laufen

- v o r g e p l a n t e Aktivitäten mit jeweils
- genau f e s t g e l e g t e n Aufgabenstellungen in einer
- k o n t r o l l i e r t e n Umgebung

ab.

Unter dem A n f o r d e r u n g s p r o f i l einer Anwendung ("Benutzungscharakteristik" in /Sc 76/) werden hier die Anforderungen der Anwendung bezüglich der Datenhaltung verstanden, also Angaben wie: die zur Datenhaltung erforderlichen Funktionen, die Häufigkeit, mit der sie aufgerufen werden, maximale Ausführungszeiten, etc.
Aus der Charakterisierung von oben folgt für die Datenhaltung in dedizierten Systemen, daß für jeden Zeitpunkt das Anforderungsprofil der Anwendung bekannt ist.
Man kann daher den Ablauf solcher Systeme in (Betriebs-) P h a s e n unterteilen, die jeweils durch ihr Anforderungsprofil charakterisiert sind und nacheinander, in der Regel wiederholt durchlaufen werden. Typische Betriebsphasen von DV-Systemen im technisch-naturwissenschaftlichen Bereich sind (vgl. Beispiele von oben):

- "Datenerfassung"
  Wesentlich für diese Phase ist das schnelle Abspeichern von Daten; Lese-Operationen (retrieval) finden nicht statt oder sind von niederer Priorität; Kontrolle der Zugriffsberechtigung ist häufig wenig sinnvoll, etwa wenn von einer Meßapparatur gelieferte Daten on-line mittels eines Programms abgespeichert werden; aus Gründen der Effizienz verbietet sich eine Protokollierung der Datenbankaktivitäten, in der Regel kann man hierauf auch wegen der Reproduzierbarkeit der Abläufe verzichten.
- "Datenanalyse"
  Es sind, etwa vom Experimentator, Datenbestände interaktiv zu durchsuchen und analysieren: die anfallenden Ergebnisdatenmengen sind klein, es wird Wert gelegt auf kurze Antwortzeiten bei Anfragen, Update-Operationen werden entweder gar nicht benötigt oder sind zumindest nicht zeitkritisch.
- "Berichtserstellung"
  Diese Phase ist charakterisiert durch das retrieval umfangreicher Datenmengen, meist verbunden mit Sortierläufen und rechenintensiven Auswertungen.

Die Haltung von Daten auf Externspeichern ist eine allgemeine, nicht anwendungsspezifische Aufgabenstellung, die isoliert von der eigentlichen Anwendung betrachtet und durch Einsatz allgemeiner, wiederverwendbarer Software gelöst werden kann /Se 77, FS 76/. Ein Datenbank-Anwendungssystem besteht somit aus (Bild 1.1):

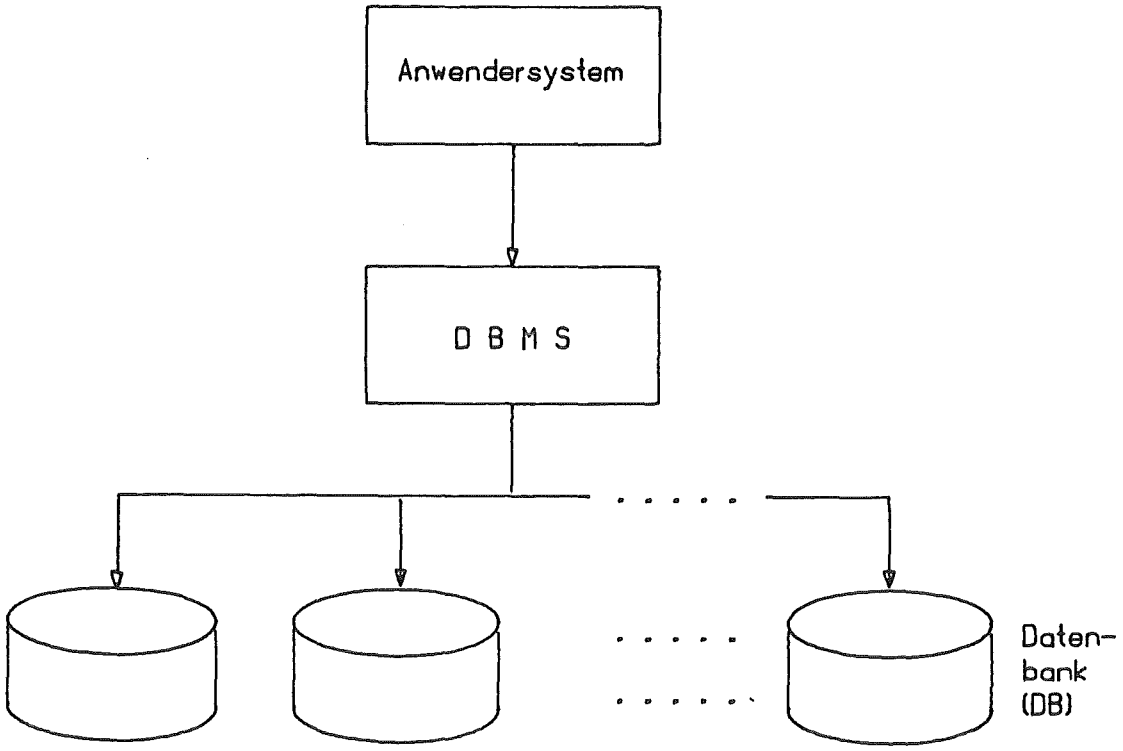


Bild 1.1: Aufbau eines DB-Anwendungssystems

- dem anwendungsspezifischen A n w e n d e r s y s t e m
- dem allgemeinen Datenbanksystem (D B M S), das die Aufgaben der Datenhaltung übernimmt
- der D a t e n b a n k (DB), also den im Rechnersystem gespeicherten Datenbeständen, auf die das Anwendersystem über das DBMS zugreift.

Wegen der Bedeutung der Datenbestände für ein Unternehmen (vgl. z.B. /FS 76, No 73/; "operational data" bei /Da 77/) entstanden zunächst auf Großrechnern Datenbanksysteme als allgemein verwendbare Programmsysteme zur Datenhaltung. Wesentliches Ziel (neben der Zentralisierung der Kontrolle der Datenbestände) war die Reduktion des Aufwandes und der Kosten bei der System-Erstellung durch Verwendung erprobter und bewährter Datenhaltungs-Software. Dieses Argument spricht ganz besonders auch für den Einsatz von Datenbanksystemen als Komponenten von DB-Anwendungssystemen auf Kleinrechnern:

- der eingangs aufgezeigte Bedarf an Systemen zur Datenhaltung erfordert ökonomische und rationelle Verfahren zur System-Erstellung
- Programmierung und Wartung von Kleinrechnersystemen werden in der Regel von den Anwendern selbst vorgenommen /Wi 78/, insbesondere steht üblicherweise kein Stab von (DBMS-)Spezialisten zur Verfügung.

Trotzdem ist jedoch festzustellen, daß Datenbanksysteme praktisch ausschließlich auf Großrechnern eingesetzt werden. Wie im nächsten Abschnitt dargelegt wird, ist der Grund hierfür darin zu sehen, daß ein modernes, datenunabhängiges DBMS ein komplexes und umfangreiches Programmsystem darstellt.

1.2. Zur Architektur von Datenbanksystemen

Die Schnittstelle eines Datenbanksystems wird gebildet von (s. /LM 78, LM 80, TL 82/)

- einer Menge von Objekten gemäß dem Schema der Datenbank, den zulässigen Zuständen der Datenbank
- einer Menge von Operationen, der Datenmanipulationssprache DML, deren Anwendung die DB von einen Zustand in einen anderen überführt.

Aufgabe eines DBMS als Programmsystem ist die Abbildung dieser (Anwender-)Schnittstelle auf eine Schnittstelle des Rechnersystems. Für ein DBMS, das einen hohen Grad an Datenunabhängigkeit bietet, ist dies ein komplexer Vorgang, der in mehreren Schritten durchzuführen ist. Ähnlich wie für Betriebssysteme gelangt man so zu einem Schichtenmodell für DBMS. In Anlehnung an Senko's DIAM /Se 73/ wird in /LH 81/ eine Gliederung in fünf Schichten angegeben (Bild 1.2):
Die Schicht S0, "Speicherverwaltung", implementiert eine "Datei-Schnittstelle", die von den spezifischen Eigenheiten und Charakteristika der Externspeichergeräte abstrahiert, also für Geräteunabhängigkeit sorgt.
Die Schicht S1, "Systempufferverwaltung", enthält die Seitenzuordnungsstrukturen des DBMS und bildet "Seiten" des Hauptspeichers auf "Blöcke" der Datei-Schnittstelle ab.
Die Schicht S2, "Satz- und Zugriffspfadverwaltung", stellt die Speicherungsstrukturen und Zugriffspfadtypen des DBMS zur Verfügung.
Die Schicht S3 verwirklicht mit der "satzorientierten DB-Schnittstelle" ein zugriffspfadorientiertes Datenmodell mit "externen Sätzen" und "logischen Zugriffspfaden" als Objekttypen.
Die Schicht S4 abstrahiert vollständig von Zugriffspfaden und realisiert ein mengenorientiertes Datenmodell mit ausschließlich logischen Datenstrukturen und deskriptiven Sprachen.

Neben der Überführung der logischen Datenstrukturen der Anwenderschnittstelle in die physischen Speicherzuordnungsstrukturen der Externspeichergeräte hat ein DBMS weitere Aufgaben zu übernehmen, wie z.B. Erhaltung der Datenintegrität, Zugriffskontrolle, Transaktionsverwaltung. Für die Implementierung eines (relationalen) DBMS gibt Härder in /Hä 78/ ausgehend vom Schichtenmodell nach Bild 1.2 drei virtuelle Maschinen[1] an (Bild 1.3):

- das Speichersystem
- das Zugriffssystem
- das Datensystem

Die virtuelle Maschine S p e i c h e r s y s t e m realisiert aufbauend auf einer Schnittstelle des Rechners die (DBMS-interne) Speicher-Schnittstelle.
In Bild 1.3. wird für die Implementierung eines DBMS von einer niederen Betriebssystem-Schnittstelle mit Primitiven für den externen Speicherzugriff ("Basiszugriffsmethode" /Hä 78/) ausgegangen; sie bietet

---

[1] Die folgende Darstellung stammt im wesentlichen aus /Hä 78/, p. 200 ff

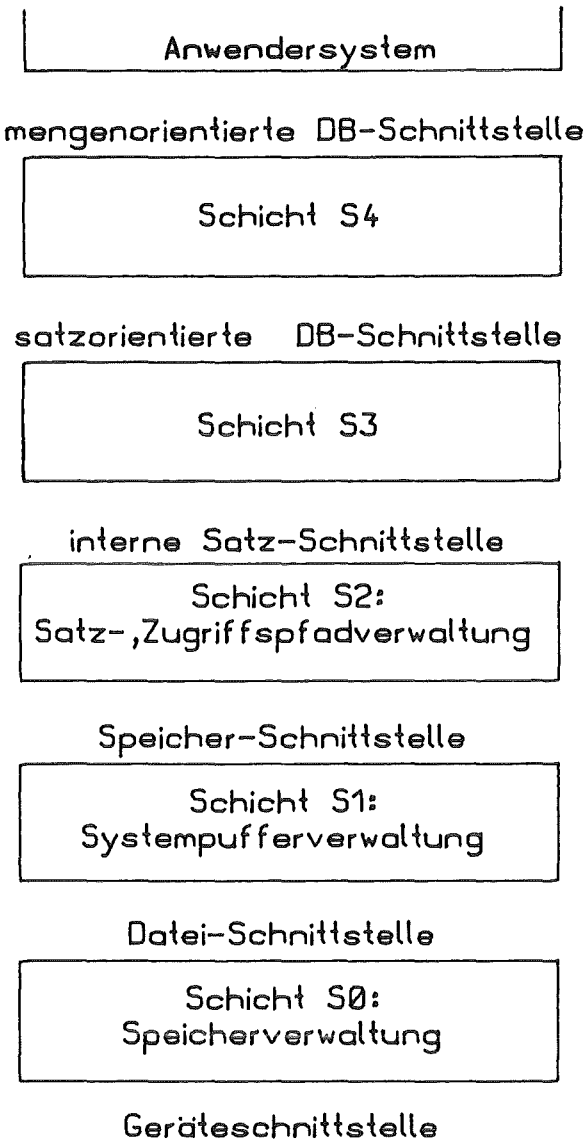


Bild 1.2: Ein Schichtenmodell für Datenbanksysteme (nach /LH 81/)

die Objekttypen "Block" und "Datei" mit den Operationen "Datei einrichten", "Datei löschen", "Datei eröffnen", "Datei schließen", "Block lesen", "Block schreiben".
Diese virtuelle Maschine entspricht also der Schicht S1 von Bild 1.2 [2].
Die Speicher-Schnittstelle weist nur noch Eigenschaften interner Speicher auf, als Objekte stehen Seiten und Segmente (geordnete Mengen von Seiten) zur Verfügung, mit Operationen zur Erzeugung und Entfernung (ALLOCATE, DEALLOCATE) von Segmenten, Verwaltung von Segmenten (OPEN, CLOSE), und

---

[2] Ist als Basismaschine lediglich die Geräte-Schnittstelle gegeben, so implementiert das Speichersystem neben der Schicht S1 ebenfalls die Schicht S0.

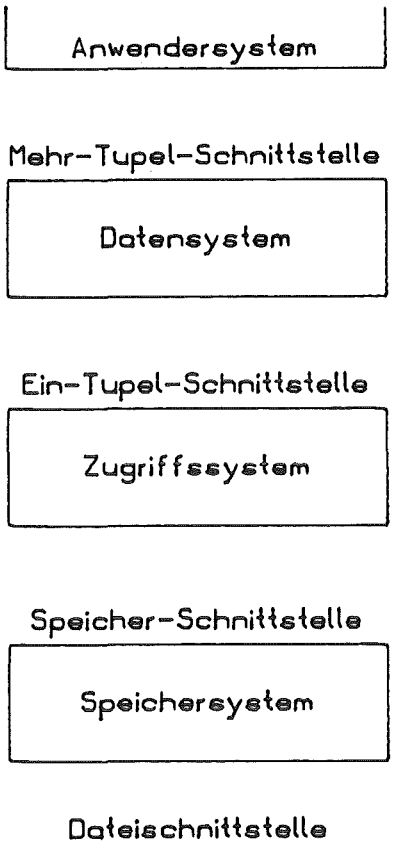


Bild 1.3: Allgemeiner Aufbau eines Datenbanksystems (vgl. /Hä 78/)

zum Lesen, Schreiben und Freigeben einzelner Seiten.
Die zur Durchführung einer DBMS-Operation erforderliche Zeit wird wesentlich durch die Zahl der hierfür erforderlichen Externspeicherzugriffe, also Operationen der Datei-Schnittstelle, bestimmt /GS 70, Hä 78/. Eine wichtige Aufgabe dieser virtuellen Maschine ist daher die Minimierung der Zahl der physischen E/A-Aufrufe. Das Speichersystem besteht hierzu aus (vgl. Bild 1.4)

- einem S y s t e m p u f f e r, der eine Menge von Blöcken aufnehmen kann
- der S y s t e m p u f f e r v e r w a l t u n g zur Verwaltung der Blöcke im Systempuffer nach einer geeigneten Ersetzungsstrategie, z.B. einem LRU-Algorithmus /GS 70/
- der S p e i c h e r v e r w a l t u n g zur Abwicklung der physischen E/A-Operationen.
  Durch eine geeignete Einbringstrategie, etwa nach dem Schattenspeicher- /Lo 77/ oder Zusatzdatei-Konzept /SL 76/, kann diese Komponente Recovery-Maßnahmen zur Gewährleistung der Datenintegrität unterstützen /Hä 78/.

Die virtuelle Maschine Z u g r i f f s s y s t e m implementiert die Schichten S2 und S3 von Bild 1.2; sie baut also auf der Speicher-Schnittstelle auf und realisiert eine Ein-Tupel-Schnittstelle, die navigierenden Ein-Tupel-Zugriff erlaubt /Ba 73/ und Unabhängigkeit von

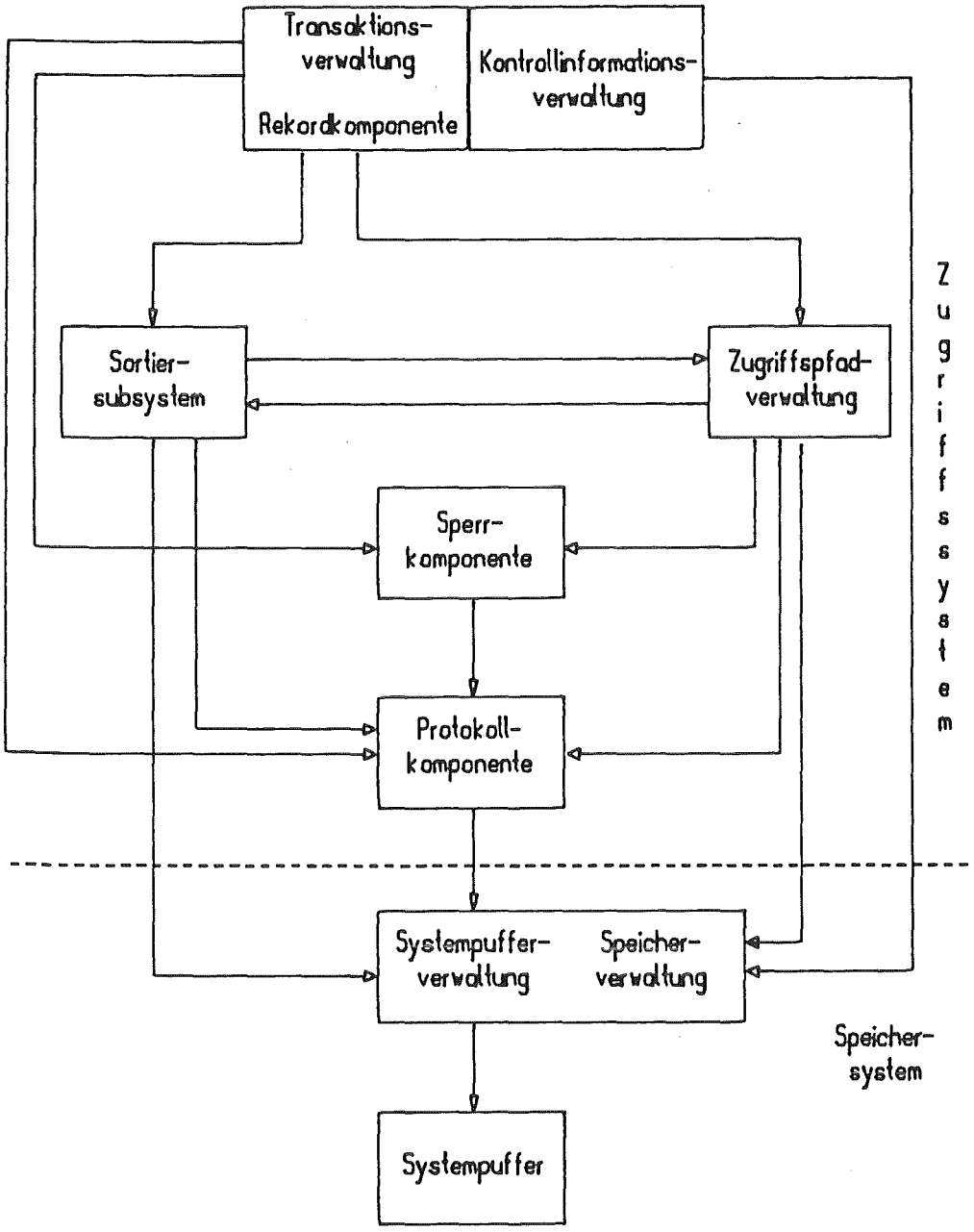


Bild 1.4: Architektur eines Zugriffs- und Speichersystems (nach /Hä 78/)

den Implementierungsmerkmalen der Speicherungs- und Zugriffspfadstrukturen bietet.
Datenbanksysteme mit einem Netzwerk- oder hierarchischen Datenmodell bieten eine solche Ein-Tupel-Schnittstelle bereits als externe Anwenderschnittstelle an. Für relationale Datenbanksysteme ist sie eine DBMS-interne Schnittstelle, die geeignete Objekte und Operationen für die Implementierung des Datensystems bereitzustellen hat. Härder schlägt hierfür vor (für eine detaillierte Beschreibung sei auf /Hä 78/ verwiesen):

- Tupel und Relationen als "Träger der eigentlichen Information der

Miniwelt"
- Segmente zur Clusterbildung von Tupeln und Relationen
- Listen als Hilfsmittel zur schnellen, fortlaufenden Verarbeitung von Tupeln, z.B. bei der Sortierung
- Index als Zugriffspfadtyp zur Auswahl von Tupeln einer Relation und Sortierung der Tupel einer Relation nach einem oder mehreren Attributen
- Link als Zugriffspfadtyp, der Tupeln in zwei Relationen verbindet, zur Unterstützung des navigierenden Zugriffs zwischen Relationen.

Die Architektur des Zugriffssystems zeigt Bild 1.4:
Die Rekordkomponente und Zugriffspfadverwaltung sind für die Speicherung und Wartung von Tupeln bzw. Zugriffspfaden verantwortlich; sie implementieren die Speicherungsstrukturen des DBMS und führen die Abbildung der Objekte der Ein-Tupel-Schnittstelle auf die Objekte der Speicher-Schnittstelle durch. Da es "keine in jeder Hinsicht optimale Speicherungstechnik gibt, sondern die jeweiligen Anwendungen die Eignung der einzelnen Speicherungsstrukturen oder Zugriffspfade in hohem Maße bestimmen, hat folglich ein Datenbanksystem ein Spektrum von verschiedenen Strukturen, die eine breite Anwendungsmöglichkeit versprechen, anzubieten" (/Hä 78/, p.69).
Rekordkomponente und Zugriffspfadverwaltung bieten also jeweils mehrere Möglichkeiten zur Implementierung der durch sie realisierten Objekte (Relation, Tupel bzw. Index, Link), es wird für diese beiden Subsysteme ein Aufbau nach Bild 1.5 zugrunde gelegt (vgl. "Konstruktionsprinzip zur Flexibilität" von /Sc 76/):
Für ein Objekt O gebe es die n Möglichkeiten I1, I2, ..., In zur Implementierung, für jede Art der Implementierung sei ein Algorithmus (A1, A2, ..., An in Bild 1.5) bereitgestellt. Die Kontrollinformation beschreibt die momentan existierenden Objekte, insbesondere ihre Implementierung:
Operationen auf ein Objekt O werden an einen Verteiler erteilt, dieser ermittelt über die Kontrollinformationsverwaltung, auf welche Art dieses Objekt implementiert ist und ruft den entsprechenden Algorithmus auf.

Die Kontrollinformation wird in "System-Katalogen" unter Benutzung bereits vorhandener Objekttypen (z.B. Relationen, vgl. auch /SW 76/) abgespeichert und vom Subsystem Kontrollinformationsverwaltung des Zugriffssystems verwaltet; es bietet Operationen zum
- Einfügen von Kontrollinformation (als "Kontrolltupel")
- selektiven Lesen der Beschreibung eines Objektes
- Ändern der Beschreibung eines Objektes
- Löschen der Beschreibung eines Objektes

Aufgabe des Sortier-Subsystems ist die Bereitstellung sortierter Folgen von Tupeln; es unterstützt damit auch die effiziente Ausführung von Relationenoperationen durch das Datensystem.

Die Sperr- und Protokoll-Komponenten sorgen für die Konsistenz der Transaktion, für die Einhaltung von Sperrprotokollen, für die Entdeckung und Auflösung von Verklemmungen, für das Zurücksetzen und Wiederherstellen von Transaktionen und für die Sicherheit bei Systemfehlern. Die Transaktionsverwaltung garantiert mit Hilfe der übrigen Komponenten die Transaktion eines Benutzers als Einheit der Konsistenz.

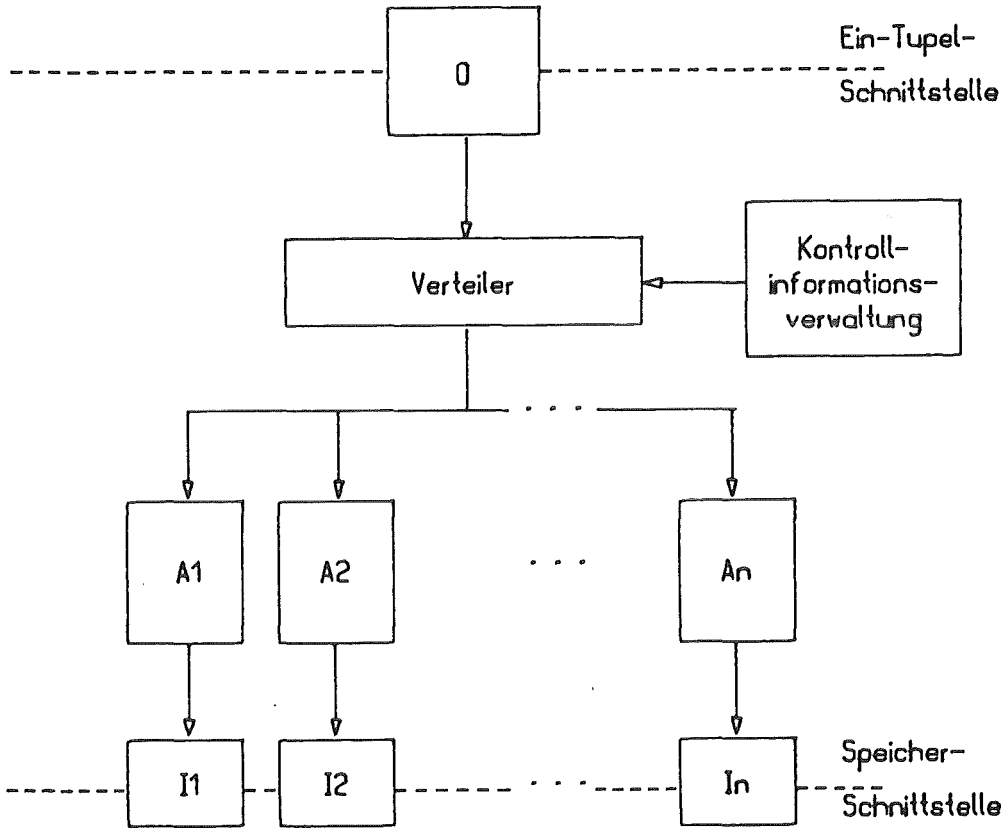


Bild 1.5: Prinzipieller Aufbau der Subsysteme Rekordkomponente und Zugriffspfadverwaltung

Die virtuelle Maschine D a t e n s y s t e m implementiert die Schicht S4: sie übersetzt die Anweisungen der deskriptiven Sprache der Anwenderschnittstelle, so daß sie mit Hilfe der Operationen der Ein-Tupel-Schnittstelle ausgeführt werden können. /Hä 78/ gibt hierzu drei Schritte an (Bild 1.6):

(1) Syntaxanalyse
(2) Optimierung der Anfrage; dieser Schritt wird nochmals unterteilt in
    - "algebraische Optimierung",
    - "nicht-algebraische Optimierung" und
    - "Optimierung der Durchführung der einzelnen Operationen"
(3) Codeerzeugung, d.h. Erzeugung eines Programms für die Ausführung der Anweisungen auf der Ein-Tupel-Schnittstelle, mit Programmausführung

Weiterhin übernimmt das Datensystem die Zugriffs- und Integritätskontrolle.

Später wird benötigt:
Das i n t e r n e (DB-) S c h e m a ist die Beschreibung der Realisierung der Datenbank "in einem Speichersystem unter Einbeziehung sämtlicher physikalischer Angaben" /LM 78/; hierzu gehören z.B. die Kontrollinformation des Zugriffssystems (Realisierung von Objekten durch

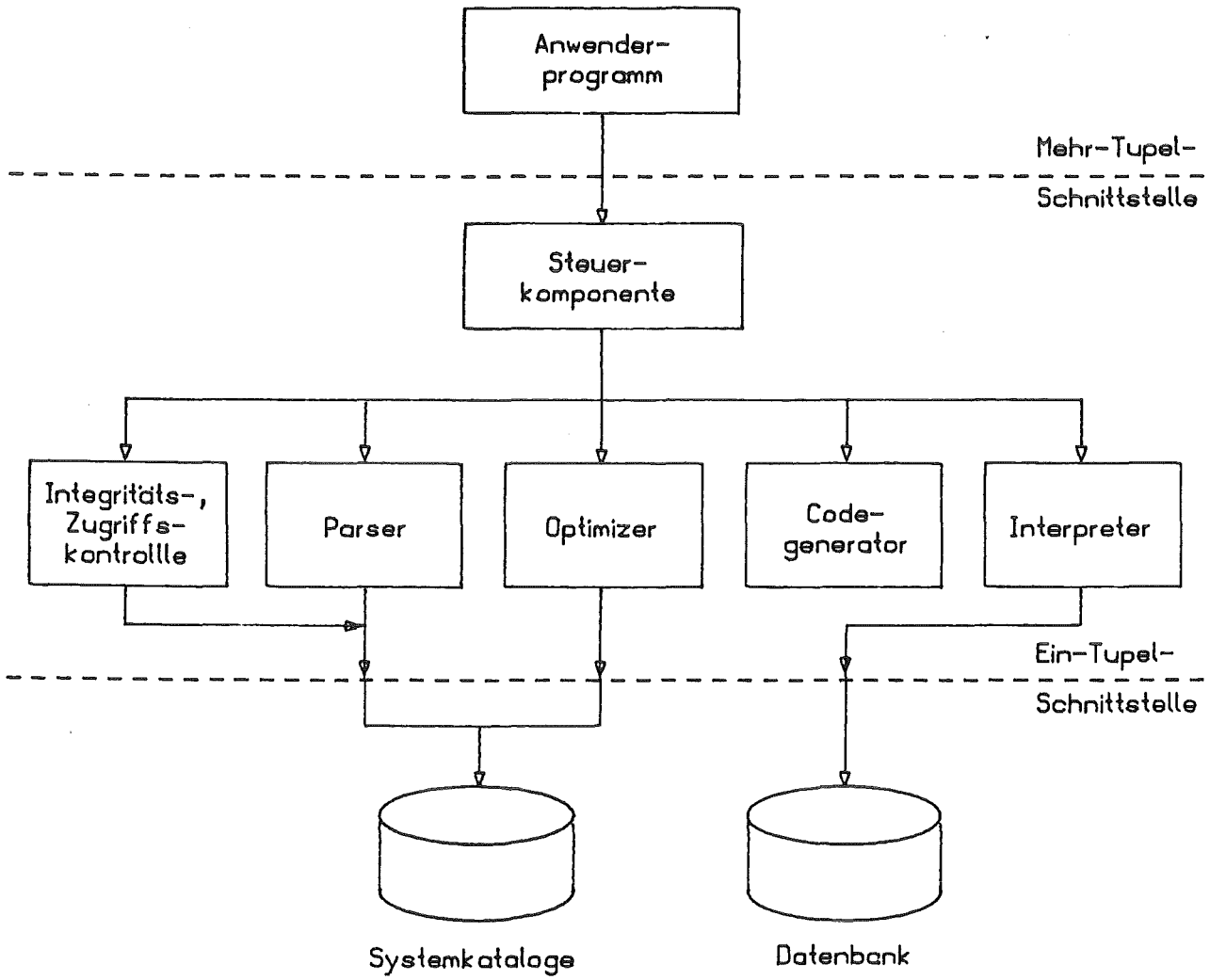


Bild 1.6: Aufbau des Datensystems (nach /Hä 78/)

Speicherungsstrukturen bzw. Zugriffspfadtypen), Größen (in Bytes) von Seiten, Blöcken u.ä.

Zur Beachtung: "DB-Schema" bezeichnet in dieser Arbeit stets das Schema der Anwenderschnittstelle!

1.3. Problemstellung, Übersicht

Ein DBMS nach Abschnitt 1.2 stellt offensichtlich ein komplexes und umfangreiches Programmsystem dar, eine Implementierung auf Kleinrechnern wird durch die für diese Rechnerklasse typischen Restriktionen vor allem bezüglich des verfügbaren Hauptspeichers und der maximalen Programmlänge zumindest erschwert.
Hierauf geht /Fi 79/ im Zusammenhang mit der Übertragung des Datenbanksystems ADABAS auf einen Kleinrechner vom Typ PDP-11 ein. ADABAS, obwohl es nur zum Teil die Fähigkeiten eines DBMS nach Abschnitt 1.2 realisiert, belegt auf einer IBM 370 einen Hauptspeicherbereich von ca. 200kB, auf einer PDP-11 ist man jedoch gezwungen, das System in einem Adreßbereich von maximal 64kB ablaufen zu lassen. Wie in /Fi 79/ im einzelnen ausgeführt ist, hat man daher mit "ADABAS-M" auf dem Kleinrechner eine ADABAS-Version zur Verfügung gestellt,

- die nur einen Teil des Funktionsumfangs der Großrechner-Version realisiert
- deren "Systemkonstanten" (im wesentlichen Puffergrößen, z.B. "maximale Satzlänge") kleiner gewählt worden sind
- die drei overlay-Bereiche benötigt.

Hiermit sind bereits die Möglichkeiten und Techniken zur Reduktion des Codeumfangs genannt, die bei der Konzipierung bzw. Implementierung von Datenhaltungssoftware für Kleinrechner eingesetzt werden:

(1) Abmagern von virtuellen Maschinen
   - Es werden nicht alle Operationen und Funktionen eines DBMS nach 1.2. angeboten, im Extremfall führt dies zum Entfernen einer ganzen virtuellen Maschine. So ist ganz besonders für Kleinrechner festzustellen (vgl. /CN 80/), daß die zur Zeit verfügbaren Systeme zur Datenhaltung meist niedere, im Sinne von Abschnitt 1.2 "interne" Schnittstellen realisieren und in der Regel Funktionen wie Zugriffskontrolle, Wahrung der Datenintegrität, Verhinderung bzw. Erkennung von Verklemmungen nicht oder eingeschränkt anbieten.
   - Vereinfachung der Abbildungsprozesse:
     Die Zahl der Algorithmen zur Implementierung virtueller Maschinen wird verringert. So kann z.B. der für die Rekordkomponente erforderliche Codeumfang durch Bereitstellung nur weniger Speicherungsstrukturen reduziert werden, Entsprechendes gilt für die Zugriffspfadverwaltung (vgl. Bild 1.5). So stellt mehr als die Hälfte der in /SF 78/ untersuchten Kleinrechner-DBMS nur eine einzige Speicherungsstruktur zur Verfügung.
     Das Datensystem kann vereinfacht werden, indem auf eine Optimierung von Anfragen gänzlich verzichtet wird, d.h. die Komponente "Optimizer" von Bild 1.6 vollständig entfällt, oder nicht alle Möglichkeiten zur Optimierung (vgl. 1.2., Datensystem) bereitgestellt werden.

(2) Verkleinern von DBMS-Puffern
   Ein DBMS enthält Puffer, deren Zweck es "lediglich" ist, auf Externspeichern abgelegte Daten im Arbeitsspeicher (Hauptspeicher bzw. virtueller Speicher) präsent zu halten; hierzu gehört insbesondere auch der Systempuffer des Speichersystems. Eine Reduktion der Größe solcher DBMS-Puffer verringert zwar die Programmlänge, wirkt sich aber i.allg. negativ auf die Effizienz des

Systems aus /GS 70/.
Zusätzlich kann die Verkleinerung von Puffern, Tabellen etc. aber auch die "Leistungsfähigkeit" des DBMS verringern, die durch Systemkonstanten wie "maximale Satzlänge", "maximale Zahl konkurrierender Transaktionen" beschrieben wird (vgl. "Leistungskenndaten" von Abschnitt 2.2).

(3) Code-Segmentierung
Der erforderliche Arbeitsspeicherbedarf kann durch die Einführung von overlays oder die Implementierung des DBMS durch mehrere Rechenprozesse ("tasks") klein gehalten werden /SW 76/. Das dann erforderliche Ein- und Austransferieren ("swapping" /Br 73/) von Programmen bzw. Programmteilen kann das Laufzeitverhalten drastisch verschlechtern /Sa 80/; selbst wenn bei Implementierung durch mehrere tasks alle tasks im Hauptspeicher verbleiben können, hat die Intertask-Kommunikation eine Verschlechterung der Effizienz zur Folge /St 80/ und stellt eine zusätzliche Belastung für das Gesamtsystem dar.

Die Notwendigkeit, mit dem beschränkt verfügbaren Speicherplatz ökonomisch umzugehen, hat dazu geführt, daß die auf Kleinrechnern angebotenen Systeme zur Datenhaltung den Anforderungen und Bedürfnissen gerade des Kleinrechner-Anwenders nicht genügen:

- "Für Mini-Computer wird mini Datenbank-Software angeboten" /De 79/
- " ... the limited amount of available software and the quality of the available software typify minicomputer systems" /SF 78/.

Aber auch auf Großrechnern, wo dieses Speicherplatz-Argument entfällt, ist die Situation unbefriedigend: DBMS "haben sich häufig als wenig anpassungsfähig, aufgrund ihrer Größe als langsam, unhandlich und ... nicht so recht wirtschaftlich erwiesen" /LH 81/.

Unabhängig von der Rechnergröße ist der Grund für diese Schwierigkeiten darin zu suchen, daß DBMS monolithische "Universal"-Systeme darstellen, die nicht an das Anforderungsprofil einer speziellen Anwendung anpaßbar sind; dies führt

- auf Großrechnern zu aufwendigen Systemen, von denen oft nur ein Bruchteil des potentiellen Leistungsumfangs für eine spezielle Anwendung ausgenutzt wird
- bei Kleinrechnern zu Systemen mit eingeschränktem Funktions- und Leistungsumfang.

Aus der Arbeit zur Entwicklung des Datenbanksystems FADABS /Po 78/ und den Erfahrungen mit seinem Einsatz als Komponente dedizierter Systeme /JL 80/ (Informationssysteme zur Kernmaterialüberwachung, die auf Kleinrechnern realisiert sind), entstand die Anregung, zur Verbesserung dieser Situation Datenbanksysteme (ähnlich wie Betriebssysteme, vgl. /RS 80, Ne 77/) an die jeweiligen Erfordernisse von Anwendungen anpaßbar zu machen.
Die Idee besteht darin auszunutzen, daß ein dediziertes System als Ganzes durchaus den vollen Leistungsumfang eines existierenden, monolithischen DBMS erfordern kann, für jede einzelne Phase aber jeweils ein Teil seiner Fähigkeiten ausreicht (vgl. 1.1.). Ein g e n e r i e r b a r e s Datenbanksystem (GDBMS) soll es erlauben, V e r s i o n e n eines umfassenden (oder auch: "vollständigen") DBMS zu erzeugen, die genau auf das Anforderungsprofil einer gegebenen Anwendung hin zugeschnitten sind,

also in der Regel nur einen Teil der Operationen der Anwenderschnittstelle des vollständigen DBMS realisieren und als Programmsystem nur die tatsächlich erforderlichen Programmteile enthalten. Es bietet sich damit die Möglichkeit,

- DB-Anwendungssysteme unter Einsatz eines DBMS auch in einer Umgebung zu realisieren, in der das vollständige DBMS nicht oder nicht mehr sinnvoll betrieben werden könnte
- durch "Weglassen" von Subsystemen, die für eine Anwendung "überflüssigen overhead" darstellen (z.B. Protokollkomponente, Optimizer), die Effizienz zu verbessern und das Betriebssystem zu entlasten.

Die Anpassung eines DBMS an ein gegebenes Anforderungsprofil wird als D e d i z i e r u n g des DBMS bezeichnet. Es sind drei Fälle zu unterscheiden:

1) Die Installation eines Anwendungssystems beinhaltet u.a. das Einrichten der Datenbank. Zur Dedizierung genügt in diesem Fall also die Generierung einer an das Anforderungsprofil der Anwendung angepaßten DBMS-Version, eine bereits existierende Datenbank ist nicht zu berücksichtigen. Die Dedizierung bei Initialisierung eines Anwendungssystems wird I - D e d i z i e r u n g genannt.
   Eine I-Dedizierung reicht offensichtlich aus, wenn die Anforderungen an die Datenhaltung für die gesamte Lebensdauer der Anwendung bekannt sind und sich nicht ändern. Dies ist z.B. bei Anwendungssystemen, wie den in 1.1. angeführten Branchenpaketen, der Fall, die eine (durch Vorschriften, Gesetze o.ä.) fest definierte Dienstleistung erbringen und wo Weiterentwicklungen und Modifikationen nicht vorgesehen oder zu erwarten sind.
   Die Dedizierung des DBMS kann hier vom Ersteller des Gesamtsystems bzw. vom DBMS-Hersteller anhand der Spezifikation des Anwenders vorgenommen werden.

2) DV-Systeme unterliegen im allgemeinen nicht vorhersehbaren Änderungen, die sich aus geändertem Benutzerverhalten und/oder neuen Anforderungen ergeben /WS 82/; dies trifft ganz besonders in der ersten Zeit nach Inbetriebnahme des Anwendersystems zu, wo Unzulänglichkeiten, etwa bezüglich des Antwortzeit-Verhaltens, zu beseitigen sind oder neue Benutzerwünsche angemeldet und berücksichtigt werden. Solche Modifikationen können zu einem neuen Anforderungsprofil führen, dem die bis dahin verwendete DBMS-Version nicht mehr gerecht wird. Es ist dann eine "Re-Dedizierung", R - D e d i z i e r u n g vorzunehmen:
   Wie bei der I-Dedizierung gehört hierzu die Generierung einer auf das neue Anforderungsprofil hin zugeschnittenen DBMS-Version. Weiterhin ist es im allgemeinen erforderlich, das interne DB-Schema zu ändern: neue Anforderungen können z.B. die Implementierung von Objekten der Anwenderschnittstelle durch andere Speicherungsstrukturen bedingen. Es ist sicherzustellen, daß mit der neuen DBMS-Version auf die DB zugegriffen werden kann, insbesondere das Zugriffssystem die erforderlichen Algorithmen enthält.
   Durch eine R-Dedizierung wird also die Komponente DBMS des Anwendungssystems (Bild 1.1) gegen eine neue Version ausgetauscht, der Inhalt der Datenbank und das DB-Schema bleiben davon unberührt.
   Eine R-Dedizierung wird sinnvollerweise "vor Ort", auf der Rechenanlage des Anwenders vorgenommen (etwa von einem "Anwendungssystem-Verwalter").

3) Anwendungssysteme, die in wiederkehrenden Betriebsphasen mit unterschiedlichen Anforderungsprofilen ablaufen (Abschnitt 1.1), benötigen für jede Betriebsphase eine eigene DBMS-Version. Da die Anforderungsprofile der Phasen im voraus (prinzipiell bereits mit der Systemanalyse) bekannt sind, bedeutet Dedizierung in diesem Fall Generierung mehrerer DBMS-Versionen (sonst müßten bei jedem Phasenwechsel die jeweiligen Versionen wieder neu generiert werden!). Es wird deshalb hier von M - D e d i z i e r u n g ("Mehrfach-Dedizierung") gesprochen.
Das Anwendungssystem enthält also mehrere DBMS-Versionen, von denen jedoch maximal eine, die der aktuellen Betriebsphase, aktiv ist, anders als bei R-Dedizierung findet hier bei Phasenwechsel keine Erzeugung einer DBMS-Version statt; wie bei R-Dedizierung kann auch hier ein Phasenwechsel, also ein Wechsel der DBMS-Version, mit einer Änderung des internen DB-Schemas verknüpft sein.

Ziel dieser Arbeit ist es, zu untersuchen, wie ein DBMS in diesem Sinne anpaßbar, d.h. zu einem GDBMS gemacht werden kann. Es sollen hierbei keine besonderen Annahmen über das DBMS, etwa das Datenmodell, gemacht werden, insbesondere soll diese Technik auf bereits existierende Systeme anwendbar sein. Es wird lediglich von der Modellbildung nach 1.2. ausgegangen.
Wie oben dargestellt, erfordert die Dedizierung eines DBMS die Generierung, d.h. Spezifikation und Erzeugung eines Programmsystems, das nur einen Teil der Anwender-Schnittstelle des DBMS implementiert; weiterhin: die Dedizierung nimmt in der Regel der Anwendungsprogrammierer vor, zur Dedizierung sollte die Kenntnis der Anwender-Schnittstelle genügen (vgl. 1.1). Insbesondere verbietet sich hierbei manuelle Programmierung, die Programmierung hat vielmehr rechnergestützt zu erfolgen; da die Dedizierung i.allg. auf der Rechenanlage des Anwenders stattfindet, ergibt sich als weitere Anforderung, daß zumindest die Konzepte und Techniken zur Generierung von DBMS-Versionen rechnerunabhängig sein sollten: die Generiertechnik und das Generiersystem müssen ebenso portabel wie das DBMS selbst sein!

Nach einem Überblick über den Stand der Technik zum rechnergestützten Programmieren werden in Abschnitt 2 die Besonderheiten der Generierung von Versionen eines DBMS aufgezeigt und die Architektur eines Generiersystems hierzu skizziert:
- Die Spezifikation einer Version erfolgt als ein "Auswählen" der erforderlichen Fähigkeiten des vollständigen DBMS
- Die Programmerzeugung selbst wird analog als ein Auswählen der nach der Spezifikation erforderlichen Programmteile des Programms des vollständigen DBMS verstanden.

Die Aufgabe der Programmerzeugung wird hier als das allgemeine Problem der Erzeugung von Teilsystemen untersucht:
Abschnitt 3 behandelt zunächst die Zerlegung des Programmtextes eines vollständigen Systems PS in sog. "Fragmente", die die Bausteine zur Erzeugung der Teilsysteme von PS darstellen. Es wird ein heuristisches Verfahren hierzu angegeben und aufgezeigt, inwieweit Möglichkeiten zur Automatisierung der Definition von Fragmenten gegeben sind. Hiervon ausgehend wird mittels sog. "abstrakter B-Programme" die Erzeugung von

Teilsystemen formal als ein Durchsuchen einer Baumstruktur eingeführt.

