EIN VERFAHREN
ZUR
RECHNERUNTERSTÜTZTEN PROGRAMMKONSTRUKTION

Von der Fakultät Mathematik und Informatik
der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften(Dr.rer.nat.)
genehmigte Abhandlung

vorgelegt von
DALTRO JOSE NUNES
aus Orleaes

Hauptberichter: Prof.Dr.R.Gunzenhäuser
Mitberichter: Prof.Dr.E.J.Neuhold
Tag der mündlichen Prüfung: 29. März 1985

Institut für Informatik der Universität Stuttgart
1985

Vorwort

Die vorliegende Arbeit entstand während eines vierjährigen Aufenthaltes am Institut für Informatik der Universität Stuttgart.

Herrn Professor Gunzenhäuser danke ich für die wissenschaftliche Betreuung und kritische Auseinandersetzung mit dieser Arbeit. Für die Übernahme des Mitberichts danke ich Herrn Professor Neuhold.

Frau Horndasch danke ich für die aufmerksame Durchsicht meiner Arbeit.

Herrn Dr. Pletat danke ich für die fruchtbare Zusammenarbeit im CASD-Projekt.

Den Mitgliedern der Abteilung Anwendersoftware danke ich für die freundliche Aufnahme und für interessante Diskussionen.

Der 'Universidade Federal do Rio Grande do Sul', Porto Alegre, Brasilien, danke ich, daß sie mich für die Erstellung dieser Arbeit beurlaubt hat.

Der Alexander von Humboldt-Stiftung und der Alfried Krupp von Bohlen und Halbach-Stiftung danke ich für die in den Jahren 1981 bis 1982 gegebene Förderung.

Der 'Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior-CAPES' danke ich für die in den Jahren 1983 bis 1985 gegebene Förderung.

Meiner lieben Frau Suzana
und meinen Kindern
Gustavo und Ingrid
gewidmet

Inhaltsverzeichnis

I - EINLEITUNG

1 - Software-Engineering: Ein Überblick

Der Begriff des Software-Engineering ist als Reaktion auf die Softwarekrise der 60er Jahre entstanden. Schon 1975 hat Steel /Ste-75/ aus Anlaß der ersten 'IEEE National Conference' über Software Engineering ausgesagt:

> '... Software engineering today is a collection of prescriptions, techniques and disciplines that show promise of bringing some order into the chaos that is computer programming...'

Bei den Bemühungen, Ordnung in dieses Chaos zu bringen, ist als Folge ein Chaos in der Terminologie entstanden. Im folgenden wird versucht, die wichtigsten Begriffe aus diesem Fachgebiet überblicksartig darzustellen, obwohl diese Begriffe in der Fachterminologie nicht einheitlich verwendet werden.

1.1 Der Software-Lebens-Zyklus (SLZ)

Unter SLZ versteht man die verschiedenen Zustände, die ein Softwaresystem von seiner Grundidee bis zur Anwendung durchläuft oder die verschiedenen Phasen, die das Softwaresystem von einem Zustand in den nächsten überführen. Viele Autoren, unter anderem Boehm /Boe-76/, Freeman /Fre-78/ und Kimm /Kim-79/ haben den SLZ beschrieben. Aus diesen verschiedenen Beschreibungen kann man Phasen des SLZ wie folgt zusammenfassen:

- Anforderungsdefintionsphase,
- Entwurfsphase,
- Codierungsphase,
- Verifikationsphase,
- Installationsphase und
- Wartungsphase.

Nach Lewis /Lev-77/ entsprechen die verschiedenen Zustände eines Softwaresystems auch verschiedenen Sprachebenen: Für die Anfangsphase des Softwaresystems ist die Sprachebene natürlich, abstrakt und nicht prozedural; für die Endphase konkreter und prozedural (Bild 1.1).

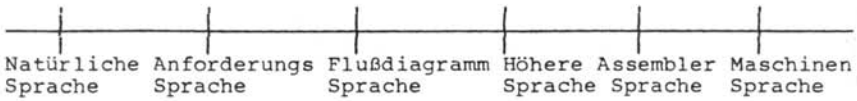


Bild 1.1: Die Beschreibungssprachen des Softwaresystems

1.1.1 Anforderungsdefinitonsphase

Boehm /Boe-77/ definiert die Anforderungsdefinition als Disziplin, um eine vollständige, konsistente und eindeutige Beschreibung des Softwaresystems festzulegen. Diese Beschreibung zeigt, was das Softwaresystem zu leisten hat. Der Beschreibung soll durch alle Beteiligten zugestimmt werden können. Diese Beschreibung soll auch möglichst fehlerfrei sein, da jeder mögliche Fehler sich in die Folgenphasen mit erheblicher Wirkung fortpflanzt.

Normalerweise wird für die Anforderungsdefinition die natürliche Sprache verwendet, obwohl auch künstliche Sprachen, wie PSL /Tei-77/, SADT /Ros77a/ und HIPO /IBM-75/ verwendet werden.

1.1.2 Entwurfsphase

Die Anforderungsdefiniton ist die Quelle für die Entwurfsphase des Software-Systems. Nach Disjkstra /Dij-76/ versucht der Software-Entwickler hier die Komplexität des Problems, das als Modul betrachtet werden kann, so zu zerlegen, daß jedes Teilproblem für sich getrennt leichter gelöst werden kann.

In der Entwurfsphase werden Verfahren wie Modularisierung, schrittweise Verfeinerung, top-down Entwurf, hierarchische Zerlegung, funktionale Zerlegung usw. mit unterschiedlicher Interpretation verwendet.

Beim Prozeß der Modularisierung wird das Softwaresystem 'top-down' in Module zerlegt. So entsteht meist ein Baum von Modulen. Da ein Modul auch rekursiv wiederverwendet werden kann, kann auch ein anderer Strukturtyp entstehen (Bild 1.2).

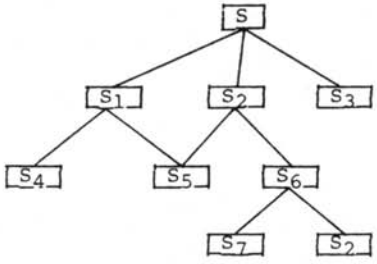


Bild 1.2: Die Modulardarstellung eines Softwaresystem

In Bild xxx stellt das Modul S das ganze Softwaresystem dar. Die Module S_1, S_2 und S_3 stellen Teilsysteme von S dar. Das Modul S_5 ist ein Bestandteil sowohl des Moduls S_1 als auch des Moduls S_2. Das Modul S_2 ist rekursiv.

Nach dem Modularisierungsprinzip von Parnas /Par-72/, das als 'information hiding' bekannt ist, ist jedes Modul so aufgebaut, daß es nur die Variablen (Schnittstelle) und die Funktionen derjenige Module kennt, mit denen es kommuniziert, nicht aber den Typ der Variablen und den Code solcher Module. So kann jedes Modul getrennt übersetzt und getestet werden.

Nach dem Modularisierungprinzip von Mayers /Mye-76,Mye-75/ muß ein Softwaresystem (Modul), das ein Problem darstellt, in eine Kette von 3 bis 10 Funktionen aufgelöst werden. Diese Kette wird durch die Transformation der Eingabedaten ('major input stream') erstellt. Durch 'points of highest abstraction' in jeder Kette wird sie in Module zerlegt. Für jedes neue Modul muß dann die Schnittstelle mit dem übergeordneten Modul festgelegt werden. Das Verfahren wird dann rekursiv angewandt.

McClure /McC-78/ zeigt, wie die Komplexität eines Softwaresystems während des Modularisierungsprozeß bestimmt werden kann, wenn für jedes Modul angegeben ist, welche Variable es ändert oder verwendet. Dies ist wichtig, damit der Benutzer entscheiden kann, ob er weiter oder andersartig modularisieren soll.

Bei anderen Modularisierungsprozessen wird das Softwaresystem 'bottom-up' aufgebaut, d.h. aus vorhandenen einfachen Modulen wird ein neues komplexeres Modul aufgebaut /Was-77/.

In der Praxis werden aber beide Typen von Modularisierungsprozessen verwendet.

Bei der Entwurfsmethode SADT wird das Softwaresystem noch präsiser spezifiziert: Jedes Modul wird durch ein Diagramm dargestellt, das den Datenfluß beschreibt. Bei anderen Entwurfsmethoden beschreibt ein Diagramm den Kontrollfluß.

Das in dieser Arbeit entwickelte 'Verfahren zur rechnerunterstützten Programmkonstruktion' gehört zur Klasse solcher Entwurfsmethoden.

1.1.3 Codierungsphase

Bei der Codierungsphase eines Softwaresystems wird der Programmcode für jedes Modul erstellt. Zuerst wird versucht, einen Algorithmus zu finden, der die Funktion des Moduls durchführt.

Eine solche Sammlung von Algorithmen wurde beispielsweise von Knuth /Knu-68/ entwickelt.

Auch bei der Codeerstellung wird die Technik des Structured Programming (s. Dijkstra /Dij68a/) und der Schrittweisen Verfeinerung angewandt.

Die Technik der Schrittweisen Verfeinerung wurde von Dijkstra /Dij68b/ entwickelt und von Wirth /Wir-71,Wir-74/ verbessert.

1.1.4 Verifikationsphase

Zu dieser Phase wird das Softwaresystem getestet. Hier werden Begriffe wie 'Test', Fehlersuche ('debug'), Beweis ('proof'), Verifikation, Validation mit unterschiedlichen Bedeutungen verwendet.

Der Programmbeweis besteht in der Tätigkeit, formal zu überprüfen, ob ein Programm Fehler hat. Die bekannteste Methode, um die Korrektheit eines Programms formal zu überprüfen, ist die Methode der 'inductive assertions' von Floyd /Flo-67/ und Naur /Nau-76/. Ihre Idee ist, in verschiedenen Punkten des Programms 'assertions' (eine boolesche Variable oder einen logischen Ausdruck) einzufügen. Ein Programm ist genau dann korrekt, wenn die 'assertions' überprüft werden können. Dershowitz /Der-78/ hat gezeigt, wie ein Programm, das in einer an Algol orientierten Sprache erstellt wird, auch mit 'invariant assertions' dokumentiert werden kann. Dies kann dann verwendet werden, um die Korrektheit des Programms zu überprüfen oder als Hilfe bei der Analyse ('debugging') des inkorrekten Programms.

1.1.5 Installationsphase

In dieser Phase wird das Softwaresystem in einen bestimmten Anwendungsumgebung installiert und getestet, um zu überprüfen, ob durch diese Installation neue Fehler eingeführt wurden.

1.1.6 Wartungsphase

In diese Phase werden sämtliche Fehler, die in der Verifikationsphase nicht gefunden wurden, korrigiert.

Da sich die Welt, in der die Computeranwendung eingebettet ist, mit der Zeit ändert, muß die Anforderungsdefinition des Softwaresystem stets mitgeändert werden. Diese Tätigkeit kann bis zu 50% der Kosten einer DV-Anwendung entsprechen. Trotz dieser hohen Summe gibt es keine Methode, um die Wartungsphase zu verbessern.

2 - Überblick über die Arbeit

Die vorliegende Arbeit beruht hauptsächlich auf der Jackson-Methode. Sie wird zusammen mit der SADT-Methode, der Nassi-Shneiderman Methode und der ISAC-Methode, aus denen weitere Ideen für die vorliegende Arbeit entnommen werden, in Kapitel II kurz beschrieben.

Bei der Anwendung unseres 'Verfahrens zur rechnerunterstüzten Programmkonstruktion' (Bild 1.3) muß der Benutzer zunächst durch Analyse der Anforderungsbeschreibung, die durch Texte in natürlicher Sprache, Bilder, Tabellen, formale Ausdrücke usw. beschreibt, was das Anwenderprogramm ausführen soll, eine Datenstruktur (Kapitel III) erstellen.

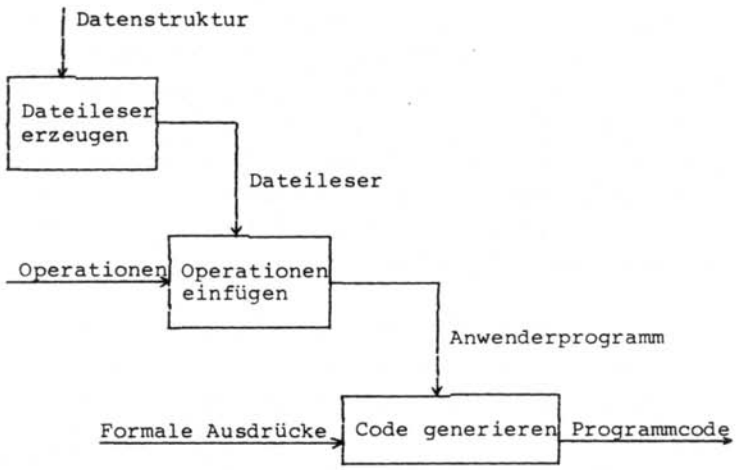


Bild 1.3: Das Verfahren zur rechnerunterstützten Programmkonstruktion

Diese Datenstruktur stellt die Operationen der realen Welt dar und besteht, wie bei der Jackson-Methode, aus vier Typen von Datenobjekten: disjunktive, konjunktive, iterative und primitive Datenobjekte. Sie kann durch eine einfache Grammatik beschrieben werden.

Ergänzend zu dieser Datenstruktur wird das Konzept der Datenabhängigkeit zwischen disjunktiven Datenobjekten und der Bereichsgrenzen für die iterativen Datenobjekte eingeführt.

Da die Datenobjekte der Datenstruktur als boolesche Variable betrachtet werden können, kann es wegen der vorhandenen Abhängigkeiten vorkommen, daß für alle möglichen Werte der primitiven Datenobjekten bestimmte Datenobjekte nie 'true' werden. In Anlehnung an das Konzept der 'dead transition' bei Petri-Netzen /Gen-81/ bezeichnet man solche als 'tote' Datenobjekte. In der vorliegenden Arbeit wird ein Algorithmus entwickelt, der die toten Datenobjekte ausgibt.

Wir zeigen in Kapitel IV, wie ein Programm aus der Datenstruktur automatisch erzeugt werden kann. Für die Erzeugung dieses Programms werden die Konzepte von 'tags' für disjunktive Datenobjekte und EOL ('end-of-list') für iterative Datenobjekte zusätzlich eingeführt. Das Programm selbst ist ein 'Parser', der als Dateileser bezeichnet wird. Er hat die Aufgabe, eine Datei der entsprechenden Datenstruktur zu lesen und die Daten der Datenobjekte bereitzustellen.

Der Anwender unseres 'Verfahrens zur rechnerunterstützten Programmkonstruktion' spezifiziert in einem zweiten Schritt wiederum aus der Anforderungsbeschreibung die internen Operationen (Eingabe-, Ausgabedaten und die Funktion. Kapitel V) sowie den Datenfluß und integriert sie in den Dateileser. Damit ist das Anwenderprogramm (Kapitel VI) fertiggestellt.

Schließlich muß der Anwender noch den Code für das Anwenderprogramm erzeugen (Kapitel VII). Als Beispiel zeigen wir, wie für ein Anwenderprogramm Code in Pascal halbautomatisch generiert werden kann. Dabei muß der Benutzer formale Ausdrücke für die Funktion der Operationen und die Typen der Ausgabedaten der Operationen angeben.

Zwei Prototypen von CAD-Systemen werden in Kapitel VII spezifiziert und implementiert. Sie unterstützen den Anwender bei der Erstellung von Datenstrukturen ('Computer aided data structure design-CADSD') und Anwenderprogrammen ('Computer aided software design - CASD'). CASD ist dabei ein System, womit Anwenderprogramme auch unabhängig von unserer Technik durch Nassi-Shneiderman Diagramme spezifiziert werden können. Neue Operationen in CASD unterstützen den Anwender beim Schritt 'Operationen einfügen' (Bild 1.3).

Im Kapitel VIII machen wir abschließende Kommentare über die Anwendung unserer Technik und zeigen, wie sie ergänzt und erweitert werden kann.

II - SOFTWARE-ENGINEERING METHODEN

Im folgenden werden als Grundlage des ab Kapitel III darzustellenden Systems die Software-Engineering Methoden SADT, Jackson-Methode, Nassi-Shneiderman-Methode und ISAC kurz beschrieben.

1 - Structured Analysis and Design Technique-SADT

Diese Methode wurde von Douglas T. Ross /Ros77b/ entwickelt. Später hat Dickover /Dic-78/ den 'Sequencing' Mechanismus eingeführt, wodurch die Standardregeln zur Durchführung der Aktivitäten geändert werden: Es wird für jede Aktivität spezifiziert, welche Eingabedaten ('Pre-Conditions') nötig sind, um eine Untermenge von Ausgabedaten ('Post-Conditions') zu erzeugen. Thomas /Tho-78/ hat gezeigt, daß der SADT-Verfeinerungsmechanismus auf dem Prinzip der funktionalen Zerlegung beruht.

Der Software-Entwickler erstellt bei dieser Methode zwei Darstellungen (bei SADT als Modell bezeichnet) des Software-Systems: ein Aktigramm, das eine Beschreibung der Aktivitäten des Softwaresystems ist, und ein Datengramm, das eine Beschreibung der Daten des Softwaresystems darstellt.

Im Aktigramm sind die Daten die Schnittstellen der Aktivitäten und im Datengramm die Aktivitäten die Schnittstellen der Daten.

Der Software-Entwickler soll die beiden Modelle getrennt entwickeln. Sind die Modelle fertig, müssen Aktivitäten und Daten sowohl im einen als auch im anderen Modell auftreten. Dieser 'check' erhöht nach Thomas /Tho-78/ die 'consistency' und die 'completeness' der Modelle.

In der Praxis wird aber vorwiegend das Aktigramm verwendet.

Die Methode

Da die Verfahren, um beide Modelle zu erstellen, dual sind, beschreiben wir hier nur das Verfahren, ein Aktigramm zu erstellen.

Ein Kasten ('black box') stellt eine Aktivität, die komplex oder elementar ist, dar (Bild 2.1).

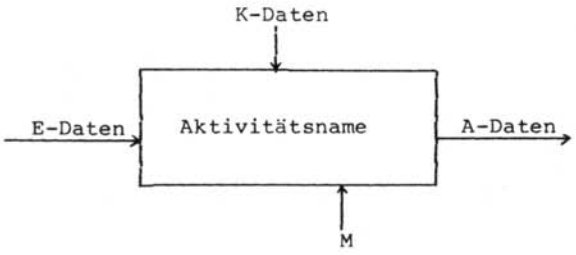


Bild 2.1: Der SADT-Kasten.

E-Daten stellen die Menge der Eingabedaten dar. Sie kann auch leer sein; K-Daten stellen die Menge der Kontrolldaten dar; A-Daten stellen die Menge der Ausgabedaten dar; M stellt einen Mechanismus (der ein Modul, eine Prozedur oder eine Funktion sein kann) dar.

Die K-Daten werden bei dieser Methode als auch Eingabedaten betrachtet. Die Aktivität wird durchgeführt, wenn sämtliche Eingabedaten zur Verfügung stehen. Als Ergebnis erzeugt die Aktivität sämtliche Ausgabedaten.

Wenn die Aktivität eine komplexe Operation darstellt und sie noch nicht vorhanden ist, muß der Software-Entwickler

beschreiben, aus was sie sich zusammensetzt.

Ein Diagramm (Bild 2.2) ist die Beschreibung einer komplexen Aktivität. Es besteht aus minimal 3 und maximal 6 Aktivitäten, die auf einer Seite diagonal angeordnet sind. Die Ein- und Ausgabedaten für das Diagramm sind genau diejenigen, die bei den Aktivitäten auftreten. Die E-Daten kommen von links in das Diagramm herein, die K-Daten von oben und die A-Daten gehen nach rechts hinaus. Der Name der Aktivität und des Diagramms, das sie beschreibt, sind gleich.

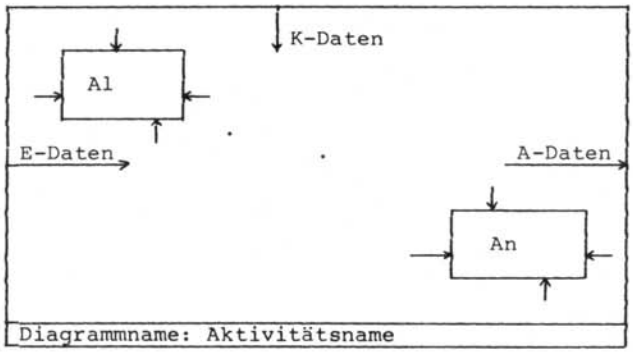


Bild 2.2: Das SADT-Diagramm.

Im Diagramm sind die Pfeile Schnittstellen zwischen den Aktivitäten und stellen einen Datenfluß dar.

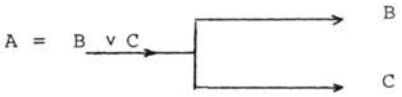
Die Daten im Diagramm können (a) sich verzweigen oder (b) sich vereinigen. Damit wird die Struktur der Daten beschrieben. Die verschiedenen Möglichkeiten werden im folgenden dargestellt:

a) Verzweigung der Daten:

a.1 - 'Branch'. Beispiel:

a.2 - 'Spread'. Beispiel:

a.3 - 'OR-Branch'. Beispiel:

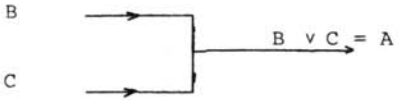


b) Vereinigung der Daten:

b.1 - 'Join'. Beispiel:

b.2 - 'Bundle'. Beispiel:

b.3 - 'OR-Join'. Beispiel:



2 - Die Jackson Methode

Ausgehend von Erfahrungen im DV-Gebiet hat Jackson /Jac-75/ eine Methode entwickelt, um Programme zu entwerfen, die sequentielle Dateien lesen und schreiben können.

Die Methode eignet sich auch, um die von Jackson selbst definierten einfachen Programme zu entwerfen.

Da in der DV-Praxis Programme normalerweise sequentielle Dateien lesen und schreiben, wird diese Methode dort häufig verwendet. Es gibt sogar CAD-Systeme, die den Entwurf von Programmen durch diese Methode unterstützen /kre-83/.

Die Methode

Als erster Schritt sollen die Eingabe- und Ausgabe-Datenstrukturen graphisch erstellt werden und als zweiter Schritt die Programmstruktur. Die primitiven Operationen, die ausgeführt werden sollen, werden in die Programmstruktur eingefügt. Jackson hat gezeigt, wie man die Programmstruktur in verbale Form ('pseudo code') übersetzen kann. Diese Form der Programmstruktur wird als 'schematic logic' bezeichnet. Die vollständige 'schematic logic' der Programmstruktur enthält unterdessen auch die Bedingungen für die bedingten Anweisungen.

Die Datenstrukturen der Jackson-Methode sind Bäume und stellen die Problemumgebung dar. Die Datentruktur kann aus vier Komponenten bestehen: 'selection', 'iteration', 'concatenation' und 'elementary'. Jede Komponente wird graphisch dargestellt.

Die Programmstruktur ist ein Baum und besteht auch aus den vier genannten Komponenten. Sie wird ausgehend von den Datenstrukturen entworfen.

Die Übersetzung der Programmstruktur in eine verbale Form verläuft wie folgt:

a) Wenn der Knoten der Programmstruktur eine 'concatenation' ist, dann gilt:

```
<Knotenname > seq
   .............
   Übesetzung des i-ten Unterknoten der 'Concatenation'
   .............
<Knotenname > end
```

b) Wenn der Knoten der Programmstruktur eine 'selection' ist, dann gilt:

```
<Knotenname > select  <Bedingung >
                      .............
<Knotenname > or  <Bedingung >
                      Übersetzung der i-ten Unterkomponente der
                      'selection'
                      .............
<Knotenname > end
```

c) Wenn der Knoten der Programmstruktur eine 'iteration' ist, dann gilt:

```
<Knotenname > iter while  <Bedingung >
                      Übersetzung der Unterkonponente der
                      'iteration'
<Knotenname > end
```

d) Wenn der Knoten der Programmstruktur 'elementary' ist, dann gilt:

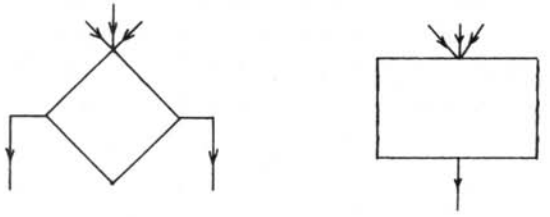
```
do  <Knotenname > .
```

3 - Nassi-Shneiderman Methode

3.1 - Historische Entwicklung

Zur Erstellung von Programmen war das Flußdiagramm das zuerst entwickelte Hilfsmittel.

Ein Flußdiagramm wird durch eine Sprache erzeugt, die zwei Komponenten hat: die prädikative (Bedingung) und die funktionale (Anweisung).



Prädikative Komponente Funktionale Komponente

Bild 2.3: Flußdiagramm: Bestehende Komponenten

Die Komponenten haben einen oder mehrere Eingangspunkte ('entry-point'), aber genau einen Ausgangspunkt (bei der funktionalen Komponente) oder zwei Ausgangspunkte (bei der prädikativen Komponente) (Bild 2.3).

Als die Hardwarekosten für Anwendersysteme noch die der Software überstiegen, waren Flußdiagramme sehr verbreitet, weil sie einen weitgerend optimierten Code liefern. Die Nachteile des optimierten Programms liegen im schlechteren Verständnis und als Folge davon in hohen Wartungskonsten.

Jacopini hat /Boh-66/ bewiesen, daß jedes beliebige Flußdiagramm in ein Flußdiagramm verwandelt werden kann, das in die rekur-

siven Konstrukte Sequenz, Auswahl, Iteration und funktionale Komponenten zerlegt werden kann.

3.2 - Die Methode

Nassi und Shneiderman /Nas-73/ haben eine neue Sprache entwickelt, die grundsätzlich aus vier Komponenten (Sequenz, Auswahl, Iterativen und Anweisung) besteht und die auch als Struktrogramm bezeichnet wird.

Aus den Komponenten dieser Sprache erzeugt der Benutzer ein Diagramm, das leicht zu verstehen ist, weil es eine Baumstruktur hat. Jacopini hat gezeigt, daß jedes Problem, das durch eine Sprache mit den zwei Komponenten (Bild 2.3) gelöst werden kann, auch durch eine Sprache mit diesen vier Komponenten dargestellt werden kann.

Die Erstellung eines Flußdiagramms und seine automatische Transformation in ein Nassi-Shneiderman Diagramm bringt wenig Vorteile, da dabei fremde prädikative und funktionale Komponente entstehen.

Ein NS-Diagramm liefert zwar keinen optimalen Code; beim heutigen Preisverhältnis von Hardware und Anwendersoftware ist diese Tatsache nicht mehr ausschlaggebend. Große Anwerdersysteme werden deswegen immer häufiger mit NS-Diagramme entwickelt und dokumentiert.

4 - Die ISAC Methode

Die ISAC Methode /Lun-79/ wurde in der Universität Stockholm entwickelt, damit das Informationssystem eines Unternehmens dieselbe Sprache wie die eingebettete Datenverarbeitung verwenden kann.

Das Informationssystem eines Unternehmens enthält sowohl die Beschreibungen die vorhandener Objekte wie Formulare,

Zeichungen, Materialien, Personen als Daten, als auch die Veränderungen und Verfahren von und mit diesen Objekten. Die Verarbeitung der im Informationssystem beschriebenen Objekte erfolgt durch Menschen, Computerprogramme oder Maschinen.

Unter einem DV-System wird die Zusammenfassung aller DV-Prozesse (Programme, Moduln, Unterprogramme, Transaktionen) und der dazugehörenden Daten (Datenbanken, Dateien, Datenfeldern) verstanden.

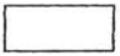
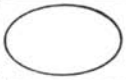
Die ISAC Methode basiert auf der K/I-Netz Notation /Ric-82/ und liefert als Ergebnis ein solches Netz.

Die Beschreibung der Methode

Die Grundelemente der ISAC Methode sind Aktivitäten und Informationen. Im Bespiel der Datenverarbeitung entsprechen den Informationen die Daten und den Aktivitäten die DV-Prozesse.

Ein ISAC Diagramm eines Anwenderprogramms stellt ein solches K/I-Netz dar. Bei K/I Netzen entspricht jede Aktivität einer Instanz und jede Information einem Kanal.

Die Elemente der Sprache sind in folgender Übersicht dargestellt.

Symbol	Interpretation
	Dieses Symbol stellt eine Aktivität dar. Eine Aktivität verbraucht, ändert, erzeugt, speichert oder verteilt Informationen.
	Dieses Symbol stellt eine Transition dar und dient dazu, Zugriffspfade (Strömungslinen) zu synchronisieren und/oder zu verzweigen.
	Dieses Symbol stellt eine Informationseinheit dar.
	Eine senkrechte Gerade stellt den normalen Datenfluß dar, der von oben nach unten geht. Sie verbindet eine Aktivität oder eine Transition mit einer Informationseinheit.
	Ein Pfeil, der von unten nach oben geht, zeigt eine Veränderung des Flusses. Er verbindet eine Aktivität oder eine Transition mit einer Informationseinheit.
	Ein Rechteckt stellt ein Diagramm dar. Oben außerhalb des Rahmens werden die Eingabe-Informationen für das Diagramm angezeigt, an der Unterseite die Ausgabeinformationen. Von oben nach unten ist die zeitliche Präzedenz zu berücksichtigen.
	Dieses Symbol stellt die Verfeinerung der Daten dar. Innerhalb dieser runden Umrandung wird angezeigt, aus welchen Informationen die übergeordnete Information besteht.