Zwischen den Fragmenten eines Programmsystems bestehen Beziehungen (die z.B. den Kontrollfluß widerspiegeln), die die Menge der möglichen Teilsysteme beschreiben: nicht jede beliebige Teilmenge von Fragmenten ergibt ein funktionsfähiges Teilsystem. Zur Modellierung dieser Beziehungen in Abschnitt 4 dienen sog. "Fragmentsysteme". Hiermit kann die Existenz von Teilmengen von Fragmenten nachgewiesen werden, die die Eigenschaft haben, daß mit der Angabe, welche von diesen für ein Teilsystem benötigt werden, bereits alle Fragmente für dieses Teilsystem festgelegt sind. Insbesondere erlaubt dies eine explizite Beschreibung der Menge aller Teilsysteme. Ein Verfahren zur Konstruktion einer minimalen Teilmenge mit dieser Eigenschaft wird angegeben.

Abschnitt 5 behandelt die Implementierung des Verfahrens zur Erzeugung von Teilsystemen, es wird ein Generiersystem entwickelt, das eine automatische Programmerzeugung (Erzeugung von Lademoduln) unter Beachtung der genannten allgemeinen Anforderungen leistet.

Es ist oben auf den Zusammenhang zwischen dem internen Schema einer Datenbank und der Generierung von Versionen eines DBMS hingewiesen worden: nicht jede beliebige Version kann mit einem Anwendersystem bei gegebener Datenbank als Komponente DBMS (Bild 1.1) eingesetzt werden. Abschnitt 6 gibt an, welche Versionen mit einem Anwendersystem und einer DB "verträglich" sind.

2. Die Generierung von DBMS-Versionen

Zur Dedizierung eines DBMS ist eine DBMS-Version in Form eines ausführbaren Programmsystems, eines L a d e m o d u l s [1], zu erzeugen. Es sollen hierzu lediglich Kenntnisse der Anwenderschnittstelle erforderlich sein, insbesondere verbietet sich eine manuelle Programmierung.
Die Programmerzeugung hat also rechnergestützt, möglichst weitgehend automatisch zu erfolgen. Nach einer Übersicht über den Stand der Technik auf dem Gebiet des rechnergestützten Programmierens wird auf die Generierung von DBMS-Versionen eingegangen und hierbei zu lösende Probleme aufgezeigt.

2.1. Rechnergestützte Programmierung

Schon mit Beginn der Entwicklung der EDV war man bestrebt, zur Reduktion des Aufwandes und der Kosten seiner Programmierung den Rechner selbst einzusetzen. Es entstanden so Hilfsmittel wie Assembler, Compiler, Binder, Editoren, etc., die den Anwendungsprogrammierer von mechanischen, wenig kreativen Tätigkeiten entlasten und es ihm so ermöglichen sollen, sich auf die Lösung seines eigentlichen Problems zu konzentrieren.
Das Fernziel der Forschung zum Thema "rechnergestützte Programmierung" sind Systeme, die es dem EDV-Laien ermöglichen, von ihm benötigte DV-Systeme selbst zu erstellen, indem er in der ihm vertrauten Terminologie des Anwendungsgebiets seine Anforderungen formuliert /Pr 77, Go 74/; aus dieser Beschreibung des Anwenders, zu deren Erstellung also keinerlei Programmier- oder Rechnerkenntnisse erforderlich sein sollen, erzeugen solche Systeme das gewünschte DV-System, ohne Beteiligung von Anwendungsprogrammierern (Goldberg spricht in diesem Zusammenhang von einer "programmerless environment" /Go 74/). Prywes /Pr 77/ bezeichnet derartige Programmiersysteme "automatic program generation systems", A P G - S y s t e m e ; ein APG-System besteht nach Prywes zumindest konzeptionell aus zwei Teilen, einem "top part" und einem "bottom part" (vgl. /Pr 77/):

- der Anwender des zu erzeugenden DV-Systems teilt dem APG-System die Aufgabenstellung und seine Anforderungen mit. Der top part klärt etwaige Mehrdeutigkeiten und Widersprüche durch Fragen oder Vorschläge und erzeugt dann hieraus eine Spezifikation des erforderlichen Programmsystems. Es steht ihm hierzu eine "Wissensbasis" über das

---

[1] Ein DBMS kann prinzipiell als selbständiger Rechenprozeß oder als "Unterprogramm" (das in die Anwendersoftware eingebunden wird) realisiert sein, vgl. "Stellung des Datenbanksystems zum Anwenderprogramm und Betriebssystem" in /Hä 78/. Im ersten Fall ist ein ladefähiges, im zweiten ein bindefähiges Objekt zu erzeugen. Diese Unterscheidung ist für das folgende unwesentlich, so daß mit "Lademodul" auch bindefähige Objekte (Objektmoduln) bezeichnet werden.

Anwendungsgebiet ("application knowledge base") zur Verfügung.
- der bottom part erzeugt aus der Spezifikation des Programmsystems, eventuell nach Kommunikation mit dem Anwender, das gewünschte Programmsystem; dies geschieht in den von der "konventionellen" Pogrammierung her bekannten Schritten "Systementwurf und -optimierung" (Spezifikation der input- und output-Daten, Programme und Dateien), "Modulentwurf und -optimierung" und schließlich "Codierung".
  Der bottom part benutzt hierzu eine allgemeine Wissensbasis über Logik, Mathematik, Programmierung.

Derartige allgemeine, "ideale" APG-Systeme haben bisher rein experimentellen Charakter und sind für den praktischen Einsatz noch nicht geeignet /Pr 77, Bo 81, Pe 81/. Prywes weist jedoch darauf hin, daß APG-Systeme umso erfolgreicher sind, je mehr der vorgesehene Anwendungsbereich eingeschränkt ist.

Ein Ansatz, bei dem man sich zur rechnergestützten Programmerzeugung auf eine bestimmte Klasse von Anwendungssystemen beschränkt, besteht in der Entwicklung "anwendungsorientierter sehr hoher Programmiersprachen" /Pe 81/ V H L L ("very high level languages" bei /Bo 81/):
Wie bei APG-Systemen formuliert der Benutzer des zu erstellenden Anwendungssystems selbst seine Aufgabenstellung. Er benutzt hierzu eine VHLL, die Mittel und Konzepte zur Beschreibung einer ganz bestimmten Klasse von Anwendungssystemen bereitstellt. Die application knowledge base eines APG-Systems ist hier also mit den Konzepten und Sprachmitteln der VHLL gegeben.
Auch dieser Ansatz hat noch experimentellen Charakter, zu den noch weitgehend ungelösten Problemen gehören z.B.
- die Techniken, anhand derer der Anwender, bei dem ja keine Programmierkenntnisse vorausgesetzt werden, das von ihm gewünschte System beschreibt
- die Übersetzung der resultierenden VHLL-Programme in effizient ausführbaren Code /Go 74/.

Im praktischen Einsatz dagegen befinden sich Systeme zur rechnergestützten Programmierung, die Anwendungssysteme durch Anpassung von Programmpaketen an Benutzerwünsche erzeugen /Bo 81, Ja 80, Pe 81/. Sie werden hier als P r o g r a m m - G e n e r a t o r e n bezeichnet ("application generator" bei /Bo 81/). Ein Programm-Generator besteht im Prinzip (vgl. Bild 2.1) aus
- einer Menge von Programm-Bausteinen ("program fragments" bei /Go 74/) zu einem bestimmten Anwendungsbereich
- einem Generiersystem, das den Wünschen des Anwenders entsprechend durch Auswahl von Bausteinen, deren Modifikation und/oder Parametrisierung ein maßgeschneidertes Anwendungssystem erzeugt.

Dem Einsatz eines Programm-Generators geht also die Definition und Erstellung, d.h. Programmierung, der Programm-Bausteine voraus (Bild 2.1), Pernard nennt dies die "Entwurfsphase" /Pe 81/. Ziel der Entwurfsphase ist die Erstellung einer "umfassenden Problemlösung", dies sind die Bausteine ("Applikationsmoduln"), aus denen sich Anwendungssysteme generieren lassen. Eine Problemlösung wird dabei nach /Pe 81/

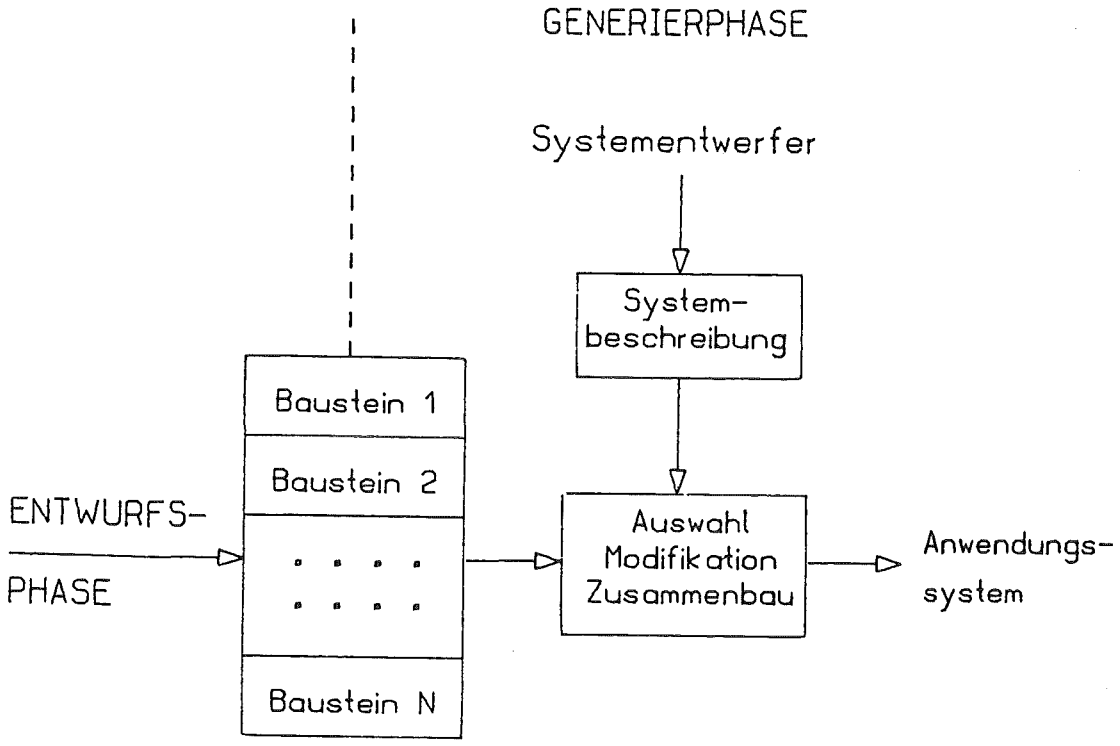


Bild 2.1: Die Erzeugung von Anwendungssystemen mit Programm-Generatoren

bereits "umfassend" genannt, wenn die Applikationsmoduln "eine Problembereichsabdeckung von etwa 60% bis 80%" ermöglichen. Diese Werte stimmen mit den Erfahrungen von /LD 81/ über die Verwendung von "reusable code" überein.
Die Erstellung und Verwendung von Programm-Generatoren ist lohnend, wenn man vor der Aufgabe steht, eine größere Anzahl von DV-Systemen zum gleichen Aufgabenbereich zu produzieren.
Dies ist z.B. in der kommerziellen Datenverarbeitung der Fall /Pe 81/; so weist /Ja 80/ darauf hin, daß hier in verschiedenen Teilbereichen (er nennt als Beispiel das Rechnungswesen) die Bedürfnisse vieler Anwender identisch sind, während in anderen (z.B. Auftragsabwicklung) individuelle Lösungen (und damit spezielle Software) unumgänglich sind. Ein Programm-Generator übernimmt hierbei die Aufgabe der Erzeugung der allgemeinen Programmteile aus den Programm-Bausteinen und deren Integration mit der anwendungsspezifischen Software zum gewünschten Anwendungssystem.
In der Literatur wird weiterhin berichtet über die Verwendung von Programm-Generatoren zur

- Erzeugung von maßgeschneiderter Software für Systeme zur Rechnerkommunikation ("data communications networks", /DK 72/) und "switching systems" /Ka 78, Ko 79/
- Lösung von Automatisierungsaufgaben in der Anlagentechnik /Re 77/
- zur automatischen Generierung problemangepaßter Prozeßrechner-Betriebssysteme /RS 80/.

2.2. Die Generierung von DBMS-Versionen

Die Generierung von Versionen eines DBMS erfordert nach Abschnitt 1.3 die Erzeugung von Versionen des vollständigen DBMS, die nur einen Teil der Anwenderschnittstelle dieses Systems realisieren. Dies ist also ein Spezialfall der Programmerzeugung mittels Programm-Generatoren: in der Terminologie von 2.1. ist hier eine umfassende Problemlösung mit einer Problembereichsabdeckung von 100% gegeben, nämlich das vollständige System, aus dem durch Auswahl der erforderlichen Teile die gesuchten Programmsysteme, hier DBMS-Versionen, erzeugt werden (vgl. Bild 2.1).
Analog zur allgemeinen Vorgehensweise von 2.1. erfolgt die Generierung einer DBMS-Version in zwei Schritten:

- Spezifikation der gewünschten Version, in der Regel durch den Anwender (vgl. 1.3.)
- die eigentliche Programmerzeugung, Erzeugung eines Lademoduls, anhand der Spezifikation.

Der erste Schritt, die Erstellung einer Beschreibung der gewünschten DBMS-Version läßt sich (zumindest konzeptionell) in drei Teilaufgaben zerlegen (vgl. Bild 2.2):

D1: Bestimmung der erforderlichen Operationen:
Es ist anzugeben, welche Operationen der Anwenderschnittstelle für die gegebene Anwendung erforderlich sind.
Eine Liste der gewünschten Operationen kann bereits im Verlauf der Systemanalyse oder des -entwurfs erstellt werden; liegen die Quellprogramme des Anwendersystems (Bild 1.1) vor, so kann die Herleitung der Anwenderwünsche rechnergestützt erfolgen, etwa unter Benutzung eines DML-Vorverarbeiters (preprocessor).
Das Generiersystem überprüft die Liste der gewünschten Operationen auf Korrektheit und Vollständigkeit und ermittelt hieraus, gegebenenfalls nach Interaktion mit dem Anwender, die Menge der "erforderlichen Operationen".

D2: Bestimmung der erforderlichen DBMS-Algorithmen:
a) Zu jeder der nach D1 erforderlichen Operationen ist anzugeben, in welcher Weise sie implementiert sein soll, d.h. welche Algorithmen die DBMS-Version zu ihrer Realisierung enthalten soll. Zu Operationen, für die das vollständige DBMS keine Alternativen zur Implementierung anbietet, erübrigen sich weitere Angaben, sonst ist die gewünschte Realisierung zu spezifizieren. Insbesondere sind die durch die Subsysteme Rekordkomponente und Zugriffspfadverwaltung bereitzustellenden Algorithmen anzugeben, es besteht hier also ein enger Zusammenhang zwischen Programmgenerierung und physischem DB-Entwurf.
b) Die Festlegungen nach a) können als Verfeinerung der Spezifikation nach D1 betrachtet werden. I.allg. erhält man aber mit der Spezifikation der Implementierung der erforderlichen Operationen nicht alle für eine DBMS-Version erforderlichen Algorithmen, da es nicht zu allen Funktionen eines DBMS explizit eine Operation der Anwenderschnittstelle geben muß: die Frage, ob z.B. die Protokollkomponente vollständig, zum Teil (etwa nur zur Führung einer

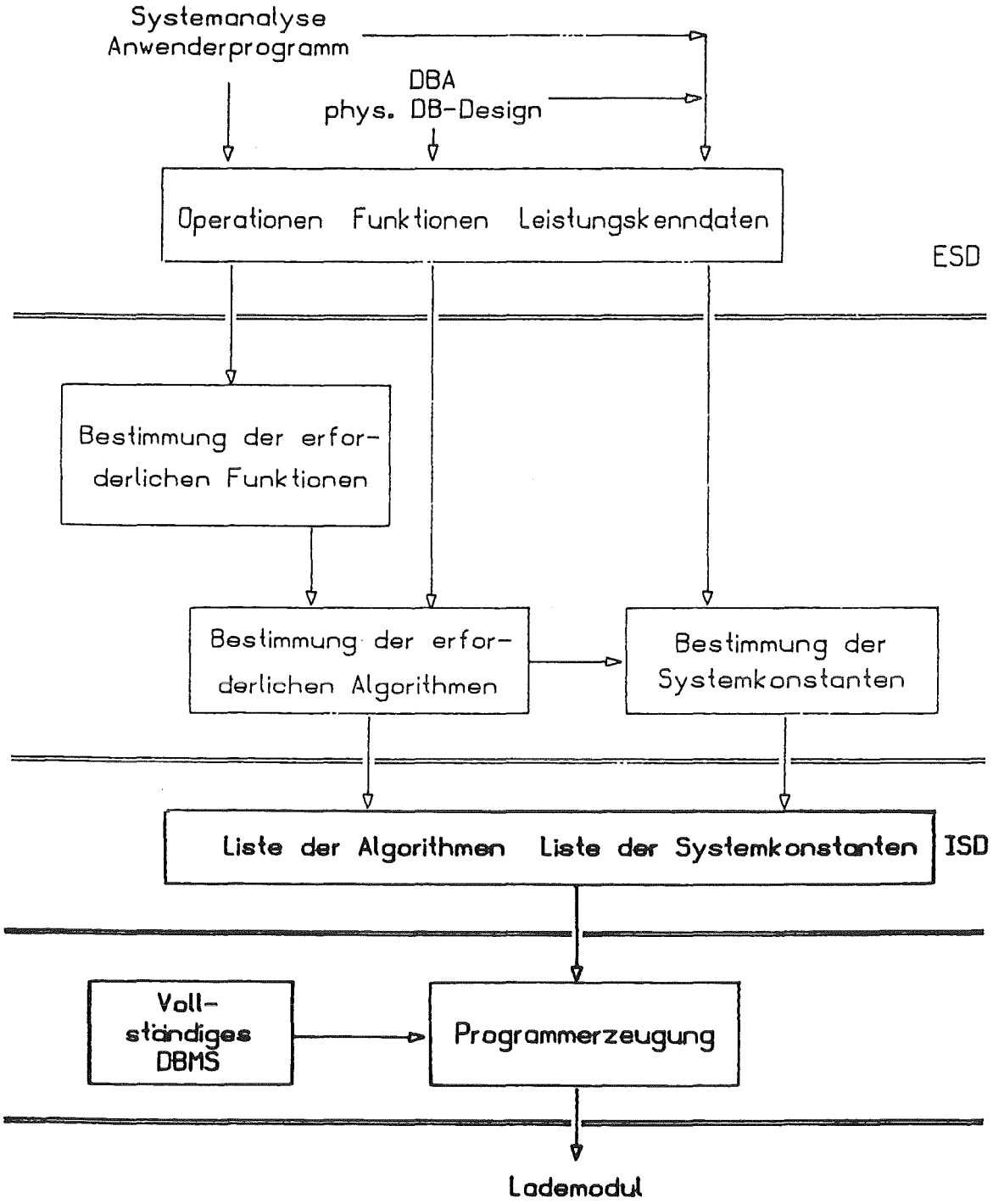


Bild 2.2: Die Generierung von DBMS-Versionen

Log-Datei) oder gar nicht erforderlich bzw. erwünscht ist, kann den DML-Anweisungen der Anwenderprogramme als solchen nicht notwendigerweise entnommen werden.

D3: Bestimmung der Leistungskenndaten
Die "funktionale" Beschreibung nach D1 und D2 ist zu ergänzen um "quantitative" Angaben. Diese können sein
- funktionsspezifisch: z.B. die maximale Zahl von Prädikaten in Qualifikationen bei Anfragen, die maximale Satzlänge für Retrieval- oder Update-Operationen, die obere Grenze für die maximale Satzanzahl bei Sortierläufen
- systemorientiert: z.B. die maximale Zahl gleichläufiger Transaktionen, die maximale Zahl gleichzeitig gesperrter Datenobjekte, maximale Zahl gleichzeitig eröffneter Dateien und Segmente.

Die zu D2 und D3 zu treffenden Entscheidungen und Festlegungen verlangen sowohl Kenntnis der Erfordernisse der Anwendung (etwa inwieweit das DBMS für Datensicherheit, Schutz vor unberechtigtem Zugriff, Datenintegrität verantwortlich sein soll), die über die eigentliche Datenmanipulation hinausgehen, als auch der Fähigkeiten und Arbeitsweise des Datenbanksystems. Diese Teile der Systembeschreibung sind von einem System-Verwalter (etwa dem Datenbankadministrator DBA) in Zusammenarbeit mit den Anwendern zu erstellen. Das Generiersystem kann hierbei Unterstützung und Anleitung bieten, etwa durch
- Vorschläge, z.B. zur Realisierung von DML-Operationen
- Angaben über die Auswirkungen des Weglassens/Hinzugenerierens von Algorithmen z.B. auf Programmlänge, Effizienz der resultierenden Version.
- Aussagen über die Auswirkungen von Änderungen von Leistungskenndaten

Das Ergebnis von Schritt D2 ist die "Liste der Algorithmen" der DBMS-Version, D3 liefert die "Liste der Systemkonstanten".

Bemerkung:
Mit dem hier gewählten Ansatz, zur Erzeugung von maßgeschneiderter DB-Software von einem vollständigen DBMS auszugehen, ist man nicht frei bei der Wahl der Konzepte und Verfahren zur Implementierung der Komponenten einer DBMS-Version. Diese Beschränkung in den Wahlmöglichkeiten ist aber durchaus sinnvoll, da Abhängigkeiten und Wechselwirkungen zwischen den Komponenten eines DBMS bestehen und zu berücksichtigen sind. /HR 80/ unterscheidet zwischen "konzeptbedingten" und "optimierungsbedingten" Abhängigkeiten:
Eine Komponente A heißt k o n z e p t b e d i n g t abhängig von einer Komponente B, wenn sich aus der Festlegung der Implementierung von B zwingende Konsequenzen für die Implementierung von A ergeben, so daß nicht mehr alle an sich denkbaren Lösungsmöglichkeiten für die Realisierung von A in Frage kommen können. Solche Wechselwirkungen zwischen den DBMS-Komponenten Sperrkomponente, Protokollkomponente, Systempufferverwaltung und Speicherverwaltung werden in /HR 80/ diskutiert.
Eine Komponente A heißt o p t i m i e r u n g s b e d i n g t abhängig von einer Komponente B, wenn die Entwurfsentscheidungen zur Implementierung von A zwar grundsätzlich unabhängig von den Konzepten und Verfahren zur Realisierung von B getroffen werden können, aus Gründen der Effizienz jedoch eine Abstimmung unverzichtbar ist. Abhängigkeiten aus

Performance-Gründen liegen vor z.B. zwischen den DBMS-Komponenten Transaktionsverwaltung, Sortierkomponente, Sperrkomponente, Zugriffspfadverwaltung, Protokollkomponente und Systempufferverwaltung /HR 80/.

Wie bereits in Bild 2.2 angedeutet, können D1, D2 und D3 in der Terminologie von APG-Systemen (Abschnitt 2.1) als die Aufgaben des "top part" eines Generiersystems für DBMS verstanden werden ("Generierphase" bei Programm-Generatoren, Bild 2.1): diese Komponente unterstützt den Generierer bei der Erstellung seiner Beschreibung der DBMS-Version, der "externen Systembeschreibung" ESD, und erzeugt dann aus dieser mit der "Liste der Algorithmen" und der "Liste der Systemkonstanten" eine interne Systembeschreibung ISD; die ISD stellt die Spezifikation für die Programmerzeugung durch die zweite Komponente des Generiersystems ("bottom part") dar.

Die Problematik der Spezifikation von DBMS-Versionen wird nicht weiter behandelt; hierzu gehören z.B. Fragen

- der Interaktion Mensch-Generiersystem: das Ziel muß es sein, vom Generierer, in der Regel der Anwender, lediglich Kenntnis der Anwenderschnittstellen (DML, DDL) des vollständigen DBMS zu erwarten (vgl. /RS 80, Sc 77/)
- des Zusammenwirkens von Generiersystem und Werkzeugen zum physischen Datenbankentwurf
- der Einbettung der DML-Anweisungen in eine Programmiersprache, die eine möglichst weitgehend automatisierte Ermittlung des Anforderungsprofils einer Anwendung bei Vorliegen der Quellprogramme erlaubt, etwa in Verbindung mit einem DML-Vorübersetzer.

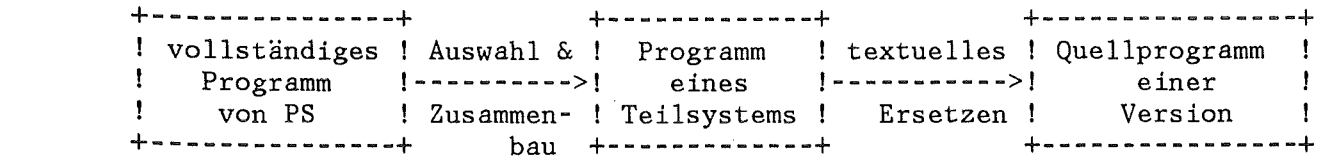
Zu untersuchen wäre auch die Frage, inwieweit z.B. das Datenmodell des DBMS die Wahl der Konzepte und Techniken zur Spezifikation beeinflußt.

Die Untersuchungen der folgenden Abschnitte befassen sich mit der eigentlichen Programmerzeugung ausgehend von der Spezifikation der Version in Form der ISD, d.h. mit der zweiten Komponente (bottom part) eines Generiersystems.

2.3. Die Erzeugung von Teilsystemen: Problemstellung

An die Erstellung der ISD als Spezifikation der Version eines DBMS schließt sich die Programmerzeugung an (Bild 2.2). Man kann hierzu vom vollständigen DBMS ausgehen.
Dieses liege in Form eines "Programmtextes", also einer Zeichenkette über einem beliebigen Alphabet, vor; es wird lediglich folgende Eigenschaft benötigt:
Mit der Angabe eines Algorithmus sind die hierfür erforderlichen ausführbaren Anweisungen des Programmtextes festgelegt, variabel sind nur noch die Werte bzw. Größen von Datenobjekten.
Für den Programmtext bedeutet dies, daß zur Änderung von Werten oder Größen von Datenobjekten (nicht ihres Typs!) ein textuelles Ersetzen im Programmtext des alten Wertes durch den neuen ausreicht, insbesondere also keine Änderung der Programmstruktur oder Neuprogrammierung erforderlich ist. Dies ist keine echte Einschränkung, die gängigen Programmiersprachen haben diese Eigenschaft!

Es gibt somit beliebig viele verschiedene Versionen eines DBMS, die eine gegebene Menge von Algorithmen realisieren, deren Quellprogrammtexte sich aber nur in den Werten und Größen von Datenobjekten (d.h. in Konstanten von Deklarationsanweisungen, Wertzuweisungen, etc.) unterscheiden. Ersetzt man diese durch symbolische Bezeichner, im weiteren P l a t z - h a l t e r genannt, so sind die aus diesen Quellprogrammen resultierenden Zeichenketten identisch (i.allg. aber keine syntaktisch korrekten Quellprogramme mehr!).
Die Programmerzeugung zur Dedizierung eines Datenbanksystems stellt sich hiermit als das folgende a l l g e m e i n e Problem dar:
Gegeben ist ein Programmsystem PS in Form eines Programmtextes mit Platzhaltern, dieser heißt das vollständige Programm von PS. Durch Auswahl (gemäß der Liste der Algorithmen der ISD) und Zusammenbau von Teilen des vollständigen Programms erhält man die P r o g r a m m e der T e i l s y s t e m e von PS. Textuelles Ersetzen der Platzhalter des Programms eines Teilsystems t durch Konstanten (gemäß der Liste der Systemkonstanten der ISD) ergibt das Q u e l l p r o g r a m m einer V e r s i o n von PS zum Teilsystem t:



Bezeichnungen:
- PS selbst kann als das Teilsystem aufgefaßt werden, das alle Algorithmen enthält, und wird daher auch "das vollständige Teilsystem" genannt.
- Die Erzeugung einer System-Version aus einem Teilsystem, also das Ersetzen der Platzhalter, wird D i m e n s i o n i e r u n g genannt.

Wie in 2.2. festgestellt, ist die Erzeugung von Versionen eines Systems ein Spezialfall der Programmerzeugung mittels Programmgeneratoren: die Bausteine zur Programmerzeugung sind Teile des vollständigen Programms,

die das Generiersystem auswählt, zum Programm eines Teilsystems zusammenbaut und durch das Ersetzen von Platzhaltern modifiziert (vgl. Bild 2.1).

Die Dimensionierung, also die Erzeugung von Versionen zu einem Teilsystem, bedarf keiner weiteren Erörterung, die folgenden Untersuchungen behandeln ausschließlich die Erzeugung von Teilsystemen eines Programmsystems.

Zur Bestimmung der Möglichkeiten zur Realisierung der Teilsystem-Erzeugung, also der Code-Auswahl, wird von dem allgemeinen Verfahren zur Erstellung ablauffähiger Objekte (Lademoduln) nach Bild 2.3 ausgegangen /BG 74/:

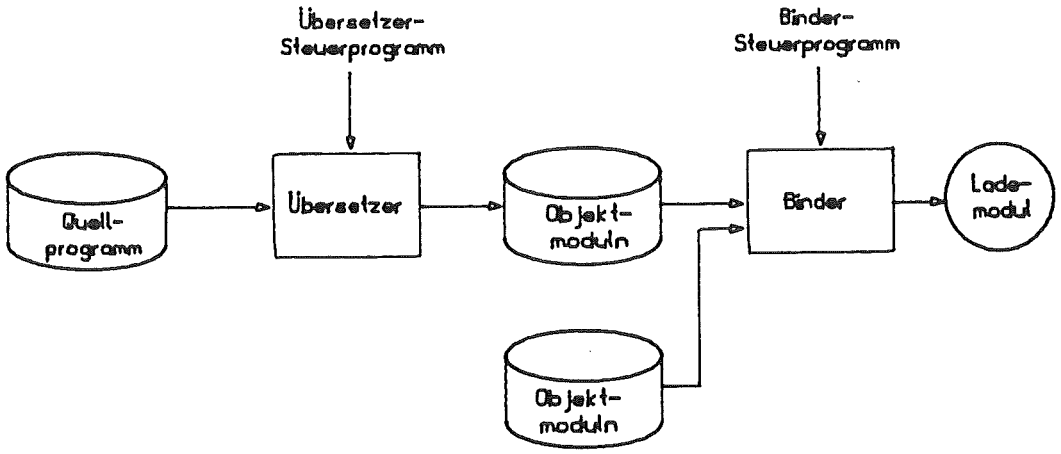


Bild 2.3: Die Erstellung ablauffähiger Objekte (Lademoduln)

Ein Quellprogramm ist gegeben als eine Menge von Haupt- und Unterprogrammen, weiterhin P r o g r a m m e i n h e i t e n genannt. Ein Übersetzer übersetzt die Programmeinheiten in je einen O b j e k t m o d u l , diese resultierenden Objektmoduln werden durch einen B i n d e r (gegebenenfalls mit weiteren Objektmoduln[2]) zu einem Lademodul zusammengeschlossen /BG 74/.
Die für die Steuerung des Übersetzers und Binders erforderlichen Anweisungen (z.B. Bedienkommandos, INCLUDE-Anweisungen des Binders) bilden das Ü b e r s e t z e r - bzw. B i n d e r - S t e u e r p r o g r a m m . Die Steuerprogramme liegen als "Programme" einer Betriebssprache /BG 74/ (engl. job control language, JCL), wie auch das Quellprogramm, in Dateien vor.

Codeauswahl zur Erzeugung von Versionen kann hier an drei Stellen vorgenommen werden:

---

[2] Objektmoduln des Anwenders oder z.B. aus Laufzeitbibliotheken höherer Programmiersprachen

- beim Binden
- beim Übersetzen
- vor dem Übersetzen

Andere Möglichkeiten, wie z.B. Modifizieren eines "Ur-Lademoduls", erfordern i.allg. Kenntnis der Konventionen eines Betriebssystems und der Struktur des vom Übersetzer und Binder erzeugten Codes, und werden hier daher (vgl. allgemeine Anforderungen von 1.3.) nicht betrachtet.

2.3.1. Codeauswahl durch den Binder

Das Verfahren:

Dem Binder wird explizit durch INCLUDE-Anweisungen des Binder-Steuerprogramms mitgeteilt, welche Objektmoduln für den Lademodul einzubinden sind. Außerdem kann implizit durch LIBRARY-Anweisungen angegeben werden, in welcher Bibliothek Objektmoduln für noch nicht aufgelöste Referenzen zu finden sind. Ein solches "LIBRARY-feature" ist in jedem Fall bei der Programmerstellung in einer höheren Programmiersprache notwendig, um so die im Quellprogramm aufgerufenen Standardprozeduren oder implizit benötigte Prozeduren (etwa zur Speicherbereinigung, Parameterübergabe, E/A-Anweisungen) anzuschließen, ohne die im allgemeinen sehr umfangreiche vollständige Laufzeitbibliothek miteinzubinden (/BG 74/, S.93).
Codeauswahl beim Binden bedeutet also Modifizieren des Binder-Steuerprogramms des vollständigen Systems: Durch Streichen einer INCLUDE-Anweisung wird das Einbinden des entsprechenden Objektmoduls und aller nur durch diesen Modul aufgerufenen Objektmoduln verhindert, die implizit aufgrund von LIBRARY-Anweisungen Bestandteil des vollständigen Systems sind.

Vorteile:

Codeauswahl und Zusammenbau bedeuten in diesem Falle lediglich Modifizieren eines bereits vorliegenden Binder-Steuerprogramms mit anschließendem Bindelauf, insbesondere entfällt das Übersetzen von Quellprogrammen. Ein solches Verfahren ist also wenig zeitaufwendig und das zugehörige Generiersystem relativ einfach: es sind lediglich die erforderlichen Anweisungen des Binder-Steuerprogramms des vollständigen Systems auszuwählen und ein Bindelauf zu initiieren.

Nachteile:

N1: Im allgemeinen reicht es nicht aus, irrelevante Objektmodule einfach nicht einzubinden, da dies zu einem n i c h t - f e h l e r - t o l e r a n t e n Lademodul führen würde: der Aufruf einer Operation, die ein so erzeugter Lademodul nicht realisiert, würde ein irreguläres Verhalten (z.B. Programmabbruch durch das Betriebssystem)

bewirken, in jedem Fall ist die gewünschte Reaktion, nämlich eine Anzeige "Operation nicht realisiert!" offensichtlich durch einfaches Weglassen von Code nicht zu erreichen.
Es ist vielmehr anstelle solcher Objektmoduln "Ersatzcode" auszuführen mit der Aufgabe, entsprechende Anzeigen zu setzen.
Um dies ausschließlich mit dem Bindelauf zu bewerkstelligen, müßte der Ersatzcode ein gleichnamiges Unterprogramm mit der gleichen Anzahl und Art von formalen Parametern wie das zu Ersetzende sein, das jedoch die Funktion der Anzeigenerzeugung hat. Solche Ersatzroutinen könnten bereits übersetzt in einer separaten "Ersatz-Bibliothek" (wegen der notwendigen Namensgleichheit!) vorliegen; damit anstelle eines Objektmoduls Ersatzcode eingebunden wird, ist in der entsprechenden INCLUDE-Anweisung der Name der Objektbibliothek durch den der Ersatzbibliothek zu ersetzen, sonst, wenn also ein Objektmodul ohne Ersatz weggelassen werden kann, genügt das Entfernen der jeweiligen INCLUDE-Anweisung des vollständigen Systems.
Die Notwendigkeit der Existenz gleichnamiger Programmeinheiten und Objektmodule stellt eine Komplizierung der Programmwartung und -pflege dar.

N2: Bei der Codeauswahl durch den Binder ist die Auswahleinheit der Objektmodul, diese Einheit ist im allgemeinen zu grob und führt zu Lademoduln mit ü b e r f l ü s s i g e m C o d e :
- Datenobjekte, wie z.B. große Felder, die mit anderen in einer separaten Programmeinheit (BLOCK DATA in FORTRAN) deklariert sind, können i.allg. nicht mit den Programmeinheiten, die sie referieren, entfernt werden.
- Unterprogrammaufrufe sind häufig von Anweisungen umgeben, die sinnvoll nur bei tatsächlicher Ausführung der betreffenden Programmeinheit sind (z.B. Bereitstellung der aktuellen Parameter in einer durch den Typ der formalen Parameter vorgeschriebenen Form, Auswertung des Returncodes). Nichtrelevante Unterprogrammaufrufe und solche hiermit assoziierten Anweisungen einer Programmeinheit können mit dieser Auswahltechnik ebenfalls nicht elimiert werden.

In beiden Fällen ist ein Modifizieren des Quellprogramms erforderlich.

N3: Aus den Überlegungen von N2 folgt auch unmittelbar, daß man von der Programmstruktur des zu betrachtenden Systems folgende Eigenschaft fordern muß, wenn durch Anwendung nur dieser Auswahltechnik Lademodule ohne überflüssigen Code erzeugbar sein sollen:
Jeder Algorithmus, der in Teilsystemen unabhängig von anderen enthalten sein kann, ist so durch Programmeinheiten realisiert, daß für jede Programmeinheit alle ihre Anweisungen, und nicht nur ein Teil davon, für diesen Algorithmus benötigt werden.
Beim Entwurf eines neuen Programmsystems kann man versuchen, diesen Aspekt zu berücksichtigen und auf diese Anforderung bei der Spezifikation der Unterprogramme einzugehen. Es erhebt sich die Frage, inwieweit dies praktikabel und sinnvoll ist und nicht zu Konflikten mit Entwurfs- und Konstruktionsprinzipien des software engineering führt. Man wird jedoch davon ausgehen müssen, daß die Unterprogrammstruktur eines bereits existierenden Programmsystems (vgl. allgemeine Anforderungen von 1.3.) dieser Anforderung nicht genügt.

N4: Dieses Verfahren bietet keine Möglichkeiten zur Dimensionierung, auch hierzu ist ein Modifizieren des Quellprogramms erforderlich.

2.3.2. Codeauswahl durch den Übersetzer

Dem Übersetzer wird durch Anweisungen des Übersetzer-Steuerprogramms mitgeteilt, aus welchen Programmeinheiten Objektmodule zu erzeugen sind. Codeauswahl durch den Übersetzer bedeutet daher analog zu 2.3.1. das Erzeugen eines Übersetzer-Steuerprogramms, das die Erzeugung nur der tatsächlich benötigten Objektmodule bewirkt: dieses Steuerprogramm ist wieder eine Teilmenge der Anweisungen des Übersetzer-Steuerprogramms für das vollständige System.
Zusätzlich ist wie in 2.3.1. das Binder-Steuerprogramm für den anschließenden Bindelauf zu erzeugen.

Ein Generiersystem zu diesem Verfahren ist komplexer als eines nach 2.3.1., da noch das Übersetzer-Steuerprogramm zu erzeugen ist, außerdem wird durch die Quellprogrammübersetzung die Generierung zeitaufwendiger. Trotzdem ergeben sich hieraus gegenüber Codeauswahl durch den Binder keine Vorteile: da jeder Programmeinheit genau ein Objektmodul entspricht, hat man hier die gleichen Schwierigkeiten und Einschränkungen wie bei Auswahl durch den Binder, es ist jetzt lediglich von Programmeinheiten anstelle von Objektmoduln als Auswahleinheit die Rede.

2.3.3. Codeauswahl vor dem Übersetzen

Wie in 2.3.1. (N2) bemerkt, erfordert die Erzeugung von Teilsystemen ohne überflüssigen Code eine feinere Auswahleinheit als die Programmeinheit bzw. den Objektmodul. Bei dem hier gewählten Ansatz muß daher die Auswahl der erforderlichen Teile vor der Übersetzung erfolgen, d.h. es sind Quellprogramme zu erzeugen:
Quellprogrammerzeugung bedeutet Auswählen von benötigten Zeichenketten des vollständigen Programms, F r a g m e n t e genannt. Ein Fragment, das hier die Auswahleinheit darstellt, kann "beliebig" klein, insbesondere auch ein Teilstring (des Textes) einer Programmeinheit sein, nach 2.3.1. eine notwendige Voraussetzung für die Erzeugung von Lademoduln ohne überflüssigen Code.
Programmerzeugung kann hier als ein allgemeines Textverarbeitungsproblem betrachtet werden, insbesondere bietet sich als weiterer Vorteil gegenüber 2.3.1. und 2.3.2. die Möglichkeit, mit der Programmerzeugung zugleich die Dimensionierung vorzunehmen.

Für die folgenden Untersuchungen zur Erzeugung von Versionen eines Programmsystems wird daher Bild 2.4 zugrunde gelegt:

- Es wird ein B - P r o g r a m m ("Basis-Programm") benötigt, das u.a. die zur Erzeugung von Teilsystemen erforderlichen Fragmente des vollständigen Systems enthält, weiter enthält es aus den in 2.3.1. (N1) genannten Gründen zu jedem Fragment als Ersatzcode eine Ersatz-Zeichenkette.
- Ein S e l e k t o r wählt die für das Teilsystem relevanten Bausteine, also Fragmente bzw. Ersatzstrings, aus und integriert sie zu einem korrekten Quellprogramm. Die Code-Auswahl und Dimensionierung

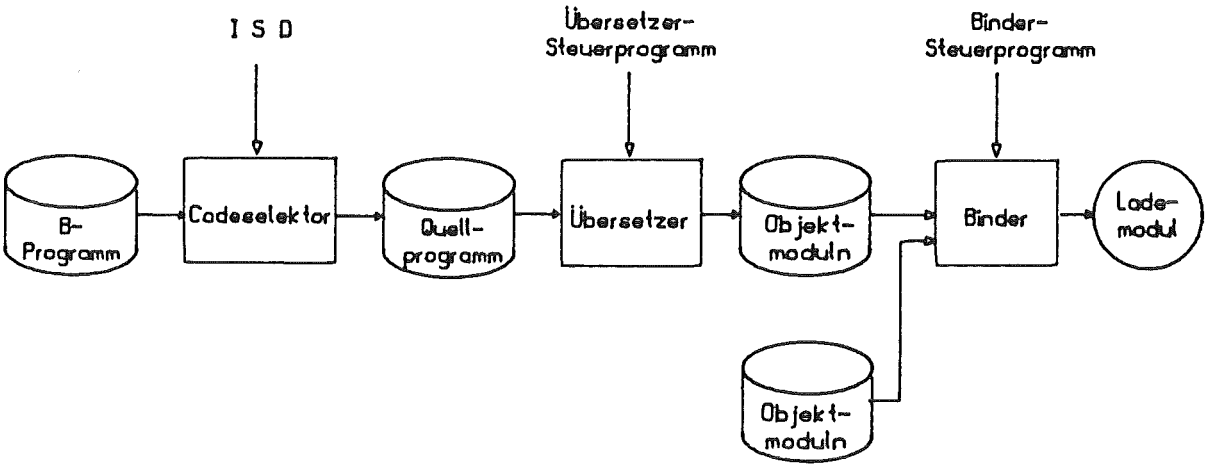


Bild 2.4: Komponenten eines Programmerzeugungssystems

wird durch die ISD gesteuert, die ISD hat also die Aufgabe eines "Selektor-Steuerprogramms".

- Übersetzung und Binden erfolgt wieder wie in Bild 2.3.

3. Die Erzeugung von Teilsystemen

Die Erzeugung von Versionen eines DBMS ist in 2.3. auf das allgemeine Problem der Erzeugung von Teilsystemen eines Programmsystems PS zurückgeführt worden, wobei das vollständige Programm von PS als ein Element der Menge Σ aller Zeichenketten über einem Alphabet A (vgl. /Gr 71, AU 72/) betrachtet wird.
Es wird zunächst das Problem der Herleitung der Fragmente für die Teilsystem-Erzeugung aus dem vollständigen Programm untersucht und ein heuristisches Verfahren zur Konstruktion eines B-Programms aus dem vollständigen Programm angegeben. Hiervon ausgehend werden "abstrakte B-Programme" eingeführt und eine formale Definition der Teilsystem-Erzeugung angegeben. Dies erlaubt u.a. die Herleitung von Transformationen zur Reduktion, d.h. Vereinfachung von B-Programmen.

3.1. Konzepte, ein Beispielsystem

Gegeben sei das vollständige Programm eines Programmsystems PS, also ein Element von Σ mit Platzhaltern. Diese Zeichenkette wird in "Fragmente" zerlegt, die als Bausteine zur Erzeugung der Programme der Teilsysteme dienen.

Bezeichnung:
Die M e n g e d e r F r a g m e n t e des Programmsystems wird mit F bezeichnet, T sei die Menge der T e i l s y s t e m e des Programmsystems. NIL bezeichne den l e e r e n String, es wird vereinbart:
NIL ε Σ.

Zu jedem Fragment f gibt es eine "Relevanz", d.h. eine Beschreibung, für welche Teilsysteme aus T es "relevant" ist, d.h. benötigt wird; hierzu wird definiert:

DEFINITION 3.1:
- B := {0, 1} heißt die Menge der R e l e v a n z w e r t e
- Die Abbildung ρ: T×F --> B gibt an, ob ein Fragment für ein Teilsystem r e l e v a n t ist:

$$\rho(t,f) := \begin{cases} 0 : f \text{ ist für } t \text{ nicht relevant} \\ 1 : f \text{ ist für } t \text{ relevant} \end{cases}$$

Für jedes f ε F ist mit ρ eine Abbildung ρ_f: T --> B definiert:

$$\rho_f(t) := \rho(t,f)$$

ρ_f heißt die R e l e v a n z (-funktion) von f.

Man sagt, zwei Fragmente f und g haben die g l e i c h e R e l e v a n z oder sie "stimmen in der Relevanz überein", wenn $\rho_f \equiv \rho_g$ gilt (d.h. $\rho(t,f)=\rho(t,g)$ für jedes t ε T, vgl. Anhang 1)

• Der Operator OR: B×B --> B :

$$x \text{ OR } y := \begin{cases} 0 : x=0 \text{ und } y=0 \\ 1 : \text{sonst} \end{cases}$$

Für Relevanzfunktionen ist der Operator OR wie folgt definiert:

$$\rho_f \text{ OR } \rho_g \ (t) := \rho_f(t) \text{ OR } \rho_g(t)$$

Ein R e l e v a n z a u s d r u c k ist eine Relevanz oder die OR-Verknüpfung von Relevanzen.