Regeln der ISAC Methode:

1 - Eine Aktivität hat neben den oben angegebenen Interpretationen auch die Aufgabe, den Datenfluß zu synchronisieren und zu verzweigen. Sie kann jedoch im Gegensatz zur Transition noch verfeinert werden; dies soll die Schraffur verdeutlichen.

2 - Zwei Aktivitäten sind nur über Informationen bzw. Dateneinheiten miteinander verbunden.

3 - Eine Aktivität wird durchgeführt, wenn die benötigen Informationen eingetroffen sind und an der Aktivität anstehen.

4 - Transitionen werden eingeführt, um Informationen zu trennen bzw. zusammenzuführen, ohne daß eine Tätigkeit erfolgt.

5 - Eingabe-/ Ausgabe-Informationen können innerhalb eines Diagramms auftauchen, sofern die Übergangsbedingungen eingehalten werden (gleiche Eingabe- und Ausgabeinformationen für eine Aktivität).

6 - Bei der Verfeinerung oder Vergröberung von Informationen ist erlaubt, daß Informationen erhalten bleiben, wegfallen oder neu entstehen.

5 - Kommentare zu den besprochenen Methoden

Wir gehen davon aus, daß ein Software-System zunächst durch Texte in natürliche Sprache, Bilder, Tabellen, formale Ausdrücke usw. beschrieben wird. In einem zweiten Schritt wird das Software-System spezifiziert. Schließlich wird das Software-System codiert.

Die Spezifikation des Software-Systems durch eine SW-E-Methode soll machinenunabhängig sein, d.h. Operationen über z.B. 'Zeiger' oder 'Puffer' sollen in dieser Beschreibung nicht enthalten sein.

Wir halten eine derartige Methode dann für gut, wenn sie eine verständliche und vollständige Spezifikation des Software-Systems liefert und es einfach ist, aus der Spezifikation den Code zu generieren.

5.1 - SADT

5.1.1 - Software Spezifikation

Da jede Aktivität durch ein Diagramm beschrieben (verfeinert) werden kann, wird die Spezifikation des Software-Systems 'top

down' erzeugt. Die Methode liefert so einen Baum von Diagrammen.

Das erste Diagramm, das genau eine Aktivität enthält, stellt das Software-System dar.

Bei der Erstellung eines Diagramms ist es schwer zu entscheiden, ob die Daten, die eine Aktivität benötigt, als Eingabe- oder als Kontrolldaten charakterisiert werden sollen.

Die verschiedenen Datenstrukturen, insbesondere diejenigen, die den den Datenfluß vereinigen, sind nicht eindeutig zu interpretieren. Aus diesem Grund ist es auch nicht einfach zu verstehen, ob eine Aktivität ablaufen darf.

5.1.2 - Code Generierung

Ein SADT-Diagramm stellt ein Netz dar, dessen Komponenten Transitionen (Verzweigung und Vereinigung von Daten) und Aktivitäten sind.

Wenn man für ein Diagramm Code generieren möchte, kann man die Aktivitäten als elementare Anweisungen oder als Prozeduraufrufe und die verschiedenen Typen von Transitionen als Kontrollstrukturen (if-then-else, while usw.) betrachten. Die Transitionen haben aber zu den Kontrollstrukturen keine Entsprechungsregeln. Wenn eine Entsprechungsregel vorliegen würde, enthielten nach unserer Meinung die Transitionen immer noch zu wenige Informationen, um eine Kontrollstruktur völlig aufzubauen. Da die Kontrollstrukturen nicht ableitbar sind, kann man auch nicht feststellen, welche Aktivitäten ablaufen und in welcher Reihenfolge sie ausgeführt werden können.

5.2 - Die Jackson Methode

5.2.1 - Software Spezifikation

Wenn der Software-Entwickler die Beschreibung eines Soft-

ware-Systems erhält, hat er zu entscheiden, ob ein oder mehrere Programme erstellt werden sollen. Es kann dadurch zu einem Netz von Programmen kommen, die Dateien als Schnittstellen haben.

Es kann auch der Fall auftreten, daß durch das Einsetzen neuer und/oder das Herausnehmen alter Programme ein vorhandenes Netz geändert werden soll. Programme können geändert werden, wenn die Beschreibung der Daten einer Datei geändert wurde.

Bei der Erstellung eines Programms soll der Anwender zuerst die Eingabe- und die Ausgabe-Datenstruktur entwerfen; sie entsprechen je einer sequentiellen Datei. Die Bedingungen für die verschiedenen bedingten Datentypen werden nicht dargestellt.

Für eine bestimmte Klasse, insbesondere von numerischen Problemen, ist es nicht oder nur unter künstlichen Annahmen möglich /Neu-80/, die Eingabe- und die Ausgabe-Datenstrukturen zu erzeugen. Wenn die Primzahlen Eingabedaten für einen Algorithmus sind, ist die Datenstruktur z.B. der Primzahlen nicht einfach zu entwickeln.

Aus den Datenstrukturen hat der Software-Entwickler die Kontrollstruktur des Programms zu entwerfen. Bei dieser Tätigkeit besteht als Hilfsmittel die sogennante Datenentsprechung, die normalerweise für die iterativen Daten verwendet wird. Wenn alle Datenstrukturen gleich sind, haben wir dann als triviale Lösung, daß die Kontrollstruktur dieselbe Struktur wie die Datenstrukturen hat. Wenn sie aber nicht gleich ist, tritt beim Software-Entwickler entweder der sogenante 'structur clash' auf (der nur durch eine Zwischendatei gelöst werden kann) oder es gibt folgende Lösung: Die Programmstruktur wird so aufgebaut, daß sie quasi formal jede Datenstruktur enthält/Jac-76/.

Damit bietet die Methode keine präzisen Regeln, um die Programmstruktur zu erstellen; Komponenten wie Erfahrungen und Faustregel werden notwendig.

Ist nun die Kontrollstruktur fertig, sollen die Anweisungen, die auszuführen sind, in die sequentielle Komponente der Kontrollstruktur eingefügt werden. Zusammen mit den Bedingungen wird dann die 'schematic logic' erzeugt.

5.2.2 - Code Generierung

Da die 'schematic logic' ähnlich aussieht wie ein Programm in einer prozeduralen Programiersprache (z.B. PL/I), ist die Tätigkeit der Code-Generierung mehr oder wenig intuitiv ableitbar.

5.3 - Nassi-Shneiderman Methode

5.3.1 - Software Spezifikation

Das NS-Diagramm stellt nur die Kontrollstruktur eines Anwenderprogramms dar. Datenstrukturen können bei dieser Methode nicht dargestellt werden.

Für eine Klasse von Problemen, insbesondere für numerische Probleme, wo die Eingabe- und Ausgabedaten keine richtige Struktur haben, ist diese Methode gut, da sie eine 'GO-TO-LESS' /Dij68a/ Darstellung liefert. Dadurch ist sie leicht zu verstehen und das entsprechende Programm ist einfach zu testen.

5.3.2 - Code Generierung

Der Software-Entwickler muß bei dieser Methode für den Algorithmusteil der Softwarespezifikation den auftretende Bedingungen und elementaren Anweisungen eine formale Gestalt - z.B. Code in Pascal - geben. Wir sind aber der Auffassung, daß sich der Software-Entwickler bei der Erstellung eines NS-Diagramms ohnedies bemüht, diese Bedingungen und die Anweisungen formal auszudrücken /Rap-82/. Deswegen wird die Code Generierung leichter.

Eine automatische Code-Generierung für die NS-Diagramme ist dann möglich, wenn seine Struktogramme eindeutig identifiziert werden können /Rap-82/. Damit müssen z.B. unterschiedliche Typen von 'For-statements' durch unterschiedliche Struktogramme dargestellt werden.

5.4. - Die ISAC Methode

Die ISAC Methode beruht auf K/I-Netzen und dadurch ist auch ein ISAC-Diagramm ein Netz. Die Methode ist ähnlich der SADT-Methode und die Bemerkungen, die wir dort gemacht haben, gelten auch hier.

Zusätzlich kann man noch bemerken: Ein ISAC-Diagramm sieht wegen der Transitionen besser aus als ein SADT-Diagramm. Da eine Aktivität auch als eine Transition interpretiert werden kann, (erste Regel der Methode), ist das ISAC-Diagramm nicht richtig zu interpretieren. Es ist erlaubt, an Rückpfeile (Veränderung des normalen Flusses) Texte zu schreiben (z.B. 'While Satz existiert'), die Ähnlichkeit mit der bedingten Anweisung haben. Dadurch wird die Code Generierung etwas einfacher.

6 - Abschließende Bemerkungen über die Methoden

Da alle erwähnten Methoden informell sind, liefert ihre Anwendung eine Softwarespezifikation, die weder vollständig noch eindeutig ist. Dadurch ist eine automatische Code-Generierung nicht möglich.

Der Vorteil ihrer Anwendung liegt darin, daß die erzeugte Softwarespezifikation schon eine grobe Struktur hat. Um diese Struktur zu erzeugen, gewinnt der Software-Entwickler damit ein besseres Verständnis des Problems, ohne sich mit Implementierungsproblemen beschäftigen zu müssen. Bei der Implementierung versucht der Software-Entwickler, diese Struktur zu erhalten (d.h. die Struktur der Softwarespezifikation und des Programms passen zusammen).

Außer den erwähnten Methoden werden bei der Softwarespezifikation auch die informellen Methode'Structured-Design-SD' /Con-78/, 'Hierarchy plus Input-Process-Output-HIPO' /IBM-74/, 'Program Design Language-PDL' /Cai-75/ verwendet sowie die formale Methoden 'Vienna Development Method-VDM /Bjo-78/, 'Data Type Specifications' /Gut-78/ und das Verfahren der Petri-Netze /Pet-77/.

III - DATENSTRUKTUREN

1 - Informale Beschreibung

Eine Datenstruktur ist ein Baum von Datenobjekten, von denen es vier Typen gibt: konjunktive, disjunktive, iterative und primitive Datenobjekte. Um die Datenstuktur besser darstellen zu können, ordnen wir jedem Datentyp eine graphische Darstellung zu, die wir von der Jackson-Methode übernommen haben.

1.1 - Ein Datenobjekt ist vom Typ konjunktiv (K-Datenobjekt), wenn es mehrere (≥ 2) Nachfolger hat, die das Datenobjekt bestimmen. Einem solchen Datenobjekt entspricht logisch eine Konjunktion seiner Nachfolger.

Beispiel:

A besteht aus B UND C UND D

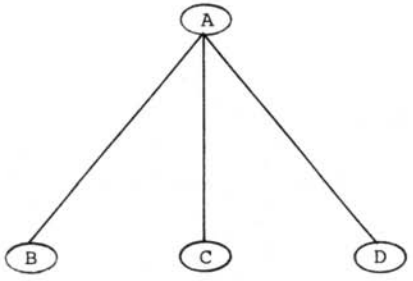


Bild 3.1: Graphische Darstellung des konjuntiven Datenobjekts A

1.2 - Ein Datenobjekt ist vom Typ disjunktiv (D-Datenobjekt), wenn es mehrere (≥ 2) Nachfolger hat und nur einer von ihnen das Datenobjekt bestimmt. Einem solchen Datenobjekt entspricht eine Disjunktion seiner Nachfolger.

Beispiel: A ist B ODER C ODER D

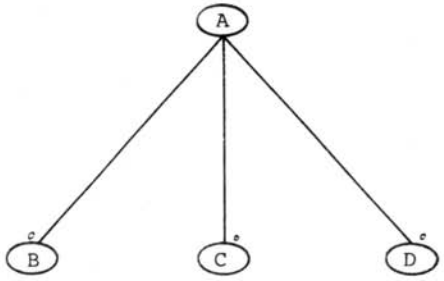


Bild 3.2: Graphische Darstellung des disjunktiven Datenobjekts A.

In der graphischen Darstellung bezeichnet ein kleiner Kreis am Nachfolger das D-Datenobjekt.

1.3 - Ein Datenobjekt ist vom Typ iterativ (I-Datenobjekt), wenn es genau einen Nachfolger hat. Man kann ein solches Datenobjekt als ein K-Datenobjekt interpretieren, bei dem alle Nachfolger gleich sind.

Beispiel:

A besteht aus B

Bei diesem Datentyp ist A eine Liste von B.

Bild 3.3: Graphische Darstellung des iterativen Datenobjekts A.

1.4 - Ein Datenobjekt ist vom Typ primitiv (P-Datenobjekt), wenn es keinen Nachfolger hat. Primitive Datenobjekte sind Konstante oder Variablen. Die Variablen sind vom Typ Zeichenkette, Integer oder Real.

Solche primitiven Datenobjekten werden beispielsweise wie folgt dargestellt:

Bild 3.4: Beispiele von primitiven Datenobjekten.

2 - Bildung der Datenstruktur

Da die Nachfolger eines Datenobjekts selbst Datenobjekte sind, wird eine Datenstruktur rekursiv gebildet. Der Rekursionsprozeß hört bei den primitiven Datenobjekten auf.

Da jedes I-Datenobjekt eine Rekursion darstellt , darf kein Datenobjekt zwei- oder mehrfach auf der Datenstruktur erscheinen. Deswegen erzeugt dieser Rekursionsprozeß einen Baum von Datenobjekten.

Die Kanten, die zwei Datenobjekte verbinden, stellen anonyme Prädikate ('hat','ist'...) dar. Die Datenobjekte müssen, je nach ihrem Typ, diese Prädikate erfüllen.

3 - Beispiele

Im folgenden werden Beispiele für Datenstrukturen angeben.

3.1 - Bespiel 1:

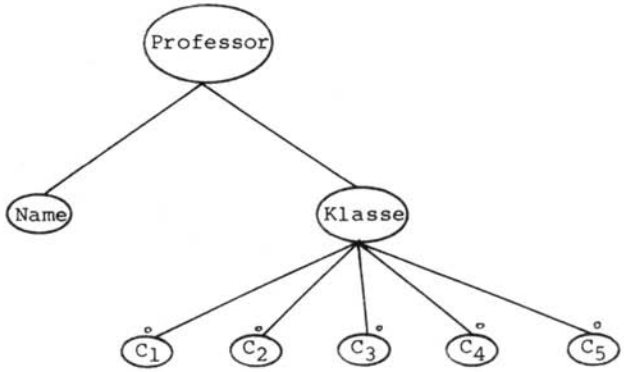


Bild 3.5: Beispiel einer Datenstruktur 'Professor'

Interpretation: Das Datenobjekt Professor hat einen Namen und gehört einer Klasse C_1,C_2,C_3,C_4 oder C_5 an.