Notation:
Die OR-Verknüpfung einer Menge von Elementen wird durch Angabe von Indizes bzw. der Menge wie folgt geschrieben (OR ist kommutativ!):

$$OR_{i=m}^{n} x_i \quad \text{bzw.} \quad OR_{x \varepsilon S} x$$

Sonderfälle: $OR_{i=m}^{m} x_i = x_m$ $\qquad$ $OR_{x \varepsilon S} x=s$ bei $S=\{s\}$

Weiter ist nach 2.3. zu einem Fragment f ein Ersatz anzugeben, d.h. ein Element von Σ:
DEFINITION 3.2:
Die Abbildung σ: F --> Σ ordnet jedem Fragment f seinen E r s a t z (-string) σ(f) zu; σ(f) kann insbesondere auch der leere string NIL sein.

Zur Verdeutlichung wird im folgenden als Beispiel das Programm des Programmsystems DBMS von Bild 3.1 benutzt. Es skizziert die Realisierung eines stark vereinfachten Zugriffssystems (vgl. Abschnitt 1.3, Bild 1.5) und spiegelt die Struktur des FADABS-Nucleus wieder /Po 78/. Die Programmeinheiten von Bild 3.1 sind nur soweit wie für die Erläuterungen zu "Teilsystem-Erzeugung" erforderlich ausgearbeitet, programmtechnische Details (z.B. Deklarationen von Variablen, formalen Parametern, etc.) sind nur aufgeführt, wenn dies für Demonstrationszwecke interessant ist; aus Gründen der Übersichtlichkeit werden zum Teil informelle Pseudo-Anweisungen (in Kleinschreibung) benutzt, die durchaus umfangreiche Programmabläufe, etwa mit Aufrufen weiterer Programmeinheiten, beschreiben können.
Die "package"-Programmeinheit dient der Vereinbarung des array INDEX_TABLE als globales Datenobjekt (vgl. ADA /ADA/; BLOCK DATA von FORTRAN /FTN/). Die USE-Anweisung (vgl. /ADA/) macht Programmeinheiten globale Datenobjekte zugänglich.
Das Programmsystem wird von einem Anwendersystem als Unterprogramm DBMS aufgerufen (s. Bild 1.1) und stellt sechs Operationen bereit:

OPEN : zum Sperren von Relationen; zu Beginn einer Transaktion muß der Anwender für die von ihm benötigten Relationen Sperren anfordern
CLOSE : zum Freigeben von Sperren; spätestens zum Abschluß der Transaktion muß der Anwender die von ihm (mit OPEN!) belegten Relationen wieder freigeben
FIND : zum Auffinden einer Menge von Tupeln einer Relation auf Grund von Attributwerten; übergeben wird der Bezeichner eines Objektes vom Typ QSS /Po 78/ ("Scan" bei /Hä 78/), auf das sich der Anwender zum eigentlichen Retrieval (mit GET) bezieht
GET : zur Bereitstellung eines Tupels einer QSS
INSERT: zum Einfügen eines Tupels in eine Relation
DELETE: zum Löschen eines Tupels aus einer Relation

```
PROCEDURE DBMS

   IF (OP<1 OR OP>6)
      THEN return 'operation unknown'
   CASE OP OF
     1: OPEN
     2: CLOSE
     3: FIND
     4: GET
     5: INSERT
     6: DELETE
   END
END
```

```
PACKAGE INDEXES
  INDEX_TABLE: ARRAY OF INTEGER
END
```

```
PROCEDURE OPEN
   OPEN_RF
   OPEN_IF
END
```

```
PROCEDURE CLOSE
   CLOSE_RF
   CLOSE_IF
END
```

```
PROCEDURE OPEN_RF
  . . . . . .

  GET
  . . . . . .
END
```

```
PROCEDURE OPEN_IF
  USE INDEXES
  . . . . . .
  GET
  . . . . . .
END
```

```
PROCEDURE CLOSE_RF
  . . . . . .

END
```

```
PROCEDURE CLOSE_IF
  USE INDEXES
  . . . . . .
END
```

```
PROCEDURE FIND
   USE INDEXES
   evaluate INDEX_TABLE
   STRTGY
   return qss
END
```

```
PROCEDURE STRTGY
   determine access-strategy and
    set ACCESS_TYPE
   CASE ACCESS_TYPE OF
     1: build seq.search qss
     2: BEGIN
        CASE FILE_TYPE OF
          1: calculate tid
          2: . . . . . .
             RETRIEVE_TID_LIST
             . . . . . .
        END
        build direct-access qss
        END
   END
END
```

```
PROCEDURE GET
   NEXT_TUPLE:
   CASE ACCESS_TYPE OF
     1: NEXT_SEQ
        . . . . . .

     2: NEXT_TID
        . . . . . .

   END
   IF (qualifikation is not satisfied)
      THEN GO TO NEXT_TUPLE
END
```

```
PROCEDURE INSERT
  CASE FILE_TYPE OF
    1: INSERT_1
    2: INSERT_2
  END
  INSERT_TID
END
```

```
PROCEDURE DELETE
  CASE FILE_TYPE OF
    1: DELETE_1
    2: DELETE_2
  END
  DELETE_TID
END
```

```
PROCEDURE NEXT_SEQ
   CASE FILE_TYPE OF
     1: next_1
     2: next_2
   END
END
```

```
PROCEDURE INSERT_1
  . . . . . .
END
```

```
PROCEDURE DELETE_1
  . . . . . .
END
```

```
PROCEDURE INSERT_2
  . . . . . .
END
```

```
PROCEDURE DELETE_2
  . . . . . .
END
```

```
PROCEDURE NEXT_TID
   return next tid of tid-list
END
```

```
PROCEDURE INSERT_TID
  USE INDEXES
  . . . . . .
END
```

```
PROCEDURE DELETE_TID
  USE INDEXES
  . . . . . .
END
```

```
PROCEDURE RETRIEVE_TID_LIST
  . . . . . .
END
```

Bild 3.1: Das Beispiel-Programmsystem DBMS

DBMS bietet zur Implementierung
- von Relationen zwei Speicherungsstrukturen (Variablen FILE_TYPE)
- von Zugriffspfaden neben "sequentielles Suchen nach physischer Abspeicherung" ein Hash-Verfahren (gestreute Speicherungsstruktur) und eine B-Baum-Struktur, Index oder invertierte Datei genannt.

Entsprechend gibt es für das Datenretrieval zwei Zugriffsverfahren (Variablen ACCESS_TYPE): "sequentielles Suchen" und "direkter Zugriff" (mittels TID-Listen nach dem hash-Verfahren bzw. invertierter Dateien)

Die folgende Übersicht erläutert die Implementierung der Operationen (in Klammern die entsprechenden Anweisungen bzw. Teile des Programms) und gibt dabei die Algorithmen an (A1 bis A17 der rechten Spalte):

OPEN	- Belegen der Relation, Kontrollinformation bereitstellen (OPEN_RF)	A1
	- Überprüfen, ob invertierte Dateien existieren, diese gegebenenfalls belegen und INDEX_TABLE aktualisieren (OPEN_IF)	A2
CLOSE	- Freigeben der Relation (CLOSE_RF)	A3
	- Überprüfen, ob invertierte Dateien existieren, diese gegebenenfalls freigeben und INDEX_TABLE aktualisieren (CLOSE_IF)	A4
FIND	- Ermittlung (in INDEX_TABLE) der verfügbaren invertierten Dateien (evaluate INDEX_TABLE)	
	- Ermittlung des Zugriffsverfahrens und Erzeugung eines Objektes QSS (STRTGY) für	
	sequentielles Suchen (build seq.search qss) oder	A5
	direkten Zugriff mittels:	
	hash-Verfahren (calculate tid)	A6
	TID-Liste aus invert. Datei (RETRIEVE_TID_LIST)	A7
GET	- Aufsuchen des nächsten Tupels mittels:	
	sequentielles Suchen (NEXT_SEQ) gemäß Speicherungs-	
	struktur 1 (next_1)	A8
	oder 2 (next_2)	A9
	direkten Zugriff anhand einer TID-Liste (NEXT_TID)	A10
	- Überprüfung, ob eine Qualifikation erfüllt ist	A11
INSERT	- Einfügen eines Records gemäß Speicherungs-	
	struktur 1 (INSERT_1)	A12
	oder 2 (INSERT_2)	A13
	- Ermittlung (in INDEX_TABLE), ob invertierte Dateien	
	existieren, gegebenenfalls diese aktualisieren	A14
	(INSERT_TID)	
DELETE	- Löschen eines Records gemäß Speicherungs-	
	struktur 1 (DELETE_1)	A15
	oder 2 (DELETE_2)	A16
	- Ermittlung (in INDEX_TABLE), ob invertierte Dateien	
	existieren, gegebenenfalls diese aktualisieren	A17
	(DELETE_TID)	

Das Teilsystem t_ins von DBMS:
Gegeben sei ein Anwendungssystem mit einer Betriebsphase, in der ausschließlich Daten abzuspeichern sind (vgl. 1.1., "Datenerfassung"). Eignet sich z.B. Speicherungsstruktur 1 zum schnellen Abspeichern von Datensätzen, so wird man zumindest für diese Betriebsphase die Relationen, die diese Daten aufnehmen, mittels Speicherungsstruktur 1 realisieren und weiterhin auf invertierte Dateien zu diesen Relationen verzichten (es sind keine Retrieval-Operationen zu unterstützen). Für die Operation INSERT selbst ist in dieser Phase also lediglich Algorithmus A12 erforderlich. Zum Belegen und Freigeben von Relationen (Operationen OPEN, CLOSE) genügen die Algorithmen A1 bzw. A3; zum Zugriff auf die Kontrollinformation (Aufruf von GET in OPEN_RF) in Systemkatalogen (vgl. Abschnitt 1.2) seien die Algorithmen A8 und A11 erforderlich.
Das Teilsystem mit diesen Algorithmen wird im folgenden mit t_ins bezeichnet.

3.2. Die Fragmente eines Programmsystems, das B-Programm

3.2.1. Die Fragmente: Anforderungen, Eigenschaften

Fragmente stellen die Teile des vollständigen Programms dar, die als Bausteine für die Erzeugung von Teilsystemen benötigt werden bzw. überflüssig sind, d.h. weggelassen werden. Dies sind nach 2.3. sowohl ganze Programmeinheiten, aber auch nur Teile von Programmeinheiten. Anhand des Beispielsystems wird dies in 3.2.1.1. zunächst verdeutlicht und anschließend weitere Anforderungen an ein Konstrukt "Fragment" hergeleitet.

3.2.1.1. Fragmente: Programmeinheiten, Teile von Programmeinheiten

Nach 2.3.1. (N1) muß es zur Erzeugung von Teilsystemen möglich sein, zumindest Programmeinheiten des vollständigen Programms auszuwählen.
Beispiel:
Für Teilsysteme von DBMS, die die Operation INSERT nicht implementieren, sind die Programmeinheiten INSERT, INSERT_1, INSERT_2, INSERT_TID nicht relevant; es sind Fragmente erforderlich, die diese Programmeinheiten des vollständigen Programms umfassen und deren Relevanz für solche Teilsysteme den Relevanzwert 0 ergeben.

Es muß möglich sein, auch Teile einer Programmeinheit als für ein Teilsystem "relevant" bzw. "überflüssig" zu kennzeichnen (2.3.1., N2 und N3) Man betrachte hierzu die detailliertere Darstellung der Programmeinheit INSERT in Bild 3.2:
Teilsysteme (wie t_ins) für Anwendungen, die die Operation INSERT nur für eine Teilmenge der insgesamt verfügbaren Speicherungsstrukturen erfordern und/oder beim Einfügen keine invertierten Dateien zu warten haben, benötigen jeweils nur eine Teilmenge der in Bild 3.2 mit 1, 2 und 3 gekennzeichneten Anweisungen. Damit solche Teilsysteme nur relevante und keine überflüssigen Anweisungen enthalten, sind diese Teile der Programmeinheit des vollständigen Programms als je ein Fragment zu definieren[1].

Die Einführung von Fragmenten kann die Definition weiterer Fragmente zur Folge haben: Bild 3.2 entnimmt man unmittelbar, daß die Anweisungen von A2 höchstens dann zur Ausführung gelangen können, wenn die von Fragment 3 Bestandteil von INSERT sind; umgekehrt benötigt man mit Fragment 3 auch die Anweisungen von A2. Damit man keine Programme mit überflüssigem Code erhält, ist mit Fragment 3 auch ein Fragment A2 erforderlich, wobei wie

---

[1] Zur Darstellung der Fragmente werden links vom Programmtext eine oder mehrere Spalten für "Fragment-Bezeichner" (meist Ziffernfolgen) eingeführt: die Zeilen des Programmtextes mit gleichem Fragment-Bezeichner bilden das Fragment mit diesem Bezeichner. So stellen z.B. die Zeilen von Bild 3.2 mit dem Fragment-Bezeichner "1" das Fragment 1 dar.

```
Fragment-
Bezeichner

   | |          PROCEDURE INSERT
   | |
   | |D1|       S_RET: INTEGER
   | |D2|       Z_RET: INTEGER
   | |          FTYPE: INTEGER

    :             :   :   :   :   :   :   :   :

   | |          ermittlung der speicherungsstruktur und
   | |          zuweisung an FILE_TYPE
   | |          CASE FILE_TYPE OF
   | |            1: BEGIN
   | | 1|             /* Einfuegen: Speicherungsstruktur 1 */
   | | 1|             INSERT_1( . . . . . ,S_RET)
   | | 1|             IF (S_RET # 0) THEN GO TO 930
   | |                END
   | |              2: BEGIN
   | | 2|             /* Einfuegen: Speicherungsstruktur 2 */
   | | 2|             INSERT_2( . . . . . ,S_RET)
   | | 2|             IF (S_RET # 0) THEN GO TO 930
   | |                END
   | |          END
   | | 3|       /* aktualisieren der invert. Dateien */
   | | 3|       INSERT_TID( . . . . . ,Z_RET)
   | | 3|       IF (Z_RET # 0) THEN GO TO 990
   | |
   | |      900: /* REGULAR EXIT */
   | |           GO TO 999
   | |
   | |A1| 930: s_error-action
   | |A1|      GO TO 999
   | |A2| 990: z_error-action
   | |A2|      GO TO 999
   | |
   | |      999: RETURN
   | |           END
```

Bild 3.2: Fragmente als Folgen von Anweisungen

gezeigt, beide in der Relevanz übereinstimmen: $\rho_3 \equiv \rho_{A2}$ (s. Definition 3.1).
Fragment A1 ist stets bereitzustellen, wenn ein Fragment der CASE-Anweisung erforderlich ist, d.h. die Relevanz von A1 ist gleich der OR-Verknüpfung der Relevanzen der Fragmente 1 und 2: $\rho_{A1} \equiv \rho_1$ OR ρ_2.

Der Ersatz für die Fragmente 1 und 2, also $\sigma(1)$ und $\sigma(2)$, besteht in Anweisungen zur Übergabe eines Returncodes, etwa "storage structure not accessible" (vgl. Bild 3.5).
$\sigma(A1)$, $\sigma(A2)$ sind NIL. $\sigma(3)$ kann ebenfalls NIL sein, falls garantiert werden kann, daß die Teilsysteme von DBMS ohne Algorithmus A14 (INSERT_TID) nur wie vorgesehen eingesetzt werden; andernfalls sind als Ersatz Anweisungen vorzusehen, die Fehlertoleranz gewährleisten (es ist zu verhindern, daß durch das Einfügen eines Tupels die interne Konsistenz der DB verloren geht!).

Die Einführung dieser fünf Fragmente mit ausführbaren Anweisungen hat die Definition von zwei Fragmenten mit Deklarationen von Datenobjekten zur Folge: die Variable Z_RET wird nur von Anweisungen des Fragmentes 3 referiert, d.h. nur mit diesem benötigt; entsprechendes gilt für S_RET.

Man gelangt so zu den Fragmenten D2 bzw. D1 von Bild 3.2 mit

$$\rho_{D2} \equiv \rho_3 \qquad \rho_{D1} \equiv \rho_1 \text{ OR } \rho_2 .$$

3.2.1.2. Geschachtelte Fragmente

Die Fragmente 1, 2 und 3 von Bild 3.2 sind notwendig zur Erzeugung von Teilsystemen, die nur eine Teilmenge der insgesamt für die Operation INSERT verfügbaren Algorithmen bereitstellen, für Teilsysteme, die die Operation INSERT nicht realisieren, ist die Programmeinheit INSERT als ganzes überflüssig.
Da nicht alle Anweisungen von INSERT in den Fragmenten von Bild 3.2 enthalten sind, könnte man zur Erzeugung von Programmen ohne Programmeinheit INSERT versuchen, weitere Fragmente zu denen von Bild 3.2 so definieren, daß INSERT vollständig durch Fragmente "überdeckt" wird: eine Programmeinheit INSERT wird dann nicht erzeugt, wenn die Relevanzen dieser Fragmente den Wert 0 ergeben. Bei diesem Ansatz ist man aber i.allg. gezwungen, für ein Fragment m e h r e r e Ersatz-strings vorzusehen: für das Fragment 1 z.B. wären dies

- die Anweisungen zum Setzen eines Returncodes von oben für Teilsysteme, die zwar die Operation INSERT realisieren, aber z.B. nur mit Algorithmus A12 (vgl. Teilsystem t_ins)
- NIL für Teilsysteme ohne die Operation INSERT.

Der Ersatz eines Fragments ist dann aber nicht mehr so einfach wie in Definition 3.2 durch eine Abbildung F --> Σ zu beschreiben.
Diese Schwierigkeit entfällt, wenn ein Fragment neben Teilen des Programmtextes selbst wieder Fragmente, "Unterfragmente", enthalten kann: es ist dann z.B. möglich, ein Fragment f zu definieren, in Bild 3.2 angedeutet durch die leere linke Spalte (vgl. Fragment 8 von Bild 3.4), das die Fragmente von oben u n d die in diesen nicht enthaltenen Teile der Programmeinheit umfaßt. Die Erzeugung einer Programmeinheit INSERT unterbleibt für Teilsysteme, für die die Relevanz von f den Relevanzwert 0 liefert; sonst werden für die Programmerzeugung alle Teile von f, die kein Fragment darstellen, und alle Unterfragmente von f mit dem Relevanzwert 1 benötigt.

DEFINITION 3.3:

- Ein F r a g m e n t f ist eine geordnete Menge von Zeichenketten $q \in \Sigma$ und Fragmenten f_i; die Fragmente f_i heißen die U n t e r f r a g m e n t e von f, f das O b e r f r a g m e n t für jedes f_i.
  Ein Fragment, das nur aus Elementen von Σ besteht, wird ein e i n f a c h e s Fragment genannt.
- Ein Fragment g heißt in f g e s c h a c h t e l t , wenn g ein Unterfragment von f ist oder in einem Unterfragment von f geschachtelt ist.
  Die (Schachtelungs-) T i e f e DEPTH(g) eines Fragments g ist gegeben durch:

$$\text{DEPTH}(g) := \begin{cases} 1 & : \text{es gibt kein Oberfragment zu } g \\ \text{DEPTH}(f)+1 & : f \text{ ist das Oberfragment von } g \end{cases}$$

Aus dem obigen ergibt sich für die Relevanz geschachtelter Fragmente folgende Eigenschaft:
Ist $t \in T$ und g in f geschachtelt, dann gilt: $\rho(t,f)=0 \Longrightarrow \rho(t,g)=0$;

d.h. für ein Teilsystem t sind mit f stets auch die in f geschachtelten Fragmente für t nicht relevant, zur Programmerzeugung wird in diesem Fall f (mit den in f geschachtelten Fragmenten!) durch σ(f) ersetzt.

Vereinbarung:
Zur eindeutigen Bezeichnung werden die Unterfragmente eines Fragments f im folgenden (in der Regel mit 1 beginnend) durchnummeriert. Der Bezeichner des Unterfragmentes i eines Fragments mit dem Bezeichner f hat den Bezeichner "f.i" .

3.2.1.3. Fragmente als Teile von Anweisungen

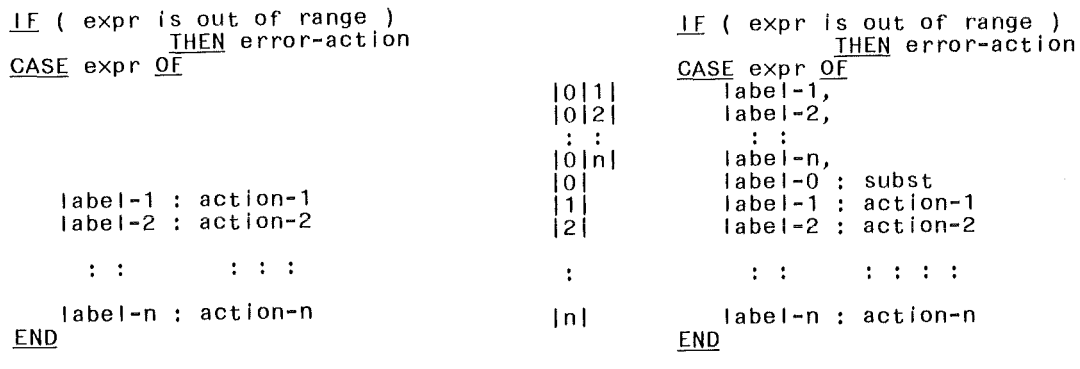
Bei der Definition der Fragmente eines Programmsystems spielt die Zerlegung von CASE-Konstrukten eine besondere Rolle (ein Verteiler nach Abschnitt 1.2, Bild 1.5, i s t ein CASE-Konstrukt), zwei Möglichkeiten hierzu werden aufgezeigt.
Den allgemeinen Aufbau eines CASE-Konstruktes mit n Alternativen action-i zeigt Bild 3.3a; die IF-Abfrage stellt sicher, daß die Auswertung des CASE-Ausdrucks expr einen zulässigen Wert liefert (vgl. Semantik der CASE-Anweisung von PASCAL /JW 74/).

Zur Zerlegung kann stets so vorgegangen werden, daß die n Alternativen, also die n Teilstrings

action-i 1≤i≤n

je ein Fragment bilden. Man erhält so in Bild 3.2 die Fragmente 1 und 2.

Eine andere Zerlegung des CASE-Konstruktes ist unter der Voraussetzung möglich, daß für alle Alternativen der gleiche Ersatz subst vorgesehen ist, eine in der Praxis häufige Situation. In diesem Fall kann man wie folgt vorgehen (Bild 3.3b):



a) b)

Bild 3.3: Fragmentierung von CASE-Konstrukten

1) Für 1≤i≤n bildet der string ' label-i : action-i ' der CASE-Anweisung das Fragment i mit σ(i)=NIL.

Wird also die Alternative i nicht benötigt, so werden die Anweisungen action-i m i t der Marke label-i ersatzlos entfernt, das CASE-Konstrukt also "verkürzt".

2) Damit nun aber die Überprüfung auf Zulässigkeit des Wertes von expr nicht vom Teilsystem abhängig wird, wird ein Fragment 0 mit n Unterfragmenten 0.i, $1 \leq i \leq n$, gemäß Bild 3.3b eingeführt. Die Relevanz eines Unterfragments 0.i ist die "Negation" der Relevanz des Fragments i:

$$\rho(t,0.i) = \begin{cases} 0 : \rho(t,i)=1 \\ 1 : \rho(t,i)=0 \end{cases} \qquad 1 \leq i \leq n$$

Fragment 0 ist genau dann relevant, wenn eines seiner Unterfragmente relevant ist: $\rho_0 \equiv OR_{i=1}^{n} \rho_{0.i}$

Wie man sich leicht überzeugt, bewirkt dies, daß für jedes Teilsystem stets alle Marken Bestandteil des CASE-Konstrukts sind.
Ersatz für das Fragment 0 und seine Unterfragmente ist jeweils NIL.

Bemerkungen, Erläuterungen:

- 'label-0' dient lediglich zur Vereinfachung der Fragmentstruktur: ohne diese Hilfskonstruktion müßten die Kommas der Labelliste des Fragments 0 eigene Fragmente bilden, deren Relevanzen sich nicht mehr einfach als OR-Verknüpfung von Relevanzen $\rho_{0.i}$ darstellen lassen!
- Die Zeichenketten des Fragmentes 0 und seiner Unterfragmente sind n i c h t Teile des vollständigen Programms!
- Man erhält hier Fragmente, die keine syntaktisch vollständigen Anweisungen enthalten!

Diese Art der Zerlegung ist dem allgemeinen Ansatz vorzuziehen, wenn das CASE-Konstrukt eine große Anzahl von Alternativen umfaßt: der Programmtext wird kürzer, die Codestücke subst werden nicht dupliziert; das CASE-Konstrukt wird außerdem "lesbarer", da man unmittelbar erkennen kann, welche der Alternativen nicht implementiert sind.
In Abschnitt 3.2.2, Bild 3.4, wird diese Technik zur Definition der Fragmente der Programmeinheit DBMS angewandt; zur Illustrierung sei weiter auf das Programm des Teilsystems t_ins von Bild 3.6 verwiesen.

Fragmente, die keine syntaktisch vollständigen Anweisungen enthalten, können sich auch bei der Zerlegung von Deklarationsanweisungen ergeben. Es ist i.allg. zulässig, mit einer Anweisung, etwa von der Form

var1,var2,var3,... : type

durch Angabe einer Liste von Bezeichnern eine Menge von Datenobjekten (var1,var2,...) zu deklarieren. Haben diese unterschiedliche Relevanz, so kann man Fragmente einführen, die nur einen Teil der Bezeichnerliste umfassen. Enthielte die Programmeinheit INSERT anstelle der drei Deklarationen von Bild 3.2 die eine Anweisung

S_RET,Z_RET,F_TYPE: INTEGER

so wäre das Fragment D1 der string 'S_RET,' und D2 der string 'Z_RET,'.

3.2.2. Zur Zerlegung eines Programmsystems in Fragmente

Mit den Ergebnissen von 3.2.1. wird nun das Problem der Zerlegung des vollständigen Programms untersucht und ein heuristisches Verfahren hierzu angegeben.

Da ein Teilsystem nur relevante Teile des vollständigen Systems enthalten soll (keine nichterreichbaren ausführbaren Anweisungen, keine unbenutzten Datenobjekte), liegt es nahe, zur Ermittlung der Fragmente von einer statischen Analyse /GG 80, VG 80/, insbesondere dem Kontrollfluß, des vollständigen Programms auszugehen. Man beachte, daß diese Information für die Herleitung der Fragmente der Programmeinheit INSERT in 3.2.1. bestimmend war.

Zur Vermeidung von nicht erreichbaren Anweisungen, ist bei der Definition von Fragmenten darauf zu achten, daß der Programmteil, den ein Fragment umfaßt, in folgendem Sinne m a x i m a l gewählt wird:
s sei eine ausführbare Anweisung, und p‖s oder s‖p ein Teilstring des Programmtextes. Ein Fragment f, das p umfaßt, muß auch s enthalten, wenn s nicht in einem Fragment mit einer Tiefe größer/gleich DEPTH(f) enthalten ist und die Ausführung von s notwendigerweise die Ausführung einer Anweisung von p impliziert.

Bei der Ermittlung solcher maximaler Anweisungsfolgen p können allgemeine Testsysteme, wie z.B. RXVP /GG 80/, oder spezielle Analysesysteme (z.B. BRNANL /Fo 74/) Unterstützung bieten.

Das folgende heuristische Verfahren zur Herleitung der Fragmente eines Programmsystems ist eine Verallgemeinerung der Vorgehensweise von 3.2.1.; es werden die folgenden Begriffe benötigt:
- Ein Fragment wird a u s g e f ü h r t , wenn mindestens eine Anweisung des Fragments ausgeführt wird.
- Ein Algorithmus ist durch Anweisungen implementiert, die in einer oder mehreren Programmeinheiten enthalten sind. Eine Anweisung s heißt i n i t i i e r e n d für einen Algorithmus A, wenn mit der Ausführung von s auch A ausgeführt wird (z.B. ein Unterprogrammaufruf s).

Zur Illustrierung dient wieder das Programmsystem DBMS von 3.1. Es wird hierzu angenommen, daß der Programmtext als zeilenorientierte Zeichenkette nach Bild 3.4 vorliegt.

Zur Herleitung einer F r a g m e n t i e r u n g geht man wie folgt vor:

SCHRITT 1:
Jede Programmeinheit des Programmsystems bildet je ein Fragment.

Beispiel:
Für das Beispielsystem ergibt dies die Fragmente 1 bis 22 von Bild 3.4.

Bemerkung:
- Mit Schritt 1 werden die Voraussetzungen zur Auswahl einer beliebigen

```
1|     PROCEDURE DBMS                                    1
1|        . . . . . .
1|        IF (OP<1 OR OP>6)
1|           THEN return 'operation unknown'
1|        CASE OP OF
1|0|1|        1,                                        2
1|0|2|        2,                                        3
1|0|3|        3,                                        4
1|0|4|        4,                                        5
1|0|5|        5,                                        6
1|0|6|        6,                                        7
1|0|          0: return 'operation not                  8
1|0|                  implemented'
1|1|          1: OPEN                                   9
1|2|          2: CLOSE                                  10
1|3|          3: FIND                                   11
1|4|          4: GET                                    12
1|5|          5: INSERT                                 13
1|6|          6: DELETE                                 14
1|        END                                           15
1|     END
2|     PROCEDURE OPEN                                   16
2|        OPEN_RF
2|1|     OPEN_IF                                        17
2|     END                                              18
3|     PROCEDURE FIND                                   19
3|        USE INDEXES
3|        evaluate INDEX_TABLE
3|        STRTGY
3|        return qss
3|     END
4|     PROCEDURE STRTGY                                 20
4|        determine access-strategy and
4|         set ACCESS_TYPE
4|        CASE ACCESS_TYPE OF
4|          1:
4|1|           build seq.search qss                     21
4|          2: BEGIN                                    22
4|2|           CASE FILE_TYPE OF                        23
4|2|1|          1: calculate tid                       24
4|2|2|          2: . . . . . .                         25
4|2|2|             RETRIEVE_TID_LIST
4|2|2|             . . . . . .
4|2|           END                                     26
4|2|           build direct-access qss
4|             END                                     27
4|        END
4|     END
5|     PROCEDURE GET                                    28
5|        NEXT_TUPLE:
5|        CASE ACCESS_TYPE OF
5|          1: . . . . . .
5|1|           NEXT_SEQ                                29
5|1|           . . . . . .
5|1|
5|          2: . . . . . .                             30
5|2|           NEXT_TID                                 31
5|2|           . . . . . .
5|2|
5|        END                                           32
5|3|     IF (qualifikation is not satisf                 33
5|3|        THEN GO TO NEXT_TUPLE
5|     END                                              34
6|     PROCEDURE NEXT_SEQ                               35
6|        CASE FILE_TYPE OF
6|          1:
6|1|           next_1                                   36
6|          2:                                          37
6|2|           next_2                                   38
6|        END                                           39
6|     END
7|     PROCEDURE NEXT_TID                               40
7|       return next tid of tid-list
7|     END
```

```
8|     PROCEDURE INSERT                                 41
8|        CASE FILE_TYPE OF
8|          1:
8|1|           INSERT_1                                 42
8|          2:                                          43
8|2|           INSERT_2                                 44
8|        END                                           45
8|3|     INSERT_TID                                     46
8|     END                                              47
9|     PROCEDURE CLOSE                                  48
9|        CLOSE_RF
9|1|     CLOSE_IF                                       49
9|     END                                              50
10|    PROCEDURE DELETE                                 51
10|       CASE FILE_TYPE OF
10|         1:
10|1|          DELETE_1                                 52
10|         2:                                          53
10|2|          DELETE_2                                 54
10|       END                                           55
10|3|    DELETE_TID                                     56
10|    END                                              57
11|    PROCEDURE OPEN_RF                               58
11|       . . . . . .
11|      GET
11|       . . . . . .
11|    END
12|    PROCEDURE CLOSE_RF                              59
12|       . . . . . .
12|    END
13|    PROCEDURE OPEN_IF                               60
13|      USE INDEXES
13|       . . . . . .
13|      GET
13|       . . . . . .
13|    END
14|    PROCEDURE CLOSE_IF                              61
14|      USE INDEXES
14|       . . . . . .
14|    END
15|    PROCEDURE INSERT_1                              62
15|       . . . . . .
15|    END
16|    PROCEDURE INSERT_2                              63
16|       . . . . . .
16|    END
17|    PROCEDURE DELETE_1                              64
17|       . . . . . .
17|    END
18|    PROCEDURE DELETE_2                              65
18|       . . . . . .
18|    END
19|    PROCEDURE INSERT_TID                            66
19|      USE INDEXES
19|       . . . . . .
19|    END
20|    PROCEDURE DELETE_TID                            67
20|      USE INDEXES
20|       . . . . . .
20|    END
21|    PROCEDURE RETRIEVE_TI                           68
21|       . . . . . .
21|    END
22|    PACKAGE INDEXES                                 69
22|1|     INDEX_TABLE: ARRAY                           70
22|    END                                             71
```

Bild 3.4: Fragmentierung des Beispielsystems

Teilmenge der Programmeinheiten des vollständigen Programms geschaffen (vgl. Abschnitt 3.2.1).

- Diese Fragmente erhält man mit der syntaktischen Analyse des Programms, Schritt 1 ist somit vollständig a u t o m a t i s i e r b a r .
- Lokale Programmeinheiten, d.h. Programmeinheiten, die in anderen Programmeinheiten definiert sind, führen zu geschachtelten Fragmenten.

SCHRITT 2:
Für jedes Fragment wird überprüft, ob es Anweisungen enthält, die initiierend für Algorithmen sind, aber nicht notwendigerweise mit jeder Ausführung des Fragments auszuführen sind. Ist dies der Fall, so sind Unterfragmente zu definieren, die diese initiierenden Anweisungen enthalten.

Man beachte die Rekursivität: für ein so eingeführtes Fragment kann (wegen der Maximalitätseigenschaft) die Voraussetzung zur Definition von Unterfragmenten erfüllt sein, Schritt 2 ist dann auch auf dieses Fragment anzuwenden, etc.
Für alle Unterfragmente, die man nach Schritt 2 zu einem Fragment f erhält, gilt, daß sie nicht notwendigerweise mit f auszuführen sind. Man kann jedoch Mengen X von Unterfragmenten erhalten, mit der Eigenschaft, daß mit jeder Ausführung von f genau ein Fragment von X ausgeführt wird. Solche Fragmente werden X - F r a g m e n t e von f genannt, die anderen, die also ohne jede Einschränkung "optional" sind, heißen die O - F r a g m e n t e von f.

Beispiele:

- Fragment 4 enthält Anweisungen, die initiierend für die Algorithmen A5, A6 und A7 sind, die aber nicht notwendigerweise mit jedem Aufruf von STRTGY ausgeführt werden. Nach Schritt 2 sind daher zu Fragment 4 zunächst die Unterfragmente 4.1 und 4.2 einzuführen. Damit Fragment 4.2 maximal ist, muß es so definiert sein, daß es (neben den Anweisungen von 'build direct-access qss') zwei Anweisungen enthält, die initiierend für je einen optionalen Algorithmus (A6 bzw. A7) sind: offensichtlich wird bei jeder Ausführung von 4.2 genau eine von beiden ausgeführt, es sind somit nach Schritt 2 zu 4.2 die Unterfragmente 4.2.1 und 4.2.2 zu definieren.
- Die Alternativen der CASE-Anweisung der Programmeinheit DBMS sind initiierend für die Algorithmen, die zur Implementierung der jeweiligen Operation zur Verfügung stehen.
  Alle Alternativen haben als gemeinsamen Ersatz die Anzeige 'operation not implemented', so daß hier die Zerlegung nach 3.2.1.3, Bild 3.3b, vorgenommen wird.
- Für das Beispielsystem erhält man die folgenden Mengen von X-Fragmenten: { 1.1 , 1.2 , 1.3 , 1.4 , 1.5 , 1.6 } ,
  { 4.1 , 4.2 } , { 4.2.1 , 4.2.2 } , { 5.1 , 5.2 } , { 6.1 , 6.2 } ,
  { 8.1 , 8.2 } , {10.1 ,10.2 }.
  Die O-Fragmente sind: 2.1 , 5.3 , 8.3 , 9.1 , 10.3
- Nicht zu allen Anweisungen einer Programmeinheit, die initiierend für einen Algorithmus sind, ist ein Fragment einzuführen: Die Aufrufe von OPEN_RF (Algorithmus A1) bzw. CLOSE_RF (Algorithmus A3) der Fragmente 2 bzw. 9 sind notwendigerweise mit der Ausführung des jeweiligen Fragments auszuführen, es wäre daher sinnlos, entsprechende Unterfragmente einzuführen!
  Die in Schritt 2 angegebene Bedingung zur Definition von Fragmenten ist

also wesentlich!

Bemerkung:
Offensichtlich ist zur Durchführung von Schritt 2 eine detaillierte Kenntnis der Semantik des Programms erforderlich, die hier für die Definition von Fragmenten erforderliche Information ist n i c h t aus der Syntax des Programms abzuleiten:

- nicht jedes CASE-Konstrukt des Quellprogramms gibt Anlaß zur Definition von X-Fragmenten; entsprechend läßt nicht jede IF-Anweisung auf ein O-Fragment schließen
- umgekehrt können O-Fragmente definiert werden, zu denen das Programm keine Verzweigungsanweisungen enthält, ein Beispiel hierfür ist das Fragment 2.1 von Bild 3.4: das Wissen, daß dieser Aufruf unnötig ist, falls keine invertierten Dateien zu eröffnen sind, setzt eine detaillierte Kenntnis der Implementierung des Systems voraus. Analog sind X-Fragmente nicht notwendigerweise Alternativen von CASE-Anweisungen.

Schritt 2 ist daher i.allg. n i c h t a u t o m a t i s i e r b a r !

Ausgehend von den X- und O-Fragmenten werden in Schritt 3 und 4 weitere Fragmente eingeführt. Sie sind eine Folge dieser bereits existierenden Fragmente und heißen deshalb a b g e l e i t e t e Fragmente.

SCHRITT 3: abgeleitete Fragmente ausführbarer Anweisungen
Für jedes Fragment f wird überprüft, ob es Anweisungen enthält, die nur mit Anweisungen von in f geschachtelten Fragmenten ausgeführt werden können. Es werden weitere Fragmente definiert, die diese Anweisungen umfassen.

Auch dies ist ein iterativer Vorgang: die Definition eines neuen Fragmentes kann zur Folge haben, daß weitere Anweisungen die Eigenschaft von oben haben und somit auch zu diesen Fragmente zu definieren sind, etc.

Beispiel:
In Bild 3.2 erhält man so die abgeleiteten Fragmente A1 und A2.

Bemerkung:
Der Programmtext eines Teilsystems enthält ein abgeleitetes Fragment f mit ausführbaren Anweisungen genau dann, wenn er mindestens eines der Fragmente enthält, mit deren Ausführung auch f ausgeführt wird.

SCHRITT 4: abgeleitete Fragmente zu Deklarationen
Für jedes Fragment f wird überprüft, ob es Deklarationsanweisungen von Datenobjekten enthält, die nur von Anweisungen der in ihm geschachtelten Fragmente referiert werden.
Zu diesen und allen globalen Datenobjekten werden Fragmente definiert (vgl. 3.2.1.3.), sog. D - F r a g m e n t e .

Die Definition von D-Fragmenten ist prinzipiell ein iterativer Vorgang, wenn eine Deklarationsanweisung eine andere Deklaration (z.B. eines Datentyps) referiert oder (wie z.B in FORTRAN) der Anfangswert eines Datenobjektes getrennt von seiner Typ-Vereinbarung in einer separaten Anweisung spezifiziert werden kann.

Beispiele:
- Für die Programmeinheit INSERT erhält man so die Fragmente D1 und D2 von Bild 3.2
- Fragment 22.1 von Bild 3.4 umfaßt die Deklaration des globalen Datenobjektes INDEX_TABLE.

Bemerkung:
- Der Programmtext eines Teilsystems enthält ein abgeleitetes Fragment f mit ausführbaren Anweisungen genau dann, wenn er mindestens eines der Fragmente mit Anweisungen enthält, die Deklarationsanweisungen von f referieren.
- Dieser Schritt ist wieder a u t o m a t i s i e r b a r, Querverweis-Listen von Compilern oder Testsysteme /GG 80/ enthalten die hierzu erforderlichen Informationen.

3.2.3. Konstruktion eines B-Programmes, Teilsystemerzeugung

Das B-Programm bildet die Basis für die Erzeugung von Teilsystemen (vgl. Bild 2.4!):
a) es muß das vollständige Programm enthalten, damit insbesondere auch das vollständige Teilsystem (s. Abschnitt 2.3) erzeugt werden kann
b) da sich aus dem Programmtext als solchem keine Fragmentierung ableiten läßt, muß es "Fragment-beschreibende" Information enthalten
c) Zu jedem t ε T und f ε F sind $\rho(t,f)$ und $\sigma(f)$ aus dem B-Programm abzuleiten

Ein B-Programm wird als die Menge

$$\{ (\rho_f, \sigma(f), f) \mid f \varepsilon F \}$$

aufgefaßt. Diese Tripel werden B l ö c k e genannt und enthalten mit den Fragmenten die Teilstrings des vollständigen Programms und die Ersatz-Zeichenketten, die Bausteine für die Teilsystemerzeugung.

Die Spezifikation der Ersatzstrings, also der Abbildung σ, ist Aufgabe des System-Entwerfers bzw. -Implementierers, dies hat 3.2.1.1. gezeigt; ein allgemeines Verfahren zur Bestimmung der Relevanzen eines Programmsystems wird in Abschnitt 4 entwickelt. Für den Rest dieses Abschnittes wird daher angenommen, daß σ und die Relevanzen des Programmsystems gegeben sind.

Die Anforderung a) legt es nahe, zur Konstruktion eines B-Programms vom Programmtext des vollständigen Systems auszugehen und es zu jedem Fragment f um ρ_f und $\sigma(f)$ zu erweitern; dies kann zusammen mit der Definition der Fragmente geschehen:
Wird zu einem Teilstring q des Programmtextes ein Fragment eingeführt, so wird im Programmtext q durch die Zeichenkette

$$[\ \rho_f\ \sigma(f) < q >\ ]$$

ersetzt, zur Definition eines Unterfragmentes f' von f zum Teilstring q' mit $q = q_1 \| q' \| q_2$ wird analog q' durch $[\ \rho_{f'}\ \sigma(f') < q' >\ ]$ ersetzt, insgesamt also enthält der Programmtext danach anstelle q die Zeichenkette

$$[\ \rho_f\ \sigma(f) < q_1\ [\ \rho_{f'}\ \sigma(f') < q' >\ ]\ q_2 >\ ]$$

Aus Definition 3.3 und der Herleitung der Fragmente nach 3.2.2. ergibt sich:

a) ein B-Programm ist eine geordnete Menge von Blöcken, wobei die dritte Komponente eines Blocks, der B l o c k t e x t , selbst wieder eine geordnete Menge von Zeichenketten des vollständigen Programms und Blöcken sein kann
b) ein B-Programm enthält in Teilstrings zerlegt das vollständige Programm, und zwar jeden Teilstring genau einmal.

Bild 3.5 zeigt die Zeichenkette, die man auf diese Weise für das Beispielsystem zur Fragmentierung nach Bild 3.4 erhält:

$$
\begin{aligned}
&[\ \rho_1\ \mathrm{NIL} < q_1\ [\ \rho_{1.0}\ \mathrm{NIL} < [\ \rho_{1.0.1}\ \mathrm{NIL} < q_2 >]\\
&\quad [\ \rho_{1.0.2}\ \mathrm{NIL} < q_3 >]\ [\ \rho_{1.0.3}\ \mathrm{NIL} < q_4 >]\ [\ \rho_{1.0.4}\ \mathrm{NIL} < q_5 >]\\
&\quad [\ \rho_{1.0.5}\ \mathrm{NIL} < q_6 >]\ [\ \rho_{1.0.6}\ \mathrm{NIL} < q_7 >]\ q_8 >]\\
&\quad [\ \rho_{1.1}\ \mathrm{NIL} < q_9 >]\ [\ \rho_{1.2}\ \mathrm{NIL} < q_{10} >]\ [\ \rho_{1.3}\ \mathrm{NIL} < q_{11} >]\\
&\quad [\ \rho_{1.4}\ \mathrm{NIL} < q_{12} >]\ [\ \rho_{1.5}\ \mathrm{NIL} < q_{13} >]\ [\ \rho_{1.6}\ \mathrm{NIL} < q_{14} >]\ q_{15} >]\\
&[\ \rho_2\ \mathrm{NIL} < q_{16}\ [\ \rho_{2.1}\ \mathrm{NIL} < q_{17} >]\ q_{18} >]\ [\ \rho_3\ \mathrm{NIL} < q_{19} >]\\
&[\ \rho_4\ \mathrm{NIL} < q_{20}\ [\ \rho_{4.1}\ \mathrm{NIL} < q_{21} >]\ q_{22}\ [\ \rho_{4.2}\ \mathrm{NIL} < q_{23}\\
&[\ \rho_{4.2.1}\ \mathrm{NIL} < q_{24} >]\ [\ \rho_{4.2.2}\ \mathrm{NIL} < q_{25} >]\ q_{26} >]\ q_{27}]\\
&[\ \rho_5\ \mathrm{NIL} < q_{28}\ [\ \rho_{5.1}\ \sigma(5.1) < q_{29} >]\ q_{30}\ [\ \rho_{5.2}\ \sigma(5.2) < q_{31} >]\ q_{32}\\
&[\ \rho_{5.3}\ \mathrm{NIL} < q_{33} >]\ q_{34} >]\\
&[\ \rho_6\ \mathrm{NIL} < q_{35}\ [\ \rho_{6.1}\ \sigma(6.1) < q_{36} >]\ q_{37}\ [\ \rho_{6.2}\ \sigma(6.2) < q_{36} >]\ q_{39} >]\\
&[\ \rho_7\ \mathrm{NIL} < q_{40} >]\\
&[\ \rho_8\ \mathrm{NIL} < q_{41}\ [\ \rho_{8.1}\ \sigma(8.1) < q_{42} >]\ q_{43}\ [\ \rho_{8.2}\ \sigma(8.2) < q_{44} >]\ q_{45}\\
&[\ \rho_{8.3}\ \mathrm{NIL} < q_{46} >]\ q_{47} >]\ [\ \rho_9\ \mathrm{NIL} < q_{48}\ [\ \rho_{9.1}\ \mathrm{NIL} < q_{49} >]\ q_{50} >]\\
&[\ \rho_{10}\ \mathrm{NIL} < q_{51}\ [\ \rho_{10.1}\ \mathrm{NIL} < q_{52} >]\ q_{53}\ [\ \rho_{10.2}\ \mathrm{NIL} < q_{54} >]\ q_{55}\\
&[\ \rho_{10.3}\ \mathrm{NIL} < q_{56} >]\ q_{57} >]\ [\ \rho_{11}\ \mathrm{NIL} < q_{58} >]\ [\ \rho_{12}\ \mathrm{NIL} < q_{59} >]\\
&[\ \rho_{13}\ \mathrm{NIL} < q_{60} >]\ [\ \rho_{14}\ \mathrm{NIL} < q_{61} >]\ [\ \rho_{15}\ \mathrm{NIL} < q_{62} >]\\
&[\ \rho_{16}\ \mathrm{NIL} < q_{63} >]\ [\ \rho_{17}\ \mathrm{NIL} < q_{64} >]\ [\ \rho_{18}\ \mathrm{NIL} < q_{65} >]\\
&[\ \rho_{19}\ \mathrm{NIL} < q_{66} >]\ [\ \rho_{20}\ \mathrm{NIL} < q_{67} >]\ [\ \rho_{21}\ \mathrm{NIL} < q_{68} >]\\
&[\ \rho_{22}\ \mathrm{NIL} < q_{69}\ [\ \rho_{22.1}\ \mathrm{NIL} < q_{70} >]\ q_{71} >]
\end{aligned}
$$

$\sigma(5.1)=\sigma(5.2)$:= return 'illegal access-type'
$\sigma(6.1)=\sigma(6.2)=\sigma(8.1)=\sigma(8.2)$:= return 'storage structure not accessible'

Bild 3.5: Das B-Programm des Beispielsystems

< und > sind die Begrenzer der Komponente Blocktext eines Blocks, einen Block selbst begrenzen [und] ; die 71 Teilstrings, in die das vollständige Programm durch diese Fragmentierung zerlegt wird und die Elemente je eines Blocktextes sind, werden in Bild 3.5 durch q_i, $1 \leq i \leq 71$, dargestellt; in Bild 3.4 sind diese Zeichenketten durch Angabe der Indizes rechts vom Programmtext gekennzeichnet: q_i ist die Zeichenkette von Bild 3.4, die aus der Zeile mit der Markierung i und allen folgenden bis zur Zeile mit der Markierung i+1 ausschließlich besteht.

Bemerkung:
Es ist hier noch offen, wie ρ, σ, die Teilstrings q_i und die Begrenzer realisiert werden, diese Fragen behandelt Abschnitt 5.

Die Erzeugung des Programmtextes für ein Teilsystem t, also Auswahl von Teilstrings und deren Zusammenbau i.allg. mit Ersatz-Zeichenketten, ist im wesentlichen ein Konkatenieren von Zeichenketten:
Beginnend mit dem ersten Block des B-Programms wird pro Block der Relevanzwert der Komponente Blocktext bestimmt. Ist dieser 0, d.h. das Fragment, dem der Block entspricht, ist für t nicht relevant, so wird die Block-Komponente $\sigma(f)$ an den bisher erzeugten Text (der Anfangswert hierfür ist der leere string NIL!) angehängt; sonst wird der Blocktext "ausgewertet":

- ist der Blocktext ein Element von Σ, so wird dieses an den bisher erzeugten Text angehängt
- ist der Blocktext eine geordnete Menge mit Blöcken, so werden diese in der gegebenen Reihenfolge ausgewertet: Elemente von Σ werden an den bisher erzeugten Text angefügt, bei einem Block ist wieder zunächst der Relevanzwert zu ermitteln und dann wie oben zu verfahren, etc.

Zur Erzeugung eines Teilsystems genügt somit ein sequentielles "Lesen" des B-Programms!

Bemerkung:
Offensichtlich ist die Reihenfolge der Blöcke des B-Programms für die Teilsystem-Erzeugung wesentlich: sie bestimmt die Reihenfolge, in der die Teilstrings des vollständigen Systems zum Programm eines Teilsystems integriert werden.

Beispiel:
Wie in Abschnitt 4 (vgl. auch Anhang 2) gezeigt wird sind für das Teilsystem t=t_ins von Abschnitt 3.1 die folgenden Relevanzwerte gleich 1:
$\rho(t,1)$, $\rho(t,1.0)$, $\rho(t,1.0.3)$, $\rho(t,1.0.4)$, $\rho(t,1.0.6)$, $\rho(t,1.1)$, $\rho(t,1.2)$, $\rho(t,1.5)$, $\rho(t,2)$, $\rho(t,5)$, $\rho(t,5.1)$, $\rho(t,5.3)$, $\rho(t,6)$, $\rho(t,6.1)$, $\rho(t,8)$, $\rho(t,8.1)$, $\rho(t,9)$, $\rho(t,11)$, $\rho(t,12)$, $\rho(t,15)$;
alle anderen Relevanzen liefern für t=t_ins den Wert 0.
Man erhält damit für das Teilsystem t_ins das Programm von Bild 3.6.

```
1|     PROCEDURE DBMS
1|
1|        IF (OP<1 OR OP>6)
1|           THEN return 'operation unknown'
1|        CASE OP OF
**
1|0|3|      3,
1|0|4|      4,
**
1|0|6|      6,
1|0|        0: return 'operation not
1|0|                   implemented'
1|1|        1: OPEN
1|2|        2: CLOSE
**
1|5|        5: INSERT
**
1|        END
1|     END
2|     PROCEDURE OPEN
2|        OPEN_RF
**
2|     END
5|     PROCEDURE GET
5|        NEXT_TUPLE:
5|        CASE ACCESS_TYPE OF
5|          1: . . . . . .
5|1|           NEXT_SEQ
5|1|           . . . . . .
5|1|
5|          2: . . . . . .
**            return 'illegal access-type'
5|        END
5|3|        IF (qualifikation is not satisfied)
5|3|           THEN GO TO NEXT_TUPLE
5|     END
```

```
6|    PROCEDURE NEXT_SEQ
6|       CASE FILE_TYPE OF
6|          1:
6|1|          next_1
6|          2:
**             return 'storage structure
**                     not accessible'
6|       END
6|    END
8|    PROCEDURE INSERT
8|       CASE FILE_TYPE OF
8|          1:
8|1|          INSERT_1
8|          2:
**             return 'storage structure
**                     not accessible'
8|       END
**
8|    END
9|    PROCEDURE CLOSE
9|       CLOSE_RF
**
9|    END
11|   PROCEDURE OPEN_RF
11|     . . . . . .
11|     GET
11|     . . . . . .
11|   END
12|   PROCEDURE CLOSE_RF
12|     . . . . . .
12|   END
15|   PROCEDURE INSERT_1
15|     . . . . . .
15|   END
```

** : Ersatzstring (nur innerhalb von Programmeinheiten)

Bild 3.6: Das Programm des Teilsystems t_ins

3.3. Ein Modell zur Erzeugung von Teilsystemen

Die Erzeugung von Teilsystemen nach Abschnitt 3.2 läßt sich formal als eine Operation auf einer Baumstruktur beschreiben.
Ein Fragment $f \varepsilon F$ ist nach Definition 3.3 eine geordnete Menge (vgl. Anhang 1)

$$f = < f[1], f[2], \ldots, f[n] > \qquad n \geq 1$$

mit $f[i] \varepsilon F$ oder $f[i] \varepsilon \Sigma$ (d.h. f[i] ist ein Unterfragment von f oder ein Teilstring des vollständigen Programms).
Bezeichnet man mit Q die Menge der Teilstrings des Programmsystems, so ist auf der Menge F+Q eine Relation S definiert durch

$$S = \{ (f,g) \mid f \varepsilon F, g \varepsilon F+Q, g \text{ ist Element von } f \}.$$

Hat man k Fragmente der Tiefe 1, so ist (F+Q,S) eine Menge von k geordneten Bäumen mit diesen k Fragmenten als Wurzeln: nach Definition 3.3 hat ein Fragment höchstens ein Oberfragment und ein Teilstring des vollständigen Programms ist Element von höchstens einem Fragment.
Ergänzt man F um das "Pseudo-Fragment" SYS := $< f_1, \ldots, f_k >$ (man beachte: SYS genügt Definition 3.3 für Fragmente!), so erhält man einen geordneten Baum mit der Wurzel SYS: die Teilstrings des Programmsystems sind die Blattknoten, die Fragmente des Programmsystems die inneren Knoten des Baumes.

DEFINITION 3.4:
F und Q seien nichtleere Mengen mit $Q \subseteq \Sigma$ und $F \cap Q = \emptyset$; es sei P:=F+Q und $S \subseteq P \times P$ eine Relation auf P; weiterhin gebe es zwei Abbildungen $\sigma: F \rightarrow \Sigma$ und $\rho: T \times F \rightarrow B$.
(P,S,σ,ρ) heißt ein a b s t r a k t e s B - P r o g r a m m , wenn erfüllt ist:
B1: (P,S) ist ein geordneter Baum mit den Elementen von Q als Blattknoten
B2: $(f,g) \varepsilon S$, $g \varepsilon F$, $\rho(t,f)=0 \Longrightarrow \rho(t,g) = 0$
B3: für die Wurzel r von (P,S) gilt: $\rho(t,r) = 1$ für $t \varepsilon T$, $\sigma(r)=NIL$

Bemerkung: Eigenschaft B1 impliziert $S \subseteq F \times P$!

Wie oben gezeigt, haben B-Programme nach 3.2. ergänzt um das Pseudo-Fragment die Eigenschaft B1 eines abstrakten B-Programmes. Sie erfüllen auch B2 nach Definition 3.3. B3 beschreibt die Erweiterung des Definitionsbereichs der Abbildung ρ bzw. σ von 3.1. um das Pseudo-Fragment: diese formalen Festlegungen werden unten zur Definition von Teilsystem-Erzeugung in Verfahren 3.1 benötigt.

Beispiel:
Bild 3.7 zeigt den geordneten Baum (F+Q,S) des abstrakten B-Programmes zur Fragmentierung des Beispielsystems von Bild 3.4. Für die graphische Darstellung von abstrakten B-Programmen wird für diese Arbeit festgelegt: Die Knoten von F, also Fragmente, werden durch Rechtecke, Zeichenketten aus Q (also die Blattknoten) durch Kreise dargestellt.
Zu jedem Knoten ist sein Bezeichner angegeben, bei Blattknoten sind dies (anders als im Text!) lediglich die Indizes, die zur eindeutigen Unterscheidung von Fragment-Bezeichnern mit einem Querstrich versehen sind:

$\bar{i}$ steht hier also anstelle von q_i !

Der Bezeichner des Pseudo-Fragments ist "DBMS".

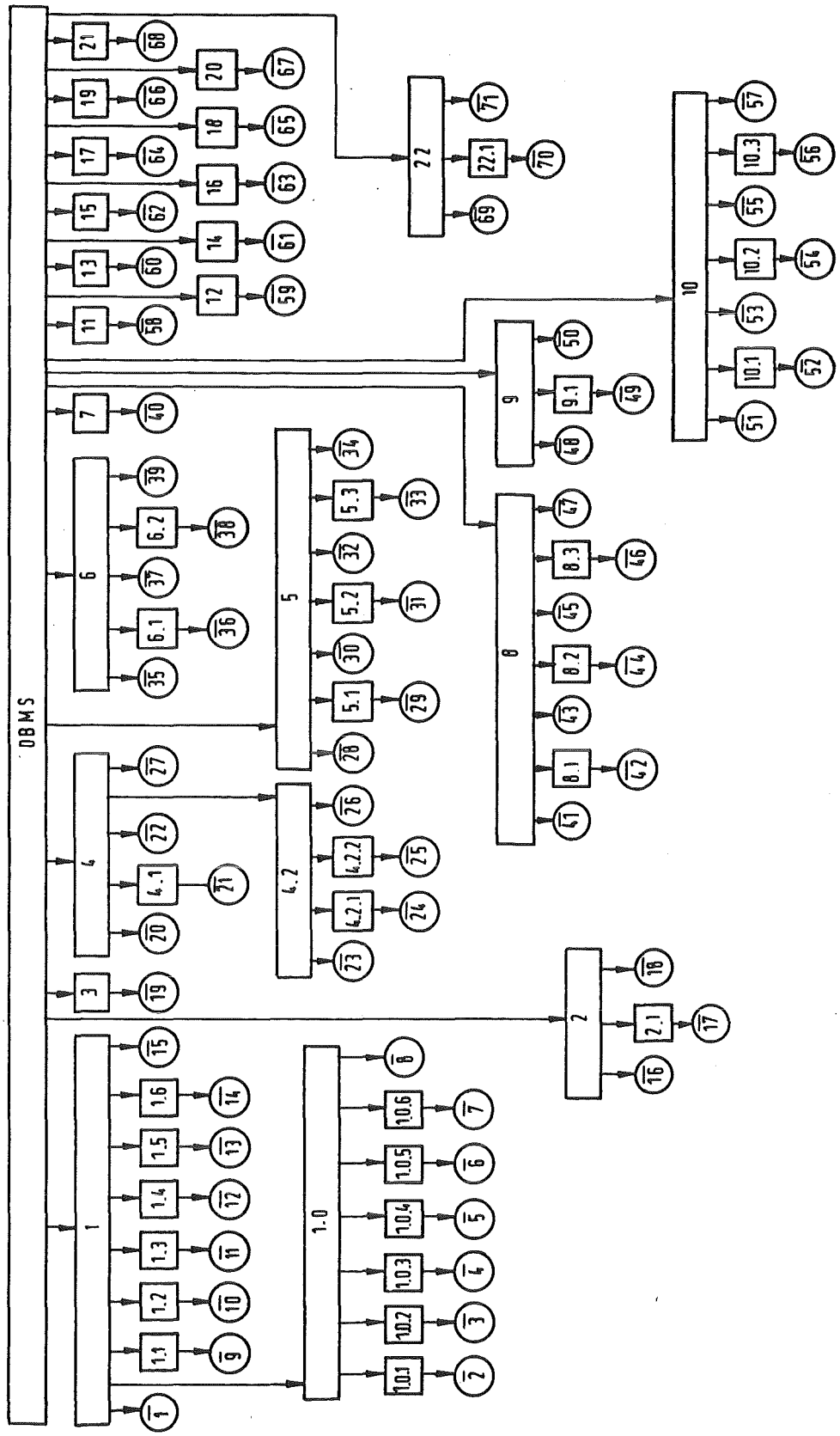


Bild 3.7: Die Baumstruktur des abstrakten B-Programms zum Beispielsystem

Die Anordnung der Nachfolger eines Knotens von links nach rechts zeigt die Ordnung der Menge SUCC(f) an.

Die Erzeugung von Teilsystemen wird formal als als ein Durchsuchen des geordneten Baumes eines abstrakten B-Programms beschrieben:

VERFAHREN 3.1: Erzeugung eines Teilsystems
Input : B-Programm BP=(P,S,σ,ρ), t ε T
Output: Der Programmtext PROG=PROG(t,BP) des Teilsystems t
Algorithmus:

```
            r sei die Wurzel von (P,S)
            PROG = NIL
            EVAL(t,r)
mit
            PROCEDURE EVAL(t,x)
               IF (x ε F)
                  THEN
                       IF (ρ(t,x)=0)
                          THEN PROG = PROG ‖ σ(x)
                          ELSE FOR I=1 TO |SUCC(x)|
                               DO
                               EVAL(t,x[I])
                               END
                  ELSE PROG = PROG ‖ x
            END
```

Zur Erzeugung eines Teilsystems wird mit Verfahren 3.1 eine Teilmenge der Knoten des B-Programmes "ausgewertet", wobei die A u s w e r t u n g eines Knotens x für t ε T, notiert: P(t,x), wie folgt definiert wird:

```
                          +-
                          | bestimme ρ(t,x) : x ε F
              P(t,x) := <
                          | PROG=PROG‖x     : x ε Q
                          +-
```

DEFINITION 3.5:
(P,S,σ,ρ) sei ein B-Programm, P=F+Q, r die Wurzel von (P,S) und t ε T. Der Teilbaum (P_t,S_t) von (P,S) mit

$P_t := \{ p \mid p \varepsilon P, p=r$ oder es gibt ein x ε PRED(p) mit $\rho(t,x)=1 \}$

$S_t := S^*(P_t \times P_t)$

heißt der für t r e l e v a n t e T e i l b a u m des B-Programms.

Beispiel: Anhang 2 enthält den für t_ins relevanten Teilbaum.

Bezeichnung:
$SUCC_t(x)$ bezeichnet die Menge der Nachfolger des Knotens x des relevanten Teilbaums (P_t,S_t), also: $SUCC_t(x) := SUCC(x)^*P_t$

Bemerkung: Wegen Eigenschaft B2 gilt

- $x \in P_t^*F$, $\rho(t,x)=0$ ==> x ist ein Blattknoten von (P_t,S_t), d.h.

$$SUCC_t(x) = \emptyset$$

- x ist ein Blattknoten von (P_t,S_t) <==> ($x \in P_t^*F$ und $\rho(t,x)=0$) oder $x \in Q$

SATZ 3.1:
Zur Erzeugung eines Teilsystems t nach Verfahren 3.1 werden genau die Knoten des relevanten Teilbaumes (P_t,S_t) ausgewertet, dieser wird hierzu in preorder-Reihenfolge durchlaufen.
Beweis:
Aus der Bemerkung zu Definition 3.5 und Verfahren 3.1 ergibt sich unmittelbar der erste Teil der Behauptung.
EVAL(t,x) und damit P(t,x) wird für jedes $x \in P_t$ ausgeführt; ist x kein Blattknoten von (P_t,S_t), ist also $|SUCC_t(x)|>0$, so wird nach Verfahren 3.1 EVAL auf die Nachfolger von x in der durch den geordneten Baum gegebenen Reihenfolge angewandt. Die Reihenfolge der Auswertung der Knoten des relevanten Teilbaumes (P_t,S_t) läßt sich damit rekursiv wie folgt beschreiben[2] (r ist wieder die Wurzel von (P,S)):

PREORDER(t,r)

mit

```
PROCEDURE PREORDER(t,x)
   P(t,x)
   FOR I=1 TO |SUCC_t(x)|
   DO
      PREORDER(t,x[i])
   END
END
```

Dies ist genau die Definition für preorder-Durchlauf geordneter Bäume /AH 74, Kn 69, Wi 79/.
□

Notation:
$<_{BP}$ bezeichnet die preorder-Ordnung der Knoten des geordneten Baumes eines B-Programms BP: $x <_{BP} y$ <==>
<==> x wird bei preorder-Durchlauf vor y aufgesucht, d.h. die durch Numerierung der Knoten des geordneten Baumes nach preorder-Durchlauf dem Knoten x zugewiesene Zahl ist kleiner als die von y /Wi 79/.

---

[2] Es wird die Konvention zugrunde gelegt, daß eine Wiederholungsanweisung
FOR I = m TO n DO statements END
im Fall n<m gleich der leeren Anweisung ist, d.h. statements nicht ausgeführt wird.

Aus Satz 3.1 ergibt sich unmittelbar, daß der Programmtext PROG(t,BP) die Konkatenation
- der Blattknoten $q \varepsilon Q$ und
- der Ersatzstrings $\sigma(f)$ der Blattknoten $f \varepsilon F$

des relevanten Teilbaumes (P_t,S_t) ist.

Weiter gilt mit $x_1,x_2,x_3 \varepsilon \Sigma$:

- $q_1,q_2 \varepsilon Q^*P_t$, $q_1 <_{BP} q_2$ ==> PROG (t,BP) = $x_1 \| q_1 \| x_2 \| q_2 \| x_3$
- $q \varepsilon Q^*P_t$, $f \varepsilon F$ ist Blattknoten von (P_t,S_t), $q <_{BP} f$ ==>

==> PROG(t,BP) = $x_1 \| q \| x_2 \| \sigma(f) \| x_3$

Insbesondere folgt, daß der Programmtext eines Teilsystems t bereits mit der Menge der für t relevanten Fragmente eindeutig beschrieben ist: die Reihenfolge, in der die Teilstrings des vollständigen Programms und Ersatz-Zeichenketten konkateniert werden, ist implizit mit der Struktur des abstrakten B-Programms festgelegt. Es gilt also:

KOROLLAR 3.1:
Ein Teilsystem $t \varepsilon T$ ist eindeutig beschrieben durch die Menge F_t der für t relevanten Fragmente: $F_t = \{ f \mid f \varepsilon F, \rho_f(t)=1 \}$

DEFINITION 3.6:
- Zwei B-Programme $BP_1=(P_1,S_1,\sigma_1,\rho_1)$, $BP_2=(P_2,S_2,\sigma_2,\rho_2)$ heißen ä q u i - v a l e n t , wenn mit Verfahren 3.1 $PROG(t,BP_1) = PROG(t,BP_2)$ für alle $t \varepsilon T$ gilt.
- Sind BP_1, BP_2 äquivalente B-Programme, so heißt BP_2 e i n f a c h e r als BP_1, wenn $|P_2| < |P_1|$ ist.

Jedem $f \varepsilon F-\{r\}$ eines abstrakten B-Programms entspricht umkehrbar eindeutig ein Block eines B-Programmes bzw. ein Fragment. Bei Herleitung einer Fragmentierung nach 3.2. kann man Fragmente f erhalten, die die gleiche Relevanz wie das Oberfragment haben, d.h. f wird genau dann benötigt, wenn sein Oberfragment benötigt wird. Solche Fragmente sind offensichtlich sinnlos und können aufgelöst werden. Dies bedeutet, daß ein einfacheres B-Programm zur Erzeugung der Teilsysteme von T genügt.

Die folgenden drei Transformationen geben an, unter welchen Bedingungen und wie ein B-Programm BP in ein äquivalentes, einfacheres B-Programm BP' übergeführt werden kann.

Bezeichnung:
Zur Beschreibung der Ordnung der geordneten Bäume von B-Programmen BP bzw. BP' wird im folgenden mit $\nu(x)$ bzw. $\nu'(x)$ der Index des Knotens x von BP bzw. BP' bezeichnet, vgl. Anhang 1. (ν, ν' dienen also nur der "Zählung", n i c h t der Benennung von Knoten!.)
Weiter ist zu beachten:
SUCC(f) bzw. PRED(f) bezieht sich bei der folgenden Beschreibung der Transformationen 1, 2 und 3 stets auf f als Knoten des geordneten Baumes des B-Programms BP (und nicht BP'!)

TRANSFORMATION 1:
Es sei BP=(P,S,σ,ρ) ein B-Programm, P=F+Q, weiter gebe es f,g ε F mit

T1.1: $(f,g) \in S$

T1.2: $\rho_f \equiv \rho_g$

BP':=(P',S',σ',ρ'), P'=F'+Q', erhält man aus BP wie folgt (vgl. Bild 3.8):

$F' := F - \{f,g\} + \{f'\}$ mit $f' \notin F$, $\quad Q' := Q$

$S' := S - \{(x,y) \mid (x,y)\in S,\ x=f \text{ oder } x=g\} - \{(x,f) \mid x\in PRED(f)\} + \{(f',x) \mid x\in P,\ x\neq g,\ (g,x)\in S \text{ oder } (f,x)\in S\} + \{(x,f') \mid x\in PRED(f)\}$

$$\rho'(t,x) := \begin{cases} \rho(t,x) & : x \in F'-\{f'\} \\ \rho(t,f) & : x=f' \end{cases} \qquad \sigma'(x) := \begin{cases} \sigma(x) & : x \in F'-\{f'\} \\ \sigma(f) & : x=f' \end{cases}$$

Mit m=|SUCC(g)|, n=|SUCC(f)| gilt:

$$\nu'(x) := \begin{cases} \nu(x) & : x \in P'-(SUCC(f)+SUCC(g)+\{f'\}) \\ \nu(f) & : x = f' \\ \nu(x) & : x \in SUCC(f),\ \nu(x)<\nu(g) \\ \nu(x)+\nu(g)-1 & : x \in SUCC(g) \\ \nu(x)+m & : x \in SUCC(f),\ \nu(x)>\nu(g) \end{cases}$$

Erläuterung: Es werden zwei Knoten f und g zu e i n e m neuen Knoten f' zusammengefaßt, der als Nachfolger die Knoten SUCC(f)+SUCC(g)-{g} hat (Bild 3.8); die Definition von ν' stellt sicher, daß die Ordnung der Knoten erhalten bleibt:

$$f'[i] = \begin{cases} f[i] & : 1\le i\le \nu(g)-1 \\ g[i+1-\nu(g)] & : \nu(g)\le i\le \nu(g)+m-1 \\ f[i+1-m] & : \nu(g)+m\le i\le m+n-1 \end{cases}$$

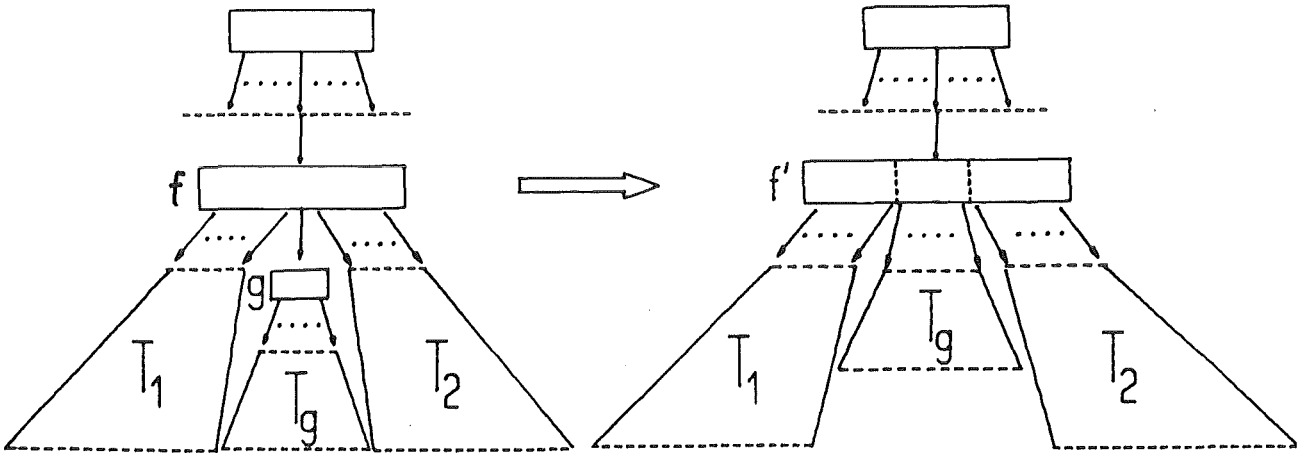


Bild 3.8: Vereinfachung eines B-Programmes mittels Transformation 1

TRANSFORMATION 2:
Es sei BP=(P,S,σ,ρ) ein B-Programm, P=F+Q, weiter gebe es f,g,h ε F mit

T2.1: $f,g \in SUCC(h)$

T2.2: $\nu(f)+1=\nu(g)$

T2.3: $\rho_f \equiv \rho_g$

BP':=(P',S',σ',ρ'), P'=F'+Q', erhält man aus B wie folgt (vgl. Bild 3.9):

F' := F - {f,g,h} + {f',h'} mit f',h' ¬ε F, Q' := Q

S' := S - {(x,y) |(x,y) ε S, x=f oder x=g oder x=h} - {(x,h)|xεPRED(h)}
+ {(f',x)|x ε P, (f,x) ε S oder (g,x) ε S }
+ {(h',x)|x ε P, (h,x) ε S, x≠f, x≠g } + {(h',f')}
+ {(x,h')|x ε PRED(h) }

$$\rho'(t,x) := \begin{cases} \rho(t,x) & : x \in F'-\{f',h'\} \\ \rho(t,h) & : x=h' \\ \rho(t,f) & : x=f' \end{cases} \qquad \sigma'(x) := \begin{cases} \sigma(x) & : x \in F'-\{f',h'\} \\ \sigma(h) & : x=h' \\ \sigma(f)\|\sigma(g) & : x=f' \end{cases}$$

Mit m=|SUCC(g)|, n=|SUCC(f)| gilt:

$$\nu'(x) := \begin{cases} \nu(x) & : x \in P'-(SUCC(h)+SUCC(g)+\{f',h'\}) \\ \nu(f) & : x = f' \\ \nu(h) & : x = h' \\ \nu(x) & : x \in SUCC(h),\ \nu(x)<\nu(f) \\ \nu(x)+n & : x \in SUCC(g) \\ \nu(x)-1 & : x \in SUCC(h),\ \nu(x)>\nu(g) \end{cases}$$

Erläuterung: Es werden hier zwei benachbarte Knoten f und g zu e i n e m neuen Knoten zusammengefaßt (Bild 3.9), der als Nachfolger die Knoten von SUCC(f)+SUCC(g) hat; die Definition von ν' stellt sicher, daß die Ordnung der Knoten erhalten bleibt. Für die Nachfolger von f' und h' in (P',S') hat man also:

$$f'[i] = \begin{cases} f[i] & : 1 \le i \le n \\ g[i-n] & : n+1 \le i \le m+n \end{cases} \qquad h'[i] = \begin{cases} h[i] & : 1 \le i \le \nu(f)-1 \\ f' & : i=\nu(f) \\ h[i+1] & : \nu(f)+1 \le i \le |SUCC(h)|-1 \end{cases}$$

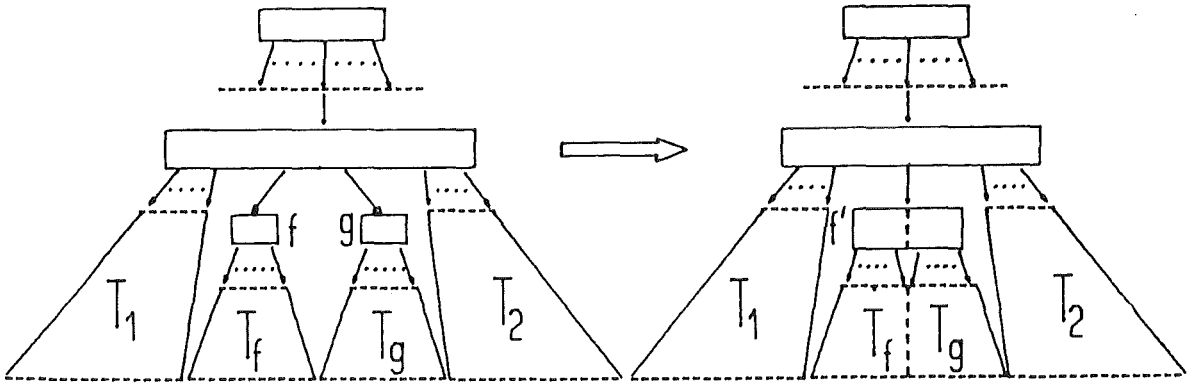


Bild 3.9: Vereinfachung eines B-Programmes mittels Transformation 2

TRANSFORMATION 3:
Es sei BP=(P,S,σ,ρ) ein B-Programm, P=F+Q, weiter gebe es f ε F und u,v ε Q mit
T3.1: u,v ε SUCC(f)
T3.2: ν(u)+1=ν(v)

BP':=(P',S',σ',ρ'), P'=F'+Q', erhält man aus BP wie folgt:

F' := F - {f} + {f'} mit f' ¬ε F
Q' := Q - {u,v} + {q} mit q = u‖v
S' := S - {(f,x) |(f,x) ε S} - {(x,f)|xεPRED(f)}
+ {(x,f')|x ε PRED(f) }
+ {(f',q)} + {(f',x)|x ε P, (f,x) ε S, x≠u, x≠v }

$$\rho'(t,x) := \begin{cases} \rho(t,x) & : x \in F'-\{f'\} \\ \rho(t,f) & : x=f' \end{cases} \qquad \sigma'(x) := \begin{cases} \sigma(x) & : x \in F'-\{f'\} \\ \sigma(f) & : x=f' \end{cases}$$

$$\nu'(x) := \begin{cases} \nu(x) & : x \in P'-(SUCC(f)+\{q,f'\}) \\ \nu(u) & : x = q \\ \nu(f) & : x = f' \\ \nu(x) & : x \in SUCC(f),\ \nu(x)<\nu(u) \\ \nu(x)-1 & : x \in SUCC(f),\ \nu(x)>\nu(v) \end{cases}$$

Erläuterung: Der Unterschied zu Transformation 2 besteht lediglich darin, daß hier zwei B l a t t k n o t e n von BP (durch ihre Konkatenation) ersetzt werden:

$$f'[i] = \begin{cases} f[i] & : 1 \le i \le \nu(u)-1 \\ u \,\|\, v & : i=\nu(u) \\ f[i+1] & : \nu(u)+1 \le i \le |SUCC(f)|-1 \end{cases}$$

SATZ 3.2:
BP'=(P',S',σ',ρ') sei aus dem B-Programm BP=(P,S,σ,ρ) durch Anwendung einer der Transformationen von oben hervorgegangen. Es gilt:
- BP' ist ein B-Programm
- BP und BP' sind äquivalent, BP' ist einfacher als BP.

Beweis:
a) Die Konstruktionsvorschriften besagen, daß in (P,S) zwei Knoten x und y durch einen neuen Knoten z ersetzt werden, d.h. |P'|=|P|-1.
Nach Definition von S' ist (P',S') ein Baum, d.h. es gilt B1:
- der Vorgänger von z in (P',S') ist der Vorgänger von x in (P,S) (h und h' von Transformation 2 und f und f' von Transformation 3 können für diese Überlegungen als jeweils identisch betrachtet werden!)
- die Knoten in (P',S'), die in (P,S) als (einzigen!) Vorgänger x oder y haben, haben in (P',S') den Vorgänger z
- (P,S) und (P',S') stimmen in den übrigen Kanten und Knoten überein

- Aus der Definition von ν' folgt unmittelbar, daß (P',S') ein geordneter Baum ist; die Transformationen sind sogar in folgendem Sinne ordnungserhaltend:

$$p_1,p_2 \in P*P',\ p_1 <_{BP} p_2 \Longrightarrow p_1 <_{BP'} p_2$$

Aus der Definition von ρ' und σ' folgt unmittelbar, daß BP' auch den Bedingungen B2 und B3 genügt (Transformation 2 ist mit f=r nicht möglich!): BP' ist somit ein B-Programm.

b) Es ist noch PROG(t,BP)=PROG(t,BP'), $t \in T$ zu zeigen.
(Es werden die bei den jeweiligen Transformationen eingeführten Bezeichnungen benutzt)

b1) zu Transformation 1:

Wegen Satz 3.1 genügt es zu zeigen, daß EVAL(t,f) für BP gleich EVAL(t,f') für BP' ist, d.h. gleiche Zeichenketten konkateniert werden.
Wegen T1.2 sind lediglich zwei Fälle zu unterscheiden:
Fall 1: $\rho'(t,f')=\rho(t,f)=0$
Die Behauptung folgt hier unmittelbar aus $\sigma(f)=\sigma'(f')$.
Fall 2: $\rho'(t,f')=\rho(t,f)=1$
Unter Beachtung von T1.1 und mit n:=|SUCC(f)|, m:=|SUCC(g)| läßt sich EVAL(t,f') bzw. EVAL(t,f) wie folgt schreiben:

EVAL(t,f'):	EVAL(t,f):
FOR I=1 TO ν(g)-1	FOR I=1 TO ν(g)-1
DO	DO
EVAL(t,f'[I])	EVAL(t,f[I])
END	END
FOR I=ν(g) TO ν(g)+m-1	
DO	
EVAL(t,f'[I])	EVAL(t,g)
END	
FOR I=ν(g)+m TO m+n-1	FOR I=ν(g)+1 TO n
DO	DO
EVAL(t,f'[I])	EVAL(t,f[I])
END	END

Aus der Erläuterung zu Transformation 1 ergibt sich, daß
- jeweils die ersten und letzten Wiederholungsanweisungen der beiden Fälle identisch sind
- die zweite Wiederholungsanweisung zu EVAL(t,f') geschrieben werden kann als

```
FOR I=1 TO m
DO
   EVAL(t,g[I])
END
```

was gerade EVAL(t,g) der rechten Spalte darstellt.

Die Behauptung des Satzes ist somit für Transformation 1 gezeigt.

b2) zu Transformation 2:

Analog zu b1) ist nachzuweisen, daß EVAL(t,h) (für BP) und EVAL(t,h') (für BP') identisch sind. Berücksichtigt man T2.2 und die Erläuterungen zu den Nachfolgern von h, so ist hierzu die Gleichheit von

EVAL(t,f)	|	EVAL(t,f')
EVAL(t,g)	|	

zu demonstrieren:
Fall 1: $\rho'(t,f')=\rho(t,f)=0$
Die Behauptung folgt hier unmittelbar aus der Definition von σ': $\sigma'(f')=\sigma(f)\|\sigma(g)$
Fall 2: $\rho'(t,f')=\rho(t,f)=1$
Mit den Erläuterungen zu den Nachfolgern von f' in (P',S') erhält man die Behauptung analog zu b1) unmittelbar aus der Definition von EVAL.

b3) zu Transformation 3:
Wegen $\sigma'(f')=\sigma(f)$ und $q = u\|v$ sind EVAL(t,f) (für BP) und EVAL(t,f') (für BP') identisch und somit die Behauptung auch für Transformation 3 gezeigt.
□

Von einem abstrakten B-Programm (insbesondere also einer Fragmentierung) ausgehend kann man durch Anwendung der Transformationen versuchen, ein einfacheres B-Programm (und damit eine einfachere Fragmentierung!) zu erhalten; dies kann geschehen durch

a) Beseitigung unnötiger Unterfragmente und damit Blockschachtelungen mittels Transformation 1
b) Zusammenfassung benachbarter Knoten gleicher Relevanz zu e i n e m Knoten, was der Bildung eines Blockes aus zwei unmittelbar aufeinander folgenden Blöcken zu Fragmenten gleicher Tiefe entspricht.

Beispiele:
zu a):
Bild 3.10 zeigt den Teilbaum eines abstrakten B-Programms des Beispielsystems zur Fragmentierung der Programmeinheit INSERT nach Bild 3.2. Nach Abschnitt 3.2.1 haben die Fragmente D1 und A1 die gleiche Relevanz wie Fragment 8, ihr Oberfragment (Programmeinheit INSERT):

$$\rho_8 \equiv \rho_{D1} \equiv \rho_{A1}.$$

Zweimalige Anwendung von Transformation 1 führt zum B-Programm von Bild 3.10b, das zwei Knoten weniger enthält als das von Bild 3.10a. Mit Transformation 3 kann man schließlich von den Nachfolgern des Knotens 8' die zwei Paare benachbarter Blattknoten zu je einem Knoten zusammenfassen und gelangt so zur Baumstruktur von Bild 3.10c.

zu b):
In Abschnitt 4 (vgl. auch Anhang 2) wird gezeigt, daß u.a. $\rho_3 \equiv \rho_4$ gilt (dies ist auch unmittelbar einsichtig, da STRTGY stets und nur von FIND aufgerufen wird!). Das B-Programm von Bild 3.4 kann damit durch Anwendung von Transformation 2 (Bild 3.11b) und Transformation 3 (Bild 3.11c) vereinfacht werden.

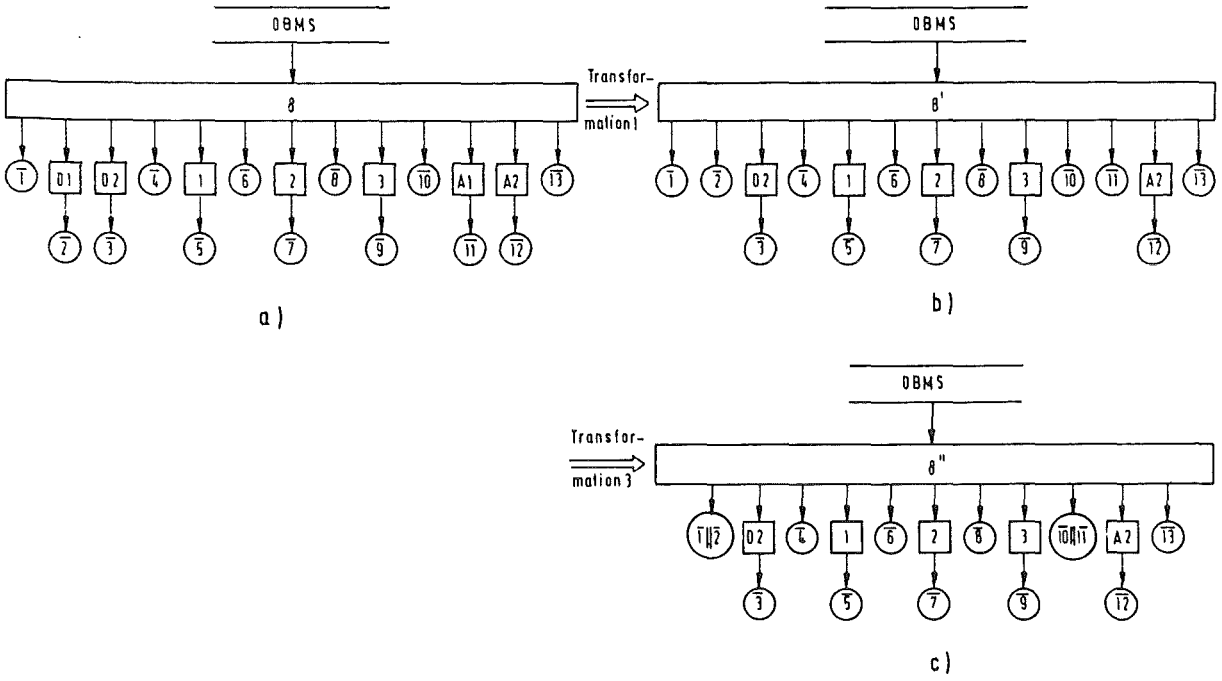


Bild 3.10: Anwendung von Transformation 1

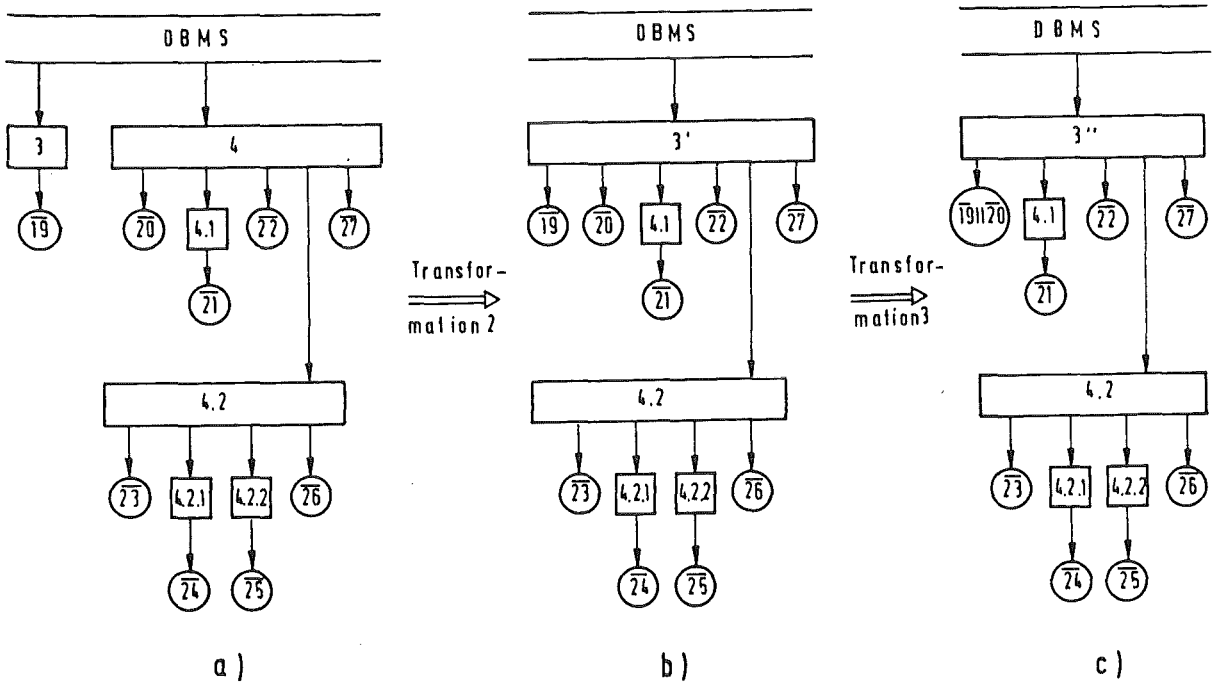


Bild 3.11: Anwendung von Transformation 2

Zur Vereinfachung nach Transformation 2 und 3 ist die Ordnung der Knoten des B-Programms wesentlich: da benachbarte Knoten gleicher Relevanz nach Transformation 2 zu einem einzigen zusammengefaßt werden können, wird man versuchen, durch Umordnen ein abstraktes B-Programm mit möglichst langen Folgen solcher Knoten zu bilden.
Die Ordnung der Knoten des B-Programms bestimmt die Reihenfolge der Teilstrings des vollständigen Programmes und umgekehrt, eine Voraussetzung für eine Änderung der Ordnung der Knoten (dies kann als eine vierte Transformation interpretiert werden, bei der nur eine neue Abbildung ν' definiert wird!) ist daher, daß die Vertauschung der entsprechenden Programmteile im vollständigen Programm dieses semantisch unverändert läßt:

- Die Reihenfolge separat übersetzbarer Programmeinheiten eines Programmsystems ist irrelevant, sie sind miteinander vertauschbar
- Entsprechendes gilt in der Regel auch für Deklarationsanweisungen einer Programmeinheit.

Beispiel:
Spielt die Reihenfolge der Programmeinheiten des Beispielsystems im Programmtext keine Rolle, so kann der geordnete Baum von Bild 3.7 so umgeformt werden, daß Fragment 11 der rechte Nachbar von Fragment 2 ist (im Programmtext folgt dann die Programmeinheit OPEN_RF unmittelbar auf OPEN); wegen $\rho_2 \equiv \rho_{11}$ sind dann wieder die Voraussetzungen zur Vereinfachung des B-Programms Transformation 2 (und 3) gegeben.