3.2 - Beispiel 2:

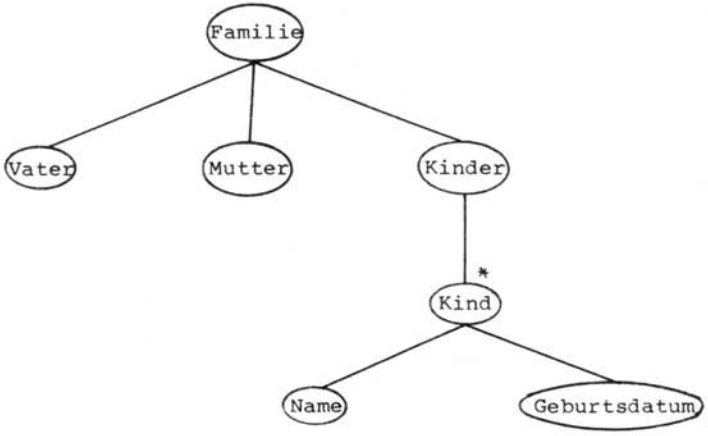


Bild 3.6: Beispiel einer Datenstuktur 'Familie'

Interpretation: Das Datenobjekt Familie besteht aus Vater, Mutter und Kindern. Jedes Kind hat einen Namen und ein Geburtsdatum.

3.3 - Beispiel 3:

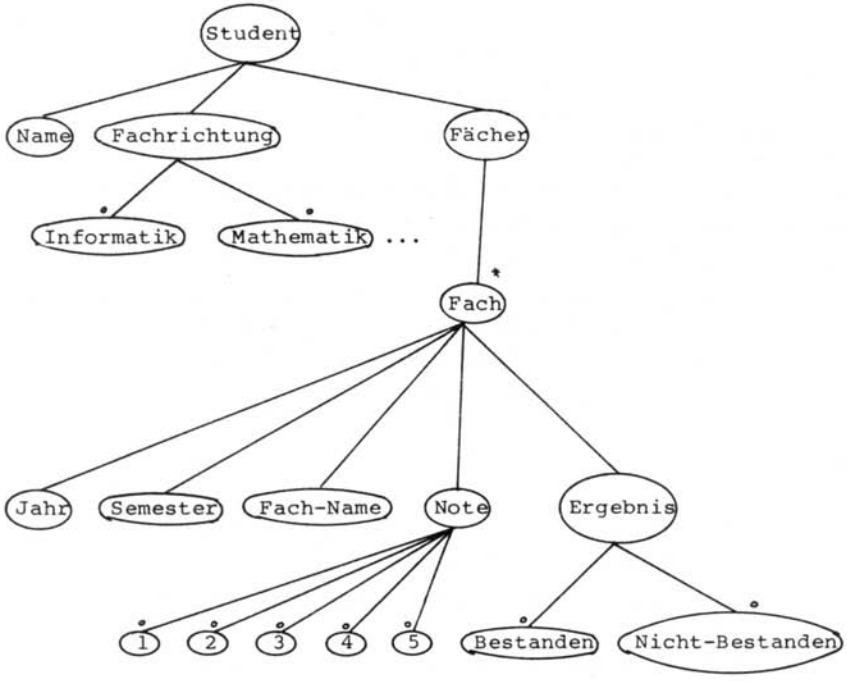


Bild 3.7: Beispiel einer Datenstruktur 'Student'

Interpretation: Das Datenobjekt Student hat einen Namen, eine Fachrichtung und eine Liste von Fächern. Die Fachrichtung kann entweder Informatik oder Mathematik usw. sein. Jedes Fach wurde in einem bestimmten Jahr und Semester gehört. Der Student hat für jedes Fach eine Note und ein Ergebnis erhalten. Die Note kann 1,2,3,4 oder 5 sein und das Ergebnis bestanden oder nicht bestanden.

Die Datenobjekte dieses Bespiel sind:

K-Datenobjekte = {Student,Fach }
D-Datenobjekte = {Fachrichtung,Note,Ergebnis }
I-Datenobjekte = {Fächer }
P-Datenobjekte = {Name, Informatik ,Mathematik ,Jahr,
Semester, Fach-Name, 1, 2, 3, 4, 5, Bestanden,
Nicht-Bestanden }

4 - Datenstruktur und Grammatik

Jede Datenstruktur kann als graphische Darstellung einer Grammatik angesehen werden, die auf der Metasprache BNF /Gri-71/ basiert, wobei auf der rechten Seite einer Regel dieser Grammatik genau eine Verknüpfung, eine Alternative oder eine Iteration von Datenobjekten steht. Die Rekursion wird durch die Iteration dargestellt. Die P-Datenobjekte der Datenstruktur machen die terminalen Symbole der Grammatik aus.

Beispiel.

<Familie> ::= <Vater><Mutter><Kinder>
<Kinder> ::= {<Kind>}
<Kind> ::= <Namen><Geburtsdatum>

Die graphische Darstellung dieser Grammatik wird im Beispiel 3.2 gezeigt.

5 - Datenstruktur und die reale Welt

Jede Datenstruktur kann, wie die Beispiele gezeigt haben, verwendet werden zur abstrakten Beschreibung eines Objektes der realen Welt.

Eine Ausprägung ('Instance') der Datenstruktur ist ein Baum von Daten, die einen endlichen Zustand (true oder false) haben. Diese Ausprägung stellt die konkrete Beschreibung des Objekts

dar. (In der Datenverarbeitung spricht man hier von Daten eines Datentyps.)

Wenn die Datenstruktur verwendet wird, um ein Objekt der realen Welt zu beschreiben, spiegelt die Ausprägung den Zustand des Objektes zu einem bestimmten Zeitpunkt wider. Wenn sich das Objekt in der realen Welt ändert, spiegelt sich diese Änderung auch in einer neuen Ausprägung des Objekts wider. Dabei interessieren nur diejenigen Daten der Ausprägung, die 'true' sind. (Die anderen existieren nicht.)

In dieser Arbeit werden externe Operationen, die eine Ausprägung in eine andere überführen, nicht dargestellt.

6 - Die Datei.

6.1 - Dateiinhalt

Es genügt unter der Berücksichtigung von Speichereffizienz, nur die primitiven Daten (P-Daten) einer Ausprägung im Zustand 'true' zu speichern. Es ist dann Aufgabe der syntaktischen Analyse, den gesamten Baum abzuleiten.

Jede Datei ist damit ein physikalisches Medium, das die P-Daten im Zustand 'true' einer Ausprägung enthält, aus denen dann der syntaktische Analysator ('Analyser') die true Ausprägung ableitet.

Die lexikalische Analyse hat die Aufgabe, die P-Daten, die auf der Datei stehen, abzuleiten. Wenn ein P-Datenobjekt eine Variable ist, enthält die Datei Elemente vom Typ String, Integer oder Real und nicht den Namen des P-Datenobjekts selbst. Wenn ein P-Datenobjekt eine Konstante ist, enthält die Datei diese Konstante selbst.

6.2 - Die sequenzielle Datei

Eine sequenzielle Datei ist eine Datei, deren Daten sequenziell

eingelesen oder geschrieben werden. Gleichzeitiges Lesen und Schreiben ist nicht möglich.

Sequenzielle Dateien können beispielsweise auf Magnetbandeinheiten gespeichert werden.

Dabei wird ein P-Datum, das auf dem Band steht, verbraucht, wenn es gelesen wird, d.h. es kann weder gelesen noch überschrieben werden. (Das Band gehe immer vorwärts).

Da ein Modell einer realen Welt durch verschiedene Datenobjekte, die miteinander eine Beziehung haben, beschrieben wird, liest ein Programm normalerweise verschiedene Eingabedateien und erzeugt verschiedene Ausgabedateien.

7 - Datenstruktur: Formale Beschreibung

Im folgenden wird nun der Begrif der Datenstruktur definiert, wobei eine zur Graphentheorie ähnliche Notation verwendet wird.

Ein 6-Tupel

$$DS(D_0)=(K,D,I,P,D_0,F)$$

ist genau dann eine Datenstruktur, wenn

1) K die Menge der K-Datenobjekte ist
2) D die Menge der D-Datenobjekte ist
3) I die Menge der I-Datenobjekte ist
4) P die Menge der P-Datenobjekte ist
5) D_0 die Wurzel der Datenstruktur ist

Sei L_0 die Menge der Datenobjekte, dann gilt:

$L_0=K \cup D \cup I \cup P$,
$D_0 \in L_0$,
$L=L_0 - \{D_0\}$ sowie
$(K \cap D) \cup (K \cap I) \cup (K \cap P) \cup (D \cap I) \cup (D \cap P) \cup (I \cap P) = \emptyset$.

6) F sind die Kanten

$$F = K \times D \cup K \times I \cup K \times P \cup$$
$$D \times K \cup D \times I \cup D \times P \cup$$
$$I \times K \cup I \times D \cup I \times P \cup$$
$$K \times K \cup D \times D \cup I \times I$$

Sei $x \in L_0$ und $N(x)$ die Menge der Nachfolger von x und $V(x)$ die Menge der Vorgänger von x, dann gilt:

$$N(x) = \{n \mid [x,n] \in F\}$$
$$V(x) = \{y \mid [x,y] \in F\}$$

6.1 Wenn $x \in L \cap D$, dann gilt

$$|N(x)| \geq 2$$
und $$|V(x)| = 1.$$

6.2 Wenn $x \in L \cap K$, dann gilt

$$|N(x)| \geq 2$$
und $$|V(x)| = 1$$

6.3 Wenn $x \in L \cap I$, dann gilt

$$|N(x)| = 1$$
und $$|V(x)| = 1$$

6.4 Wenn $x \in L \cap P$, dann gilt

$$|N(x)| = 0$$
$$|V(x)| = 1$$

6.5 Für D_0 als Wurzel der Datenstruktur, gilt

$$|V(D_0)| = 0$$
$$\text{und } |N(D_0)| \geq 2 \text{ wenn } D_0 \in K$$
$$\text{und } |N(D_0)| \geq 2 \text{ wenn } D0 \in D$$
$$\text{und } |N(D_0)| = 1 \text{ wenn } D_0 \in I$$
$$\text{und } |N(D_0)| = 0 \text{ wenn } D_0 \in P$$

8 - Ausprägung: Formale Beschreibung.

Eine Ausprägung der Datenstruktur ist ein Baum, der von der Datenstruktur abgeleitet wird.

Es sei $DS(D_0) = (K,D,I,P,D_0,F)$ eine Datenstruktur.

Jedes Datenobjekt x aus $DS(D_0)$ stellt eine Liste dar:

$$x = \{x_1,\ldots,x_i,\ldots,x_n\},\ n \geq 0,\ x_i \in x.$$

Die Wurzel D_0 hat nur ein Element:

$$D_0 = \{d_1\}$$

Ein 6-tupel

$$A(D(D_0)) = (K',D',I',P',d_1,F')$$

ist eine Ausprägung von $DS(D_0)$ genau dann, wenn

1) $K' = \bigcup_{\substack{x \in K \\ x_i \in x}} x_i$ (K' ist die Menge von K-Daten)

2) $D' = \bigcup_{\substack{x \in D \\ x_i \in x}} x_i$ (D' ist die Menge von D-Daten)

$$3)\ I' = \bigcup_{\substack{x \in I \\ x_i \in x}} x_i \quad \text{(I' ist die Menge von I-Daten)}$$

$$4)\ P' = \bigcup_{\substack{x \in P \\ x_i \in x}} x_i \quad \text{(P' ist die Menge von P-Daten)}$$

L'_0 sei die Menge von Daten, dann gilt

$$L'_0 = K' \cup D' \cup I' \cup P'.$$

5) F' sind die Kanten der Ausprägung in folgender Darstellung:

$$\begin{aligned} F' = {} & K' \times D' \cup K' \times I' \cup K' \times P' \cup \\ & D' \times K' \cup D' \times I' \cup D' \times P' \cup \\ & I' \times K' \cup I' \times D' \cup I' \times P' \cup \\ & K' \times K' \cup D' \times D' \cup I' \times I' \end{aligned}$$

Es gelten noch folgende Definitionen, um den Baum der Ausprägung herzuleiten:

5.1 Ein D-Datum x_i mit $x_i \in x$ und $x \in D$ ist selbst aus D' und hat folgende Eingeschaften für Nachfolger $N(x_i)$ und Vorgänger $V(x_i)$:

$$\begin{aligned} & N(x_i) = \{y_i \mid [x_i, y_i] \in F' \wedge [x,y] \in F\}, \text{ mit } y_i \in y \\ \text{und} \quad & |N(x_i)| = |N(x)| \\ \text{und} \quad & V(x_i) = y_j, \text{ so daß } [y_j, x_i] \in F' \wedge [y,x] \in F, \text{ mit } y_j \in y \\ \text{und} \quad & |V(x_i)| = |V(x)| \end{aligned}$$

5.2 Ein K-Datum x_i mit $x_i \in x$ und $x \in K$ ist selbst aus K' und hat folgende Eigenschaften für Nachfolger $N(x_i)$ und Vorgänger $V(x_i)$:

$$\begin{aligned} & N(x_i) = \{y_i \mid [x_i, y_i] \in F' \wedge [x,y] \in F\}, \text{ wobei } y_i \in y; \\ & |N(x_i)| = |N(x)|. \\ & V(x_i) = y_j, \text{ so daß } [y_j, x_i] \in F' \text{ und } [y,x] \in F, \text{ mit } y_j \in y \\ & |V(x_i)| = |V(x)|. \end{aligned}$$

5.3 Bei der Datenstruktur haben die I-Datenobjekte genau einen Nachfolger; bei einer Ausprägung haben die I-Daten aber keinen, einen oder mehrere Nachfolger. Ein I-Datum x_i mit $x_i \epsilon x$ und $x \epsilon I$ ist selbst aus I' und hat folgende Eigenschaften für Nachfolger $N(x_i)$ und Vorgänger $V(x_i)$:

$N(x_i) = \{y_k, \ldots, y_{k+j}, \ldots, y_m\}$, so daß
$[x_i, y_{k+j}] \epsilon F$ und $y_{k+j} \epsilon y$, wobei
$[x,y] \epsilon F$.
Wenn $n > i$ und $[x_n, y_l] \epsilon F'$ dann gilt: $l > m$.
Wenn $n < i$ und $[x_n, y_l] \epsilon F'$ dann gilt: $l < k$.
$|N(x_i)| \geq 0$
$V(x_i) = y_j$, so daß $[y_j, x_i] \epsilon F'$ und $[y,x] \epsilon F$
$|V(x_i)| = |V(x)|$

Die Nachfolger von x_i sind, falls vorhanden, sequenziell angeordnet; das erste Element ist j_k und das letzte y_m (Bild 3.8).