4. Die Menge der Teilsysteme

In Abschnitt 3.2 ist festgestellt worden, daß i.allg. nicht aus jeder beliebigen Teilmenge der Fragmente F eines Programmsystems ein Teilsystem erzeugt werden kann; vielmehr kann für ein Fragment f gelten: "ist f für ein $t \varepsilon T$ relevant, dann wird auch ein Fragment g für t benötigt" (vgl. 3.2.1.1. und Bemerkungen zum Verfahren von Abschnitt 3.2.2). Der Relevanzwert $\rho(t,f)$ ist also i.allg. nicht frei wählbar, sondern durch die Relevanzwerte $\rho(t,g)$ anderer Fragmente g festgelegt.
Ziel dieses Abschnittes ist es, aus diesen Beziehungen zwischen den Relevanzen von Fragmenten, die auch als Eigenschaften der Abbildung ρ aufgefaßt werden können, eine explizite Darstellung für ρ und, was nach Korollar 3.1 äquivalent dazu ist, eine Beschreibung der Menge T der Teilsysteme eines Programmsystems herzuleiten.

4.1. Fragmentsysteme

Zur Ermittlung der Fragmente untersucht man in 3.2. den Kontrollfluß des Programmsystems und in welcher Weise Datenobjekte referiert werden. Für die Abhängigkeiten zwischen den Relevanzen von Fragmenten ist es dabei unwesentlich, ob die Fragmente ausführbare Anweisungen oder Deklarationen enthalten, es genügt (als Verallgemeinerung) die Feststellung, daß zwischen Fragmenten eine Relation R besteht, wobei $R \subseteq F \times F$ wie folgt definiert ist:
$(f,g) \varepsilon R$, in Worten: "f referiert g", genau dann, wenn einer der folgenden Fälle vorliegt (vgl. Schritt 1 bis 4 von 3.2.):
(1) g enthält eine Programmeinheit, die von einer Anweisung von f aufgerufen wird (Unterprogrammaufruf)
(2) g ist ein X- oder O-Fragment von f
(3) g ist ein abgeleitetes Fragment und mit der Ausführung von f wird eine Anweisung von g ausgeführt
(4) g ist ein D-Fragment, das mit f benötigt wird.

Der gerichtete Graph (F,R) bildet die Grundlage für die Modellierung der Abhängigkeiten zwischen den Elementen von F durch "Fragmentsysteme" nach Definition 4.1 .
Man beachte, daß man R bereits mit der Herleitung der Fragmente nach 3.2. erhält!

Hinweis:
PRED(f) und SUCC(f) eines Fragmentes f bezeichnen in diesem Abschnitt die Vorgänger bzw. Nachfolger von f als Knoten des Graphen (F,R) (nicht mehr als Knoten des geordneten Baumes eines B-Programms von Abschnitt 3!)

DEFINITION 4.1:
$F \neq \emptyset$ sei eine endliche Menge, $R \subseteq F \times F$ eine Relation auf F; G sei der gerichtete Graph (F,R), E die Menge $\{f \mid f \in F, PRED(f)=\emptyset\}$. Weiter gebe es Abbildungen $X: F \rightarrow \P(F)$, $O: F \rightarrow \P(F)$ und $\rho: T \times F \rightarrow B$. [1]
(F,R,X,O,E,ρ) heißt ein F r a g m e n t s y s t e m , wenn G keine Zyklen enthält, X und O FG2 erfüllen und ρ für alle $t \in T$ FG1 und FG3 bis FG5 genügt.

FG1: Für jedes $e \in E$ gibt es ein $t \in T$ mit $\rho(t,e)=1$

FG2: Für jedes $f \in F$ gilt:

- $X^{-1}(f) \neq \emptyset$ oder $O^{-1}(f) \neq \emptyset \quad \Longrightarrow \quad |PRED(f)| = 1$
- $X(f) \subseteq SUCC(f)$, $O(f) \subseteq SUCC(f)$, $X(f) * O(f) = \emptyset$
- $X(f) = \emptyset$ oder $|X(f)| > 1$

FG3: Für $f \in F-E$ gilt:
$\rho(t,f)=1 \quad \Longrightarrow$ es existiert ein $g \in PRED(f)$ mit $\rho(t,g)=1$

FG4: Für $f \in F$ mit $X(f) \neq \emptyset$ gilt:
$\rho(t,f)=1 \quad \Longrightarrow$ es existiert ein $g \in X(f)$ mit $\rho(t,g)=1$

FG5: Für jedes $g \in PRED(f)$ mit $f \in F$ und $X^{-1}(f)=O^{-1}(f)=\emptyset$ gilt:
$\rho(t,g)=1 \quad \Longrightarrow \quad \rho(t,f)=1$

Bemerkung:
Da (F,R) keine Zyklen enthält und |F| endlich ist, gilt für jedes Fragmentsystem FG1':

FG1': $E \neq \emptyset$; jedes $f \in F$ ist von mindestens einem $e \in E$ erreichbar.

Bezeichnungen:

- (F,R,X,O,E) heißt der F r a g m e n t g r a p h des Fragmentsystems
- Für $f \in F$ werden die Elemente von X(f) die X - F r a g m e n t e von f, die von O(f) die O - F r a g m e n t e von f genannt. Die Elemente von E heißen die e n t r y - F r a g m e n t e des Fragmentsystems.
- ρ_f hat die gleiche Bedeutung wie in Abschnitt 3. Sofern nicht explizit anders vereinbart bezeichnet im folgenden ρ_i bzw. $\rho_{i,j}$ die Relevanz eines Fragmentes f_i bzw. $f_{i,j}$

Die Herleitung von Fragmenten nach dem Verfahren von Abschnitt 3.2 liefert Fragmentsysteme:

a) F ist die Menge der Fragmente, die Interpretation der Relation R ist

---

[1] Abweichend von der allgemein üblichen Notation wird für das Folgende vereinbart:
$X^{-1}(f) := \{ g \mid g \in F, f \in X(g) \}$ $\quad O^{-1}(f) := \{ g \mid g \in F, f \in O(g) \}$
X^{-1} und O^{-1} bezeichnen hier also nicht die Umkehrabbildungen von X bzw. O, die ja als Definitionsbereich jeweils eine Teilmenge von $\P(F)$ haben!

oben gegeben.
Die Forderung, daß G keine Zyklen enthält, impliziert nicht notwendigerweise, daß auf Programmsysteme mit rekursiven Programmeinheiten die folgenden Ergebnisse zu Fragmentsystemen nicht anwendbar sind:
Die Programmeinheit U_0 sei rekursiv, d.h. es gibt Programmeinheiten U_i, so daß U_i U_{i+1} aufruft mit $U_n=U_0$ für ein $n\geq 0$. Dies führt zunächst zu einem Graphen (F,R) mit einem Zyklus $<f_0, \ldots, f_n, f_0>$. Jede Kante (f_i,f_{i+1}) stellt die Aussage "f_i wird genau mit f_{i+1} benötigt" dar; es geht also keine Information verloren, wenn man die Kante (f_n,f_0) entfernt und damit den Zyklus auflöst.

b) entry-Fragmente stellen Programmteile dar, deren Ausführung vom Anwender beim Aufruf einer Operation angestoßen wird. Dies sind z.B. Hauptprogramme bei Programmsystemen , die als separate tasks ablaufen, oder Unterprogramme, falls das Programmsystem als Unterprogrammpaket zur Anwendersoftware hinzugebunden wird. Im letzteren Fall hat man in der Regel mehrere entry-Fragmente: $|E|>1$.
Für das Beispielsystem (Bild 3.4) ist $E=\{1\}$; man hätte sechs entry-Fragmente, wenn die Anwenderprogramme anstelle der Programmeinheit DBMS unmittelbar die Programmeinheiten der CASE-Anweisung aufrufen müßten.

c) FG1, FG1' stellen eine notwendige Voraussetzung dafür dar, daß es keinen überflüssigen Code gibt.

d) Die Abbildungen X bzw. O ordnen jedem Fragment seine X- bzw. O-Fragmente zu; für das Beispielsystem gilt (s. 3.2.2., Schritt 2):

```
                +-
                | { 1.1 , 1.2 , 1.3 ,                    +-
                |   1.4 , 1.5 , 1.6 } : f= 1             | { 2.1 } : f= 2
                | { 4.1 , 4.2 }       : f= 4             | { 5.3 } : f= 5
                | { 4.2.1 , 4.2.2 }   : f= 4.2           | { 8.3 } : f= 8
X(f) = <        { 5.1 , 5.2 }       : f= 5     O(f) = <
                | { 6.1 , 6.2 }       : f= 6             | { 9.1 } : f= 9
                | { 8.1 , 8.2 }       : f= 8             | { 10.3} : f= 10
                | {10.1 ,10.2 }       : f= 10            |   ∅     : sonst
                |       ∅             : sonst            +-
                +-
```

FG2 besagt (vgl. Schritt 2 von 3.2.), daß
- ein X- oder O-Fragment von genau einem Fragment referiert wird (in 3.2.2.: seinem Oberfragment)
- ein X-Fragment nicht zugleich auch ein O-Fragment sein kann und umgekehrt.
- ein Fragment entweder kein oder mindestens zwei X-Fragmente hat.

e) Ein Fragment f eines Fragmentsystems hat genau eine Menge von X-Fragmenten $X(f) \subseteq \P(F)$. Bei der Zerlegung eines Programms in Fragmente nach Abschnitt 3.2 kann man aber Fragmente mit mehr als einer Menge von X-Fragmenten erhalten, z.B. wenn eine Programmeinheit mehrere CASE-Konstrukte mit Anweisungen enthält, die initiierend für optionale Algorithmen sind.
Eine Möglichkeit, hieraus dennoch ein Fragmentsystem zu erhalten, besteht in der Definition von Unterfragmenten, die je eines dieser CASE-Anweisungen enthalten und die vom Oberfragment (anstelle der X-Fragmente) referiert werden. (Eine andere Möglichkeit ist die Modifikation des Programms in der Weise, daß Programmeinheiten eingeführt werden, die jeweils eines dieser CASE-Konstrukte enthalten

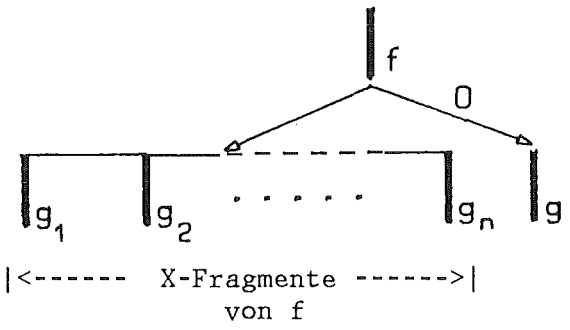
und von der ursprünglichen Programmeinheit zur Ausführung der CASE-Anweisungen aufgerufen werden.)

f) Die Abbildung ρ ist die von Definition 3.1.
FG3 besagt, daß, wenn ein Fragment für ein Teilsystem relevant ist, mindestens eines der Fragmente, die es referieren, ebenfalls für dieses Teilsystem benötigt wird; für Fragmente, die keine X- oder O-Fragmente sind, gilt auch die Umkehrung, dies ist die Eigenschaft FG5. X-Fragmente haben nach 3.2. die Eigenschaft FG4. (Wegen X(f) $\subseteq$ SUCC(f) stellt FG4 eine schwächere Forderung als FG5 dar!)

g) Die Fragmente, die man bei dem Spezialfall der Zerlegung eines CASE-Konstrukts nach 3.2.1.3 erhält, sind n i c h t Knoten eines Fragmentsystems: wie in 3.2.1.3. definiert, sind die Relevanzen solcher Fragmente durch Relevanzen von Knoten des Fragmentsystems festgelegt.
(Man beachte: solche Fragmente enthalten keinen Teilstring des vollständigen Programms!)

Bild 4.1 zeigt den Fragmentgraph des Beispielsystems von Abschnitt 3. Generell wird zur graphischen Darstellung von Fragmentgraphen vereinbart:

- Ein Fragment, also ein Knoten von (F,R) wird als ein breiter, senkrechter Strich versehen mit dem Bezeichner des Fragmentes gezeichnet (als Symbol für den Abschnitt des Programmtextes, den das Fragment umfaßt).
- eine gerichtete Kante (f,g) ε R wird durch eine Verbindungslinie von f nach g, mit einem Pfeil nach g, dargestellt.
  Der Übersichtlichkeit halber kann eine Kante (f,g) auch in zwei Teilen als eine Verbindungslinie vom Knoten f nach dem Bezeichner von g und eine Verbindunglinie vom Bezeichner von f nach dem Knoten g gezeichnet werden (in Bild 4.1 z.B. die Kante (13,22)).
- ist g ein O-Fragment von f, dann ist die Verbindungslinie mit "O" markiert, hat f n X-Fragmente, so wird (um graphisch auszudrücken, daß von der Menge der X-Fragmente mit f genau eines ausgeführt wird, vgl. Bild 1.5), e i n Pfeil von f zu einer Linie gezeichnet, die die X-Fragmente von f zusammenfaßt:



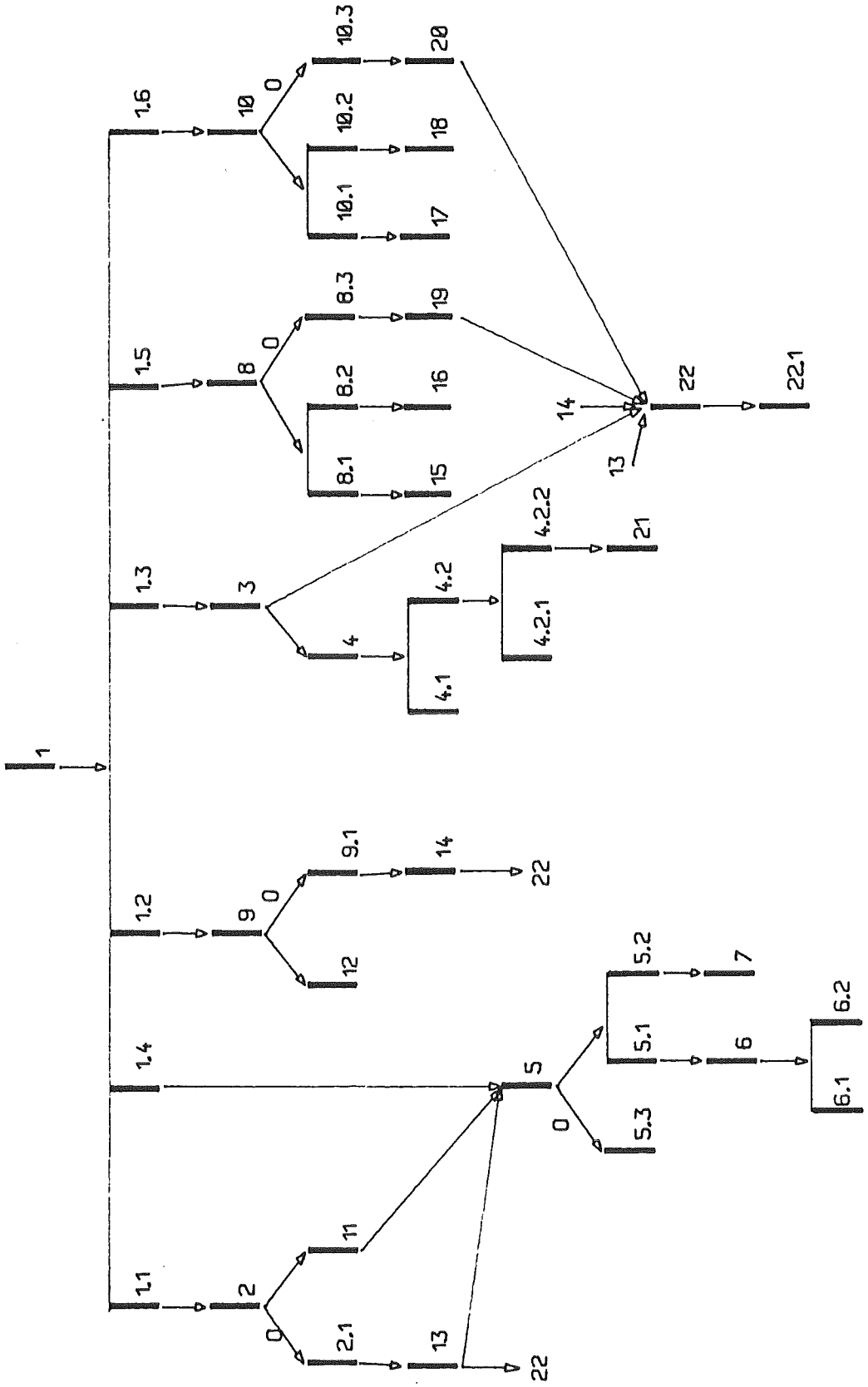


Bild 4.1: Der Fragmentgraph des Beispielsystems

4.2. Beziehungen zwischen den Relevanzen von Fragmenten

Die Axiome FG3 bis FG5 beschreiben die Abhängigkeiten zwischen den Relevanzen von Fragmenten von F. Es wird zunächst untersucht, unter welchen Voraussetzungen eine Relevanz durch andere Relevanzen festgelegt ist.

4.2.1. Beziehungen zwischen f und PRED(f)

f ε F habe die Relevanz ρ_f und n>0 Vorgänger $PRED(f)=\{f_i \mid 1\leq i\leq n\}$, ρ_i sei die Relevanz von f_i, $1\leq i\leq n$.

f ist insbesondere kein entry-Fragment, nach FG3 gilt somit stets die Implikation

(I1) $$\rho(t,f)=1 \Longrightarrow OR_{i=1}^{n}\rho(t,f_i)=1$$

und folglich auch

(I2) $$OR_{i=1}^{n}\rho(t,f_i)=0 \Longrightarrow \rho(t,f)=0$$

Ist f weiterhin auch kein O- oder X-Fragment, so folgt aus FG5

$$OR_{i=1}^{n}\rho(t,f_i)=1 \Longrightarrow \rho(t,f)=1$$

Für Fragmente f ε F-E, die keine X- oder O-Fragmente sind, erhält man so insgesamt die Gleichung:

(G1) $$\rho_f \equiv OR_{i=1}^{n}\ \rho_i$$

Bemerkung:
Ist f kein O- oder X-Fragment und PRED(f)={g}, so haben f und g die gleiche Relevanz: $\rho_f \equiv \rho_g$

DEFINITION 4.2:
Ein R 1 - P f a d von f nach g eines Fragmentgraphen (F,R,X,O,E) ist ein Pfad P des Graphen (F,R) von f nach g, so daß kein Knoten x ε P mit $x\neq f$ ein O- oder X-Fragment ist.

Beispiel: Ein R1-Pfad des Fragmentgraphen von Bild 4.1 ist z.B. die Folge
< 1.1 , 2 , 11 , 5 >

Später wird folgende allgemeinere Aussage benötigt:

SATZ 4.1:
$f \varepsilon F$ sei weder ein X- noch ein O-Fragment und $E'=\{e_i \mid 1\leq i\leq m\}$ die Menge aller entry-Fragmente, von denen es einen Pfad nach f gibt: für $1\leq i\leq m$ gebe es die p_i Pfade $P_{i,j}$, $1\leq j\leq p_i$, von $e_i \varepsilon E'$ nach f. Mit

$$g_{i,j} := \begin{cases} e_i & : P_{i,j} \text{ ist ein R1-Pfad} \\ x & : x \varepsilon P_{i,j},\ x \text{ ist ein O- oder X-Fragment und der Teilpfad von } P_{i,j} \text{ von x nach f ist ein R1-Pfad} \end{cases}$$

und $C := \{g_{i,j} \mid 1\leq j\leq p_i,\ 1\leq i\leq m\}$ gilt:

$$\rho_f \equiv OR_{g\varepsilon C}\ \rho_g$$

Beweis:
Die Behauptung ergibt sich unter Beachtung der Kommutativität der OR-Verknüpfung unmittelbar durch sukzessive Anwendung von Gleichung G1:
ρ_f ist zunächst die OR-Verknüpfung der Relevanzen der Vorgänger von f. Enthält der Relevanzausdruck für ρ_f Relevanzen von Fragmenten, die weder ein O- noch ein X-Fragment sind, so werden diese jeweils durch Relevanzausdrücke gemäß G1 ersetzt. Da |F| endlich und jeder Pfad in (F,R) azyklisch ist, gelangt man so in endlich vielen Schritten zu einem Relevanzausdruck für ρ_f, der Relevanzen nur von O-, X- oder entry-Fragmenten enthält.
Es sind dies genau die Relevanzen der Behauptung, da man bei diesem Vorgang a l l e Pfade, die zu f führen, von f aus durchläuft, bis ein O-, X- oder entry-Fragment erreicht wird.
□

Beispiel:

In Bild 4.1 gibt es drei Pfade von $E'=E=\{1\}$ nach 5:

$P_{1,1} = < 1 , 1.1 , 2 , 2.1 , 13 , 5 >$ $\quad g_{1,1} = 2.1$
$P_{1,2} = < 1 , 1.1 , 2 , 11 , 5 >$ $\quad g_{1,2} = 1.1$
$P_{1,3} = < 1 , 1.4 , 5 >$ $\quad g_{1,3} = 1.4$

und es gilt somit: $\rho_5 \equiv \rho_{2.1}\ OR\ \rho_{1.1}\ OR\ \rho_{1.4}$

Für Fragment 22 erhält man: $\rho_{22} \equiv \rho_{2.1}\ OR\ \rho_{1.3}\ OR\ \rho_{9.1}\ OR\ \rho_{8.3}\ OR\ \rho_{10.3}$

4.2.2. Beziehungen zwischen f und SUCC(f)

f ε F habe die Relevanz ρ_f und n>0 Nachfolger SUCC(f)={f_i|1≤i≤n}, ρ_i sei die Relevanz von f_i, 1≤i≤n.

Aus dieser Voraussetzung alleine, ohne weitere Annahmen vor allem bez. der Mächtigkeit der Mengen PRED(f_i), lassen sich keine Aussagen über die Relevanzen ableiten. Wegen X(f)*O(f)=∅ (Axiom FG2) kann man SUCC(f) als Vereinigung paarweise disjunkter, nicht notwendigerweise nichtleerer Mengen darstellen:

SUCC(f) := X(f) + O(f) + SUC1(f) + SUCM(f) mit

SUC1(f) := { x | x ε SUCC(f), |PRED(x)|=1, x ¬ε X(f), x ¬ε O(f) }
SUCM(f) := { x | x ε SUCC(f), |PRED(x)|>1 }

O.B.d.A. sei

X(f) := { f_i | 1≤i<nx} O(f) := { f_i | nx≤i<no}

SUC1(f) := { f_i | no≤i<n1} SUCM(f) := { f_i | n1≤i≤n }

mit 1≤nx≤no≤n1.

Beispiele:
SUCC(8) = { 8.1 , 8.2 } + { 8.3 } + ∅ + ∅ (also: nx=3, no=4, n1=4, n=3)
SUCC(2) = ∅ + { 2.1 } + { 11 } + ∅ (also: nx=1, no=2, n1=3, n=2)

Fragment 5 ist eines der Fragmente von Bild 4.1 mit mehr als einem Vorgänger: SUCM(13)=SUCM(11)=SUCM(1.4)={5}

DEFINITION 4.3:
Die Fragmente f ε F mit |PRED(f)| > 1 heißen "S-Fragmente".

4.2.2.1. f und seine X-Fragmente

Es sei X(f)≠∅, also nx>1. f_i ε X(f) ist kein entry-Fragment, aus FG2 folgt PRED(f_i)={f}, wegen FG3 gilt daher:

$\rho(t,f_i)=1$ für ein i mit 1≤i<nx ==> $\rho(t,f)=1$ oder:

$$OR_{i=1}^{nx-1}\ \rho(t,f_i)=1 \implies \rho(t,f)=1$$

Wegen FG4 gilt aber auch die Umkehrung:

$$\rho(t,f)=1 \implies OR_{i=1}^{nx-1}\ \rho(t,f_i)=1$$

und damit insgesamt

$$\rho_f \equiv OR_{i=1}^{nx-1}\ \rho_i$$

4.2.2.2. f und seine O-Fragmente

Es sei $O(f)\neq\emptyset$, also no>nx. $f_i \varepsilon O(f)$ ist kein entry-Fragment, aus FG2 folgt $PRED(f_i)=\{f\}$, wegen FG3 gilt daher:

$\rho(t,f_i)=1$ für ein i, $nx\leq i<no$ ==> $\rho(t,f)=1$ oder:

$$OR_{i=nx}^{no-1} \rho_i(t)=1 \implies \rho_f(t)=1$$

FG4 bzw. FG5 sind hier nicht anwendbar, die Umkehrung, und damit die Gleichheit gilt nicht mehr!

4.2.2.3. f und die Elemente von SUC1(f)

Es sei $SUC1(f)\neq\emptyset$, also n1>no. Wegen FG3 und FG5 haben f und die Fragmente von SUC1(f) die gleiche Relevanz (dies ergibt sich auch unmittelbar aus der Bemerkung von 4.2.1.), für $no\leq i<n1$ gilt also:

$$\rho_f \equiv \rho_i$$

4.2.2.4. f und seine S-Fragmente

Es sei $SUCM(f)\neq\emptyset$, d.h. $n1\leq n$. Für jedes Element f_i von SUCM(f), also für $n1\leq i\leq n$, gilt wegen FG5:

$$\rho_f(t)=1 \implies \rho_i(t)=1$$

Nach der Definition von SUCM(f) gilt die Umkehrung (und damit eine Gleichheit) nicht, FG3 reicht hierfür nicht aus.

Beispiel: Für Fragment 5 gilt also:
$\rho_{1.4}(t)=1$ ==> $\rho_5(t)=1$ $\rho_{13}(t)=1$ ==> $\rho_5(t)=1$ $\rho_{11}(t)=1$ ==> $\rho_5(t)=1$

4.2.3. Entry-Fragmente und f $\neg\varepsilon$ E

SATZ 4.2:
Ist $\rho_f(t)=1$ für f ε F-E, so gibt es mindestens ein e ε E und einen Pfad P von e nach f, so daß $\rho_x(t)=1$ für alle x ε P gilt, insbesondere also auch $\rho_e(t)=1$.

Beweis:
Wegen f ε F-E gibt es nach Implikation I1 von 4.2.1. (oder nach FG3) ein u ε PRED(f) mit $\rho_u(t)=1$; ist u kein entry-Fragment, dann gibt es aus den gleichen Gründen auch einen Vorgänger v von u mit $\rho_v(t)=1$, etc.:

Da |F| endlich ist und (F,R) keine Zyklen enthält, liefert die sukzessive Anwendung der Implikation I1 einen Pfad P von einem entry-Fragment nach f, so daß $\rho_x(t)=1$ für alle x ε P gilt.

□

SATZ 4.3:
Es sei $F' \subseteq F$, $F' \neq \emptyset$. Enthält jeder Pfad P von E nach $f \in F$ mindestens einen Knoten von F', so gilt:

$$OR_{g \in F'} \rho_g(t)=0 \quad ==> \quad \rho_f(t)=0$$

Beweis:
Für $f \in F'$ ist nichts zu zeigen, es sei also $f \in F-F'$, d.h. $f \neg\in E$.
Es sei $OR_{g \in F'} \rho_g(t)=0$. Die Annahme $\rho_f(t)=1$ impliziert nach Satz 4.2 einen Pfad P von einem entry-Fragment nach f, mit $\rho_x(t)=1$ für alle $x \in P$. Wegen $P^* F' \neq \emptyset$ bedeutet dies aber $OR_{f \in F'} \rho_f(t)=1$ im Widerspruch zur Voraussetzung.
□

KOROLLAR 4.1:
Es sei $E' \subseteq E$. Ist $f \in F$ von E', aber von keinem $e \in E-E'$ erreichbar, so gilt:

$$OR_{e \in E'} \rho_e(t)=0 \quad ==> \quad \rho_f(t)=0$$

Beweis:
Mit F':=E' erhält man dies unmittelbar aus Satz 4.3.
□

KOROLLAR 4.2:
Es sei $E' \subseteq E$. Ist $f \in F$ von E', aber von keinem $e \in E-E'$ erreichbar und existiert von jedem $e \in E'$ mindestens ein R1-Pfad nach f, dann gilt:

$$OR_{e \in E'} \rho_e \equiv \rho_f.$$

Beweis:
Nach Korollar 4.1 gilt:

$$OR_{e \in E'} \rho_e(t)=0 \quad ==> \quad \rho_f(t)=0$$

Nach Voraussetzung gibt es für jedes $e \in E'$ einen R1-Pfad nach f. f ist somit kein 0- oder X-Fragment (s. Definition 4.2), nach Satz 4.1 gibt es also ein $F' \subseteq F$ mit $E' \subseteq F'$ und $\rho_f(t) = OR_{x \in F'} \rho_x(t)$; es gilt damit:

$$OR_{e \in E'} \rho_e(t)=1 \quad ==> \quad OR_{x \in F'} \rho_x(t)=\rho_f(t)=1.$$

□

4.3. Charakteristische Fragmente

4.3.1. Definition

In 4.2. ist gezeigt worden, daß unter bestimmten Vorausetzungen die Relevanz eines Knotens eines Fragmentgraphen als OR-Verknüpfung der Relevanzen anderer Knoten dargestellt werden kann. Es liegt daher nahe, nach einer Teilmenge von Knoten zu suchen, so daß sich mit den Relevanzen dieser Fragmente die der übrigen Knoten des Fragmentgraphen darstellen lassen; zusätzlich sollte diese Menge möglichst klein sein. Ein formale Beschreibung dieser Eigenschaften gibt die folgende Definition:

DEFINITION 4.4:
Eine Menge CF $\subseteq$ F ist eine c h a r a k t e r i s t i s c h e M e n g e des Fragmentsystems (F,R,X,O,E,ρ), wenn sie die Eigenschaften CF1 und CF2 hat:

CF1: Für jedes f ε F gibt es eine Menge C(f) $\subseteq$ CF, C(f)$\neq\emptyset$, mit:

$$\rho_f \equiv OR_{g\varepsilon C(f)}\ \rho_g$$

CF2: Für f ε CF gibt es keine Menge C $\subseteq$ CF-{f}, C$\neq\emptyset$, mit:

$$\rho_f \equiv OR_{g\varepsilon C}\ \rho_g$$

Bezeichnungen:
- Die Elemente einer charakteristischen Menge heißen c h a r a k t e r i s t i s c h e F r a g m e n t e des Fragmentsystems
- Für f ε F wird eine Menge C(f) gemäß CF1 eine C F - D a r s t e l l u n g von f und der hiermit gegebene Ausdruck $OR_{g\varepsilon C(f)}\ \rho_g$ ein C F - A u s d r u c k von ρ_f genannt.

Eine charakteristische Menge CF eines Fragmentsystems hat also die Eigenschaft, daß zur Spezifikation der relevanten Fragmente eines Teilsystems t höchstens zu jedem Fragment f ε CF der Relevanzwert $\rho_f(t)$ anzugeben ist (Eigenschaft CF1); CF ist weiterhin minimal in dem Sinne, daß es Teilsysteme t gibt, so daß zu j e d e m Element f von CF $\rho_f(t)$ anzugeben ist, die Festlegung der Relevanzwerte zu einer Teilmenge von CF also nicht genügt.

4.3.2. Ein Verfahren zur Herleitung von charakteristischen Mengen

4.3.2.1. R1-Mengen

DEFINITION 4.5:
- Die R 1 - M e n g e von $f \varepsilon F$, notiert R1(f), ist die Menge
  $\{f\} + \{ g \mid$ es gibt einen R1-Pfad P von f nach g,
  P-{f} enthält kein S-Fragment$\}$
- f heißt das W u r z e l f r a g m e n t von R=R1(f), notiert: ROOT(R)

Bemerkungen:
a) Es gilt offensichtlich (vgl. 4.2.1., 4.2.2.3.): $g \varepsilon R1(f) \Longrightarrow \rho_g \equiv \rho_f$
b) Für ein X-, O- oder S-Fragment f gibt es kein $x \varepsilon F$ mit $x \neq f$ und $f \varepsilon R1(x)$.
c) Die Teilgraphen des Fragmentgraphen mit den Knoten von R1-Mengen sind Bäume (jeder Knoten von R1(f)-{f} hat genau einen Vorgänger), es ist daher sinnvoll, vom "Wurzelfragment" einer R1-Menge zu sprechen. Hieraus ergibt sich auch unmittelbar:
   Jeder Pfad von $x \varepsilon F$-R1(f) nach $g \varepsilon R1(f)$ enthält f.
d) Mit Verfahren 4.1 wird ROOT(M) auch für bestimmte Nicht-R1-Mengen $M \subseteq F$ definiert.

Für eine Klasse von R1-Mengen läßt sich eine Maximalitätseigenschaft nachweisen:

SATZ 4.4:
Ist f_1 ein X-, O- oder S-Fragment oder $f_1 \varepsilon E$, so gilt für $f_2 \varepsilon F$:

$$R1(f_2) \subseteq R1(f_1) \quad \text{oder} \quad R1(f_2) * R1(f_1) = \emptyset$$

Beweis:
Fall 1: $f_2 \varepsilon R1(f_1)$
Aus Definition 4.5 folgt unmittelbar: $R1(f_2) \subseteq R1(f_1)$.
Fall 2: $f_2 \neg\varepsilon R1(f_1)$
Wegen $f_2 \neq f_1$ gilt $R1(f_2) \neg\subseteq R1(f_1)$. Nimmt man $R1(f_2) * R1(f_1) \neq \emptyset$ an, so folgt, daß es ein $x \varepsilon F$ gibt mit $x \varepsilon R1(f_1)$ und $x \varepsilon R1(f_2)$. Insbesondere existiert dann ein R1-Pfad von f_1 nach x, der nach Bemerkung c) zu Definition 4.5 auch f_2 enthält; dies steht aber im Widerspruch zu $f_2 \neg\varepsilon R1(f_1)$. Es muß also $R1(f_2) * R1(f_1) = \emptyset$ gelten.
□

4.3.2.2. Das Verfahren

Verfahren 4.1 liefert in Schritt 3 zu einem Fragmentsystem (F,R,X,O,E,ρ) eine Menge $CF \subseteq F$, von der gezeigt werden wird, daß sie CF1 und CF2 erfüllt: F wird zunächst so in Teilmengen, sogenannte "Ω-Mengen", zerlegt, daß die Fragmente jeder dieser Ω-Mengen jeweils in der Relevanz übereinstimmen. Eine charakteristische Menge kann maximal ein Element je Ω-Menge enthalten: zur Konstruktion von CF wird daher ein gerichteter Graph angegeben, der als Knoten aus jeder Ω-Menge genau ein Fragment enthält, und CF als eine Teilmenge der Knoten dieses Graphen beschrieben.

VERFAHREN 4.1: Ermittlung einer charakteristischen Menge
Input : Fragmentsystem (F,R,X,O,E,ρ)
Output: eine charakteristische Menge CF
Algorithmus:

SCHRITT 1:

$$i = 0$$
$$\Omega^{(0)} = \{ R1(f) \mid f \in F,\ f \text{ ist ein X-, O- oder S-Fragment oder } f \in E \}$$

SCHRITT 2:

WHILE (es gibt $\omega_1, \omega_2 \in \Omega^{(i)}$, so daß $f=ROOT(\omega_2)$ ein S-Fragment und $PRED(f) \subseteq \omega_1$ ist)
DO

$$i = i + 1$$
$$\omega^{(i)} = \omega_1 + \omega_2$$
$$ROOT(\omega^{(i)}) = ROOT(\omega_1)$$
$$\Omega^{(i)} = \Omega^{(i-1)} + \{ \omega^{(i)} \} - \{ \omega_1, \omega_2 \}$$

END

SCHRITT 3:

$$\Omega = \Omega^{(i)}$$

$G\Omega$ sei der gerichtete Graph $(F\Omega, R\Omega)$ mit
$F\Omega = \{ f \mid f \in F,$ es gibt ein $\omega \in \Omega$ mit $ROOT(\omega)=f \}$
$R\Omega = \{ (f,g) \mid f,g \in F\Omega,\ PRED(g) * \omega \neq \emptyset$ für $\omega \in \Omega$ mit $ROOT(\omega)=f \}$

Die Abbildung $X\Omega: F\Omega \rightarrow \P(F\Omega)$ sei wie folgt definiert:

$X\Omega(f) = \{ g \mid g \in F\Omega,\ X^{-1}(g) \neq \emptyset,$
jeder Pfad in $G\Omega$ von E nach g enthält f,
$(f,g) \in R\Omega$ oder es gibt in $G\Omega$ einen Pfad P von f nach g, so daß $P-\{f,g\}$ kein X- oder O-Fragment enthält$\}$

$$CF = \{ f \mid f \in F\Omega,\ |PRED(f)| \leq 1,\ X\Omega(f)=\emptyset \}$$

Hinweis:
Für den allgemeinen Fall, daß in Schritt 2 mehrere Paare (ω_1,ω_2) die Bedingung zur Konstruktion von $\omega^{(i)}$ erfüllen, ist durch Verfahren 4.1 nicht festgelegt, mit welchem Paar $\omega^{(i)}$ zu bilden ist. In 4.3.4., Satz 4.11, wird gezeigt, daß diese Reihenfolge insofern unwesentlich ist, als Schritt 2 in jedem Fall die gleiche Menge Ω liefert.

Bezeichnung:
Die Elemente des Mengensystems Ω werden Ω - M e n g e n genannt.

Bemerkung:
Aus der Definition von $G\Omega$ ergibt sich unmittelbar:
a) $|F\Omega| = |\Omega|$, $E \subseteq F\Omega$, $|\omega * F\Omega|=1$ für $\omega \in \Omega$
b) Es sei $\omega_1,\omega_2 \in \Omega$, $f_1=ROOT(\omega_1)$, $f_2=ROOT(\omega_2)$ und P ein Pfad von f_1 nach f_2 in G.
   Dann ist $P\Omega := P*F\Omega$ ein Pfad von f_1 nach f_2 in $G\Omega$, $\{f_1,f_2\} \subseteq P\Omega$
c) $(f_1,f_2) \in R\Omega$ ==> es gibt einen Pfad P in G von f_1 nach f_2, $P-\{f_1,f_2\}$ enthält kein X- oder O-Fragment.
   Allgemeiner gilt: ist $P\Omega$ ein Pfad in $G\Omega$ von f_1 nach f_2, dann gibt es einen Pfad P in G von f_1 nach f_2 und $P\Omega \subseteq P$

Beispiel:
Verfahren 4.1 ergibt für den Fragmentgraphen von Bild 4.1 in Schritt 1 das Mengensystem $\Omega^{(0)}$ von Bild 4.2; die Bedingung von Schritt 2 ist hier für kein Paar von Elementen aus $\Omega^{(0)}$ erfüllt, somit ist $\Omega=\Omega^{(0)}$. Für die Menge $F\Omega$ des Graphen $G\Omega$ von Schritt 3 erhält man

```
FΩ = {  1   , 1.1 , 1.4 , 1.2 , 1.3 , 1.5 , 1.6 ,
       2.1 , 9.1 , 4.1 , 4.2  , 4.2.1 , 4.2.2 ,
       8.1 , 8.2 , 8.3 , 10.1 ,  10.2 , 10.3  ,
        5  , 5.1 , 5.2 , 5.3  ,  6.1  , 6.2   , 22 }
```

Bild 4.3 stellt $(F\Omega,R\Omega)$ dar.

Die Abbildung $X\Omega$:

```
          +-
          |  { 1.1 , 1.2 , 1.4 , 1.3 , 1.5 , 1.6 } : f=1
          |  { 4.1 , 4.2 }                         : f=1.3
          |  { 8.1 , 8.2 }                         : f=1.5
          |  {10.1 ,10.2 }                         : f=1.6
XΩ(f) = <
          |  { 4.2.1 , 4.2.2 }                     : f=4.2
          |  { 5.1 , 5.2 }                         : f=5
          |  { 6.1 , 6.2 }                         : f=5.1
          |     ∅                                  : sonst
          +-
```

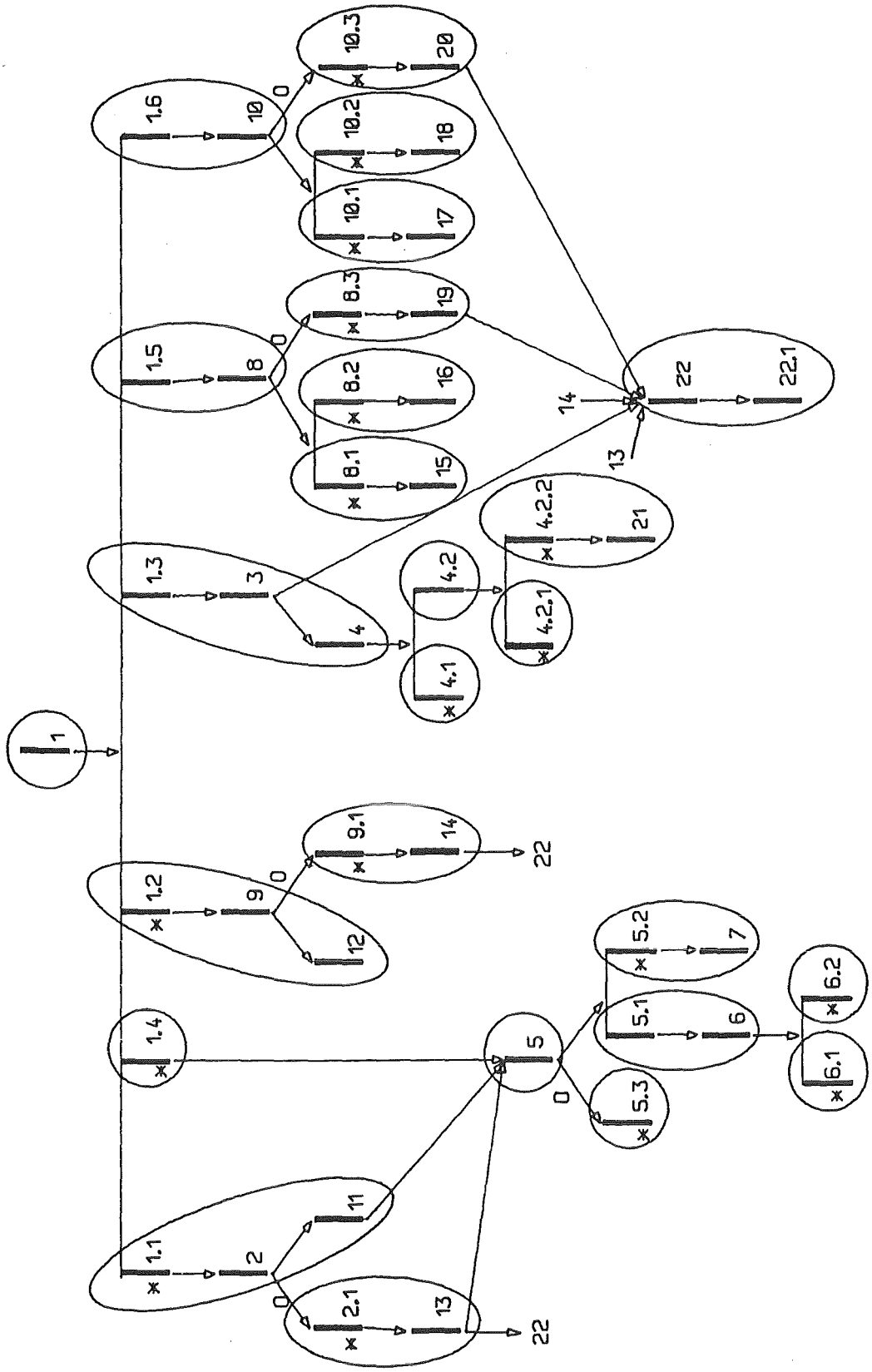


Bild 4.2: Die Ω-Mengen und die Menge CF des Beispielsystems

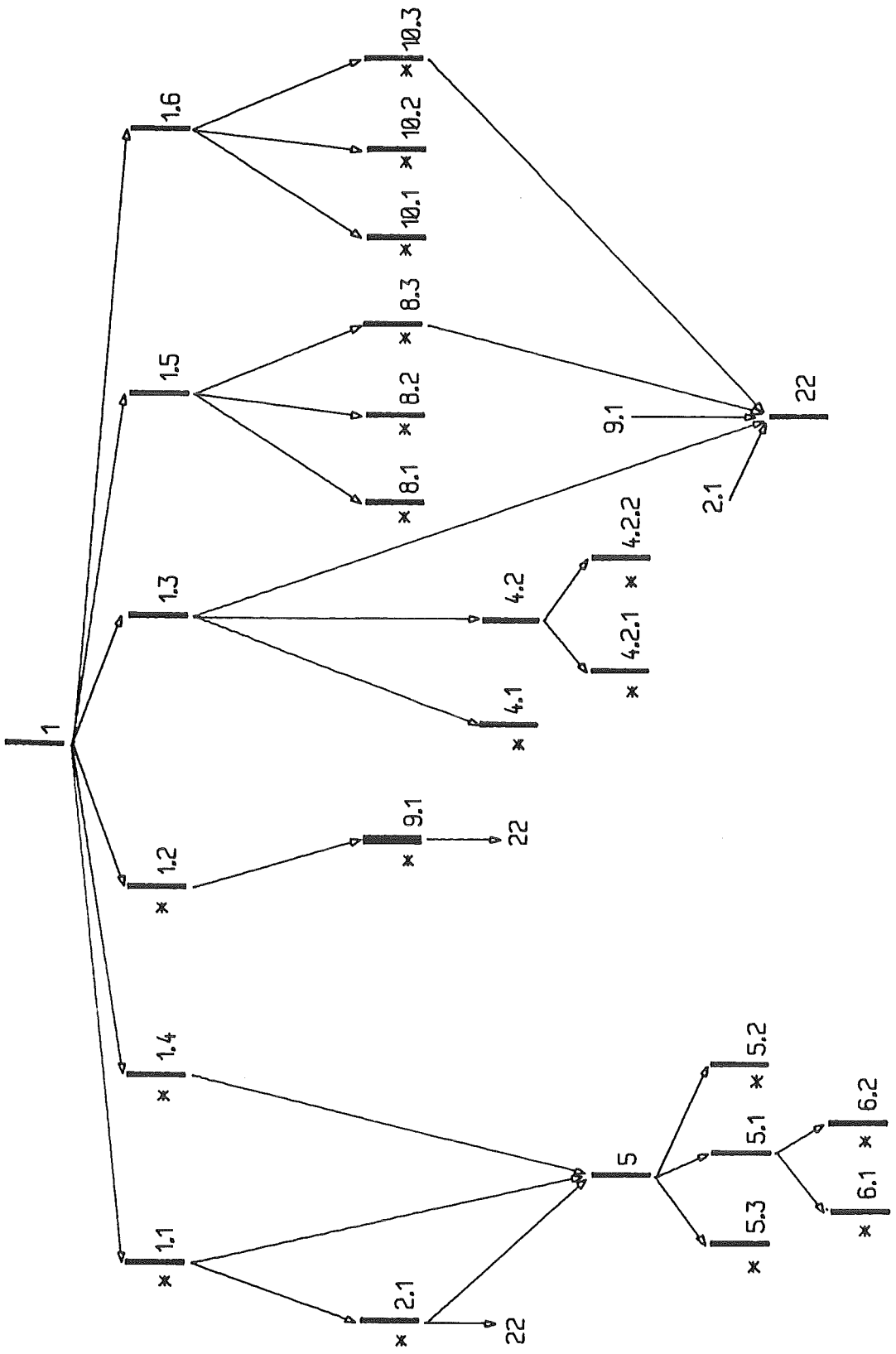


Bild 4.3: Der Graph GΩ und die Menge CF des Beispielsystems

Beachtet man noch, daß die Fragmente 5 und 22 S-Fragmente sind, so erhält man mit dem Verfahren 4.1 für das Beispielsystem:

$$CF = F\Omega - \{ 5 , 22 \} - \{ 1 , 1.3 , 1.5 , 1.6 , 4.2 , 5.1 \}$$

Diese Fragmente sind in Bild 4.2 und 4.3 mit "*" markiert.