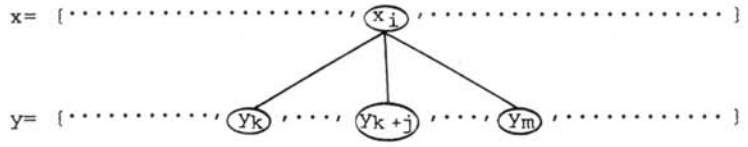


Bild 3.8: Die Nachfolger eines I-Datums.

Da die Anzahl der Nachfolger $|N(x_i)|$ von jedem x_i nicht spezifiziert wird, ist die Abbildung $x_i \rightarrow y_j$, $[x,y] \epsilon F$, mehrdeutig.

5.4 Ein P-Datum x_i mit $x_i \epsilon x$ und $x \epsilon P$ ist selbst aus P' und hat folgende Eigenschaften für Nachfolger $N(x_i)$ und Vorgänger $V(x_i)$:

$N(x_i) = \{y_i \mid [x_i, y_i] \epsilon F'\}$;
$|N(x_i)| = 0$
$V(x_i) = y_j \mid [y_j, x_i] \epsilon F' \wedge [y,x] \epsilon F$
$|V(x_i)| = |V(x)|$

9 - Länge der Liste $x \in L_0$

Es sei

1) M(x,y) die Funktion (s. Punkt 11.2.1.1), die alle Datenobjekte (ausschließlich x und einschließlich y) liefert, die auf dem Weg von x nach y in Richtung nach D_0 stehen.

2) x das Datenobjekt, dessen Länge wir feststellen möchten.

3) W die Funktion (s. Punkt 11.2.4):

$$W: L_0 \rightarrow L_0$$

3.1) $W(x) = q$ wenn $\exists$ q so, daß

$$|M(x,q) \cap I| = 1$$

3.2) $W(x) = D_0$, wenn $|M(x,D_0) \cap I| = 0$

dann wird die Länge von x definiert als

$$T(x) = \begin{cases} \sum_{q_i \in W(x)} |N(q_i)|, \text{ wenn } W(x) \in I \\ 1, \text{ wenn } W(x) \notin I \end{cases}$$

10 - Die Funktion Z(x)

Die Funktion Z(x) legt eine Abbildung zwischen L'_0 und der booleschen Menge B fest. Sie liefert den Zustand ('true' oder 'false') des Datums x. Sie ist eine Entscheidungsfunktion, die je nach dem Datentyp von x CONJ(x),DISJ(x),ITER(x) oder PRIM(x) ist. Die folgenden Ausdrücke (1),(2),(3) und (4) stellen dies genauer dar. Die Funktion Z liefert den Wert FEHLER, wenn Z(x) weder 'true' noch 'false' ist. FEHLER bedeutet, daß die Funktion unbestimmt wird.

$$Z: L'_0 \rightarrow B \cup \{FEHLER\}$$

wobei B die boolesche Menge ist.

Wenn $x' \epsilon L'_0$ dann gelten folgende Ausdrücke:

(1) - Z(x') = KONJ(x'), wenn x' ϵ K'
(2) - Z(x') = DISJ(x'), wenn x' ϵ D'
(3) - Z(x') = ITER(x'), wenn x' ϵ I'
(4) - Z(x') = PRIM(x'), wenn x' ϵ P'
(5) - KONJ(x') = 'true', wenn für $\forall$n' ϵ N(x')
Z(n') = 'true'.
(6) - KONJ(x') = 'false', wenn für $\forall$n' ϵ N(x')
Z(n') = 'false'.
(7) - DISJ(x') = 'true', wenn für $\exists$n' ϵ N(x') ,so daß
Z(n') = 'true'
und für $\forall n'_1 \epsilon$ N(x') , $n'_1 \neq$n',
$Z(n'_1)$ = 'false'.
(8) - DISJ(x') = 'false', wenn für $\forall$n' ϵ n(x')
Z(n') = 'false'.
(9) - ITER(x') = 'true', wenn
|N(x')| > 0
und für $\forall$n' ϵ N(x'),
Z(n') = 'true'.
(10)- ITER(x') = 'true', wenn |N(x')| = 0
(11)- ITER(x') = 'false', wenn |N(x')| > 0 ,
und für $\forall$n' ϵ N(x')
Z(n') = 'false'.
(12)- PRIM(x') = 'true', wenn x' = 'true'.
(13)- PRIM(x') = 'false', wenn x' = 'false'.

Beispiel

a) Gegeben sei die Datenstruktur:

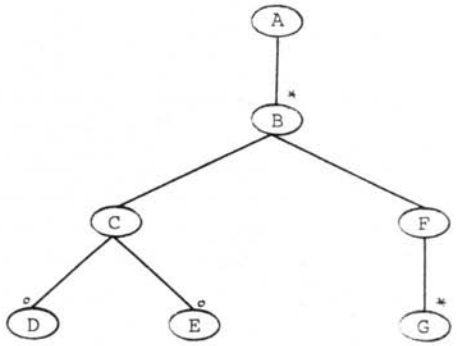


Bild 3.9: Beispiel einer Datenstruktur

$$
\begin{aligned}
DS(A) = (K &= \{B\},\\
D &= \{C\},\\
I &= \{A,F\},\\
P &= \{D,E,G\},\\
F &= \{[A,B],[B,C],\\
&\quad [B,F],[C,D],\\
&\quad [C,E],[F,G]\})
\end{aligned}
$$

b) Gegeben seien DS(A)(Bild 3.9) und

$A = \{a_1\}$ $\qquad$ $E = \{e_1, e_2\}$

$B = \{b_1, b_2\}$ $\qquad$ $F = \{f_1, f_2\}$

$C = \{c_1, c_2\}$ $\qquad$ $G = \{g_1, g_2, g_3\}$

$D = \{d_1, d_2\}$

Eine Ausprägung von DS(A) ist dann beispielsweise:

$$
\begin{aligned}
A(DS(A)) = (\ & K' = \{b_1, b_2\}, \\
& D' = \{c_1, c_2\}, \\
& I' = \{a_1, f_1, f_2\}, \\
& P' = \{d_1, e_1, g_1, d_2, e_2, g_2, g_3\}, \\
& F' = \{[a, b_1], [c, b_2], [b_1, c_1], [b_2, c_2], \\
& \quad [b_2, f_2], [c_1, d_1], [c_1, e_1], [f_1, g_1], \\
& \quad [c_2, d_2], [c_2, e_2], [f_2, g_2], [f_2, g_3]\}\ )
\end{aligned}
$$

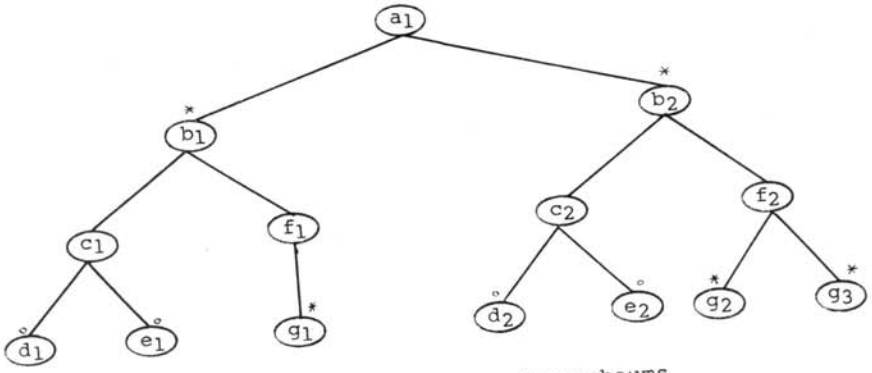


Bild 3.10: Beispiel eines Ausprägungsbaums

c) Wenn d_1 = 'true', e_1 = 'false', g_1 = 'true', d_2 = 'false', e_2 = 'true', g_2 = 'true', g_3 = 'true' ist, dann ist

$$Z(a_1) = \text{'true'}$$

Da jedes Datenobjekt eine Liste darstellt und ihre Länge von den I-Datenobjekte abhängt (s. Punkt 9), hat eine Datenstruktur, die wenigstens ein I-Datenobjekt hat, zwar eine unendliche Anzahl von Ausprägungen, aber jede Ausprägung hat eine endliche Anzahl von Lösungen*. (Bei einer Datenstruktur, die kein I-Datenobjekt hat, ist die Länge jedes Datenobjekts gleich 1).

---

* Eine Lösung einer Ausprägung ist eine bestimmte Kombination von P-Daten, so daß sich stets $Z(d_1)$= true ergibt.

11 - Disjunktive Datenobjekte

Bei der Programmerstellung überprüft der syntaktische Analysator, ob eine 'true'-Ausprägung der Datenstruktur ausgehend von der Datei abgeleitet werden kann. Es ist u.a. Aufgabe des syntaktischen Analysators, zu analysieren, welcher Nachfolger eines D-Datenobjekts im Moment gilt. Die Schwierigkeit besteht darin, daß der syntaktische Analysator dies allein aus den P-Datenobjekte ableiten muß.

Nach Jacksons Meinung soll der Benutzer bei der Erstellung der Datenstruktur die Bedingungen für die einzelnen Varianten der D-Datenobjekte nicht darstellen, da sie eine Implementierungseigenschaft tragen.

Das ist dann richtig, wenn die Bedingungen zur Syntaxanalyse der Grammatik (Implementierungseigenschaft) und nicht zur Beschreibung der Objekte gehören.

Wir haben die D-Datenobjekte in zwei Teile aufgeteilt:

- diejenigen, die unabhängig sind und
- diejenigen, die abhängig sind.

Die unabhängigen D-Datenobjekte enthalten bei der Definition der Datenstruktur keine Bedingungen für ihre Nachfolger.
Es ist Aufgabe der Software-Entwickler, Bedingungen zu finden,

- die aus den P-Datenobjekten(Terminal Symbole der Grammatik) gebildet sind,
- die disjunktiv sind.

Hier entsteht bei der syntaktischen Analyse das größte Problem und auch eine große Fehlerquelle, weil die Bedingungen vom Typ der Grammatik abhängen, die durch die Datenstruktur dargestellt wird. Ab und zu muß man weitere Daten einlesen, um zu entscheiden, welche Variante (Nachfolger) weiter verfolgt werden muß.

Bei einer bestimmten Klasse von Problemen, wobei

a) die Variante des unabhängigen D-Datenobjekts bei der Erzeugung der Datenstruktur explizit angegeben werden muß (z.B. bei Formularen),

b) und sich die Bedingungen für die abhängigen D-Datenobjekte aus den Beziehungen, die sie zueinander haben, ergeben, wird der syntaktische Analysator einfacher. Für diese Klasse von Problemen wurde unser Verfahren zur rechnerunterstützten Programmkonstruktion entwickelt; dabei wird bei der Konstruktion des Anwenderprogramms der Software-Entwickler davon befreit, sich mit formalen Sprachen beschäftigen.

11.1 - Die Kontrolldaten

Kontrolldaten sind Programm-Variable, die mit unabhängigen D-Datenobjekten verbunden sind. Ihr Wert steht in der Datei und stellt einen der Nachfolger des D-Datenobjekts dar. Man kann also die Kontrolldaten als D-Datenobjekte betrachten, deren Nachfolger Konstante sind.

11.1.1 - Die impliziten Kontrolldaten

Bei der Erstellung des sogenannten Dateilesers (ein Programm, das die Daten der Datenobjekte liest), der im folgendem näher beschrieben wird, stellen die unabhängigen D-Datenobjekte Variable (implizite Kontrolldaten) und ihre Nachfolger Konstante dar. Die Variablen werden gelesen und erhalten einen Wert. Dieser Wert entspricht einem der Nachfolger des D-Datenobjekts. Die entsprechende Kontrollstruktur steuert dann den Kontrollfluß je nach dem Wert der Variablen.

11.1.2. - Die expliziten Kontrolldaten

Wenn die Datenstruktur eines Datenobjekts n unabhängige D-Datenobjekte hat, hat der Dateileser n Variable. Um Speicher zu

sparen, kann ein Benutzer alle unabhängigen D-Datenobjekte durch eine einzige Variable (explizit) darstellten.

Wir bezeichnen als tag /You-82/ solche Variable, die mit unabhängigen D-Datenobjekten verbunden sind. Sie können die Datenobjekte selbst (implizit) oder eine bestimmte Variable (explizit) darstellen.

Die Datenobjekte werden in den Datenstrukturen durch graphische Darstellung beschrieben. Die Datenobjekte haben Namen, die innerhalb bestimmter Anwender eine Bedeutung* haben. Da eine Datenstruktur bei vielen Problemen verwendet werden kann, sollen diese Namen eine Bedeutung zum Ausdruck bringen. Dies ist bei der Dateigenerierung aber nicht notwendig. Es ist eine Kurzbezeichnung (tag-Wert) für jeden Nachfolger eines unabhängigen D-Datenobjekts zu finden, der das Abspeichern von Datenobjekt der ausgewählten Variante vereinfacht. Dazu dienen z.B. natürliche Zahlen. Man kann also die tags als D-Datenobjekte betrachten, deren Nachfolger eine Kurzbezeichnung tragen.

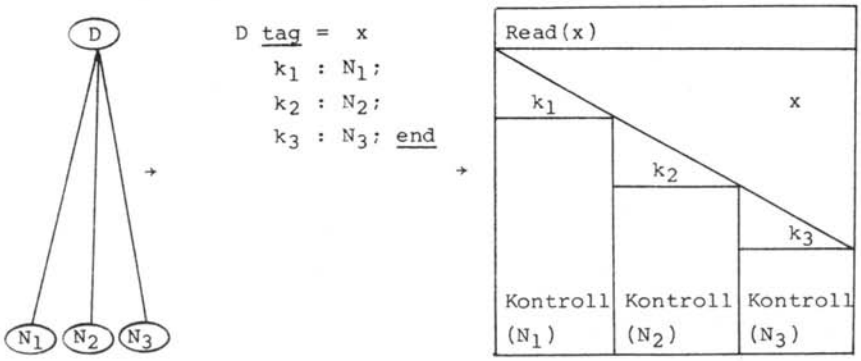
Die Bezeichungen für die Verbindung zwischen unabhängigen Datenobjekten und dem tag sind technisch und implementierungsabhängig. Deswegen ziehen wir es vor, die tags und die Kurzbezeichnungen (tag-Werte) für die Nachfolger wie folgt explizit zu beschreiben:

```
<tag-assignment> ::= <D-Dataobject>
                     [tag = <tag-name>]
                     { <constant>: <successor> ; } end
```

---

*Hier wird eine Datenstruktur dazu verwendet, um über eine Idee zu kommunizieren.

Beispiel:



Datenstruktur <tag-assignment> Kontroll(<D>)

Bild 3.11: Kontrolldaten: Beispiel

11.1.3 - Kontrolldaten: formale Beschreibung

Es sei 1) D die Menge der D-Datenobjekte;

2) TD die Menge der unabhängigen D-Datenobjekte (TD-Datenobjekte);

3) TDF die Menge der abhängigen D-Datenobjekte (TDF-Datenobjekte);

4) TAG die Menge der tags

Aus (1), (2), (3) und (4) folgen damit die Aussagen:

$$D = TD \cup TDF$$
$$TD \cap TDF = \emptyset \; ; \; TD \subseteq D \; ; \; TDF \subset D$$
$$|TAG| \leq |TD| \; ; \; TAG \cap TD \geq \emptyset$$

F_T sein die Verbindung eines tags mit einem unabhängigen Datenobjekt:

5) F_T: TAG × TD

T(x) ist das tag von x (x hat nur ein tag). A(t) sind die Datenobjekte, die mit dem tag t verbunden sind; dann gilt

a) $T(x) = \{t \mid [t,x] \in F_T\}$; $|T(x)| = 1$
b) $A(x) = \{x \mid [t,x] \in F_T\}$; $|A(x)| \geq 1$

N([t,x]) seien die Nachfolger von t, dann gilt

6) $N([t,x]) \geq 2$, $[t,x] \in F_T$

F_1 sei die Verbindung zwischen dem Nachfolger von tag t und dem Nachfolger vom dem unabhängigen Datenobjekt x, dann gilt

7) $F_1 = N([t,x]) \times N(x)$, $[t,x] \in F_T$

C(a) sei der Nachfolger von tag t im Zusammenhang mit x, der a entspricht, dann gilt:

a) $C(a) = b \mid [b,a] \in F_1$; $|C(a)| = 1$

Die folgende Aussage zeigt, daß die Beziehung zwischen dem Nachfolger von tag t und dem Nachfolger vom Datenobjekt x 'eins-zu-eins' sein muß:

$$b) \bigcap_{a \in N(x)} C(a) = \emptyset$$

11.2 - Abhängigkeit zwischen D-Datenobjekten

Der Begrif der Datenabhängigkeit, den wir im folgenden definieren und verwenden wollen, stellt eine Erweiterung der Datenstruktur, wie sie in der Jackson-Methode verwendet wurde, dar, und gehört zur Beschreibung der Datenobjekte. Die sich daraus ergebenden Bedingungen für die abhängigen Datenobjekte, sind implementierungsunabhängig.

Wenn das Datenobjekt x das Datenobjekt y steuert, vollzieht sich die Steuerung so, wie wenn x ein tag für y wäre. Das heißt, daß die Variante, die von y ausgewählt wird, von der Varianten abhängt, die von x ausgewählt wurde.

Die Abhängigkeit enthält eine gewisse Semantik, weil durch die Steuerung bestimmte Datenobjekte von anderen abgeleitet werden können.

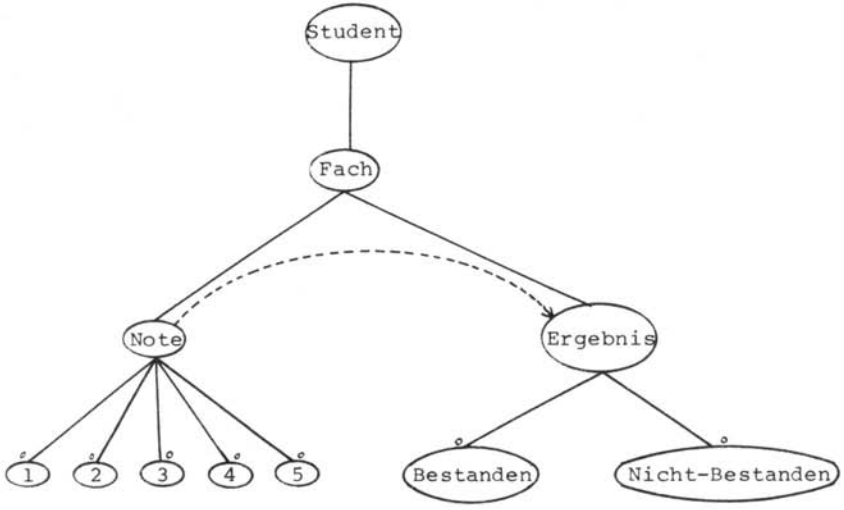


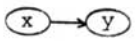
Bild 3.12: Abhängigkeit: Beispiel

Im Bild 3.12 werde das Datenobjekt 'Ergebnis' vom Datenobjekt 'Note' gesteuert. Der Pfeil zwischen 'Note' und 'Ergebnis' zeigt diese Abhängigkeit. Das Ergebnis soll Bestanden bzw. Nicht-Bestanden sei, wenn die Note 1,2,3 oder 4 bzw. 5 ist.

11.2.1 - Abhängigkeit(1:1): formale Beschreibung

Die Abhängigkeit(1:1) zwischen Datenobjekten bedeutet, daß ein Datenobjekt x maximal ein Datenobjekt steuert bzw. von einem Datenobjekt abhängen kann.

Beispiel:



x steuert y; y hängt von x ab.

Die Abhängigkeit (1:1) läßt sich wie folgt beschreiben.

Sei 1) TD die Menge der unabhängigen Datenobjekte;

2) TDF die Menge der abhängigen Datenobjekte;

Sei F_{TF} die Verbindung zwischen einem TD-Datenobjekt und einem TDF-Datenobjekt oder zwischen zwei TDF-Datenobjekte. Dann gilt

3) $F_{TF} = TD \times TDF \cup TDF \times TDF$

A(x) sind die von x gesteuerten Datenobjekte. x kann höchstens ein Datenobjekt steuern. Deshalb $|A(x)| \leq 1$. S(y) sind die Datenobjekte, wovon y abhängt. Jedes TDF-Datenobjekt hängt von genau einem Datenobjekt ab. Deshalb $|S(y)| = 1$. Daraus folgen:

a) $A(x) = \{y \mid [x,y] \in F_{TF}\}$; $|A(x)| \leq 1$
b) $S(y) = \{x \mid [x,y] \in F_{FT}\}$; $|S(y)| = 1$

Sei F_2 die Entsprechung zwischen den Nachfolger des Datenobjekts x und dem Nachfolger des Datenobjekts y, wobei $[x,y] \in F_{TF}$.

4) $F_2 = N(x) \times N(y)$;

Seien C(a) die Nachfolger vom Datenobjekt x, die dem Nachfolger a vom Datenobjekt y entsprechen. Ein Nachfolger von y kann einem oder mehreren Nachfolgern von x entsprechen.

a) $C(a) = \{b \mid [b,a] \in F_2\}$; $|C(a)| \geq 1$

Die folgende Aussage bestimmt, daß ein Nachfolger von x kann höchstens einem Nachfolger von y entsprechen kann.

$$b) \bigcap_{a \in N(y)} C(a) = \emptyset$$

Die Reihenfolge Bedingungen (RB)

Für die Paare $[x,y] \in F_T$ sollen weitere sogenannte Reihenfolge-Bedingungen dargestell werden. Die P-Daten ('true' P-Daten) und die Kontrolldaten, die zum Beispiel auf einem sequenziellen Datenträger stehen, müssen in bestimmter Reihenfolge gelesen (verbraucht) werden.

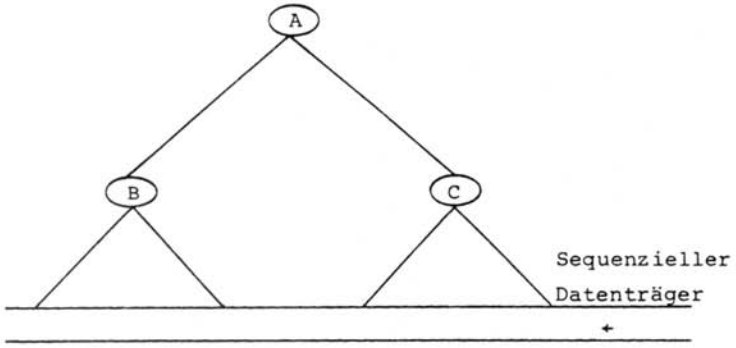


Bild 3.12: Beispiel für den Verbraucher von P-Daten.

Wenn der Datenträger zum Beispiel von rechts nach links läuft, werden zuerst die Daten von B und dann die Daten von C gelesen, wenn A ein K-Datenobjekt ist. Also wird B vor C verbraucht und die Datenobjekte werden von links nach rechts verbraucht. Damit gilt allgemein:

Wenn y von x gesteuert wird, kann y nur verbraucht werden, wenn x bereits verbraucht wurde.

Für die Reihefolgebedingung verwendet man die Funktion M(x,y):

$$M: L_0 \times L_0 \rightarrow 2^{L_0} \cup \{\text{unbestimmt}\}$$

$$\begin{cases} M(y,y) = \emptyset \\ M(x,y) = M(V(x),y) \cup V(x) \end{cases}$$

Die Funktion M(x,y) liefert alle Datenobjekte (einschließlich y und ausschließlich x), die auf dem Weg von x nach y stehen. Für $y \notin M(x,D_0)$ wird die Funktion unbestimmt.

Wir definiere nun sechs Reihenfolge-Bedingungen (RB):

RB.1: - Wenn $[x,y] \in F_T$, dann ist

$$x \notin M(y,D_0)$$

Wenn y verbraucht wird, ohne daß diese Bedingung besteht, dann liegen sowohl die Variante von x, die ausgewählt wurde, als auch die entsprechende Variante von y, die ausgewählt werden muß, fest.

RB.2: - Wenn $[x,y] \in F_T$ ist, dann gilt

$$y \notin M(x,D_0)$$

Ohne diese Bedingung kann y nicht verbraucht werden, da x noch nicht verbraucht wurde.

RB.3 - Wenn $[x,y] \in F_T$, dann kann man Wege von x und y bis zur Wurzel D_0 bauen. Die beide Wege treffen sich auf einem Datenobjekt p.

Es gilt dann die Bedingung: Der Treffpunkt(TP) beider Wege muß ein K-Datenobjekt sein.

$$p \in K$$

Das gewährleitet, daß x und y verbraucht werden können.

Damit diese Bedingung gilt, ist zu beweisen, daß nur ein p existiert. Formal ausgedrückt lautet die Behauptung:

aus $p \in M(x,D_0) \cap M(y,D_0)$, folgt
$TP(x,y) = \{p \mid |M(x,p) \cap M(y,p)| = 1\}$

Beweis: Angenommen, zwei Datenobjekte p_1 und p_2 erfüllen die Bedingung, dann gilt

1) $p_{1,2} \in M(x,D_0)$
2) $p_{1,2} \in M(y,D_0)$
3) $p_1 \in TP(x,y)$
4) $p_2 \in TP(x,y)$

aus 3) folgt: $M(x,p_1) \cap M(y,p_1) = \{p_1\}$
aus 4) folgt: $M(x,p_2) \cap M(y,p_2) = \{p_2\}$

Nehmen wir an, daß auf dem Weg von x nach D_0 das Datenobjekt p_1 vor p_2 getroffen wird. Dann gilt:

aus 1) $p_1 \in M(x,p_2)$
aus 2) $p_1 \in M(y,p_2)$

Dann gilt $p_1 \in M(x,p_2) \cap M(y,p_2)$. Das ist ein Widerspruch zu 4).

RB.4: - Es gelten die folgenden Ausdrücke:
$[x,y] \in F_T$
$p = TP(x,y)$
$l \in N(p) \cap (M(x,p) \cup \{x\})$
$r \in N(p) \cap (M(y,p) \cup \{y\})$

Angenommen, l und r seien Nachfolger von p und l stehe auf dem Weg $x \rightarrow p$ sowie r auf dem Weg $y \rightarrow p$.

Dann gilt folgende Bedingung: Der Nachfolger l muß links vom

Nachfolger r stehen.

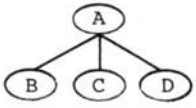
Wenn x und y verbraucht werden gewährleistet das, daß x vor y verbraucht wird.

Zur informellen Beschreibung dieser Bedingung diene die Funktion LINKS:

$$\text{LINKS} : L_O \rightarrow 2^{L_O}$$
$$\text{LINKS}(x) = T,$$

wobei T die Menge von Datenobjekte ist, die links von x stehen.

Beispiel:



$$\text{LINKS}(D) = \{B,C\}$$
$$\text{LINKS}(B) = \emptyset$$

RB.4 sagt hier aus: 1 ϵ LINKS(r); F_{TF} stellt damit eine Menge Menge von Ketten (Bild 3.13) dar, wobei jede Kette mit einem TD-Datenobjekt anfängt, das von TDF-Datenobjekten gefolgt wird.

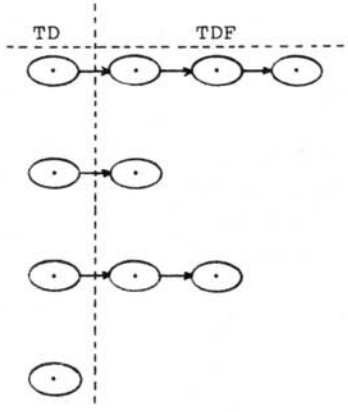


Bild 3.13: Die Abhängigkeit (1:1)

RB.5: - Wenn $[x,y] \in F_T$ dann sei

$$M(x,p) \cap K = M(x,p), \text{ wobei}$$
$$p = TP(x,y)$$

Wenn y verbraucht wird, gewährleistet diese Bedingung, daß x bereits verbraucht wurde.

RB.6: - Diese letzte Bedingung hat mit dem Gültigkeitsbereich des tag zu tun und ist implementierungsabhängig.

Angenommen, x und y haben dasselbe tag t, d.h.

$$([t,x], [t,y]) \in F_T$$

und

$$([x,p], [y,q]) \in F_{TF}$$

und die Reihenfolge, in der die Datenobjekte x,p,y,q verbraucht werden, sei

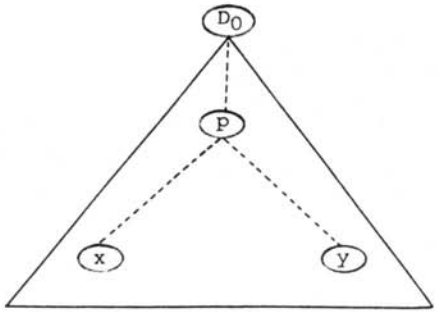
$$x \rightarrow y \rightarrow p \rightarrow q \text{ , dann gilt:}$$

Wenn y verbraucht wird, wird der Wert vom tag t geändert, bevor p verbraucht wird. Wenn p verbraucht wird, wird es daher von y und nicht von x gesteuert. Dies soll nicht sein.