Es folgt eine Charakterisierung der Menge CF als eine Teilmenge der Knoten F des Graphen G:

SATZ 4.5:
Die Menge CF des Verfahrens 4.1 ist die Menge
{ f | f ε F, f ist ein X-, O- oder entry-Fragment,
es gibt kein X-Fragment g, so daß gilt:
(*) jeder Pfad in G von E nach g enthält f und
(**) (f ε $X^{-1}(g)$ oder es gibt in G einen R1-Pfad von f nach x ε $X^{-1}(g)$) }
Beweis:
Nach Schritt 3 von Verfahren 4.1 enthält CF alle Fragmente f ε F, die X-, O- oder entry-Fragmente sind und für die es jeweils kein X-Fragment g ε F gibt, das (1) u n d (2) erfüllt:
(1) Jeder Pfad in GΩ von E nach g enthält f
(2) (f,g) ε RΩ oder es gibt in GΩ einen Pfad P von f nach g, so daß P-{f,g} kein X- oder O-Fragment enthält.

Nach den Bemerkungen b) und c) zu Verfahren 4.1 ist (1) äquivalent zu (*).

(2) impliziert (**):

- (f,g) ε RΩ ==> es gibt in G einen R1-Pfad von f nach x ε $X^{-1}(g)$ oder g ε X(f)
- Es gibt in GΩ einen Pfad P von f nach g, so daß P-{f,g} kein X- oder O-Fragment enthält ==> es gibt in G einen R1-Pfad von f nach x ε $X^{-1}(g)$.

CF von Verfahren 4.1 ist damit in der Menge der Behauptung enthalten. Es gilt aber auch die Umkehrung, d.h. (**) impliziert (2):

- f ε $X^{-1}(g)$ ==> (f,g) ε RΩ ==> (2)
- es gibt in G einen R1-Pfad von f nach x ε $X^{-1}(g)$ ==> (2)

□

Wegen $X^{-1}(x)$=PRED(x) für X-Fragmente x ergibt sich hieraus unmittelbar

KOROLLAR 4.3:

CF = { f | f ε F, f ist ein X-, O- oder entry-Fragment,
es gibt kein g ε F mit X(g)≠∅, so daß gilt:
jeder Pfad in G von E nach g enthält f und
(f=g oder es gibt in G einen R1-Pfad von f nach g) }

Es ist zu zeigen, daß die Menge CF des Verfahrens 4.1 eine charakteristische Menge ist, d.h. daß CF die Eigenschaften CF1 und CF2 hat:

- In Abschnitt 4.3.3 wird die Eigenschaft CF2, also die Minimalität von CF nachgewiesen. Hierzu wird konstruktiv gezeigt, daß es zu jedem f ε CF zwei Teilsysteme gibt, so daß f als einziges Element von CF für diese beiden Teilsysteme verschiedene Relevanzwerte hat.
- Zum Nachweis von CF1 wird in Abschnitt 4.3.4 zunächst gezeigt, daß die Elemente je einer Ω-Menge in der Relevanz übereinstimmen und Ω eine disjunkte Zerlegung von F ist. In 4.3.5. kann man sich daher darauf beschränken anzugeben, wie für die Wurzelfragmente der Ω-Mengen, d.h. die Elemente von FΩ, eine CF-Darstellung herzuleiten ist.

Vereinbarung:
Sofern nicht explizit anders angegeben, ist im folgenden ein "Pfad von f nach g" stets ein Pfad im Graphen G=(F,R) des Fragmentsystems, und nicht in GΩ!

4.3.3. Die Minimalität von CF

CF hat die Eigenschaft CF2 genau dann, wenn für jedes f ε CF gezeigt ist:

$$\rho_f \neg\equiv OR_{g\varepsilon C}\ \rho_g \text{ für jede Teilmenge } C \subseteq CF-\{f\},\ C\neq\emptyset$$

Gleichbedeutend hiermit ist:

(CF2') es gibt für jedes $C \subseteq CF-\{f\}$ ein $t_C \varepsilon T$ mit

$$\rho(t_C,f) \neq OR_{g\varepsilon C}\ \rho(t_C,g)$$

Um CF2', d.h. die Existenz eines t_C zu beweisen, wird unten gezeigt, daß man zu jedem f ε CF zwei Teilsysteme $t_1,t_2 \varepsilon T$ angeben kann mit:

$$\rho(t_1,f) \neq \rho(t_2,f)$$
$$\rho(t_1,g) = \rho(t_2,g) \quad \text{für } g \varepsilon CF-\{f\}$$

Hat man nämlich für $C \subseteq CF-\{f\}$, $C\neq\emptyset$ zwei solche Teilsysteme und setzt man

$$t_C := \begin{cases} t_1 : & \rho(t_1,f) \neq OR_{g\varepsilon C}\ \rho(t_1,g) \\ t_2 : & \rho(t_1,f) = OR_{g\varepsilon C}\ \rho(t_1,g) \end{cases}$$

so gilt $\rho(t_C,f) \neq OR_{g\varepsilon C}\ \rho(t_C,g)$:

- $t_C=t_1 \Longrightarrow \rho(t_C,f) = \rho(t_1,f) \neq OR_{g\varepsilon C}\ \rho(t_1,g) = OR_{g\varepsilon C}\ \rho(t_C,g)$
- $t_C=t_2 \Longrightarrow \rho(t_C,f) = \rho(t_2,f) \neq \rho(t_1,f) = OR_{g\varepsilon C}\ \rho(t_1,g)$
  $= OR_{g\varepsilon C}\ \rho(t_2,g) = OR_{g\varepsilon C}\ \rho(t_C,g)$

Bemerkenswert ist, daß zum Nachweis von CF2' für jedes f zwei Teilsysteme ausreichen, die zudem ausschließlich von f abhängen und nicht der Menge C: C bestimmt nur, welches von beiden t_C ist! Man vergleiche hierzu das Beispiel am Ende dieses Abschnitts.

Zur Konstruktion solcher Teilsysteme t_1, t_2 wird benötigt:

SATZ 4.6:
Es sei $D:=\{f_i \mid 1\leq i\leq n\} \subseteq F$ eine nichtleere Menge von X-, O- oder entry-Fragmenten, für jedes $g \in F$ mit $X(g)\neq\emptyset$ sei $X(g) \not\subseteq D$.
Ist F' die Menge { g | jeder Pfad von E nach $g \in F$ enthält mindestens ein Element von D }, dann sind die Elemente von F-F' die Fragmente eines Teilsystems $t \in T$, d.h. t mit

$$\rho(t,g) := \begin{cases} 0 : & g \in F' \\ 1 : & g \in F-F' \end{cases}$$

genügt FG3, FG4 und FG5.
Beweis:
a) Es sei $f \in F-E$.

$\rho(t,f)=1 \Longrightarrow f \in F-F'$
$\Longrightarrow$ es existiert ein Pfad P von E nach f mit $P*D=\emptyset$
$PRED(f)\neq\emptyset \Longrightarrow$ es gibt ein $g \in PRED(f)$ mit $g \in P$
$\Longrightarrow$ P-{f} ist ein Pfad von E nach $g \in PRED(f)$ mit $(P-\{f\})*D=\emptyset$, also $g \in F-F'$
$\Longrightarrow$ es existiert ein $g \in PRED(f)$ mit $\rho(t,g)=1$.
Es gilt somit FG3.

b) Es sei $f \in F$ mit $X(f)\neq\emptyset$.
$\rho(t,f)=1 \Longrightarrow f \in F-F'$
$\Longrightarrow$ es existiert ein Pfad P von E nach f mit $P*D=\emptyset$
Wegen $X(f)-D\neq\emptyset$ und $PRED(g)=\{f\}$ für $g \in X(f)$ folgt:
es gibt ein $g \in X(f)$, so daß $P+\{g\}$ ein Pfad von E nach g ist mit $(P+\{g\})*D=\emptyset$, also $g \in F-F'$
$\Longrightarrow$ es existiert ein $g \in X(f)$ mit $\rho(t,g)=1$
Es gilt somit FG4.

c) Es sei $f \in F$, $X^{-1}(f)=O^{-1}(f)=\emptyset$, $g \in PRED(f)$:
$\rho(t,g)=1 \Longrightarrow g \in F-F'$
$\Longrightarrow$ es existiert ein Pfad P von E nach g mit $P*D=\emptyset$
nach Voraussetzung gilt $f \not\in D$, d.h. $P+\{f\}$ ist ein Pfad von E nach f mit $(P+\{f\})*D=\emptyset$, also $f \in F-F'$
$\Longrightarrow \rho(t,f)=1$.
Es gilt somit FG5.
□

Es sei f ε CF; die zu f gesuchten Teilsysteme t_1 und t_2 sind gegeben mit den Mengen F-F_1 bzw. F-F_2, wobei

F_1 := { g | jeder Pfad von E nach g ε F enthält f }

FO := { g | g ε F_1, g≠f, g ist O-Fragment },

F_2 := { g | jeder Pfad von E nach g ε F enthält mindestens ein Element von FO }, also:

$$\rho(t_1,g) := \begin{cases} 0 : g \in F_1 \\ 1 : g \in F-F_1 \end{cases} \qquad \rho(t_2,g) := \begin{cases} 0 : g \in F_2 \\ 1 : g \in F-F_2 \end{cases}$$

- t_1 ist ein Teilsystem nach Satz 4.6 (mit D = {f})
- Ist FO≠∅, so ist wieder nach Satz 4.6 t_2 ε T; sonst ist F_2=∅ und damit $\rho(t_2,f)=1$ für alle f ε F, d.h. t_2 ε T

t_1 und t_2 sind also Teilsysteme. t_1 und t_2 haben die oben geforderten Eigenschaften:

Aus der Definition von FO folgt $F_2 \subseteq F_1$ mit f ¬ε F_2 und f ε F_1 (F_2 ist also eine echte Teilmenge von F_1!).

Es gilt daher für g ε F: $\rho(t_1,g)=1 \implies \rho(t_2,g)=1$, insbesondere:

g ε CF-{f}, $\rho(t_1,g)=1 \implies \rho(t_2,g)=1$

Es gilt aber auch:

g ε CF-{f}, $\rho(t_1,g)=0 \implies \rho(t_2,g)=0$

Beweis:
Aus $\rho(t_1,g)=0$ folgt g ε F_1. Wegen g≠f ist g kein entry-Fragment.

- Ist g ein O-Fragment, so ist g ε FO wegen g≠f und daher $\rho(t_2,g)=0$ nach Definition von F_2 (FO ⊆ F_2!).
- Ist g ein X-Fragment, so muß jeder Pfad von f nach g mindestens ein O-Fragment enthalten (auch in dieser Situation ist FO≠∅!):
  Annahme: es gibt einen Pfad P' von f nach g, so daß P'-{f} kein O-Fragment enthält, aber mindestens ein X-Fragment (nämlich g!).
  P' sei die Folge k_i, 1≤i≤j, mit k_1=f und k_j=g, m der kleinste Index, so daß $X(k_m)\neq\emptyset$, d.h. k_{m+1} ein X-Fragment von k_m ist. Nach Annahme ist m<j (k_j=g ist ein X-Fragment!) und jeder Pfad von E nach k_{m+1} enthält f.
  Ist m>1, so gibt es einen R1-Pfad von f nach k_m ε $X^{-1}(k_{m+1})$, sonst ist f ε $X^{-1}(k_{m+1})$; nach Satz 4.5 steht dies aber im Widerspruch zu f ε CF, die Annahme ist also falsch.

Da jeder Pfad von E nach g auch f enthält, folgt aus dem soeben gezeigten: jeder Pfad von E nach g enthält mindestens ein Element von FO, d.h. $g \varepsilon F_2$, also $\rho(t_2,g)=0$.

□

Es gilt also

$$\rho(t_1,g) = \rho(t_2,g) \quad \text{für } g \varepsilon CF-\{f\}$$

und $$0=\rho(t_1,f) \neq \rho(t_2,f)=1 \quad (\text{wegen } f \varepsilon F_1,\ f \neg\varepsilon F_2)$$

d.h. t_1 und t_2 sind die für f gesuchten Teilsysteme.

Für alle f ε CF ist damit gezeigt: Es gibt Teilsysteme t_1, t_2, so daß f als einziges Element von CF zu t_1 und t_2 verschiedene Relevanzwerte hat; d.h. die Relevanzwerte der Elemente von CF-{f} reichen zur Beschreibung aller möglichen Teilsysteme n i c h t aus! Dies ist die Minimalitätseigenschaft CF2 bzw. CF2'.

Beispiel:
Für das Fragment f=1.1 des Beispielsystems erhält man

F_1={ 1.1 , 2 , 11 , 2.1 , 13 }, FO={2.1}, F_2={ 2.1 , 13 },

t_1 wird somit gebildet von F-{1.1,2,11,2.1,13}, t_2 von F-{2.1,13}.
Für C={2.1} ist $t_C=t_2$, für alle anderen nichtleeren Teilmengen C von CF-{1.1} (auch die mit 2.1 ε C) ist $t_C=t_1$.

4.3.4. Eigenschaften von Ω-Mengen

SATZ 4.7:

Jedes f ε F ist Element genau einer R1-Menge von $\Omega^{(0)}$

Beweis:

1) $\omega_1, \omega_2 \varepsilon \Omega^{(0)}$ sind R1-Mengen, ihre Wurzelfragmente sind X-, O-, S- oder entry-Fragmente. Nach Satz 4.4 gilt

$$\omega_2 \subseteq \omega_1 \quad \text{oder} \quad \omega_2 * \omega_1 = \emptyset$$

und

$$\omega_1 \subseteq \omega_2 \quad \text{oder} \quad \omega_1 * \omega_2 = \emptyset$$

==> $\omega_1 = \omega_2$ oder $\omega_1 * \omega_2 = \emptyset$, d.h. ein f ε F ist in höchstens einer R1-Menge von $\Omega^{(0)}$ enthalten.

2) Annahme: f ε F ist in keiner R1-Menge enthalten.

f ist damit insbesondere kein entry- oder S-Fragment, d.h. es gilt |PRED(f)|=1. f ist weiterhin auch kein X- oder O-Fragment: damit kann der Vorgänger f' von f ebenfalls kein Element einer R1-Menge sein, da in dieser dann auch f enthalten wäre. Diese Überlegungen gelten auch für f', also auch der Vorgänger von f' ist in keiner R1-Menge enthalten, etc. Wegen der Endlichkeit von F impliziert dies die Existenz eines R1-Pfades von einem e ε E nach f, dessen Knoten jeweils höchstens einen Vorgänger haben, aber in keiner R1-Menge enthalten sind. Insbesondere wäre also e nicht Element einer R1-Menge, was im Widerspruch zu Schritt 1 von Verfahren 4.1 steht.

□

Es folgen Aussagen, die sich aus Eigenschaften der Mengensysteme $\Omega^{(i)}$ ergeben, sie werden durch Induktion über i bewiesen:

KOROLLAR 4.4:

a) Das Mengensystem Ω ist eine disjunkte Zerlegung von F

b) Jedes ω ε Ω ist die Vereinigung von R1-Mengen von $\Omega^{(0)}$

Beweis:

Zu a):

Nach Satz 4.7 ist $\Omega^{(0)}$ eine disjunkte Zerlegung von F.

$\Omega^{(i+1)}$ erhält man gemäß Schritt 2 aus $\Omega^{(i)}$, indem zwei Elemente von $\Omega^{(i)}$ durch ihre Vereinigung ersetzt werden. Ist daher $\Omega^{(i)}$ eine disjunkte Zerlegung von F, so gilt dies auch für $\Omega^{(i+1)}$.

Jedes $\Omega^{(i)}$, $i \geq 0$, ist also eine disjunkte Zerlegung von F, wegen der Endlichkeit von F gilt dies auch für Ω.

Zu b):

Die Behauptung von b) gilt trivialerweise für die Elemente von $\Omega^{(0)}$.

Analog zu a) gilt: ist jedes Element von $\Omega^{(i)}$ die Vereinigung von Elementen aus $\Omega^{(i)}$, so ist dies auch für die Elemente von $\Omega^{(i+1)}$ richtig. Damit ist b) gezeigt.

□

SATZ 4.8:

$$f,g \in \omega,\ \omega \in \Omega^{(i)},\ i\geq 0 \ \Longrightarrow\ \rho_f \equiv \rho_g$$

Beweis:
Nach Definition 4.5, Bemerkung a), gilt die Behauptung für i=0.
Induktionsannahme: Die Behauptung ist für die Elemente von $\Omega^{(i)}$ gezeigt.
Induktionsschluß: es sei $\omega \in \Omega^{(i+1)}$.
Für $\omega \in \Omega^{(i)} * \Omega^{(i+1)}$ gilt die Behauptung nach Induktionsannahme.
Für $\omega \in \Omega^{(i+1)} - \Omega^{(i)}$ folgt:
$\omega=\omega_1+\omega_2$, wobei $\omega_1,\omega_2 \in \Omega^{(i)}$ und $PRED(ROOT(\omega_2)) \subseteq \omega_1$.
Mit $u:=ROOT(\omega_1)$, $v:=ROOT(\omega_2)$ gilt nach Induktionsannahme

$$\rho_u \equiv \rho_x \text{ für } x \in \omega_1, \qquad \rho_v \equiv \rho_x \text{ für } x \in \omega_2.$$

Mit G1 von 4.2.1. (v ist ein S-Fragment!) folgt aus $PRED(v) \subseteq \omega_1$:

$$\rho_v \equiv OR_{x \in PRED(v)}\ \rho_x \equiv \rho_u\ ,$$

also $\rho_x \equiv \rho_u \equiv \rho_v$ für $x \in \omega_1+\omega_2$, womit die Behauptung auch für die Elemente von $\Omega^{(i+1)}$ gezeigt ist. Sie gilt also für alle $i\geq 0$.
□

SATZ 4.9:
Es sei $\omega \in \Omega^{(i)}$, $i\geq 0$. Jeder Pfad von $x \in F-\omega$ nach $g \in \omega$ enthält $ROOT(\omega)$.
Beweis:
Nach Definition 4.5, Bemerkung c), gilt die Behauptung für i=0.
Induktionsannahme: Die Behauptung ist für die Elemente von $\Omega^{(i)}$ gezeigt.
Induktionsschluß: es sei $\omega \in \Omega^{(i+1)}$.
Für $\omega \in \Omega^{(i)} * \Omega^{(i+1)}$ gilt die Behauptung nach Induktionsannahme.
Für $\omega \in \Omega^{(i+1)} - \Omega^{(i)}$ folgt:
$\omega=\omega_1+\omega_2$, wobei $\omega_1,\omega_2 \in \Omega^{(i)}$ und $PRED(ROOT(\omega_2)) \subseteq \omega_1$.
Mit $u:=ROOT(\omega_1)$, $v:=ROOT(\omega_2)$ gilt nach Induktionsannahme:

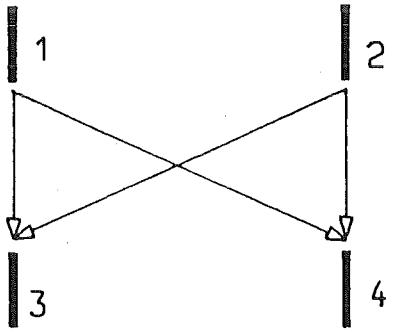
- $g \in \omega_1 \Longrightarrow$ jeder Pfad von $x \in F-(\omega_1+\omega_2)$ nach g enthält u
- $g \in \omega_2 \Longrightarrow$ jeder Pfad von $x \in F-(\omega_1+\omega_2)$ nach g enthält v
  $PRED(v) \subseteq \omega_1 \Longrightarrow$ jeder Pfad von $x \in F-(\omega_1+\omega_2)$ nach $g \in \omega_2$ enthält ein Element von ω_1 und damit auch u

$\Longrightarrow$ jeder Pfad von $x \in F-(\omega_1+\omega_2)$ nach $g \in \omega_1+\omega_2$ enthält $ROOT(\omega_1)=ROOT(\omega)$.
Die Behauptung ist damit für alle $i\geq 0$ gezeigt.
□

Da |F| endlich ist, gelten wegen Schritt 3 von Verfahren 4.1 die Aussagen der Sätze 4.8 und 4.9 auch für Ω:

KOROLLAR 4.5:
Für $\omega \in \Omega$ gilt:
a) $f,g \in \omega \Longrightarrow \rho_f \equiv \rho_g$
b) Jeder Pfad von $x \in F-\omega$ nach $g \in \omega$ enthält ROOT(ω).

Bemerkung:
Die Umkehrung von Korollar 4.5a gilt i.allg. nicht! Man betrachte hierzu den folgenden Fragmentgraphen mit den entry-Fragmenten 1 und 2 und den S-Fragmenten 3 und 4:



Es gilt $\rho_3 \equiv \rho_4 \equiv \rho_1$ OR ρ_2, die Fragmente 3 und 4 sind aber in verschiedenen Ω-Mengen enthalten: $\Omega = \{ \{1\},\{2\},\{3\},\{4\} \}$.

SATZ 4.10:
f sei ein X-, O-, S- oder entry-Fragment, $g \in F$, $\omega \in \Omega$. Enthält jeder Pfad P von einem entry-Fragment nach g das Fragment f und ist jeder Pfad von f nach g ein R1-Pfad, so gilt:

$$f \in \omega \Longrightarrow g \in \omega$$

Beweis:
Da |F| endlich ist und (F,R) keine Zyklen enthält, wird gezeigt:

Ist das Maximum der Längen der Pfade von E nach g endlich, so gilt mit den Voraussetzungen des Satzes: $f \in \omega \Longrightarrow g \in \omega$.

Dies wird durch Induktion über L(g) gezeigt, wobei L(y) das Maximum der Längen aller Pfade von f nach $y \in F$ bezeichnet.
Die Behauptung ist richtig für g mit L(g)=1: es ist $g \in R1(f)$, $f \in \omega$ impliziert $R1(f) \subseteq \omega$ nach Korollar 4.4b und somit $g \in \omega$.
Induktionsannahme: die Behauptung ist für g mit $L(g) \leq k$, $k>1$, gezeigt.
Induktionsschluß:
Es sei g mit L(g)=k+1 und $PRED(g)=\{g_i \mid 1 \leq i \leq m\}$ gegeben und $f \in \omega$. Da jeder Pfad von $e \in E$ nach g den Knoten f enthält, gilt dies auch für alle Pfade von e nach $x \in PRED(g)$. Wegen $L(x) \leq k$ für $x \in PRED(g)$ liefert die Induktionsannahme für $1 \leq i \leq m$: $f \in \omega \Longrightarrow g_i \in \omega$.

- Ist g ein S-Fragment, dann gilt $R1(g) \subseteq \omega$ wegen Schritt 2 von Verfahren 4.1 und Korollar 4.4b.

- sonst, d.h. im Falle PRED(g)={g_1}, ist nach Voraussetzung der Pfad von g_1 nach g ein R1-Pfad, also: g_1,g ε R1(x) für ein x ε F und R1(x) $\subseteq$ ω (nach Korollar 4.4b wegen g_1 ε ω).

In beiden Fällen folgt g ε ω, die Behauptung gilt also für beliebige L(g), womit der Satz gezeigt ist.
□

Für den allgemeinen Fall, daß in Schritt 2 mehrere Paare (ω_1,ω_2) die Bedingung zur Konstruktion von $\omega^{(i)}$ erfüllen, ist durch Verfahren 4.1 nicht festgelegt, mit welchem Paar $\omega^{(i)}$ zu bilden ist. Mit den Ergebnissen von oben läßt sich zeigen, daß diese Reihenfolge insofern unwesentlich ist, als Schritt 2 in jedem Fall die gleiche Menge Ω liefert, Ω also eindeutig bestimmt ist:

SATZ 4.11:
Verfahren 4.1 liefert zu einem Fragmentsystem genau eine Ω-Menge
Beweis:
Verfahren 4.1 ergebe zu einem Fragmentsystem (F,R,X,O,E,ρ) zwei Ω-Mengen Ω' und Ω". Es sei ω' ε Ω', ROOT(ω')=r'.
- Nach Korollar 4.4a und b gibt es ein ω" ε Ω" mit R1(r') $\subseteq$ ω".
- Für jede R1-Menge R=R1(f) ε $\Omega^{(0)}$ mit R $\subseteq$ ω', R≠R1(r') gilt:
  Jeder Pfad von E nach f enthält auch r' (Korollar 4.5b) und jeder Pfad von r' nach f ist ein R1-Pfad (es gibt einen solchen Pfad wegen Eigenschaft FG1' eines Fragmentsystems). Aus r' ε ω" folgt daher nach Satz 4.10 f ε ω" und damit nach Korollar 4.4b R $\subseteq$ ω".

Zu jedem ω' ε Ω' gibt es also ein ω" ε Ω", so daß ω" alle R1-Mengen von ω', und damit ω', enthält. Da Ω' und Ω" jeweils disjunkte Zerlegungen von F sind, folgt Ω'=Ω".
□

4.3.5. Die Relevanzen eines Fragmentgraphen

In Abschnitt 4.3.3 ist gezeigt worden, daß die Menge CF von Verfahren 4.1 die Eigenschaft CF2 einer charakteristischen Menge besitzt. Es wird nun nachgewiesen, daß für die Menge CF auch CF1 gilt. Hierzu wird angegeben, wie man für f ε F-CF zu einer CF-Darstellung C(f) $\subseteq$ CF und damit einer Gleichung

$$\rho_f \equiv OR_{g\varepsilon C(f)}\ \rho_g$$

gelangt.

Da die Elemente je einer Ω-Menge in der Relevanz übereinstimmen (Korollar 4.5a) und Ω eine disjunkte Zerlegung von F ist (Korollar 4.4a), kann man sich auf die Bestimmung einer CF-Darstellung für die Wurzelfragmente der Ω-Mengen, d.h. die Elemente von FΩ, beschränken.

DEFINITION 4.6:

SUCX(f) := { g | g ε F, X(g)≠∅,
jeder Pfad von E nach g enthält f,
(*) f=g oder es gibt einen R1-Pfad von f nach g }

Bemerkung:

- Nach Satz 4.5 und Korollar 4.3 gilt für f ε FΩ mit |PRED(f)|≤1:
  SUCX(f)=∅ <==> XΩ(f)=∅ <==> f ε CF
- Ist f ein entry-Fragment, so sind die Fragmente von SUCX(f) von keinem anderen entry-Fragment als f erreichbar.
- Beispiele für Mengen SUCX(f) gibt 4.3.5.4.

4.3.5.1. Die Relevanzen der X- oder entry-Fragmente von FΩ

Es sei f ε FΩ ein X- oder entry-Fragment

a) Eine CF-Darstellung für f ε CF ist {f}.
Für f ¬ε CF ist wie zu Definition 4.6 festgestellt SUCX(f) ≠ ∅. Es sei |SUCX(f)|=n≥1, SUCX(f):={f_i|1≤i≤n} und X(f_i):={$f_{i,j}$|1≤j≤m(i)}.

Für alle i mit 1≤i≤|SUCX(f)| und t ε T erhält man

$\rho(t,f)=0 \Longrightarrow \rho(t,f_i)=0$ (Satz 4.3)

$\rho(t,f)=1 \Longrightarrow \rho(t,f_i)=1$ (mit G1 von 4.2.1. wegen *)

also: $\rho_f \equiv \rho_i$

Für ρ_f gelten daher nach 4.2.2.1. die n Gleichungen

$$\rho_f \equiv OR_{j=1}^{m(i)}\rho_{i,j} \qquad 1\le i\le |SUCX(f)|$$

Die Relevanz eines X- oder entry-Fragmentes ρ_f ist also festgelegt durch die Relevanzen $\rho_{i,j}$ der X-Fragmente der Elemente von SUCX(f).
Hat man für ein i, 1≤i≤|SUCX(f)|, eine CF-Darstellung C($f_{i,j}$) für jedes $f_{i,j}$, dann gilt

$$C(f) = C(f_i) = +_{j=1}^{m(i)} C(f_{i,j})$$

Für X- oder entry-Fragmente x mit SUCX(x)=∅, also x ε CF, ist C(x) = {x}: {x} ist die einzige CF-Darstellung von x.
Für $f_{i,j}$ ε CF ist man also fertig; für $f_{i,j}$ ¬ε CF, also SUCX($f_{i,j}$)≠∅, ist C($f_{i,j}$) wieder die Vereinigung von CF-Darstellungen der X-Fragmente eines Elements von SUCX($f_{i,j}$), etc. Da |F| endlich und jeder Pfad in (F,R) azyklisch ist, gelangt man so in endlich vielen Schritten zu einer Menge C(f) mit $\rho_f \equiv OR_{g\varepsilon C(f)} \rho_g$.
Nach folgendem Verfahren erhält man somit eine CF-Darstellung für X- oder entry-Fragmente:

VERFAHREN 4.2: Ermittlung einer CF-Darstellung C(f)

Input : Fragmentgraph (F,R,X,O,E), ein X- oder entry-Fragment f
Output: eine CF-Darstellung C ε CF für f
Algorithmus:

C = C(f)

mit

```
FUNCTION C(f)
   IF (SUCX(f)=∅)
        THEN C = {f}
        ELSE DO
             man wähle ein d ε SUCX(f)
             es sei X(d)={d_i | 1≤i≤|X(d)|}, n=|X(d)|

             C = +_{i=1}^{n} C(d_i)
             END
END
```

b) Gibt es n=|SUCX(f)|>1 Gleichungen für ρ_f, so bedeutet dies, daß die Relevanzausdrücke der n Gleichungen äquivalent sind, d.h. |SUCX(f)|-1 Bedingungen

(RC1) $$OR_{j=1}^{m(1)} \rho_{1,j} \equiv OR_{j=1}^{m(i)} \rho_{i,j} \qquad 2\leq i\leq |SUCX(f)|$$

erfüllt sein müssen.

4.3.5.2. Die Relevanzen der O-Fragmente von FΩ

Es sei f ε FΩ ein O-Fragment

a) Eine CF-Darstellung für f ε CF ist {f}.
Für f ¬ε CF folgt wieder SUCX(f) ≠ ∅ und $\rho_f \equiv \rho_g$ für g ε SUCX(f), so daß ρ_f n=|SUCX(f)| Gleichungen genügen muß (Bezeichnungen wie in 4.3.5.1.):

$$\rho_f \equiv OR_{j=1}^{m(i)} \rho_{i,j} \qquad 1 \leq i \leq |SUCX(f)|$$

Zu jedem der X-Fragmente $f_{i,j}$ kann man nach Verfahren 4.2 eine CF-Darstellung angeben, so daß damit n CF-Darstellungen für f gegeben sind:

$$C(f) = +_{j=1}^{m(i)} C(f_{i,j}) \qquad 1 \leq i \leq |SUCX(f)|$$

b) Es muß stets eine Bedingung in Form einer Implikation RC2 erfüllt sein; ist g ε FΩ der Vorgänger in GΩ von f, d.h. (g,f) ε RΩ, so muß nach 4.2.2.2. für t ε T gelten:

(RC2) $$\rho_f(t)=1 \implies \rho_g(t)=1$$

Im Fall |SUCX(f)|>1 gelten zusätzlich analog zu 4.3.5.1b |SUCX(f)|-1 Bedingungen der Form RC1!

4.3.5.3. Die Relevanz der S-Fragmente von FΩ

Es sei f ε FΩ ein S-Fragment.

a) Nach Satz 4.1 läßt sich ρ_f als die OR-Verknüpfung der Relevanzen von O-, X- oder entry-Fragmenten darstellen:

$$\rho_f \equiv OR_{j=1}^{m(0)} \rho_{0,j}$$

Zu jedem Fragment $f_{0,j}$ kann man nach 4.3.5.1 bzw. 4.3.5.2. eine CF-Darstellung angeben, so daß mit dieser Gleichung auch für f eine CF-Darstellung gegeben ist:

$$C(f) = +_{j=1}^{m(0)} C(f_{0,j})$$

b) Ist |SUCX(f)|≥0, so gilt zusätzlich (mit den Bezeichnungen von 4.3.5.1.):

$$\rho_f \equiv OR_{j=1}^{m(i)} \rho_{i,j} \qquad 1 \leq i \leq |SUXC(f)|$$

so daß man |SUCX(f)| Bedingungen der Form RC1 erhält:

$$OR_{j=1}^{m(0)} \rho_{0,j} \equiv OR_{j=1}^{m(i)} \rho_{i,j} \qquad 1 \leq i \leq |SUCX(f)|$$

4.3.5.4. Die Relevanzen des Beispielsystems

Mit den Ergebnissen von oben können nun die Relevanzen des Beispielsystems angegeben werden (vgl. Anhang 2).

a) Aus der Zerlegung Ω von F (Bild 4.2) ergeben sich nach Korollar 4.5a die folgenden Gleichungen:

$^*\ \rho_{1.1} \equiv \rho_2 \equiv \rho_{11}$	$^*\ \rho_{2.1} \equiv \rho_{13}$	$^*\ \rho_{10.1} \equiv \rho_{17}$
$^*\ \rho_{1.2} \equiv \rho_9 \equiv \rho_{12}$	$^*\ \rho_{9.1} \equiv \rho_{14}$	$^*\ \rho_{10.2} \equiv \rho_{18}$
$\rho_{1.3} \equiv \rho_3 \equiv \rho_4$	$^*\ \rho_{8.1} \equiv \rho_{15}$	$^*\ \rho_{10.3} \equiv \rho_{20}$
$\rho_{1.5} \equiv \rho_8$	$^*\ \rho_{8.2} \equiv \rho_{16}$	$\rho_{5.1} \equiv \rho_6$
$\rho_{1.6} \equiv \rho_{10}$	$^*\ \rho_{8.3} \equiv \rho_{19}$	$^*\ \rho_{5.2} \equiv \rho_7$
	$\rho_{22} \equiv \rho_{22.1}$	$^*\ \rho_{4.2.2} \equiv \rho_{21}$

Mit "*" sind Gleichungen mit der Relevanz eines charakteristischen Fragments gekennzeichnet, in diesen Fällen liegt also bereits ein CF-Ausdruck und damit für die zugehörigen Fragmente eine CF-Darstellung vor!

Es sind noch CF-Darstellungen für die Fragmente von FΩ-CF anzugeben:

b) X- und entry-Fragmente:
Nicht in CF enthalten sind die X-Fragmente 1.3, 1.5, 1.6, 4.2, 5.1 und das entry-Fragment 1 (vgl. Beispiel zu Verfahren 4.1). CF-Darstellungen dieser Fragmente ergeben sich nach 4.3.5.1. wie folgt:

f	SUCX(f)	C(f)
4.2	{4.2}	{ 4.2.1 , 4.2.2 }
1.3	{4}	{ 4.1 } + C(4.2) = { 4.1 , 4.2.1 , 4.2.2 }
1.5	{8}	{ 8.1 , 8.2 }
1.6	{10}	{ 10.1,10.2}
1	{1}	{ 1.1 , 1.2 , 1.4 } + C(1.3) + C(1.5) + C(1.6) =
		= { 1.1 , 1.2 , 1.4 , 4.1 , 4.2.1 , 4.2.2 ,
		8.1 , 8.2 , 10.1 , 10.2 }
5.1	{6}	{ 6.1 , 6.2 }

c) Für die O-Fragmente von Bild 4.2 ist nichts zu zeigen, sie sind charakteristische Fragmente.

d) CF-Darstellungen zu den S-Fragmenten:

Nach Abschnitt 4.3.5.3 gibt es für das Fragment 5 zwei CF-Darstellungen: { 2.1 , 1.1 , 1.4 } nach Satz 4.1 und
{ 6.1 , 6.2 , 5.2 } wegen SUCX(5)={5}$\neq\emptyset$.
Für das folgende gilt: C(5) := { 6.1 , 6.2 , 5.2 }

Wegen SUCX(22)=$\emptyset$ ergibt sich eine CF-Darstellung für Fragment 22 nur mittels Satz 4.1: Ersetzt man im Ausdruck für ρ_{22} vom Beispiel zu Satz 4.1 die Relevanz $\rho_{1.3}$ durch den CF-Ausdruck von Fragment 1.3, so erhält man: C(22) = { 2.1 , 4.1 , 4.2.1 , 4.2.2 , 9.1 , 8.3 , 10.3 }.

4.4. Die Menge der Teilsysteme

Ein Teilsystem $t \varepsilon T$ ist eindeutig beschrieben durch die Menge seiner Fragmente $F_t=\{f|f\varepsilon F,\rho_f(t)=1\}$ (Korollar 3.1).
Es sei $CF=\{g_i|1\leq i\leq n\}$ eine charakteristische Menge, also $n=|CF|$, und für $1\leq i\leq n$ ρ_i die Relevanz von g_i. Dann ist nach Definition 4.4

$$F_t = \{ f \mid f \varepsilon F, OR_{g\varepsilon C(f)}\rho_g(t) =1 \},$$

d.h. die Menge der für t relevanten Fragmente ist mit den n Relevanzwerten $\rho_i(t)$, $1\leq i\leq n$, gegeben. Jedem $t \varepsilon T$ ist somit auf folgende Weise genau ein Element $\tau(t) \varepsilon B^n$ zugeordnet:

$$\tau(t) := (\rho_1(t), \ldots , \rho_n(t))$$

Bezeichnungen:
- $\tau(t)$ heißt die D a r s t e l l u n g von $t \varepsilon T$
- Für $f \varepsilon F$ bezeichne $I(f)$ die Menge der Indizes der charakteristischen Fragmente der CF-Darstellung $C(f)$ von f:

$$I(f) = \{ i \mid g_i \varepsilon CF*C(f) \}.$$

Hinweis: Die Darstellung des vollständigen Teilsystems ist das Element von B^n, das in allen Komponenten den Relevanzwert 1 hat!

Für die Abbildung ρ ergibt sich damit (vgl. Definition 3.1):
Hat $t \varepsilon T$ die Darstellung $\tau \varepsilon B^{|CF|}$, so gilt

$$\rho(t,f) = OR_{i\varepsilon I(f)}\ \tau[i] \quad f \varepsilon F$$

Beispiel:
Für das Beispielsystem ist $n=|CF|=18$ (vgl. Bilder 4.2, 4.3), die charakteristischen Fragmente seien wie folgt indiziert:

i	1	2	3	4	5	6	7	8	9	10
g_i	1.1	2.1	1.2	9.1	1.4	5.2	5.3	6.1	6.2	4.1

i	11	12	13	14	15	16	17	18
g_i	4.2.1	4.2.2	8.1	8.2	8.3	10.1	10.2	10.3

Die Darstellung des Teilsystems t_ins von Bild 3.6 ist damit

$$\tau=(1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0),$$

d.h. t_ins ist das Teilsystem mit den charakteristischen Fragmenten g_1, g_3, g_7, g_8 und g_{13}.
Für Fragment 22 ist $I(22)=\{2,4,10,11,12,15,18\}$ und damit $\rho(t\_ins,22)=0$.

Eine Zusammenstellung der Mengen I(f) des Beispielsystems gibt Anhang 2.

Man beachte, daß es zum Beispielsystem kein Teilsystem t mit einer Darstellung $(0,1, \ldots) \varepsilon B^n$ gibt: nach Abschnitt 4.3.5.2 (RC2) muß jedes $t \varepsilon T$ der Bedingung $\rho(t,2.1)=1 \Longrightarrow \rho(t,1.1)=1$ genügen, damit also $\tau \varepsilon B^n$ ein $t \varepsilon T$ repräsentieren kann, muß notwendigerweise mit $\tau[2]=1$ auch $\tau[1]=1$ sein.
Allgemein muß $\tau \varepsilon B^{|CF|}$ R e s t r i k t i o n e n der Form

$$OR_{i \varepsilon I1} \tau[i] = OR_{i \varepsilon I2} \tau[i]$$
$$OR_{i \varepsilon I1} \tau[i]=1 \Longrightarrow OR_{i \varepsilon I2} \tau[i]=1$$

mit $I1,I2 \subseteq \{1, \ldots ,|CF|\}$ genügen, damit es Darstellung eines $t \varepsilon T$ sein kann.

Eine Teilmenge dieser Restriktionen leitet sich aus dem Fragmentgraphen her:
Die Bedingungen RC1 und RC2, die man unmittelbar bei der Bestimmung der Relevanzen der Fragmente gemäß 4.3.5. erhält, sind notwendigerweise von jedem Teilsystem zu erfüllen.
Hieraus ergeben sich Restriktionen für $\tau \varepsilon B^{|CF|}$, indem man sie, gegebenenfalls unter Verwendung von CF-Ausdrücken, so umformt, daß sie nur noch Relevanzen charakteristischer Fragmente enthalten und dann jedes ρ_i bzw. $\rho_i(t)$ durch $\tau[i]$ ersetzt.

Da es i.allg. zu einem $f \varepsilon F\Omega$ mehrere CF-Darstellungen gibt, ergeben sich so aus einer Bedingung i.allg. mehrere, verschiedene Restriktionen für τ. Zur Herleitung aller Restriktionen zu den Bedingungen von 4.3. benötigt man also alle CF-Darstellungen von f, d.h. eine Verallgemeinerung von Verfahren 4.2. Hierzu wird definiert:

DEFINITION 4.7:
Es seien M_1 und M_2 zwei Mengen, $S_1 \subseteq \P(M_1)$, $S_2 \subseteq \P(M_2)$.

$$S_1 \text{¤} S_2 := \{ s_1+s_2 \mid s_1 \varepsilon S_1, s_2 \varepsilon S_2 \}$$

$S_1 \text{¤} S_2$ ist die Menge der Vereinigungen aller Paare von $S_1 \times S_2$. Man beachte:
$|S_1|=|S_2|=1 \Longrightarrow S_1 \text{¤} S_2 = \{s_1+s_2\}$

Es sei $f \varepsilon F\Omega$ kein S-Fragment:
Stellt C_ALL(f) die Menge aller CF-Darstellungen von f dar, so gilt (vgl. Verfahren 4.2):
- C_ALL(f) = {f} für $f \varepsilon CF$, d.h. SUCX(f)=∅
- sonst, also bei |SUCX(f)|=n>0 ist

  $C\_ALL(f) = +_{i=1}^{n} C\_ALL(f_i)$, wobei
  $C\_ALL(f_i) = C\_ALL(f_{i,1}) \text{¤} \ldots \text{¤} C\_ALL(f_{i,m(i)})$ (¤ ist kommutativ!)

Dies führt zu folgendem Verfahren:

VERFAHREN 4.3: Ermittlung aller CF-Darstellungen

Input : Fragmentgraph (F,R,X,O,E), ein X-, O- oder entry-Fragment f
Output: die Menge C der CF-Darstellungen von f
Algorithmus:

C = C_ALL(f)

mit

FUNCTION C_ALL(f)
  IF (SUCX(f)=∅)
    THEN C_ALL = {f}
    ELSE DO
      es sei SUCX(f)={f_i | 1≤i≤n}, n=|SUCX(f)|,
        X(f_i)={$f_{i,j}$ | 1≤j≤m(i)} für 1≤i≤n
      C_ALL = $+_{i=1}^{n}$(C_ALL($f_{i,1}$)¤...¤C_ALL($f_{i,m(i)}$))
      END
END

Da |F| endlich und jeder Pfad in (F,R) azyklisch ist, ergibt Verfahren 4.3 in endlich vielen Schritten zu f die Menge aller CF-Darstellungen von f, vgl. Verfahren 4.2.

Berücksichtigt man nur die RC1- und RC2-Bedingungen, so erhält man zwar Systeme, die die gewünschten Fähigkeiten realisieren, trotzdem jedoch möglicherweise nicht sinnvoll, weil unvollständig, sind:
Die RC1- und RC2-Bedingungen des Beispiels erlauben die Erzeugung von Teilsystemen, die z.B.

- die Funktion OPEN, nicht aber CLOSE implementieren und somit keine im Sinne der Schnittstellenbeschreibung (vgl. Abschnitt 3.1) korrekten DB-Transaktionen realisieren lassen.
- die nur die Operationen OPEN und CLOSE implementieren und damit keinerlei Zugriff auf Daten erlauben.

Es sind daher i.allg. weitere Restriktionen erforderlich, die gewährleisten, daß ein Teilsystem "sinnvoll" ist; diese können sich herleiten aus

a) der Semantik der System-Schnittstelle
   Die Schnittstelle des Systems kann erfordern, daß
   - zwei oder mehr Operationen stets miteinander benötigt werden
   - die Ausführung einer Operation O_0 stets die Ausführung einer von n Operationen O_i, 1≤i≤n, bedingt (als Voraussetzung oder Folge)

   Dies sind Aussagen über die Relevanz der charakteristischen Fragmente zu diesen Operationen, die die Form von RC1- bzw. RC2-Bedingungen haben, sie können damit wie oben angegeben ebenfalls als Restriktionen dargestellt werden.
b) der Implementierung, Programmstruktur
   Beziehungen zwischen den Relevanzen von Fragmenten, die sich aus der programmtechnischen Realisierung ergeben, nicht aber durch den Fragmentgraphen widergespiegelt werden, können ebenfalls zu Bedingungen der Form RC1, RC2 und damit zu Restriktionen führen.