Man muß daher vermeiden, daß ein Datenobjekt den Wert von t ändern darf, solange y nicht verbraucht ist.

Mit den Bezeichnungen

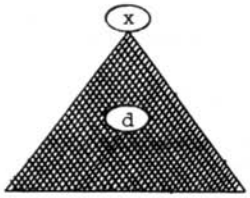
1) $DS(D_O)$
2) $[x,y] \in F_{TF}$
3) $[t,x] \in F_T$
4) $p = TP(x,y)$



stellen wir im folgenden sechs verschiedene Fälle an Einzelbeispielen dar.

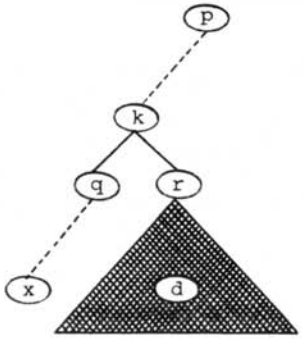
Fall 1 : Wenn d $\in$ TD - {x} und d zum Baum x gehört, dann

gilt $[t,d] \notin F_T$



Fall 2 : Sei 1) $k \in M(x,p) - \{p\}$, $k \in K$ (RB.5)
2) $q \in N(k) \cap M(x,p)$
3) $r \in N(k) - LINKS(q) - \{q\}$

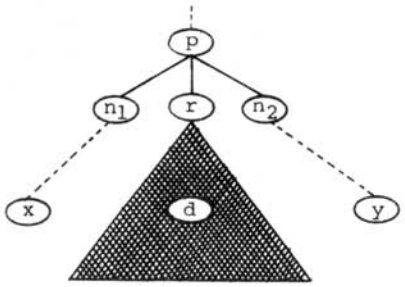
Wenn d $\in$ TD zum Baum r gehört, dann ist ebenfalls $[t,d] \notin F_T$.



Fall 3 : Sei 1) $n_1 \in N(p) \cap (M(x,p) \cup \{x\})$
2) $n_2 \in N(p) \cap (M(y,p) \cup \{y\})$
3) $r \in LINKS(n_2) - LINKS(n_1) - \{n_1\}$

Wenn $d \in TD$ zum Baum r gehört, dann ist ebenfalls

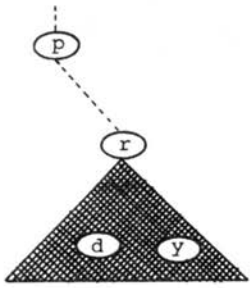
$$[t,d] \notin F_T$$



Fall 4 : Sei $r \in M(y,p) \cap I$ (r ist ein I-Datenobjekt).

Wenn $d \in TD - \{y\}$ zum Baum r gehört, dann ist ebenfalls

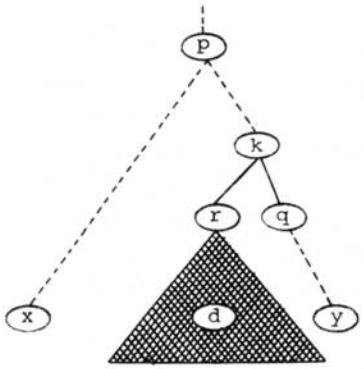
$$[t,d] \notin F_T$$



Fall 5 : Sei 1) $k \in M(y,p) \cap K$ (k ist ein K-Datenobjekt).
2) $q \in N(k) \cap M(y,p)$
3) $r \in LINKS(q)$

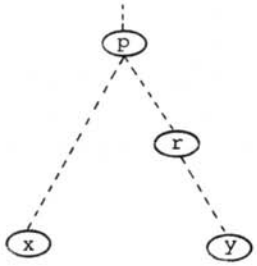
Wenn $d \in TD$ und zum Baum r gehört, dann ist ebenfalls

$$[t,d] \notin F_T$$



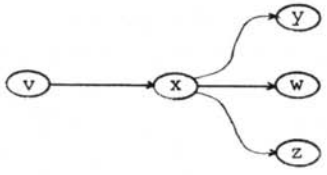
Fall 6 : Wenn r $\in$ M(y,p) $\cap$ TD (r ist ein D-Datenobjekt), dann ist ebenfalls

$$[t,r] \notin F_T$$



11.2.2 - Abhängigkeit (1: *): formale Beschreibung

Die Abhängigkeit (1: *) bedeutet, daß ein Datenobjekt viele Datenobjekte steuern, höchstens aber von einem Datenobjekt abhängen kann.



Im obigen Bild werden y,w und z werden von x gesteuert und x hängt von v ab.

In der formalen Beschreibung der Abhängigkeit (1:1) ändert sich nur die Formel 3)a) zu

$$a)\ A(x) = y \mid [x,y] \in F_{TF}\ ;\ |A(x)| \geq 0$$

Bei der Abhängigkeit (1: *) ergibt sich F_{TF} aus einem Wald von Bäumen. Die Wurzel jedes Baums ist ein TD-Datenobjekt und die anderen Knoten des Baumes sind TDF-Datenobjekte.

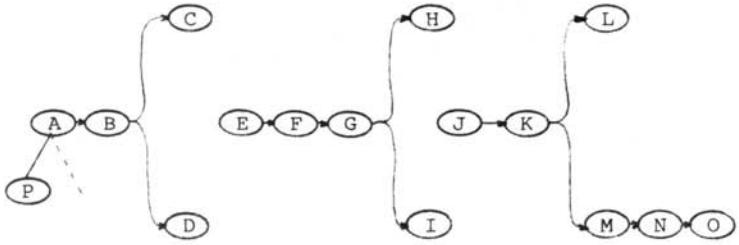


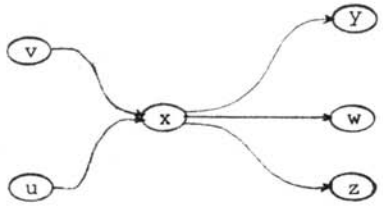
Bild 3.14: Darstellung der Abhängigkeit (1: *).

Im Bild 3.14 sind A,E und J unabhängige Datenobjekte.

Im Beispiel (Bild 3.14), werde A verbraucht und die Variante P ausgewählt. Wenn dann B verbraucht wird, so wird diejenige Variante von B gewählt, die P entspricht. Wenn B keine entsprechende Variante hat, die P entspricht, muß die Datei einen Fehler haben.

11.2.3 - Abhängigkeit(*: *): formale Beschreibung

Die Abhängigkeit (*: *) bedeutet, daß ein Datenobjekt x viele Datenobjekte steuern und von vielen Datenobjekte abhängen kann.



In der formalen Beschreibung der Abhängigkeit(1:1) ändert sich die Formel 3)b) zu

$$3)b)\ S(y) = x \mid [x,y] \in F_{TF}\ ;\ |S(y)| \geq 1$$

Für jedes Paar $[x,y] \in F_{TE}$ muß auch die Reihenfolgebedingung RB.1-6 erfüllt sein.

Bei der Abhängigkeit (*: *) ergibt sich eine Netzstruktur (Bild 3.15). Diejenigen Datenobjekte, von denen nur Pfeile ausgehen, sind TD-Datenobjekte, alle andere sind TDF-Datenobjekte. Die Pfeile zeigen eine Reihenfolge des Verbrauches. Zum Beispiel: In Bild 3.15 muß G nach F verbraucht werden.

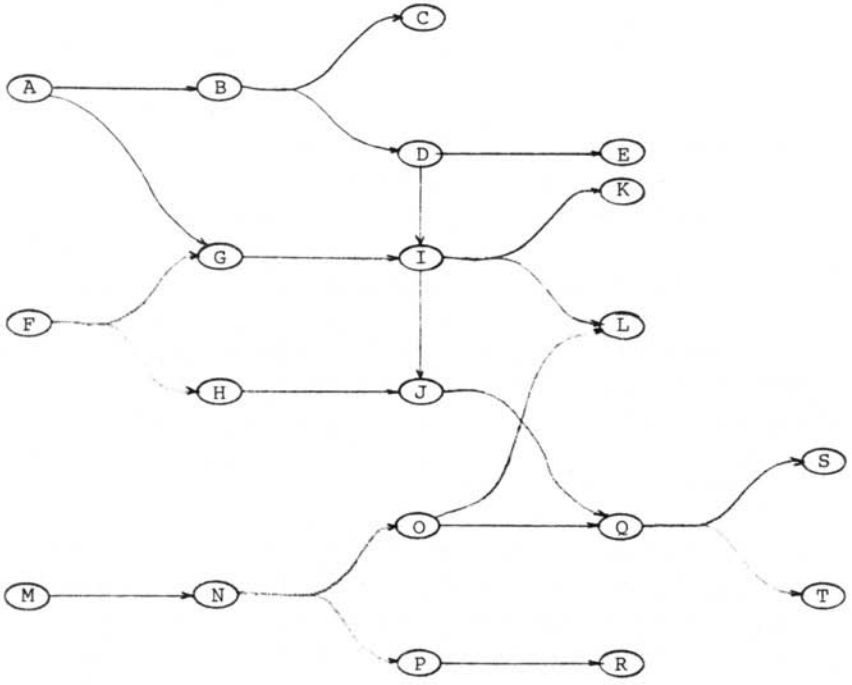


Bild 3.15: Die Darstellung der Abhängigkeit (*: *)

In Bild 3.15 hängt das Datenobjekt G von A und F ab. Das Datenobjekt A werde vor F verbraucht. Wenn G verbraucht wird, so kann dabei nur diejenige Variante von F ausgewählt werden, die gewährleistet, daß genau eine Variante von G gewählt werden kann. Wird eine andere Variante von F gewählt, so gibt es einen Fehler.

In der formalen Beschreibung der Abhängigkeit(1:1) ändert sich die Aussage 4) zu:

$$4)\ F2 = N(x) \times N(y)\ ,\ [x,y] \in F_{TF}$$

$$a)\ C(a,x,y) = b \mid [b,a] \in F_2\ ;\ |C(a,x,y)| \geq 1$$

$$b) \bigcap_{a \in N(y)} C(a,x,y) = \emptyset$$

Die Abhängigkeit(*: *) stellt den komplexesten Abhängigkeitstyp dar; die anderen Typen sind Spezialfälle. Durch eine Grammatik, die wir in Punkt 11.1.6 darstellen werden, kann der Software-Entwickler irgendeinen Typ von Abhängigkeit angeben.

11.2.4 - Der Ausprägungsbaum mit Abhängigkeit(*: *)

Der Ausprägungsbaum mit Abhängigkeit ist, ausgehend von der Datenstruktur, nicht einfach abzuleiten, weil die Datenobjekte Listen darstellen.

Dazu sei bemerkt:

1) Zwei Datenobjekte, die zu einer Datenstruktur gehören, können sich nicht bei einem I-Datenobjekt treffen, da ein solches nur genau einen Nachfolger hat. Zwei Daten können sich aber in einem Ausprägungsbaum bei einem I-Datum treffen.

2) Einem Paar $[x,y] \in F_{TF}$ der Datenstruktur entspricht im Ausprägungsbaum eine Menge von Daten-Paaren.

Für die folgenden Untersuchungen benötigen wir die Funktion W(x), die das I-Datenobjekt q, $q \in I$, $x \in L_0$, ableitet.

Definition: $W: L_0 \rightarrow L_0$

$W(x) = q$ wenn $\exists$ q so,daß $|M(x,q) \cap I| = 1$

$W(x) = D_0$ sonst

Das heißt: q ist also das erste I-Datenobjekt, das auf dem Weg von x nach D_0 getroffen wird. Wenn keines angetroffen wird, ist $W(x)=D_0$.

Die Funktion W gilt nun sowohl für die Datenstruktur als auch für den Ausprägungsbaum.

Diese Funktion W(x) wird verwendet, um die Abhängigkeit der Daten im Ausprägungsbaum aus der Abhängigkeit der Datenobjekte abzuleiten.

Wenn x das Datenobjekt y steuert, dann gilt im Ausprägungsbaum für das entsprechende Datenelement y_j ebenfalls eine Abhängigkeit von einem Elenent der Liste x. Die Menge $S(y_j)$ besteht dann aus allen Elementen der Liste x, $x \in S(y)$, die y_j steuern.

Formal gilt:

Seien 1) y ein TDF-Datenobjekt,

2) S(y) die Menge der Datenobjekte, die y steuern,

3) TD die Menge von unabhängigen Datenobjekten. Daraus wird TD' abgeleitet:

$$TD' = \bigcup_{\substack{x \in TD \\ x_i \in x}} x_i$$

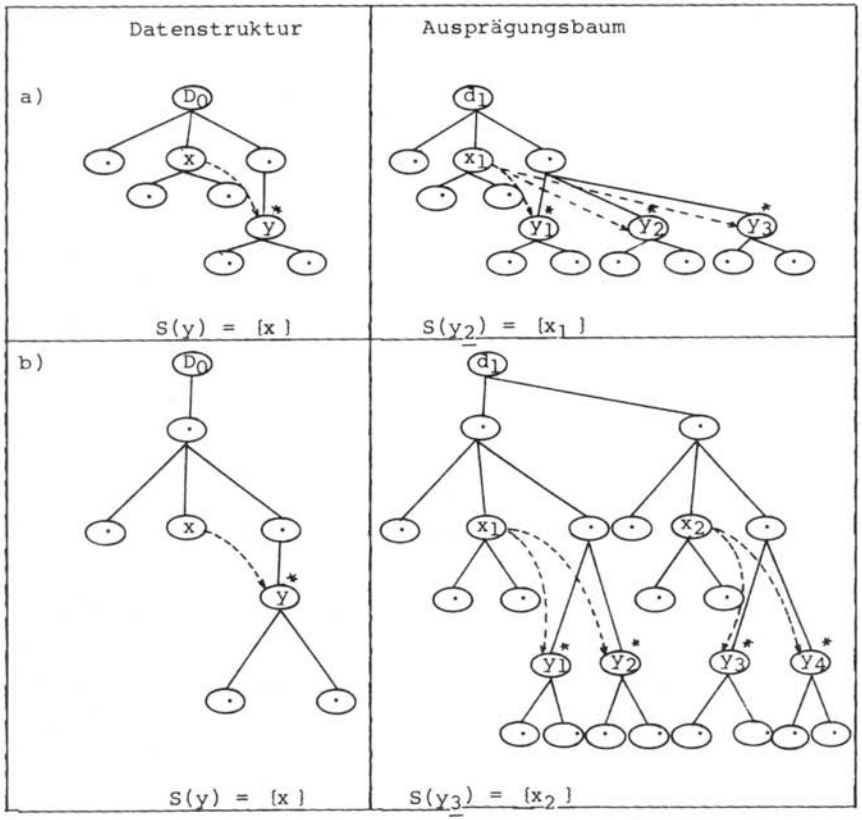
4) TDF sei die Menge von abhängigen Datenobjekten. Daraus wird TDF' abgeleitet:

$$TDF' = \bigcup_{\substack{x \in TDF \\ x_i \in x}} x_i$$

5) $F'_{TF} : TD' \times TDF' \cup TDF' \times TDF'$

$S(y_j) = \{x_i \mid [x_i, y_j] \in F'_{TF}$ mit $x_i \in x$, $x \in S(y)$ und
$|M(x_i, W(x_i)) \cap M(y_j, W(x_i))| > 1$, wenn $W(x) \in I$
oder $|M(x_i, W(x_i)) \cap M(y_j, W(x_i))| \geq 1$ sonst $\}$.