Beispiel: Die Restriktionen des Beispielsystems

(Man beachte hierzu Anhang 2: die Mengen C(f))

1) Aus dem Fragmentgraphen (Bild 4.2) ergeben sich die folgenden Bedingungen:

- zu den Ω-Mengen mit einem O-Fragment als Wurzel (Abschnitt 4.3.5.2b, RC2):

$$\rho(t, 2.1)=1 \Longrightarrow \rho(t,1.1)=1$$
$$\rho(t, 9.1)=1 \Longrightarrow \rho(t,1.2)=1$$
$$\rho(t, 5.3)=1 \Longrightarrow \rho(t, 5\ )=1$$
$$\rho(t, 8.3)=1 \Longrightarrow \rho(t,1.5)=1$$
$$\rho(t,10.3)=1 \Longrightarrow \rho(t,1.6)=1$$

- zur Ω-Menge mit dem S-Fragment 5 als Wurzel: e i n e RC1-Bedingung (vgl. Abschnitt 4.3.5.3, |SUCX(5)|=1!)

$$\rho_{2.1} \text{ OR } \rho_{1.1} \text{ OR } \rho_{1.4} \equiv \rho_{5.1} \text{ OR } \rho_{5.2}$$

Weitere RC1-Bedingungen gibt es nicht, da für alle Wurzelfragmente f der Ω-Mengen, die keine S-Fragmente sind, |SUCX(f)|≤1 ist.

Diese Bedingungen führen zu folgenden Restriktionen:

$$\tau[\ 2]=1 \Longrightarrow \tau[\ 1]=1$$
$$\tau[\ 4]=1 \Longrightarrow \tau[\ 3]=1$$
$$\tau[\ 7]=1 \Longrightarrow \tau[\ 8] \text{ OR } \tau[\ 9] \text{ OR } \tau[\ 6] = 1$$
$$\tau[15]=1 \Longrightarrow \tau[13] \text{ OR } \tau[14] = 1$$
$$\tau[18]=1 \Longrightarrow \tau[16] \text{ OR } \tau[17] = 1$$

$$\tau[2] \text{ OR } \tau[1] \text{ OR } \tau[5] = \tau[8] \text{ OR } \tau[9] \text{ OR } \tau[6]$$

2) Der Beschreibung der Schnittstelle zum Beispielsystem (Abschnitt 3.1.) entnimmt man u.a. die folgenden Beziehungen zwischen den Operationen ("-->" bedeutet "bedingt"):

- Stets miteinander benötigt werden OPEN und CLOSE: $\rho_{2.1} \equiv \rho_{9.1}$, $\rho_{1.1} \equiv \rho_{1.2}$
- INSERT --> OPEN: $\rho_{8.1}(t) \text{ OR } \rho_{8.2}(t) = 1 \Longrightarrow \rho_{1.1}(t)=1$
- FIND --> OPEN: $\rho_{4.1}(t) \text{ OR } \rho_{4.2.1}(t) \text{ OR } \rho_{4.2.2}(t) = 1 \Longrightarrow \rho_{1.1}(t)=1$
- GET --> FIND: $\rho_{6.1}(t) \text{ OR } \rho_{6.2}(t) \text{ OR } \rho_{5.2}(t) = 1 \Longrightarrow$
  $\Longrightarrow \rho_{4.1}(t) \text{ OR } \rho_{4.2.1}(t) \text{ OR } \rho_{4.2.2}(t) = 1$

Dies ergibt die folgenden fünf Restriktionen:

$$\tau[2]=\tau[4] \quad \tau[1]=\tau[3]$$
$$\tau[13] \text{ OR } \tau[14] = 1 \Longrightarrow \tau[1]=1$$
$$\tau[10] \text{ OR } \tau[11] \text{ OR } \tau[12] = 1 \Longrightarrow \tau[1]=1$$
$$\tau[8] \text{ OR } \tau[9] \text{ OR } \tau[6] = 1 \Longrightarrow \tau[10] \text{ OR } \tau[11] \text{ OR } \tau[12] = 1$$

3) Aus der Implementierung ergibt sich die Beziehung
OPEN --> Algorithmus A8: $\rho_{1.1}(t)=1 \Longrightarrow \rho_{6.1}(t)=1$
(A8: retrieval aus Systemkatalog nach Speicherungsstruktur 1, s. Erläuterungen zu t_ins von 3.1.) und hiermit die Restriktion

$$\tau[1]=1 \Longrightarrow \tau[8]=1$$

Zur Zahl der Teilsysteme eines Programmsystems:

Ist u die Zahl der charakteristischen Elemente, die in keiner Restriktion enthalten sind, so gilt für die Zahl der verschiedenen Teilsysteme

$$2^u \leq |T| \leq 2^{|CF|}$$

Zur genaueren Abschätzung nach oben und in Spezialfällen der Ermittlung von |T| kann dienen:

SATZ 4.12:

Mit $x=(x_1, \ldots, x_m) \varepsilon B^m$, $y=(y_1, \ldots, y_n) \varepsilon B^n$ gilt:

a) es gibt $(2^m-1)*(2^n-1)+1$ verschiedene Elemente $(x,y) \varepsilon B^{m+n}$ mit

$$OR_{i=1}^{m} x_i = OR_{i=1}^{n} y_i$$

b) es gibt $(2^m-1)*(2^n-1)+2^n$ verschiedene Elemente $(x,y) \varepsilon B^{m+n}$ mit

$$OR_{i=1}^{m} x_i = 1 \implies OR_{i=1}^{n} y_i = 1$$

Beweis:

Allgemein gilt: es gibt 2^i-1 Elemente von B^i, so daß mindestens eine Komponente gleich 1 ist.

Für jedes $x \varepsilon B^m$ mit $OR_{i=1}^{m} x_i = 1$ gibt es also 2^n-1 Elemente $y \varepsilon B^n$ mit $OR_{i=1}^{n} y_i = 1$, für $(2^m-1)(2^n-1)$ Elemente $(x,y) \varepsilon B^{m+n}$ ergeben beide Seiten der Gleichung von a) zugleich den Wert 1.
Da der Wert 0 auf beiden Seiten der Gleichung nur für x=(0,...,0) und y=(0,...,0) möglich ist, ist damit a) gezeigt.

Die Behauptung b) folgt aus der Tatsache, daß die Implikation im Falle $OR_{i=1}^{m} x_i = 0$ für j e d e s $y \varepsilon B^n$ erfüllt ist.

□

5. Zur Implementierung der Programmerzeugung

Der Ablauf der Programmerzeugung und die wesentlichen Komponenten eines Programmerzeugungssystems sind in Abschnitt 2.4 skizziert worden (s. Bild 2.4). Mit den Ergebnissen von Abschnitt 3 und 4 ist man nun in der Lage, die Realisierung eines Systems zur Erzeugung von Versionen eines Programmsystems anzugeben.

Es wird ein Verfahren entwickelt, bei dem das abstrakte B-Programm als lineare Zeichenkette implementiert ist, die der Selektor zur Quellprogrammerzeugung genau einmal, zeichenweise liest; Erzeugung des Programms eines Teilsystems und Dimensionierung, zwei nach 2.3. konzeptionell getrennte Teilaufgaben, bilden also e i n e n Verarbeitungsschritt.
Die ISD stellt die zur Code-Auswahl und Dimensionierung erforderlichen Informationen zur Verfügung, zusätzlich enthält sie eine Beschreibung der Menge der "korrekten" Versionen; hierzu gehören u.a. die Restriktionen für sinnvolle Teilsysteme nach 4.4. Es wird gezeigt, daß diese Beschreibung wie ein B-Programm realisiert werden kann, der Selektor übernimmt damit zur eigentlichen Programmerzeugung auch Aufgaben des top part eines Generiersystems für DBMS-Versionen nach 2.2. Weiter wird gezeigt, wie mit dem Selektor die Programmerzeugung gesteuert werden kann, so daß diese nach Erstellung der ISD automatisch abläuft.

Für das folgende sei $BP=(P,S,\sigma,\rho)$ ein abstraktes B-Programm eines Programmsystems, wie in Abschnitt 3.3 ist $P=F+Q$, wobei F die Menge der inneren Knoten (die Fragmente des Programmsystems), Q die Blattknoten (die Teilstrings des vollständigen Programms) des geordneten Baums (P,S) ist.

5.1. Die interne Systembeschreibung ISD

Nach Abschnitt 2.2 (vgl. Bild 2.2) stellt eine interne Systembeschreibung ISD die Spezifikation des zu erzeugenden Lademoduls dar. Durch die ISD sind insbesondere festzulegen (vgl. 2.3.):
- für die Teilsystemerzeugung die Relevanzwerte der Fragmente (Abschnitt 3, Verfahren 3.1)
- für die Dimensionierung die Werte der Platzhalter

5.1.1. Die Spezifikation der Relevanzwerte

τ sei die Darstellung des Teilsystems t der zu erzeugenden Version V. Nach Abschnitt 4 sind mit τ die Relevanzwerte der charakteristischen und mit diesen die aller Fragmente zu t festgelegt. Die ISD kann daher als ein "Programm" verstanden werden mit Wertzuweisungen an sog. R - V a r i a b l e :
Jedem Knoten $f \in F$ von (P,S), d.h. jedem Fragment des Programmsystems, wird eine R-Variable, notiert r_f, zugeordnet, der Wert von r_f ist $\rho(t,f)$,

also 0 oder 1.
Ist f ein charakteristisches Fragment, so enthält die ISD eine Zuweisung

$$r_f = \tau[i],$$

wobei i der Index von f als Element der charakteristischen Menge ist (d.h. I(f)={i}, s. 4.4.); sonst ist dies eine Zuweisung der Form

$$r_f = OR_{i=1}^{n} r_i$$

bzw.

$$r_f = NOT(OR_{i=1}^{n} r_i)$$

mit R-Variablen r_i. Diese Zuweisungen erhält man aus den Gleichungen von 4.2. bzw. 3.2.1.3. (die Negation), es sind lediglich die Relevanzen durch R-Variable zu ersetzen.
Die Ausdrücke der rechten Seiten dieser Zuweisungen heißen R - A u s d r ü c k e . Im wesentlichen sind dies boolesche Ausdrücke mit den booleschen Operatoren NOT und OR.

5.1.2. Die Spezifikation der Platzhalter

Jedem Platzhalter wird eine P - V a r i a b l e zugeordnet, die ISD weist den P-Variablen den jeweiligen Wert des Platzhalters zur gegebenen Version zu.

Dabei ist zunächst zu beachten, daß es i.allg. analog zu den Beziehungen zwischen Relevanzen nach Abschnitt 4 auch Beziehungen zwischen Platzhaltern gibt, die für funktionsfähige Versionen erfüllt sein müssen:

- Der Wert eines Platzhalters p kann eine Funktion der Werte von Platzhaltern p_i sein:

  $$p = \phi(p_1,\cdots,p_n)$$

  Diese Gleichungen werden P - G l e i c h u n g e n genannt.
- Es kann Bedingungen, P - R e s t r i k t i o n e n , geben, denen Platzhalter genügen müssen; Beispiele hierfür: "Array A1 muß mindestens doppelt so groß sein wie Array A2", "Array A darf höchstens n Elemente enthalten".
  Allgemein sind dies Bedingungen der Form

  $$\phi_1(p_{1,1},\cdots,p_{1,n}) \text{ rop } \phi_2(p_{2,1},\cdots,p_{2,m})$$

  mit rop als Vergleichsoperator ($<,\leq,=,\geq,>$).

  P-Restriktionen erhält man auch, wenn für einen Platzhalter mehrere P-Gleichungen gelten.

Bemerkung:
Anders als für Relevanzen lassen sich diese Beziehungen zwischen Platzhaltern nicht aus dem Programmtext herleiten, sie sind vielmehr vom Systementwerfer bzw. -implementierer zu spezifizieren!

Es gibt somit i.allg. ein Teilmenge von Platzhaltern mit der Eigenschaft, daß mit deren Werten bereits die aller anderen Platzhalter festgelegt sind. Analog zu Abschnitt 4 werden solche Platzhalter c h a r a k t e r i s t i s c h e P l a t z h a l t e r genannt.

Zur Ermittlung einer Menge charakteristischer Platzhalter hat man also ein i.allg. nichtlineares Gleichungssystem zu lösen, es sei hierzu auf die entsprechenden Verfahren der Mathematik verwiesen.
Die Bestimmung einer Menge charakteristischer Platzhalter ist für den (in der Praxis häufigen) Fall einfach, daß die Funktionen ϕ der P-Gleichungen Linearkombinationen von Platzhaltern sind:

$$\phi_i(p_1,\ldots,p_n) = a_{i,1}p_1 + \ldots + a_{i,n}p_n + a_{i,n+1}$$

mit Konstanten $a_{i,j}$. Hier liegt ein lineares Gleichungsssystem vor, so daß man mit den Verfahren der linearen Algebra (vgl. z.B. /Ko 69/) eine Menge von charakteristischen Platzhaltern errechnen kann, die Mächtigkeit jeder dieser Mengen ist die Dimension des Lösungsraums des Gleichungssystems, also n-Rang(M) (n: die Zahl der Platzhalter; Rang(M): der Rang der Koeffizientenmatrix des Gleichungssystems); zugleich ist damit die Minimalität dieser Menge gewährleistet.

Ist p die P-Variable eines charakteristischen Platzhalters, so enthält die ISD eine Zuweisung

p = p-const ,

sonst eine Zuweisung der Form

p = p-expr ,

wobei p-expr ein arithmetischer Ausdruck mit P-Variablen ist.

5.1.3. Die Spezifikation der Restriktionen

Die Menge der Versionen zu den sinnvollen Teilsystemen eines Programmsystems, deren Platzhalter den P-Restriktionen nach 5.1.2. genügen, wird die Menge der k o r r e k t e n Versionen genannt.

In 5.3. wird gezeigt, daß die Überprüfung der Restriktionen, die von jeder korrekten Version erfüllt sein müssen, als Programmerzeugung im Sinne von Abschnitt 3 betrachtet und realisiert werden kann. Diese Überprüfungen, die in 2.2. getrennt von der eigentlichen Programmerzeugung dem top part eines Generiersystems zugeordnet sind, werden daher hier mit der Programmerzeugung vorgenommen, man erhält so ein Programmerzeugungssystem, das Aufgaben des top part nach 2.2. übernimmt.
Die ISD enthält somit neben den Zuweisungen von oben noch R e s t r i k t i o n e n , die die Restriktionen von Abschnitt 4.4 zur Charakterisierung der Menge der sinnvollen Teilsysteme, im weiteren R - R e s t r i k t i o n e n genannt, und die P-Restriktionen von 5.1.2. realisieren.

Diese Restriktionen ebenso wie die Zuweisungen von Ausdrücken gelten für alle korrekten Versionen, sie bilden den v e r s i o n s i n v a r i -

a n t e n Teil der ISD. Versionsspezifisch sind lediglich die Wertzuweisungen an die R- und P-Variablen charakteristischer Fragmente bzw. Platzhalter, der top part eines Generiersystems hat also zur Programmerzeugung lediglich diesen Teil der ISD zu erstellen.

Eine ISD wird hier also als ein Programm einer "Versions-Spezifikationssprache" verstanden.

5.2. Die Quellprogrammerzeugung

Nach 2.3. besteht die Erzeugung von Versionen eines Programmsystems aus zwei, prinzipiell voneinander getrennten Aufgaben:
- Code-Auswahl: Erzeugung des Programmtextes eines Teilsystems durch Auswahl der relevanten Teile des vollständigen Programms
- Dimensionierung: Erzeugung eines Quellprogramms durch (textuelles) Ersetzen der Platzhalter des Programms des Teilsystems.

5.2.1. Das B-Programm als Zeichenkette

Ein abstraktes B-Programm BP wird mittels geschachtelter Blöcke wie bereits in 3.2.3. skizziert als eine Zeichenkette implementiert.

Zunächst ist zu beachten, daß die Blattknoten von BP Teilstrings des vollständigen Programms darstellen, das Platzhalter enthält anstelle von Konstanten, deren Werte erst mit der Generierung einer Version festgelegt werden:
Die Platzhalter eines Programmsystems werden durch die Bezeichner der P-Variablen, also Zeichenketten, dargestellt. Zur Unterscheidung dieser Bezeichner von den Symbolen des Quellprogramms werden zwei Sonderzeichen bop (begin-of-placeholder) und eop (end-of-placeholder), die nicht im Alphabet A der Programmiersprache enthalten sind, als erstes bzw. letztes Zeichen der Bezeichner von Platzhaltern eingeführt.

Jeder Knoten $k \varepsilon F$ eines B-Programms ergibt einen Block nach 3.2.3., d.h. ein Tripel mit den Komponenten Relevanz, Ersatz und Blocktext. Mit weiteren Sonderzeichen wird das Konstrukt "Block" als Zeichenkette wie folgt realisiert:
- Den Anfang bzw. das Ende eines Blocks bilden die Sonderzeichen bob (begin-of-block) bzw. eob (end-of-block); sie haben die Funktion von Begrenzern und schließen im Sinne einer Klammerung Relevanz, Ersatz und Blocktext des Blocks ein (in 3.2.3. sind dies die Zeichen [bzw.]).
- Die Relevanz wird durch den Bezeichner r_k der R-Variablen von k dargestellt, er folgt unmittelbar auf bob
- vom Bezeichner der R-Variablen durch ein Sonderzeichen bos (begin-of-substitute) getrennt schließt sich der Ersatz $\sigma(k)$ als Zeichenkette an (nicht als Bezeichner)
- das Sonderzeichen bof (begin-of-fragment) zeigt das Ende des Ersatzstrings und den Beginn der Komponente Blocktext an; eob schließt mit dem Block auch die Komponente Blocktext ab.

Die Konstruktion einer solchen linearen Zeichenkette aus einem abstrakten B-Programm beschreibt Verfahren 5.1:

VERFAHREN 5.1: Konstruktion eines B-Programms als Zeichenkette
Input : abstraktes B-Programm BP=(P,S,σ,ρ), die Bezeichner r_f der R-Variablen zu den Knoten f ε F
Output: Zeichenkette TEXT
Algorithmus:

```
            r sei die Wurzel von (P,S)
            TEXT = NIL
            BP_TEXT(r)
```

mit

```
            PROCEDURE BP_TEXT(x)
              FOR I=1 TO |SUCC(x)|
              DO
              IF (x[I] ε F)
                  THEN BEGIN
                       TEXT = TEXT ‖ bob ‖ r_x[I] ‖ bos ‖ σ(x[I]) ‖ bof
                       BP_TEXT(x[I])
                       TEXT = TEXT ‖ eob
                       END
                  ELSE TEXT = TEXT ‖ x[I]
              END
            END
```

BP_TEXT(r) liefert eine Zeichenkette $x_1 \| x_2 \| \ldots \| x_n$, wobei

$$x_i = \begin{cases} r[i] & : r[i] \in Q \\ bob\ r_{r[i]}\ bos\ \sigma(r[i])\ bof\ BP\_TEXT(r[i])\ eob & : r[i] \in F \end{cases}$$

Für BP_TEXT(x), mit x=r[i] gilt Entsprechendes, so daß Verfahren 5.1 ein abstraktes B-Programm als eine Folge von Zeichenketten des vollständigen Programms (also Elementen von Q) und Blöcken realisiert, wobei die Komponente Blocktext eines Blocks selbst wieder die Konkatenation von Zeichenketten aus Q und Blöcken sein kann; die Konkatenation erfolgt in der durch die Ordnung des abstrakten B-Programms gegebenen Reihenfolge.

Bezeichnen rel die Bezeichner von R-Variablen und subst Ersatzstrings, dann läßt sich der Aufbau des strings von TEXT formal wie folgt beschreiben (geschweifte Klammern bedeuten Wiederholung des geklammerten Konstrukts, vgl. /Wi 77/):

```
            bp_text ::=  { source | block }
              block ::=  bob rel bos subst bof bp_text eob
```

5.2.2. Das Verfahren

Verfahren 5.2 beschreibt als Implementierung von Verfahren 3.1 die Erzeugung von Teilsystemen aus einem B-Programm, das als Zeichenkette nach Verfahren 5.1 vorliegt:

VERFAHREN 5.2: Erzeugung des Programmtextes eines Teilsystems
Input : ein B-Programm BP_TEXT als Zeichenkette, die interne Systembeschreibung ISD eines Teilsystems t
Output: das Programm PROG_TEXT des Teilsystems t
Algorithmus:

```
            PROG_TEXT = NIL
            EVAL(ISD,BP_TEXT,1)
mit
            PROCEDURE EVAL(ISD,BP_TEXT,p)
              suche mit dem p-ten Zeichen von BP_TEXT beginnend
              den nächsten Blockanfang (Zeichen bob) und füge
              dabei alle Zeichen bis bob an PROG_TEXT an. Bei
              Erreichen des letzten Zeichens von BP_TEXT ist man
              fertig; bei Auftreten eines bob wird der Wert r der
              R-Variablen des Blocks bestimmt, ersatz sei die
              Komponente Ersatz des Blocks

              IF (r=0)
                 THEN
                   BEGIN
                     PROG_TEXT = PROG_TEXT || ersatz
                     suche Block-Ende: p zeigt auf das Zeichen eob
                                       des Blocks
                   END
                 ELSE
                   BEGIN
                     suche Fragment-Anfang: p zeigt auf das Zeichen
                                            bof des Blocks
                     DO UNTIL (p zeigt auf eob des Blocks)
                        EVAL(ISD,BP_TEXT,p)
                   END
            END
```

Hat man den Programmtext eines Teilsystems t vorliegen, so sind zur Erzeugung des Quellprogramms einer Version zu t, also zur Dimensionierung, die Platzhalter des Programmtextes durch die jeweiligen Werte zu ersetzen: der Programmtext ist hierzu lediglich auf die Sonderzeichen bop und eop zu durchsuchen und die so markierten Teilstrings durch den Wert des jeweiligen Platzhalters (als string!) zu ersetzen.

Da zur Erzeugung des Programmtextes des Teilsystems nach Verfahren 5.2 das B-Programm zeichenweise gelesen wird, ist es leicht möglich, Dimensionierung und Code-Auswahl in einem Schritt vorzunehmen; in Verfahren 5.2 ist hierzu lediglich beim Anfügen von Teilstrings an PROG_TEXT auf das Auftreten der Sonderzeichen bop und eop zu prüfen und anstelle der so begrenzten Teilstrings die Werte der Platzhalter an

PROG_TEXT anzufügen.
Verfahren 5.2 mit dieser Erweiterung stellt die funktionale Beschreibung der Quellprogrammerzeugung durch den Selektor (s. Abschnitt 2.4) dar.

Der Selektor benötigt zur Quellprogrammerzeugung die Werte der R- und P-Variablen des B-Programms zur jeweiligen Version, d.h. er muß vor der eigentlichen Quellprogrammerzeugung als "Initialisierungslauf" die ISD der Version lesen und auswerten. Fordert man für die ISD als Programm, daß eine R- bzw. P-Variable erst dann in einem Ausdruck einer ISD-Anweisung referiert werden darf, wenn ihr vorher (im Programmtext) ein Wert zugewiesen worden ist, so genügt zur "Auswertung" der ISD ein einmaliges Lesen der ISD und Auswerten der Anweisungen in der Reihenfolge ihres Auftretens. Nach obigem stellen R- und P-Ausdrücke im wesentlichen boolesche bzw. arithmetische Ausdrücke dar. Die Werte der R- und P-Variablen können in Tabellen oder Listen abgelegt werden, so daß "Bestimmung des Relevanzwertes" bzw. "Bestimmung des Wertes eines Platzhalters" bei der Quellprogrammerzeugung ein einfaches Aufsuchen in einer Datenstruktur bedeutet.

Es erfordert somit keine funktionale Erweiterung des Selektors, wenn man im B-Programm zur Realisierung von Relevanzen anstelle von R-Variablen allgemeiner R-Ausdrücke und analog anstelle von P-Variablen P-Ausdrücke zuläßt. Auf diese Weise kann man Zuweisungen des versionsinvarianten Teils der ISD in das B-Programm "verlagern" und damit z.B die ISD verkürzen oder die Zahl der R- bzw. P-Variablen verringern.

5.3. Die Überprüfung auf Korrektheit

Mit dem Verfahren von 5.2. zur Quellprogrammerzeugung kann auch überprüft werden, ob die Werte der R- und P-Variablen einer ISD den R- und P-Restriktionen genügen, also eine korrekte Version zu erzeugen ist.
Hierzu ist zu beachten, daß mit jeder Restriktion eine Vorschrift e r r o r - a c t i o n verknüpft ist: error-action beschreibt, was im Falle eines Verstoßes gegen die jeweilige Restriktion zu geschehen hat: generell sollen Verstöße gegen eine oder mehrere Restriktionen die Quellprogrammerzeugung verhindern! Wünschenswert ist, daß stets a l l e Restriktionen der ISD überprüft werden und gegebenenfalls anstelle der Quellprogrammerzeugung angezeigt wird, welche Bedingungen nicht erfüllt sind. Hierzu sei error-action eine Zeichenkette, z.B. eine Fehlermeldung; die Überprüfung auf Korrektheit kann dann mit Verfahren 5.2 als ein Auswählen von Fehlermeldungen, und zwar der strings error-action der Restriktionen, die nicht erfüllt sind, implementiert werden. Restriktionen sind hierzu als Blöcke wie folgt zu schreiben (Die Syntax eines Blocks ist in 5.2.1. zu Verfahren 5.1 angegeben: die Sonderzeichen bob, bos, bof, eob sind hier durch die Symbole BOB, BOS, BOF, EOB realisiert; r-expr-1, r-expr-2 bzw. p-expr-1, p-expr-2 bezeichnen R- bzw. P-Ausdrücke als Verallgemeinerung von R- bzw. P-Variablen, vgl. 5.2.2.):

a) R-Restriktionen als Blöcke:

Eine R-Restriktion der Form RC2

```
r-expr-1=1  ==> r-expr-2=1
```

mit der Fehlermeldung error-action wird abgebildet in den Block

```
BOB r-expr-1
    BOS NIL
    BOF BOB r-expr-2
            BOS error-action
            BOF NIL
        EOB
EOB
```

Eine R-Restriktion der Form RC1

```
r-expr-1 = r-expr-2
```

mit der Fehlermeldung error-action wird in die folgenden zwei Blöcke abgebildet:

```
BOB r-expr-1
    BOS NIL
    BOF BOB r-expr-2
            BOS error-action
            BOF NIL
        EOB
EOB
```

```
BOB r-expr-2
    BOS NIL
    BOF BOB r-expr-1
            BOS error-action
            BOF NIL
        EOB
EOB
```

b) P-Restriktionen als Blöcke:

Zur Realisierung einer P-Restriktion

```
p-expr-1 rop p-expr-2
```

als Block ist es erforderlich, die Definition des Konstrukts Block von oben so zu erweitern, daß als Relevanz neben Relevanzausdrücken auch Vergleiche von P-Ausdrücken zugelassen sind: der Wert eines Vergleichs ist 1, falls der Vergleich erfüllt ist, sonst 0.
Erweitert man den Selektor funktional um die Fähigkeit "Vergleich zweier numerischer Werte", so können auch P-Restriktionen in Blöcke der Form

```
BOB  p-expr-1  rop  p-expr-2
     BOS error-action
     BOF NIL
EOB
```

abgebildet und wie in 5.2. beschrieben ausgewertet werden.

Es sei BP_RESTRICTIONS die Folge von Blöcken, die die R- und P-Restriktionen realisieren. Wie man sich leicht überzeugt, hat die Anwendung von Verfahren 5.2 auf BP_RESTRICTIONS mit der ISD der Version, also

```
PROG_TEXT = NIL
EVAL(ISD,BP_RESTRICTIONS,1)
```

den gewünschten Effekt: sind alle Bedingungen erfüllt, so hat PROG_TEXT als Wert noch den Anfangswert NIL, sonst eine Liste der Fehlermeldungen der nicht erfüllten Restriktionen. Die Entscheidung, ob eine Quellprogrammerzeugung durchzuführen ist, ist damit zurückgeführt auf die Abfrage, ob PROG_TEXT nach Ausführung von EVAL(ISD,BP_RESTRICTIONS,1) als Wert den leeren string hat.

Hinweis: BP_RESTRICTIONS kann als die Implementierung nach Verfahren 5.1 eines abstrakten B-Programms mit den Fehlermeldungen error-action als Blattknoten betrachtet werden!

5.4. Die Ablaufsteuerung

Es ist noch zu zeigen, wie die Erzeugung des Lademoduls ausgehend von der ISD automatisch, ohne steuernde, manuelle Eingriffe seitens des Generierers vorgenommen werden kann.

5.4.1. Definitionen, Voraussetzungen für ein Programmgeneriersystem

Der Selektor, Übersetzer und Binder seien Dienstprogramme mit den Programmnamen SELECTOR, COMPILER bzw. LINKER. Zur Programmerzeugung sind also diese drei Programme zu initiieren und durch Steuerprogramme zu steuern (Angabe der input-Dateien, output-Dateien, etc.).

Es wird daher ein Mechanismus benötigt, der es ermöglicht, Programme des Rechnersystems
- zu initiieren
- anhand von Steuerprogrammen zu steuern.

Für diese Funktionen stehe ein (Job-) M o n i t o r zur Verfügung, der es erlaubt, mit M o n i t o r - A n w e i s u n g e n ein Programm prog zu starten und dieses mit den Anweisungen eines Steuerprogramms commands zu steuern. Zur Verdeutlichung wird für Monitor-Anweisungen folgende Form angenommen:

```
EXEC prog WITH commands
```

Zur Ausführung einer Folge von EXEC-Anweisungen durch den Monitor werden diese dem Monitor als J o b übergeben, etwa durch das Kommando

```
SUBMIT job
```

5.4.2. Architektur eines Programmgeneriersystems

Neben Selektor, Compiler und Binder ist also ein Job-Monitor eine weitere notwendige Komponente zur Programmerzeugung. Nach Erstellung der ISD (es genügt die Erstellung des versionsspezifischen Teils, s. 5.1.3.) stößt der top part des Generiersystems (s. Abschnitt 2.2), etwa mit dem Kommando "SUBMIT V_GEN", die Programmerzeugung an, wobei der Job V_GEN aus den hierfür erforderlichen Monitor-Anweisungen besteht: sind V_SEL, V_COMP und V_LINK die Steuerprogramme jeweils für SELECTOR, COMPILER und LINKER, so sind dies mindestens die folgenden drei Anweisungen:

```
EXEC SELECTOR WITH   V_SEL
EXEC COMPILER WITH   V_COMP
EXEC LINKER   WITH   V_LINK
```

Die Programmerzeugung nach diesem Verfahren zeigt Bild 5.1.
Hierbei wird allerdings angenommen, daß für jede Programmerzeugung (unter Benutzung von V_GEN) die gleichen Steuerprogramme V_SEL, V_COMP, V_LINK benutzt werden können. Dies trifft aber im allgemeinen n i c h t zu:
Mit dem Steuerprogramm für einen Binder wird u.a. beschrieben, aus welchen Objektmoduln ein Lademodul zusammenzubinden ist, etwa mittels INCLUDE-Anweisungen; da i.allg. Versionen sich in der Zahl der erforderlichen Programmeinheiten unterscheiden, werden also auch entsprechend unterschiedliche Steuerprogramme für den Bindelauf benötigt, zumindest das Steuerprogramm V_LINK ist also versionsspezifisch.

Anders als beim Bindersteuerprogramm, das notwendigerweise versionsspezifisch ist, sind es mehr praktische Gesichtspunkte und Effizienzbetrachtungen, die dazu führen, auch für V_SEL und V_COMP versionsspezifische Steuerprogramme vorzusehen:
Umfangreiche B-Programme werden in der Regel in mehreren Dateien des Rechnersystems abgespeichert sein. Dem Selektor muß mitgeteilt werden, welche Dateien das B-Programm enthalten; dies geschehe mit Steueranweisungen z.B. von der Form

```
SELECT FROM b-file INTO source-file
```

wobei b-file eine Datei spezifiziert, die das oder einen Teil des B-Programms enthält, und source-file eine Datei ist, in die der Selektor den aus b-file erzeugten Text ablegt.

Bezeichnung:
Eine Datei b-file heißt relevant für eine Version, wenn sie einen Teilstring des B-Programms (also einen Ersatz-string oder Teil des vollständigen Programms) enthält, der für das Quellprogramm der Version benötigt wird.
Bemerkung:
Eine B-Programm-Datei ist relevant für eine Version, wenn z.B. eine R-Variable eines Blocks in dieser Datei den Wert 1 hat.

Die für die Quellprogrammerzeugung erforderliche Zeit ist eine Funktion der Länge des B-Programmes (in Zeichen); daher wird man zur Beschleunigung des Generiervorganges vom Selektor nur relevante Dateien

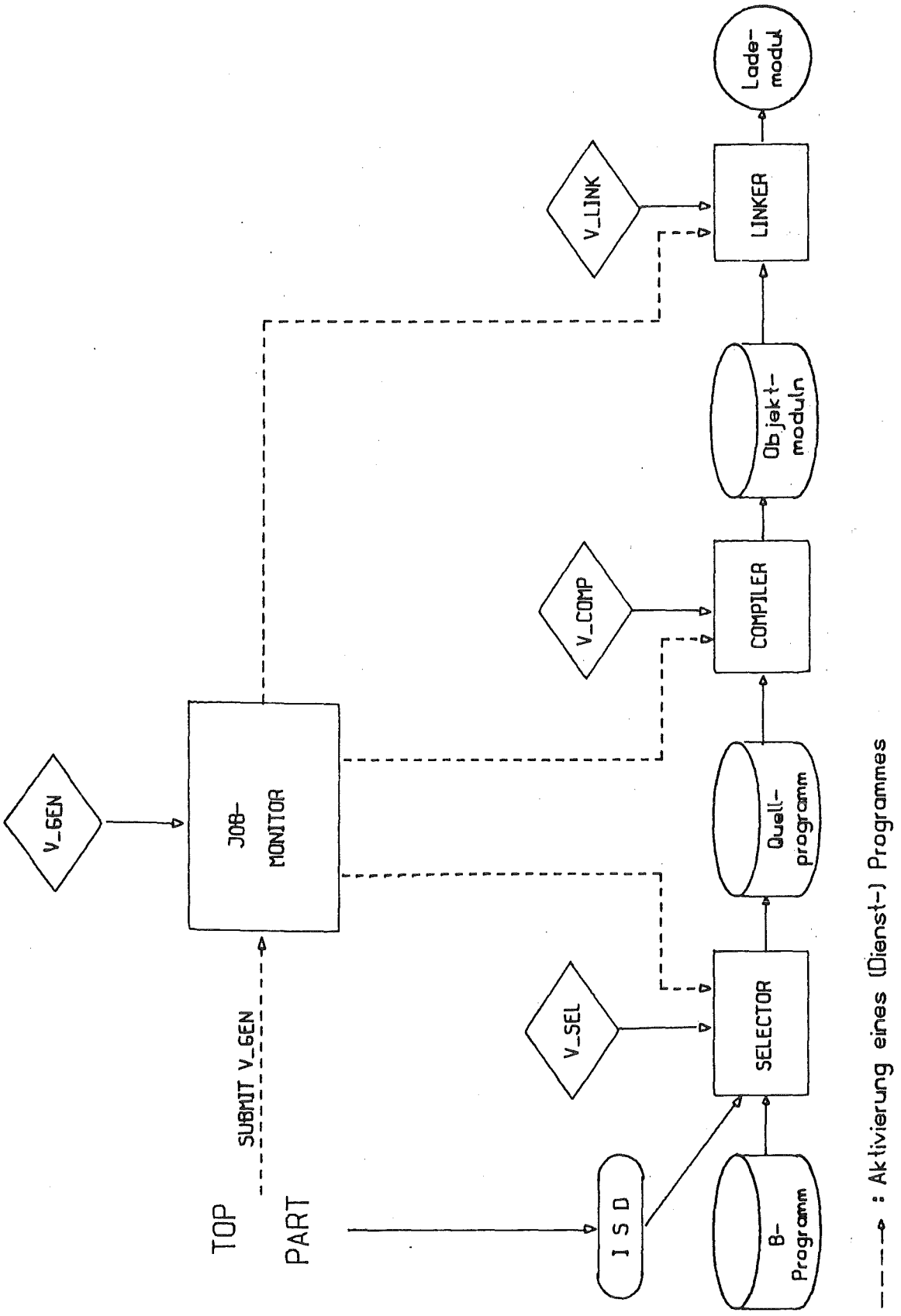


Bild 5.1: Die Erzeugung von Versionen: Spezialfall

lesen lassen, alle anderen liefern ja zur Quellprogrammerzeugung keinen Beitrag. Dies erfordert versionsspezifische Steuerprogramme zur Quellprogrammerzeugung mit SELECT-Anweisungen nur zu relevanten B-Programm-Dateien.
Ist das Selektor-Steuerprogramm in diesem Sinne versionsspezifisch, dann muß konsequenterweise ebenfalls das Steuerprogramm für den Übersetzer versionsspezifisch sein: ein Übersetzungslauf ist nur für die durch den Selektor erzeugten Quellprogramm-Dateien sinnvoll!

Für den allgemeinen Fall muß das Programmgeneriersystem damit noch zusätzlich zur Abwicklung der Schritte Quellprogrammerzeugung, Übersetzung und Binden versionsspezifische Steuerprogramme erzeugen. Dies läßt sich mit den schon bisher erforderlichen Mitteln, d.h. Selektor und ISD, realisieren, wenn folgende Eigenschaft gegeben ist:

> Für jede Version sind die Steuerprogramme Teilstrings der Steuerprogramme des vollständigen Systems.

Da die von den Dienstprogrammen zur Erzeugung einer Version auszuführenden Aufträge stets eine Teilmenge der Aufträge zur Erzeugung des vollständigen Systems sind, wird hiermit lediglich gefordert, daß es die Syntax der Steuerprogramme ermöglicht, durch Auswählen von Teilstrings die versionsspezifischen Steuerprogramme aus den entsprechenden Steuerprogrammen des vollständigen Systems zu erhalten.
Für die gängigen Jobkontrollsprachen stellt dies keine Einschränkung dar.

Dann kann die Erzeugung der Steuerprogramme wie die Quellprogrammerzeugung durch den Selektor erfolgen:
Es werden als weitere Komponenten des Generiersystems drei B-Steuerprogramme B_SEL, B_COMP, B_LINK eingeführt, aus denen der Selektor anhand der ISD die versionsspezifischen Steuerprogramme erstellt. Diese B-Steuerprogramme erhält man aus den Steuerprogrammen V_SEL, V_COMP, V_LINK des vollständigen Systems, indem die für bestimmte Versionen überflüssigen Teile durch Blöcke ersetzt werden.
Das für die Erzeugung der Steuerprogramme erforderliche Selektor-Steuerprogramm JCL_GEN ist versionsinvariant; es besteht aus den drei Anweisungen

```
SELECT FROM B_SEL  INTO   V_SEL
SELECT FROM B_COMP INTO   V_COMP
SELECT FROM B_LINK INTO   V_LINK
```

Da die Steuerprogrammerzeugung ebenfalls vom Monitor zu veranlassen ist, besteht V_GEN hier aus den folgenden Monitor-Anweisungen:

```
EXEC SELECTOR WITH JCL_GEN
EXEC SELECTOR WITH   V_SEL
EXEC COMPILER WITH   V_COMP
EXEC LINKER   WITH   V_LINK
```

Auch V_GEN ist versionsinvariant.

Bild 5.2 zeigt alle Komponenten des Programmerzeugungssystems, die Nummerierung der Monitor-Aktionen geben den Ablauf an.

Die Überprüfung, ob eine korrekte Version vorliegt, erfolgt vor der

Erzeugung der Steuerprogramme und führt daher gegebenenfalls an dieser Stelle zum Abbruch der Programmerzeugung.

Das Generiersystem zur Erzeugung von Versionen des Nucleus des DBMS FADABS /Po 78/ ist nach Bild 5.2 implementiert /Po 80/: als Selektor dient ein allgemeiner Makro-Übersetzer (im wesentlichen Strachey's GPM /St 65/), die ISD und Blöcke des B-Programms, damit nach 5.3. auch die Restriktionen, sind als Makro-Definitionen bzw. -Aufrufe realisiert.

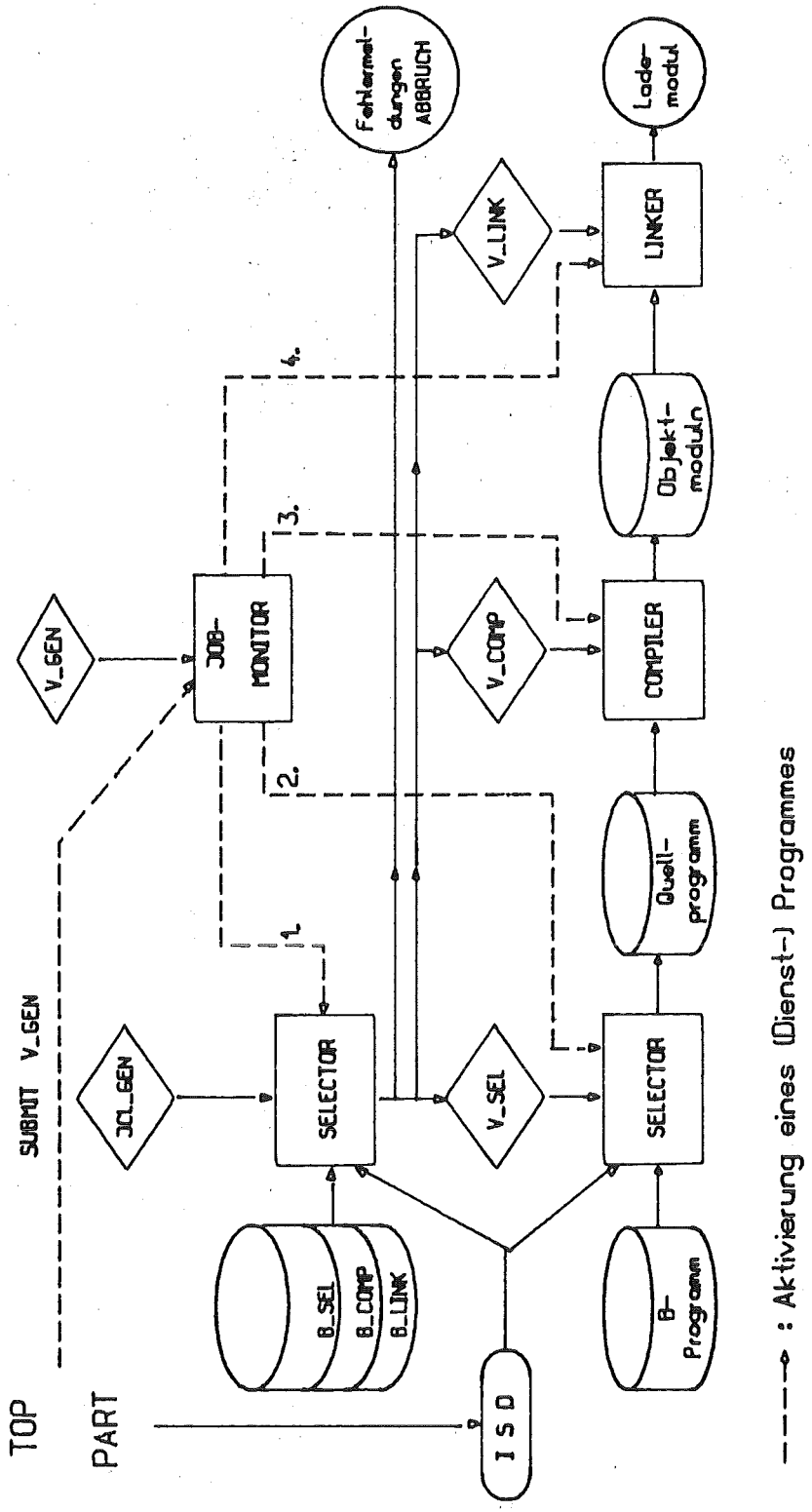


Bild 5.2: Die Erzeugung von Versionen eines Programmsystems

6. Verträgliche Teilsysteme und Versionen eines GDBMS

Es sei GDBMS ein generierbares Datenbanksystem, DML die Menge der Operationen von GDBMS, d.h. des vollständigen Teilsystems von GDBMS (vgl.: 1.3., 2.2., Hinweis von 4.4.); DB sei eine Datenbank.
Nach Abschnitt 1.3 und 2.2 geht man zur Erstellung eines Anwendersystems (vgl. Bild 1.1), im folgenden auch einfach "Anwendung" genannt, von der durch DB und DML gegebenen Schnittstelle (vgl. 1.2.) aus, beschafft sich dann aber eine Version V von GDBMS, die nur die von diesem Anwendersystem erforderlichen Operationen aus DML und diese auch nur für den jeweils erforderlichen Teil ihres Definitionsbereichs ("Menge der zulässigen DB-Zustände" in 1.2.) implementiert:
Bezeichnet $\Delta(o)$ für $o \in DML$ den Definitionsbereich von o, o(x) die Anwendung von o auf x, o(V,x) die Anwendung der Implementierung von o durch V auf $x \in \Delta(o)$, dann muß $o(V,x)=o(x)$ sein, d.h. für das Anwendersystem darf sich bei Einsatz von V anstelle des vollständigen Systems nichts an der Semantik von o ändern!

Es kann also nicht jede beliebige Version von GDBMS als Komponente DBMS eines DB-Anwendungssystems (Bild 1.1) mit einem Anwendersystem A und einer Datenbank DB eingesetzt werden:
a) V muß mindestens die von A benötigten DML-Operationen bereitstellen.
b) V muß alle Algorithmen des vollständigen Systems enthalten, die zur Ausführung der von V realisierten DML-Operationen auf DB benötigt werden.
c) Die Werte der Platzhalter von V müssen dem internen Schema von DB entsprechen.

a) und b) sind Aussagen über die Algorithmen einer Version V, nach 2.3. also Anforderungen an das Teilsystem t, aus dem V durch Dimensionierung erzeugt wird; diese Punkte werden in 6.1. und 6.2. daher für Teilsysteme untersucht. Zur Charakterisierung von Versionen, die zu einer Anwendung und einer Datenbank a), b) und c) genügen, wird in 6.3. der Begriff "Verträglichkeit" eingeführt.

Es werden die folgenden Bezeichnungen wie in den Abschnitten 3 und 4 benutzt:

F: die Menge der Fragmente von GDBMS
T: die Menge der Teilsysteme von GDBMS
$\tau(t)$: die Darstellung von $t \in T$ (s. Abschnitt 4.4)
I(f): die Menge der Indizes der charakteristischen Fragmente der CF-Darstellung von f (s. Abschnitt 4.4)
I(GDBMS): die Menge aller Indizes der charakteristischen Menge, also: $I(GDBMS)=\{i \mid 1 \leq i \leq n\}$, wobei n die Mächtigkeit der charakteristischen Menge ist

6.1. Die Operationen eines Teilsystems

Gegeben sei eine Anwendung A, OA $\subseteq$ DML sei die Menge der von A erforderlichen DML-Operationen (vgl. Abschnitt 2.2). Damit eine Version V von GDBMS mindestens die Operationen von OA realisiert, muß das Teilsystem t, aus dem V durch Dimensionierung erzeugt wird, die Algorithmen, und damit die Fragmente für mindestens eine der Realisierungsmöglichkeiten des vollständigen Systems enthalten.

Zur Beschreibung der Teilsysteme, die eine Menge O von Operationen realisieren, wird die Eigenschaft R1 benötigt:

R1: Für jede Operation o ε DML des GDBMS gibt es ein $f_o \varepsilon F$, so daß für die Relevanz ρ_o von f_o und jedes t ε T gilt:

(R1.1) $\rho(t,f_o)=1$ <==> t realisiert o ε DML

und

(R1.2) für jedes f ε F mit dieser Eigenschaft ist $\rho_f \equiv \rho_o$

Anschaulich besagt R1, daß es zu jeder Operation o der Anwenderschnittstelle mindestens ein Fragment, also einen Teilstring des vollständigen Programmtextes gibt, der stets zur Ausführung von o benötigt wird und zur Ausführung keiner weiteren Operation notwendigerweise erforderlich ist; gibt es für o mehrere Fragmente mit dieser Eigenschaft, so stimmen diese in der Relevanz überein.

Beispiel:
Die Fragmente f_o für die Operationen des Beispielsystems von Abschnitt 3.1 (vgl. Bild 3.4 und Bild 4.2):

Operation o	Fragmente f_o		
OPEN	1.1	2	11
CLOSE	1.2	9	12
FIND	1.3	3	4
GET	1.4		
INSERT	1.5	8	
DELETE	1.6	10	

Setzt man für o ε DML $I_o := I(f_o)$, so folgt wegen R1.2 nach Abschnitt 4:

t realisiert o ε DML <==> es gibt ein i ε I_o mit $\tau(t)[i]=1$

Es ergibt sich hieraus:

R2: t ε T realisiert O $\subseteq$ DML <==> für jedes o ε O gibt es ein i ε I_o mit $\tau(t)[i]=1$

Weiterhin:
t ε T realisiert o ε DML nicht <==> $\tau(t)[i]=0$ für alle i ε I_o

$T(A) \subseteq T$ bezeichne die Menge der Teilsysteme, die die Operationen OA der Anwendung A realisieren; mit R2 erhält man unmittelbar

$$T(A) = \{ t \mid t \in T, \text{ für jedes } o \in OA \text{ gibt es ein } i \in I_o \text{ mit } \tau(t)[i]=1 \}$$

Beispiel:
Die Mengen I_o des Beispielsystems von Abschnitt 3.1 (vgl. Abschnitt 4.4, Anhang 2):

Operation o	I_o
OPEN	{ 1 }
CLOSE	{ 3 }
FIND	{ 10 , 11 , 12 }
GET	{ 5 }
INSERT	{ 13 , 14 }
DELETE	{ 16 , 17 }

Für die Darstellung τ des Teilsystems t_ins gilt (vgl. 4.4.): $\tau[1]=1$, $\tau[3]=1$, $\tau[13]=1$; t_ins realisiert also die Operationen OPEN, CLOSE und INSERT.

Es seien A_1 und A_2 zwei Anwendungen mit den erforderlichen Operationen OA_1 bzw. OA_2. Die Menge der Teilsysteme, die die Operationen von A_1 und A_2 realisieren, ist

$$T(A_1)*T(A_2) =$$
$$= \{ t \mid t\varepsilon T, \text{ für alle } o \in OA_1 \text{ gilt: es gibt ein } i \in I_o \text{ mit } \tau(t)[i]=1 \} *$$
$$* \{ t \mid t\varepsilon T, \text{ für alle } o \in OA_2 \text{ gilt: es gibt ein } i \in I_o \text{ mit } \tau(t)[i]=1 \}$$
$$= \{ t \mid t\varepsilon T, \text{ für alle } o \in OA_1+OA_2 \text{ gilt: es gibt ein } i \in I_o \text{ mit } \tau(t)[i]=1\}$$

Bezeichnet man mit A_1*A_2 eine Anwendung mit den benötigten Operationen OA_1+OA_2, so ist damit gezeigt:

$$T(A_1)*T(A_2) = T(A_1*A_2)$$

d.h. realisiert $t \in T$ die Operationen von zwei Anwendungen A_1 und A_2, dann kann man A_1 und A_2 zu e i n e r Anwendung zusammenfassen, deren Operationen auch von t realisiert werden.

6.2. Die Algorithmen eines Teilsystems

Damit für eine Version V $o(V,x)=o(x)$ für $x \varepsilon \Delta(o)$ gilt, muß das Teilsystem t, aus dem V durch Dimensionierung erzeugt wird, alle Algorithmen enthalten, die das vollständige System hierfür benötigt. Es wird daher definiert:

DEFINITION 6.1:
$t \varepsilon T$ realisiere die Operationen $O \subseteq DML$, DB sei eine Datenbank. t heißt für DB g e e i g n e t , wenn t für jedes $o \varepsilon O$ alle Algorithmen enthält, die das vollständige System zur Ausführung von o auf DB benötigt.

T(DB) bezeichne die für eine Datenbank DB geeigneten Teilsysteme; das vollständige Teilsystem von GDBMS ist per definitionem für jede Datenbank geeignet, d.h. $T(DB) \neq \emptyset$.

Zur Bestimmung von T(DB):
$t \varepsilon T$ realisiere $o \varepsilon DML$. Zur Bestimmung der Algorithmen, also der Fragmente, die t zur Ausführung von o enthalten muß, genügt nach Abschnitt 4 die Ermittlung der charakteristischen Fragmente, die das vollständige System hierzu benötigt. Mit diesen sind nach Abschnitt 4.4 die Komponenten von $\tau(t)$ gegeben, die notwendigerweise den Wert 1 haben müssen, damit t die Operation o ausführen kann.
$I_{o,DB}$ bezeichne die Menge der Indizes der zur Ausführung von o zu einer Datenbank DB erforderlichen charakteristischen Fragmente des vollständigen Teilsystems.
Realisiert t die Operationen $O \varepsilon DML$, dann ist

$$+_{o \varepsilon O} I_{o,DB}$$

die Menge der Indizes der charakteristischen Fragmente, die zur Anwendung aller von t realisierten Operationen auf den jeweiligen Definitionsbereich bereitzustellen sind. Man erhält somit:

$$T(DB) = \{ t \mid t \varepsilon T, \tau(t)[i]=1 \text{ für jedes } i \varepsilon +_{o \varepsilon O} I_{o,DB} \}$$

Man beachte: $+_{o \varepsilon O} I_{o,DB}$ und damit T(DB) hängen vom internen Schema der Datenbank ab!

6.3. Verträglichkeit

Eine Version, die als Komponente eines DB-Anwendungssystems eingesetzt werden kann, muß mindestens die vom Anwendersystem benötigten DML-Operationen bereitstellen. Weiter wird gefordert, daß sie alle von ihr realisierten Operationen auf die Datenbank anwenden kann. Dies führt zu folgender Definition:

DEFINITION 6.2: Verträglichkeit von Teilsystemen
Es sei A eine Anwendung mit den erforderlichen Operationen OA, DB eine Datenbank.
$t \varepsilon T$ heißt mit (A,DB) v e r t r ä g l i c h , wenn t die Operationen OA realisiert und für DB geeignet ist.

Die Menge der mit (A,DB) verträglichen Teilsysteme ist also

$$T(A) * T(DB)$$

Es seien A_1 und A_2 zwei Anwendungen; mit 6.1. erhält man für die Menge der mit (A_1,DB) u n d (A_2,DB) verträglichen Teilsysteme

$$(T(DB)*T(A_1)) * (T(DB)*T(A_2)) = (T(DB) * (T(A_1)*T(A_2)) = T(DB) * T(A_1*A_2)$$

d.h.: ist $t \varepsilon T$ mit zwei Anwendungen A_1 und A_2 zur gleichen Datenbank DB verträglich, dann ist t auch mit A_1 und A_2 als e i n e Anwendung betrachtet zu DB verträglich.

Auf T läßt sich eine Relation $\leq$ wie folgt definieren[1]:

DEFINITION 6.3:
Es seien $t_1,t_2 \varepsilon T$ Teilsysteme mit den Darstellungen τ_1 bzw. τ_2.

$$t_1 \leq t_2 \iff \tau_1[i] \leq \tau_2[i] \text{ für } i \varepsilon I(GDBMS)$$

wobei $\leq$ auf $B=\{0,1\}$ wie folgt definiert ist: $0\leq0$, $0\leq1$, $1\leq1$.

Bemerkung:
$\leq$ auf T ist reflexiv, antisymmetrisch und transitiv, also eine Ordnungsrelation.

Für $t_1,t_2 \varepsilon T$ mit $t_1 \leq t_2$ ergibt sich unmittelbar:

- $t_1 \varepsilon T(A) \implies t_2 \varepsilon T(A)$
- $t_1 \varepsilon T(DB) \implies t_2 \varepsilon T(DB)$
- t_1 ist mit (A,DB) verträglich $\implies t_2$ ist mit (A,DB) verträglich

---

[1] Anstelle von $(x,y) \varepsilon \leq$ wird die übliche infix-Schreibweise $x\leq y$ verwendet.

Die Menge T(A) kann durch (bezüglich ≤) "kleinste" Teilsysteme beschrieben werden:
A sei eine Anwendung mit den erforderlichen Operationen OA. t(A) sei ein Teilsystem mit der Darstellung τ(A), so daß gilt:

V1: für jedes o ε OA gibt es ein i ε I_o mit τ(A)[i] = 1

V2: Hat t ε T die Darstellung τ und ist t≤t(A), dann gibt es ein o ε OA mit τ[i]=0 für jedes i ε I_o

Die Teilsysteme, die V1 und V2 genügen, sind die bezüglich der Ordnungsrelation ≤ kleinsten Teilsysteme, die in T(A) enthalten sind:
Für jedes t ε T(A) gibt es ein t(A) mit t(A)≤t, das V1 und V2 erfüllt.

Im allgemeinen gibt es mehrere kleinste Teilsysteme t(A):
Müssen die Teilsysteme von T z.B. einer Restriktion (vgl. 4.4.)

$$OR_{i\varepsilon I1}\tau[i]=1 \Longrightarrow OR_{i\varepsilon I2}\tau[i]=1$$

genügen mit $I1^*(+_{o\varepsilon OA}I_o)\neq\emptyset$ und $I2^*(+_{o\varepsilon OA}I_o)=\emptyset$, dann gibt es $2^{|I2|}-1$ Möglichkeiten (vgl. Satz 4.12), den Komponenten der Darstellung mit den Indizes von I2 Relevanzwerte zuzuweisen, so daß diese Restriktion erfüllt ist, und damit mindestens ebenso viele Teilsysteme mit den Eigenschaften V1 und V2!

Eine Version V entsteht aus einem Teilsystem durch Dimensionierung (Abschnitt 2.3), d.h. Festlegung der Größen von Datenobjekten, insbesondere Puffergrößen. Mit dem internen Schema einer Datenbank sind z.B. Blocklängen von Dateien der Datei-Schnittstelle, Längen von Seiten an der Speicher-Schnittstelle, etc. vorgegeben; dies bedeutet, daß die entsprechenden Puffer von Versionen, die als Komponente eines DB-Anwendungssystems eingesetzt werden können, bestimmte, mit dem internen Schema der Datenbank gegebene Mindestgrößen aufweisen müssen.

Es gelte also allgemein die folgende Eigenschaft:
Mit dem internen Schema einer Datenbank DB sind Werte $\pi_i(DB)$ gegeben, so daß für jede Version, die als Komponente eines Anwendungssystems mit der Datenbank DB eingesetzt werden kann,

$$\pi_i(DB) \leq p_i(V) \qquad 1\leq i\leq n$$

erfüllt sein muß, wobei $\{p_i \mid 1\leq i\leq n\}$ die Menge der Platzhalter von GDBMS ist und $p_i(V)$ den Wert des Platzhalters p_i für die Version V darstellt.

Insgesamt kann also eine Version V als Komponente eines Anwendungssystems mit der Anwendung A und der Datenbank DB eingesetzt werden, wenn V nach Definition 6.4 mit (A,DB) verträglich ist:

DEFINITION 6.4: Verträglichkeit von Versionen
Es sei A eine Anwendung, DB eine Datenbank und $\{p_i \mid 1\leq i\leq n\}$ die Menge der Platzhalter von GDBMS.
Eine Version V heißt mit (A,DB) verträglich, wenn das Teilsystem, aus dem V erzeugt wird, mit (A,DB) verträglich ist und für 1≤i≤n gilt:

$$\pi_i(DB) \leq p_i(V)$$

7. Zusammenfassung, Ausblick

In dieser Arbeit wurde die Frage untersucht, wie ein allgemeines DBMS, das Funktionen und Fähigkeiten für eine weites Spektrum von Anwendungen bieten muß, an die speziellen Anforderungen einer Anwendung angepaßt werden kann, indem man eine Version erzeugt, die nur die tatsächlich benötigten Komponenten des vollständigen DBMS enthält.

Es wurde zunächst die Problematik der Generierung von Versionen eines DBMS skizziert: die Spezifikation der gewünschten Version und die eigentliche Programmerzeugung.
Letzteres wurde auf das allgemeine Problem der Erzeugung von Teilsystemen zurückgeführt und formal ein Verfahren hierfür angegeben, das es erlaubt, aus Fragmenten des Programmtextes eines umfassenden Programmsystems PS Programme für Systeme zusammenzusetzen, die nur einen Teil der Fähigkeiten von PS realisieren.
Weiter ist gezeigt worden, daß jedes Teilsystem von PS durch ein Element von B^n $(B=\{0,1\})$ dargestellt werden kann. Ein Verfahren zur Bestimmung des kleinsten n mit dieser Eigenschaft wurde entwickelt.

Die Probleme der Zerlegung eines bereits vorliegenden Programmtextes in Fragmente wurden aufgezeigt: i.allg. wird diese Aufgabe nicht vollständig automatisierbar sein, da, wie demonstriert, Kenntnis der Semantik der Anweisungen des Programms erforderlich ist und diese i.allg. nicht aus der Syntax des Programms abgeleitet werden kann.
Zu untersuchen wäre, welche Eigenschaften einer Programmiersprache eine möglichst weitgehende Automatisierung erlauben oder ob und wie im Falle der Erstellung eines neuen Programmsystems bereits mit der Programmierung durch geeignete Sprachmittel die Fragmente festgelegt werden können, so daß man de facto anstelle eines Quellprogramms von Anfang an ein B-Programm erstellt.

Ebenfalls zu untersuchen wären Fragen der Korrektheit von Teilsystemen: Die Korrektheit des vollständigen Systems vorausgesetzt, wann sind Teilsysteme, die ja aus Codeteilen des vollständigen Systems bestehen, korrekt? Inwieweit können Korrektheitsnachweise, Testdatensätze, etc. zum vollständigen System hierfür benutzt werden?

Es wurde ein Programmerzeugungssystem entwickelt, das ausgehend von der Spezifikation einer Version unter Benutzung allgemeiner, insbesondere rechnerunabhängiger Konzepte, automatisch den zugehörigen Lademodul erzeugt. Dabei zeigt es sich, daß zur Steuerung der Programmerzeugung die gleiche Technik, d.h. Zusammenbau von Fragmenten hier zu JCL-Programmen, wie zur Erzeugung des Programmtextes eingesetzt werden kann.

Schließlich wurde das Problem der Verträglichkeit einer Version eines GDBMS mit einer existierenden Datenbank behandelt und ein Kriterium hierfür angegeben.
Damit eine Version V mit einer Anwendung und Datenbank verträglich ist, wird in Abschnitt 6 gefordert, daß V a l l e von ihm realisierten Operationen auf die g e s a m t e Datenbank ausführen kann. Es ist somit

gewährleistet, daß zumindest die interne Konsistenz der Datenbank erhalten bleibt (vgl. Bemerkung in 3.2.1.1.), stellt aber in zweierlei Hinsicht eine häufig zu starke Forderung dar:

a) bei M-Dedizierung genügt es, nur den Ausschnitt der Datenbank zu betrachten, der in der jeweiligen Phase benötigt wird, in der Regel eine echte Teilmenge der gesamten Datenbank
b) Es kann sinnvoll sein, auf Rechnersystemen, mit denen mehrere Datenbanken verwaltet werden (vgl. Beispiel von 1.1.), für mehrere, verschiedene Anwendungen zusammen eine Version zu betreiben. Hier könnte man versuchen, zur Definition von "verträglich" nur zu verlangen, daß die von der Anwendung geforderten Operationen auf die (jeweilige) Datenbank ausgeführt werden können.

Es wäre zu überprüfen,

- ob und in welcher Umgebung derartige Abschwächungen der Definition von "Verträglichkeit" ausreichen
- mit welchen Konzepten und Mitteln in diesem Fall Fehlertoleranz und Zuverlässigkeit gewährleistet werden kann, ohne daß der hierzu erforderliche Mehraufwand die Vorteile des Einsatzes eines GDBMS zunichte macht oder gar ins Gegenteil verkehrt.

Zur Überprüfung, ob eine Version mit einer Datenbank verträglich ist, ist Kenntnis des internen Schemas erforderlich. Es bleibt daher zu untersuchen, welche Konsequenz sich hieraus für die Architektur eines GDBMS ergeben. Man wird fordern müssen, daß der Verträglichkeits-Check von jeder Version, etwa mit jedem Starten des Anwendungssystems vorgenommen wird: Man kann nicht davon ausgehen, daß in einem Rechnersystem genau eine Datenbank existiert, in der Regel wird es mehrere geben, wenn auch zu jedem Zeitpunkt höchstens eine in Bearbeitung ist (vgl. Beispiel in 1.1.); selbst für den Fall, daß es genau eine Datenbank gibt, ist zu bedenken, daß sich das interne Schema bei R- bzw. M-Dedizierung (1.3.) ändern kann. Es ist daher zu fordern, daß jede Version eines GDBMS in der Lage sein muß, zumindest lesend auf das interne Schema, also die Kontrollinformation (Abschnitt 2.2), zuzugreifen: sie muß die entsprechenden Algorithmen, also Fragmente, enthalten.

Schlußwort

Die Grundideen für die vorliegenden Arbeit ergaben sich aus der Entwicklung des Datenbanksystems FADABS /Po 78/ am Kernforschungszentrum Karlsruhe und den Erfahrungen mit seinem Einsatz in Informationssystemen des KfK zur Kernmaterialüberwachung im Rahmen von PKÜ.

An dieser Stelle möchte ich Herrn Prof. Dr. H. Trauboth danken, der mir die Anfertigung dieser Arbeit ermöglichte und mich mit wertvollen Ratschlägen dabei unterstützte.
Herrn Prof. Dr. P.C. Lockemann danke ich für das Interesse, das er dieser Arbeit entgegengebracht hat; seine kritischen Anmerkungen und die Diskussionen mit ihm haben die Arbeit in der vorliegenden Form erst möglich gemacht.

Herrn Dr. Jaeschke und Herrn Dipl.-Phys. Weidemann sei für ihre Unterstützung, vor allem während des letzten Jahres, gedankt, die es mir erlaubt hat, diese Arbeit zügig durchzuführen und zu beenden.

Anhang 1: Begriffe zu Mengen, Abbildungen, Graphen

Es werden die in dieser Arbeit verwendeten mathematischen Begriffe zu Mengen, Abbildungen und Graphen zusammengestellt.

M, N seien Mengen:

Die M ä c h t i g k e i t |M| einer Menge M gibt die Anzahl der Elemente von M an.

Die P o t e n z m e n g e ¶(M) einer Menge M ist die Menge aller Teilmengen von M.

$\Pi_M \subseteq \P(M)$ ist eine d i s j u n k t e Z e r l e g u n g einer Menge M, wenn gilt:

1. für jedes $x \in M$ gibt es $\pi \in \Pi_M$ mit $x \in \pi$
2. $\pi_1, \pi_2 \in \Pi_M \Longrightarrow \pi_1 = \pi_2$ oder $\pi_1 * \pi_2 = \emptyset$

Das cartesische Produkt von M und N ist die Menge

$$M \times N = \{ (m,n) \mid m \in M, n \in N \}$$

Eine Menge $R \subseteq M \times N$ heißt eine (binäre) Relation zwischen M und N. $R \subseteq M \times M$ heißt eine O r d n u n g s r e l a t i o n über M, wenn gilt:

- $(x,x) \in R$ für alle $x \in M$ (Reflexivität)
- $(x,y) \in R, (y,x) \in R \Longrightarrow x=y$ (Antisymmetrie)
- $(x,y) \in R, (y,z) \in R \Longrightarrow (x,z) \in R$ (Transitivität)

Eine Menge, für die es eine Ordnungsrelation gibt, heißt eine g e o r d n e t e Menge oder auch eine F o l g e . Ist $L \subseteq M$ eine Folge, dann bezeichnet L[i] das i-te Element oder auch die i-te Komponente von L.
Die Ordnung einer geordneten Menge L wird auch durch folgende Schreibweise zum Ausdruck gebracht: $L = < L[1], \ldots, L[i], \ldots >$.

Eine A b b i l d u n g $f: M \dashrightarrow N$ ist eine Relation $f \in M \times N$ mit der Eigenschaft

$$(x,y) \in f, (x,z) \in f \Longrightarrow y=z$$

M heißt der D e f i n i t i o n s b e r e i c h von f; ist $(x,y) \in f$, so heißt y das Bild der Abbildung für x und wird mit f(x) bezeichnet.

Zwei Abbildungen $f: M \dashrightarrow N$, $g: M \dashrightarrow N$ heißen gleich, notiert: $f \equiv g$, wenn $f(x)=g(x)$ für alle $x \in M$ gilt.

Ein g e r i c h t e t e r G r a p h ist ein Paar G=(M,R) aus einer Menge M und einer binären Relation R $\subseteq$ M×M. Die Elemente von M heißen K n o t e n , die von R K a n t e n .

k,k_1,k_2 seien Knoten eines gerichteten Graphen (M,R):

- Die V o r g ä n g e r von k sind die Knoten der Menge
  PRED(k) := { x | x ε M, (x,k) ε R }
- Die N a c h f o l g e r von k sind die Knoten der Menge
  SUCC(k) := { x | x ε M, (k,x) ε R }
- Ein P f a d P von x nach y ist eine Folge von n≥2 Knoten k_i, 1≤i≤n, mit (k_i,k_{i+1}) ε R für 1≤i≤n-1 und k_1=x, k_n=y. Ist x=y, so heißt P ein Z y k l u s oder z y k l i s c h e r P f a d .
  n-1, also die Zahl der Kanten des Pfades, wird die L ä n g e von P genannt.
  Der Einfachheit halber, insbesondere, wenn die Ordnung der Knoten unwesentlich ist, wird mit P auch die Menge $\{k_i \mid 1\le i\le n\}$ der Knoten des Pfades P bezeichnet.
  Ein Pfad P' heißt ein T e i l p f a d eines Pfades P, wenn die Folge P' eine Teilfolge von P ist.
  Ein Pfad P heißt a z y k l i s c h , wenn es keinen Teilpfad von P gibt, der ein Zyklus ist.
- k_2 heißt von k_1 aus e r r e i c h b a r , wenn es einen Pfad von k_1 nach k_2 gibt. k heißt von M' $\subseteq$ M erreichbar, wenn es von jedem Element von M' einen Pfad nach k gibt.
- k_2 hat den A b s t a n d n von k_1, wenn es einen azyklischen Pfad von k_1 nach k_2 mit n Knoten gibt.

Ein B a u m ist ein gerichteter Graph T=(M,R) mit folgenden Eigenschaften:

1. T enthält keine Zyklen
2. es gibt genau ein r ε M mit PRED(r)=∅; r heißt die W u r z e l von T
3. k ε M, k≠r ==> |PRED(k)|=1
4. zu jedem Knoten k ε M, k≠r gibt es einen Pfad von r nach k.

Knoten eines Baumes mit Nachfolgern heißen i n n e r e Knoten, die Knoten ohne Nachfolger heißen B l a t t k n o t e n .

Ein Baum T=(M,R) heißt g e o r d n e t , wenn für jedes k ε M die Menge SUCC(k) geordnet ist.
Eine explizite Beschreibung der Ordnung der Mengen SUCC(k) eines geordneten Baumes (M,R) ermöglichen die folgenden Bezeichnungen:

- ist |SUCC(k)|>0, so bezeichnen k[i], 1≤i≤|SUCC(k)| die Nachfolger von k; für k_1,k_2 ε SUCC(k) mit k_1=k[i], k_2=k[j] gilt: $k_1<k_2$ <==> i < j
- für k ε M mit dem Vorgänger k' bezeichnet ν(k) den I n d e x von k in der geordneten Menge SUCC(k'): k = k'[ν(k)]
- ist k_1,k_2 ε SUCC(k) und $\nu(k_1)+1=\nu(k_2)$, dann heißt k_1 der "linke Nachbar" von k_2 und k_2 der "rechte Nachbar" von k_1.

ν ist damit eine Abbildung M --> { i | $1\le i\le \max_{k\varepsilon M}|SUCC(k)|$ }

Anhang 2: Die Abbildung ρ des Beispielsystems

Es werden für jedes Fragment f des Fragmentsystems zum Beispielsystem DBMS von Abschnitt 3 die Mengen C(f) und I(f) nach 4.3.5.4. bzw. 4.4. tabellarisch zusammengestellt. Weiterhin wird für jedes Fragment f des B-Programms (Abschnitt 3.3) der Relevanzwert ρ(t_ins,f) zum Teilsystem t_ins angegeben.

Zu diesem Zweck werden aus 4.4. benötigt
• die Menge CF und die Indizierung der charakteristischen Fragmente:

i	1	2	3	4	5	6	7	8	9	10
g_i	1.1	2.1	1.2	9.1	1.4	5.2	5.3	6.1	6.2	4.1

i	11	12	13	14	15	16	17	18
g_i	4.2.1	4.2.2	8.1	8.2	8.3	10.1	10.2	10.3

• die Darstellung des Teilsystems t_ins:
τ(t_ins) = (1,0,1,0,0,0,1,1,0,0,0,0,1,0,0,0,0,0)

t_ins ist das Teilsystem mit den charakteristischen Fragmenten 1.1, 1.2, 5.3, 6.1 und 8.1; es gilt somit für die Fragmente f des Fragmentsystems:
ρ(t_ins,f)=1 <==> {1.1,1.2,5.3,6.1,8.1}*C(f)≠∅ <==> {1,3,7,8,13}*I(f)≠∅

f	C(f)	I(f)	ρ(t_ins,f)
1	{ 1.1 , 1.2 , 1.4 , 4.1 , 4.2.1 , 4.2.2 , 8.1 , 8.2 , 10.1 , 10.2 }	{ 1, 3, 5 , 10, 11, 12, 13, 14, 16, 17 }	1
1.1	{ 1.1 }	{ 1 }	1
1.2	{ 1.2 }	{ 3 }	1
1.3	{ 4.1 , 4.2.1 , 4.2.2 }	{ 10, 11, 12 }	0
1.4	{ 1.4 }	{ 5 }	0
1.5	{ 8.1 , 8.2 }	{ 13, 14 }	1
1.6	{ 10.1,10.2 }	{ 16, 17 }	0
2	{ 1.1 }	{ 1 }	1
2.1	{ 2.1 }	{ 2 }	0
3	{ 4.1 , 4.2.1 , 4.2.2 }	{ 10, 11, 12 }	0
4	{ 4.1 , 4.2.1 , 4.2.2 }	{ 10, 11, 12 }	0
4.1	{ 4.1 }	{ 10 }	0
4.2	{ 4.2.1 , 4.2.2 }	{ 11, 12 }	0
4.2.1	{ 4.2.1 }	{ 11 }	0
4.2.2	{ 4.2.2 }	{ 12 }	0
5	{ 6.1 , 6.2. , 5.2 }	{ 8, 9, 6 }	1
5.1	{ 6.1 , 6.2 }	{ 8, 9 }	1
5.2	{ 5.2 }	{ 6 }	0
5.3	{ 5.3 }	{ 7 }	1
6	{ 6.1 , 6.2 }	{ 8, 9 }	1

6.1	{ 6.1 }	{ 8 }	1
6.2	{ 6.2 }	{ 9 }	0
7	{ 5.2 }	{ 6 }	0
8	{ 8.1 , 8.2 }	{ 13, 14 }	1
8.1	{ 8.1 }	{ 13 }	1
8.2	{ 8.2 }	{ 14 }	0
8.3	{ 8.3 }	{ 15 }	0
9	{ 1.2 }	{ 3 }	1
9.1	{ 9.1 }	{ 4 }	0
10	{10.1,10.2 }	{ 16, 17 }	0
10.1	{10.1 }	{ 16 }	0
10.2	{10.2 }	{ 17 }	0
10.3	{10.3 }	{ 18 }	0
11	{ 1.1 }	{ 1 }	1
12	{ 1.2 }	{ 3 }	1
13	{ 2.1 }	{ 2 }	0
14	{ 9.1 }	{ 4 }	0
15	{ 8.1 }	{ 13 }	1
16	{ 8.2 }	{ 14 }	0
17	{10.1 }	{ 16 }	0
18	{10.2 }	{ 17 }	0
19	{ 8.3 }	{ 15 }	0
20	{10.3 }	{ 18 }	0
21	{4.2.2}	{ 12 }	0
22	{ 2.1 , 4.1 , 4.2.1 , 4.2.2 , 9.1 , 8.3 , 10.3 }	{ 2, 4, 10, 11, 12, 15, 18 }	0
22.1	{ 2.1 , 4.1 , 4.2.1 , 4.2.2 , 9.1 , 8.3 , 10.3 }	{ 2, 4, 10, 11, 12, 15, 18 }	0

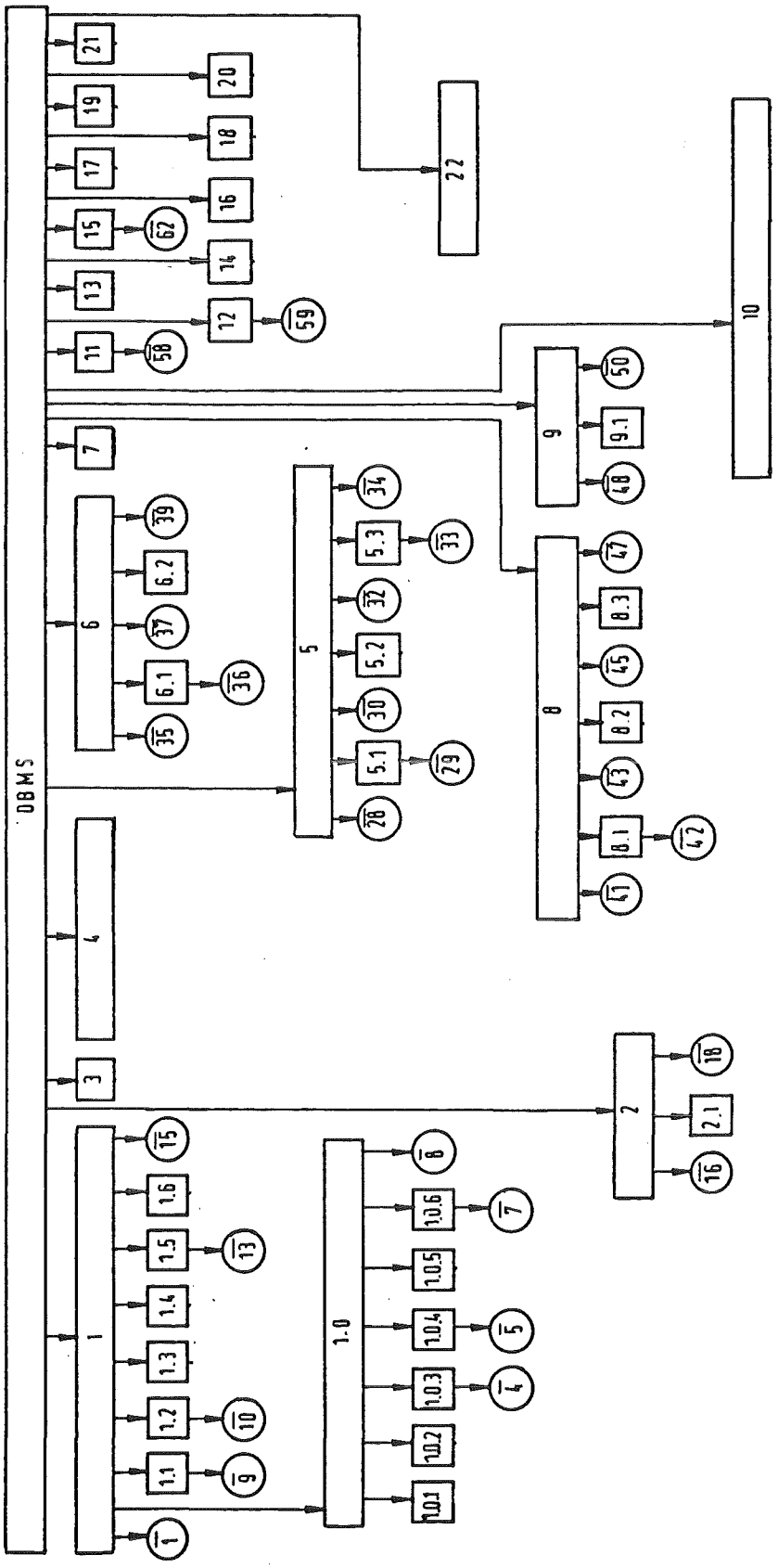
Das Fragment 1.0 und seine Unterfragmente sind n i c h t Fragmente des Fragmentsystems (vgl. 4.1., g), für diese werden die Relevanzfunktionen nach 3.2.1.3. angegeben:

f	ρ_f	ρ(t_ins,f)
1.0.1	NOT $\rho_{1.1}$	0
1.0.2	NOT $\rho_{1.2}$	0
1.0.3	NOT $\rho_{1.3}$	1
1.0.4	NOT $\rho_{1.4}$	1
1.0.5	NOT $\rho_{1.5}$	0
1.0.6	NOT $\rho_{1.6}$	1
1.0	$\rho_{1.0.1}$ OR $\rho_{1.0.2}$ OR $\rho_{1.0.3}$ OR $\rho_{1.0.4}$ OR $\rho_{1.0.5}$ OR $\rho_{1.0.6}$	1

wobei für x ε F NOT ρ_x wie folgt definiert ist (Negation von 3.2.1.3.):

$$\text{NOT } \rho(t,x) = \begin{cases} 0 : \rho(t,x)=1 \\ 1 : \rho(t,x)=0 \end{cases}$$

Der für t_ins relevante Teilbaum (P_{t_ins},S_{t_ins}) ist damit (vgl. Definition 3.5, Bild 3.7):



Literaturverzeichnis

/ADA/ Preliminary ADA Reference Manual. ACM SIGPLAN Notices 14,6 (June 1979)

/AU 72/ Aho, A.V.; Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Inc. 1972

/AH 74/ Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, 1974

/Ba 73/ Bachman, C.W.: The Programmer as Navigator. CACM 16,11 (Nov. 1973), p.653-658

/BG 74/ Bauer, F.L., Goos, G.: Informatik, Eine Einführende Übersicht, Zweiter Teil. Heidelberger Taschenbücher, Band 91. Springer-Verlag, Berlin, 1974.

/Bo 81/ Boehm, B.W.: Improving software productivity. Proc. IEEE COMPCON 81. Washington D.C., Sept. 15-17, 1981

/Br 73/ Brinch Hansen, P.: Operating Systems Principles. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973

/BW 80/ Bernstorff, P.; Weber, W.: Die Konzeption eines interaktiven Datenbanksystems für Kleinrechner. Informatik-Spektrum 3 (1980), p. 181-190

/CM 82/ Computer magazin, Heft Juni 1982, div. Beiträge zum Thema "Personalcomputer"

/CN 80/ CNR; GMD; INRIA; NCC: Beschreibung von Datenbanksystemen, Systeme für kleinere DV-Anlagen. GMD Schloß Birlinghoven, St. Augustin, 1980

/Da 77/ Date, C.J.: An Introduction to Database Systems. 2nd ed. Addison-Wesley Publishing Company, Inc. 1977

/De 77/ Desjardins, R.: Die Gestaltung der Datenbasis in Dispositionssystemen auf Prozeßrechnern. Angewandte Informatik, 3/77, p. 101-108

/De 79/ Desjardins, R.: Datenbanksysteme für Minicomputer - Anforderungen und Architekturprinzipien. In: Datenbanktechnologie, Berichte German Chapter of the ACM 2. Stuttgart: B.G. Teubner Verlag 1979 p. 195-206

/DK 72/ Detlefsen, G.D.; Kerr, R.H.; Revkin, S.B.: Software for Data Communications Networks. Fifth Australian Computer Conference, Brisbane, Australia, May 22-26, 1972, p. 130-135

/Fo 74/ Fosdick, L.D.: BRNANL, a Fortran Program to Identify Basic Blocks in Fortran Programs. Report CU-CS-040-74, Department of Computer

Science, University of Colorado. March 1974.

/Fi 79/ Fischer, H.: Back-End Implementierung von ADABAS auf PDP-11 Minicomputer. In: Datenbanktechnologie, Berichte German Chapter of the ACM 2. Stuttgart: B.G. Teubner Verlag 1979, p. 207-217

/FS 76/ Fry, J.P.; Sibley, E.H.: Evolution of Data-Base Management Systems. ACM Computing Surveys 8,1 (March 1976), p. 7-42

/FTN/ American National Standards Committee X3J3: draft proposed ANS FORTRAN. ACM SIGPLAN Notices, 11,3 (March 1976)

/GG 80/ Geiger, W.; Gmeiner, L.: Automatische Test- und Dokumentationshilfsmittel für FORTRAN-Programme. KfK-Nachrichten, 12(1980) No 1-2, S.23-28.

/Go 74/ Goldberg, P.C.: Automatic Programming. In: Hackl, C.E. (ed.): Programming Methodology. 4th Informatik Symposium, Wildbad, Sept. 25-27, 1974. Lecture Notes in Computer Science, vol 23. Springer Verlag, Berlin 1975, p. 347-361

/Gr 71/ Gries, D.: Compiler Construction for Digital Computers. John Wiley & Sons Inc., 1971.

/GS 70/ Gecsei, J.; Slutz, D.R.; Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Systems Journal 9,2(1970), p.78-117

/Hä 78/ Härder, T.: Implementierung von Datenbanksystemen. Carl Hanser Verlag, München 1978

/HR 80/ Härder, T.; Reuter, A.: Abhängigkeiten von Systemkomponenten in Datenbanksystemen. In: Wilhelm, R. (ed.): GI - 10. Jahrestagung, Saarbrücken, 30. September - 2. Oktober 1980. Informatik-Fachberichte Band 33. Springer Verlag, Berlin 1980. p. 243-257

/Ja 80/ Jappe, H.O.: Generatorsoftware - Schlagwort oder Notwendigkeit. Proc. Internationaler Kongress für Datenverarbeitung IKD 1980. Berlin 7.-10. Oktober 1980, p. 305-308

/JL 80/ Jaeschke, A.; Landmark, K.; Polster, F.J.; Stöckle, D.: Komponenten von Prozeßinformationssystemen. KfK-Nachrichten, 12(1980) No 1-2, S.35-39.

/JP 80/ Jaeschke, A.; Polster, F.J.; Trauboth, H.: A Generable Database System for the Use in Process Information Systems. Proc. COMSAC 1980, p. 670-675

/JW 74/ Jensen, K.; Wirth, N.: PASCAL User Manual And Report. 2nd ed., Springer Verlag. Berlin, 1978.

/Ka 78/ Kakuma, M.; et al.: Software Production and Maintenance for D10 System. Third International Conference on Software Engineering for Telecommunication Switching Systems, Helsinki, June 27-28, 1978, p. 7-12

/Kn 69/ Knuth, D.E.: The Art of Computer Programming, vol. 1. Addison-Wesley Publishing Company, 1969

/Ko 69/ Kowalsky, H.-J.: Lineare Algebra. de Gruyter & Co., Berlin 1969

/Ko 79/ de Koning, G.C.M.: Automatic Generation of Software for SPF Telephone Exchanges. Telecommunications 13,2 (March 1979), p. 19-21

/LD 81/ Lanergan, R.G.; Dugan, D.K.: Software engineering with reusable designs and code. Proc. IEEE COMPCON 81. Washington D.C., Sept. 15-17, 1981, p. 296-303

/LH 81/ Lockemann, P.C.; Härder, T.: Datenhaltung in Echtzeitsystemen. Fachtagung Prozeßrechner 1981. München 1981

/LM 78/ Lockemann, P.C.; Mayr, H.C.: Rechnergestützte Informationssysteme. Springer-Verlag. Berlin, 1978

/LM 80/ Lockemann, P.C.; Mayr, H.C.: Database Systems: are they special pieces of software? Universität Karlsruhe, Fakultät für Informatik. Interner Bericht 28/80, Okt. 1980

/Lo 77/ Lorie, R.A.: Physical Integrity in a large segmented Database. ACM Transactions on Database Systems 2, 1, (March 1977) p. 91-104

/Ne 77/ Nehmer, J.: Betriebssysteme für Kleinrechner. Angewandte Informatik, 1/77, p. 1-14

/No 73/ Nolan, R.: Computer data bases: The future is now. Harvard Business Review. Sept.-Oct. 1973, p. 98-114.

/Pe 81/ Pernards, P.: Das interaktive Programmiersystem IPDIP. Angewandte Informatik 8/81, p. 361-369

/Po 78/ Polster, F.J.: Ein Datenbanksystem für den Siemens Prozeßrechner 330. In: Voges, U. (Ed.): Tagungsbericht der 9.Jahrestagung des Siemens Prozeßrechner-Anwenderkreises I, Kernforschungszentrum Karlsruhe, KfK 2642, p.227-244.

/Po 80/ Polster,F.J.: Der Makro-Übersetzer SM30 als Software-Generator. In: Heim, K.(Ed.): Tagungsbericht der 12. Jahrestagung des Siemens Prozeßrechner-Anwenderkreises I, Universität Karlsruhe, 23.-25. März 1981, p. 207-218.

/Pr 77/ Prywes, N.S.: Automatic Generation of Computer Programs. In: Rubinoff, M.; Yovits, M.C. (ed.): Advances in Computers, vol 16. Academic Press 1977, p. 57-125

/Re 77/ Reitz, K.H.: Durch Systematisierung zu einem freiprojektierbaren Software-Bausteinsystem zur Lösung von Automatisierungsaufgaben im Sinne von Regeln, Steuern und Überwachen. Fachtagung Prozeßrechner 1977. Informatik Fachberichte 7, Springer-Verlag Berlin 1977, p. 116-119

/RS 80/ Rüb, W.; Schrott, G.: Automatische Generierung problemangepaßter Prozeßrechner-Betriebssysteme. Angewandte Informatik 1/80, p. 7-17

/Sa 80/ Salter, J.A.M.: SIBAS - Past, Present and Future. Database Journal 10,3 (1980), p. 21-24

/Sc 76/ Schmid, H.A.: Architektur und Implementierung von Datenbanksystemen. Der GMD-Spiegel, 3 (1976), p. 78-122

/Sc 77/ Schrott, G.: Generation of dedicated realtime operating systems by dialogue. IFAC/IFIP Workshop on Real Time Programming. Eindhoven, Netherlands. June 1977

/Sc 81/ Schulze, L.: persönliche Mitteilung. 1981

/SL 76/ Severance, D.G.; Lohman, G.M.: Differential Files: Their Application to the Maintenance of Large Databases. ACM TODS 1,3(Sept. 1976), p.256-267

/Se 73/ Senko, M.E.; et al.: Data structures and accessing in data-base systems. IBM Systems Journal 12,1 (Jan. 73) p. 30-93

/Se 77/ Senko, M.E.: Data structures and data accessing in data base systems - past, present, future. IBM Systems Journal 16,3 (1977) p. 208-257

/SF 78/ Slonim, J.; Farrell, M.W.; Fisher, P.S.: A Survey of Mini-Data Base Management Systems in 1977. Proceedings First SIGMINI Symposium on Small Systems. ACM SIGMINI, N.Y. 1978 p. 26-34

/St 80/ Stonebraker, M.: Retrospection on a Database System. ACM TODS 5,2 (June 1980) p. 320-326

/St 65/ Strachey, C.: A general purpose macrogenerator. The Computer Journal, vol 8 (1965-66) p.225-241

/SW 76/ Stonebraker, M.; Wong, E.; Kreps, P.: The Design and Implementation of INGRES. ACM TODS 1,3 (Sept. 1976), p.189-222

/TF 76/ Taylor, R.W.; Frank, R.L.: CODASYL Data-Base Management Systems. ACM Computing Surveys 8,1 (March 1976), p. 67-103

/TL 82/ Tsichritzis, D.C.; Lochowsky, F.H.: Data Models. Prentice Hall, Inc. 1972

/UB 77/ Urbanetz, M.; Bischof, D.: Prozeßrechner im Prüfstand. VDI Nachrichten 31,34 (26.8.1977), p.7-9

/Ur 80/ Urbanetz, M.: persönliche Mitteilung. 1980

/VG 80/ Voges, U., Gmeiner, L., Amschler von Mayrhauser, A.: SADAT - an automated testing tool. IEEE Transactions on Software Engineering, SE-6(1980) S.286-90.

/Wi 77/ Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? CACM 20,11 (Nov. 1977), p. 822-833

/Wi 78/ Wilkes, M.V.: Minicomputers, main frames and distributed systems. Proceedings First SIGMINI Symposium on Small Systems. ACM SIGMINI, N.Y. 1978 p. 1-6

/Wi 79/ Wirth, N.: Algorithmen und Datenstrukturen. B.G.Teubner, Stuttgart, 1979

/WS 82/ Wasserman, A.I.; Shewmake, D.: Automating the Development and Evolution of User Dialogue in an Interactive Information System. In: Hawgood, J. (ed.): Evolutionary Information Systems. Proceedings of the IFIP TC8 Working Conference on Evolutionary Information Systems, Budapest, Hungary, 1-3 Sept. 1981. North Holland Publishing Company