Beispiele:



Aus Beispiel a) ist ersichtlich:

- $y_2 \in Y$,
- $x \in S(y)$,
- $W(x) \notin I$
- Daher gilt: Das Element x_i aus x, das die Bedingung
  $|M(x_i, d_1) \cap M(y_2, d_1)| \geq 1$ erfüllt,
  ist x_1.

Aus Beispiel b) ist ersichtlich:

- $y_3 \in Y$,
- $x \in S(y)$,
- $W(x) \in I$
  Daraus gilt: Das Element x_i aus x, das die Bedingung
  $|M(x_i, W(x_i)) \cap M(y_3, W(x_i))| > 1$ erfüllt,
  ist x_2.

11.2.5 - Korrektur der Funktion DISJ

Die Funktion DISJ, die wir bei der Ausprägung vorgestellt haben, berücksichtigt keine Abhängigkeit. Bei der Definition von DISJ(x') wurden nur die Nachfolger von x' betrachtet. Die Definition der Funktion DISJ(x') muß so abgeändert werden, daß auch die Datenobjekte S(x') betrachtet werden. Die neue Definition von DISJ wird nur in einem Spezialfall benötigt, der in Punkt 11.2.7 ausführlich dargestellt ist.

11.2.6 - Spezifikation von Abhängigkeit

Als Syntax, mit der ein Software-Entwickler die Abhängigkeit spezifizieren kann, verwenden wir im folgenden

```
<dependency-assignment> ::= <depend-dataobject>
                            {{<controll-dataobject> =
                            {<controll-successor> : }}
                            <depend-successor> ; } end
```

11.2.7 - Ein Algorithmus zur Uberprüfung der Abhängigkeitslogik

Im folgenden wird ein Algorithmus generiert, der feststellt welche Datenobjekte in der Datenstruktur 'tot' sind. Wird nämlich daraus ein Programm abgeleitet, so wird aus einem 'toten' Datenobjekt ein Programmstück erzeugt, das nie ausgeführt wird.

'Tote' Datenobjekte sind demnach wie folgt zu definieren:

Definition: Für irgendeine Ausprägung A^m von $DS(D_0)$ und für alle Lösungen von A^m ergibt sich stets $Z(x_i) = \text{false}$ mit $x_i \epsilon x$, $x \epsilon L_0$.

In Bild 3.16 ist x ein solches 'totes' Datenobjekt.

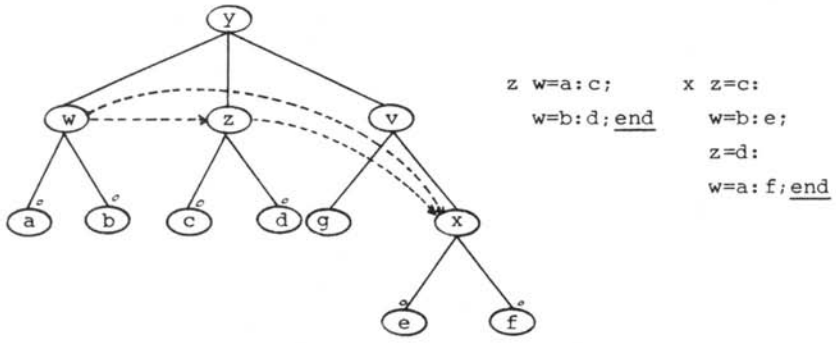


Bild 3.16: 'Totes' Datenobjekt

Wir möchten zunächst zeigen, daß man nicht alle Ausprägungen von $DS(D_0)$ untersuchen muß, sondern nur die Datenstruktur selbst, um tote Datenobjekte zu finden. Im ersten Schritt zeigen wir, daß man dazu nur eine endliche Familie von statischen Ausprägungen untersuchen muß; im zweiten Schritt zeigen wir dann, daß es

ausreicht, nur diejenigen statischen Ausprägungen zu untersuchen, für die alle I-Daten genau einen Nachfolger haben.

Da die Struktur dieser statischen Ausprägung gleich ist der Struktur von $DS(D_0)$, operiert unser Algoritmus nur über die Datenstruktur selbst.

Definition: Eine Ausprägung der Datenstruktur ist statisch, wenn alle I-Daten maximal eine Wiederholung haben. Die Anzahl von statischen Ausprägungen und auch die Anzahl von Lösungen jeder Ausprägung ist endlich.

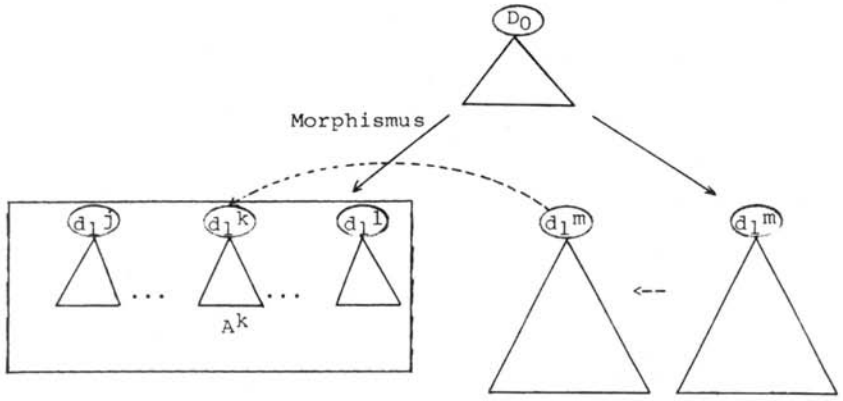
Schritt 1: Es gilt dann folgende Aussage: Wenn wir feststellen möchten, ob x ein totes Datenobjekt ist, reicht es aus, die statischen Ausprägungen von $DS(D_0)$ zu untersuchen.

Dafür ist folgendes Verfahren vorgesehen:

1 - Wähle irgendeine Ausprägung A^m von $DS(D_0)$;
2 - Finde eine Lösung für A^m;
3 - baue einen Baum B durch folgendes Verfahren auf:

- Durchsuche A^m von oben nach unten;
- Für jedes gefundene
  = K- und D-Datum: übernehme alle Nachfolger in B;
  = P-Datum : übernehme P-Daten in B;
  = I-Datum : übernehme irgendeinen Nachfolger, falls vorhanden.

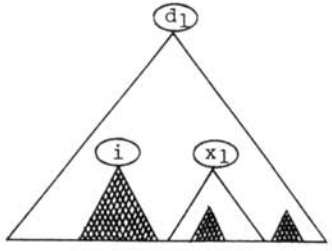
Der sich ergebende Baum B ist dann ein Morphismus einer statischen Ausprägung A^k: die Daten von B (Wurzel d_1^m) entsprechen den Daten der statischen Ausprägung A^k (Wurzel d_1^k) eins zu eins. Wenn $Z(x_i^m)$ im Morphismus 'false' ist, ist $Z(x_1^k)$ bei der entsprechenden statischen Ausprägung auch 'false'.



Familie von statischen Ausprägungen Baum B Ausprägung A^m

Schritt 2: Weiterhin gilt: Es muß nur diejenige statische Ausprägung A^k untersucht werden, in der alle I-Daten einen Nachfolger haben. Nehmen wir an, daß es für alle Lösungen von A^k ein Element x_1 gibt, $x_1 \epsilon x$, so daß $Z(x_1)$ = 'false' ist, dann folgt daraus, daß x für A^k tot ist. Die Frage ist, ob es eine andere statische Ausprägung gibt (A^n), für die dies nicht der Fall ist.

Es sei A^k der folgender Baum,



B ist die Menge der schraffierte Bäume, derer Wurzel R(i), $i \epsilon B$, I-Daten sind;

$N^*(R(i))$ sind alle direkten und indirekten Nachfolger von R(i);

A^n ist eine statische Ausprägung von $DS(D_0)$, wobei A^n aus A^k abgeleitet wird, indem für alle $i \epsilon B$ die Nachfolger $N^*(i)$ von A^k gestrichen werden.

Wenn es für A^k eine Lösung gibt, für die

$$Z(x_1) = \text{'false'}$$

und für alle $i \epsilon B$ $Z(R(i))$ = 'true' ist, und es für A^n eine Lösung gibt (s. die Funktion ITER), für die

$$Z(x_1) = \text{'true'}$$

ist, dann folgt daraus, daß $\underline{Z(x_i) \text{ in } A^n}$ 'true' wird, weil es steuernde Daten in A^k gibt, die in A^n nicht auftreten. Aber die Reihenfolgebedingung RB.5 gewährleistet, daß, wenn die abhängigen Daten in A^n auftreten, auch die steuernden Daten auftreten müssen. Das heißt, eine Ausprägung, für die

$$Z(x_1) = \text{'true'}$$

ist, existiert nicht und x stellt wie bei A^k ein totes Datenobjekt dar.

Da die Ausprägung A^k der Datenstruktur eins zu eins entspricht, betrachten wir im Algoritmus die Datenstruktur selbst.

Die Funktion DISJ, die wir in Punkt 10 definiert haben, betrachtet keine Abhängigkeiten. Wir definieren nun die Funktion DISJ im Spezialfall für A^k (also für die Datenstruktur selbst).

Es sei 1) $y \epsilon TDF$

2) $S(y)$ die von y abhängige Menge von Datenobjekten.

3) $z \epsilon N(y)$, $x \epsilon S(y)$

4) $C(z,x,y)$ (s. Punkt 11.2.3) die Menge von Nachfolgern von x, denen z entspricht, dann ist

DISJ(y) = 'true' dann genau wenn
$\exists$ z $\in$ N(y), sodaß
Z(z) = 'true',
und $\forall$n $\in$ N(y), n$\neq$z,
Z(n) = 'false'
und $\forall$x $\in$ S(y), $\exists$n $\in$ C(z,x,y), sodaß
Z(n) = 'true'
und $\forall n_1 \in C(z,x,y), n_1 \neq n$,
$Z(n_1)$ = 'false'

DISJ(y) = 'false' genau dann wenn
$\forall$n $\in$ N(y)
Z(n) = 'false'
und $\forall$x $\in$ S(y)
und $\forall n_1 \in N(y)$
und $\forall n_2 \in C(n_1,x,y)$
$Z(n_2)$ = 'false'.

Dabei bezeichnen wir ein P-Datenobjekt als 'true', wenn es auf dem Datenträger steht, andernfalls ist es false. Wir bezeichnen ferner ein Datenobjekt als 'true', wenn alle Nachfolger, die 'true' sind, auf der Datei stehen.

Algorithmus: Im folgenden wird gezeigt, wie aus der Datenstruktur ein Algorithmus generiert werden kann, der tote Datenobjekte ausgibt. (Die Beschreibungssprache ist an Algol orientiert.)

1) Die Datei, die die P-Datenobjekte enthält, wird durch eine Reihe von For-Anweisungen simuliert.

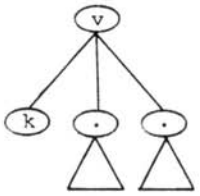
a) Es sei p ein Element der Menge P, so daß gilt

$$N(V(p)) \cap P \neq N(V(p)).$$

Nehmen wir an, daß die so ausgewählten Datenobjekte p eine Liste bilden. Für diese werden mit der 'Macro' Funktion P_1 die folgenden For-Anweisungen erzeugt:

$P_1(\langle p \rangle) \rightarrow$ for $\langle p \rangle$:= true, false do $P_1(\langle next(p) \rangle)$

Beispiel:



$P_1(k) \rightarrow$ for k := true, false do $P_1(\langle next(p) \rangle)$

b) Es seien v die Datenobjekte von L_0 außer den P-Datenobjekten, dann gilt:

$$N(v) \cap P = N(v).$$

Nehmen wir an, daß die so ausgewählten Datenobjekte v und ihre Nachfolger p Listen bilden. Die Funktion P_2 ergibt sich dann aus P_1:

$$P_1(\langle nil \rangle) \rightarrow P_2(\langle v \rangle)$$

Fall 1 - Wenn $v \in k$ oder $v \in I$, dann wird mit der Funktion P_2 folgende Anweisung erzeugt:

$P_2(\langle v \rangle) \rightarrow$ for $\langle xv \rangle^*$:= true, false do
begin
$P_3(\langle p \rangle)$
$P_2(\langle next(v) \rangle)$
end

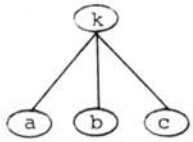
Dabei wird die Funktion P_3 für die Nachfolger von v aufgerufen:

$P_3(\langle p \rangle) \rightarrow \langle p \rangle := \langle xv \rangle ; P_3(\langle next(p) \rangle)$
$P_3(\langle nil \rangle) \rightarrow$ leere Zeichenkette

---

* xv ist der Name einer beliebigen Variable, der vom System generiert wird.

Beispiel 1 (K={...,k,...}, v={...,k,...}):



$P_2(k)$ → for ⟨xv⟩ := true,false do
begin
a:=⟨xv⟩;
b:=⟨xv⟩;
c:=⟨xv⟩;
P_2(⟨next(v)⟩)
end

Beispiel 2 (I={...,1,...}, v={...,1,...}):

$P_2(1)$ → for ⟨xv⟩ := true, false do
begin
a:=⟨xv⟩;
P_2(⟨next(v)⟩)
end

Fall 2 - Wenn v∈D ist, dann gelten die folgenden Transformationen:

P_2(⟨v⟩) → for ⟨xv⟩ := P_4(⟨p⟩) P_5(⟨next(p)⟩) ,'⟨xp⟩'* do
begin
P_6(⟨p⟩)
P_2(⟨next(v)⟩)
end

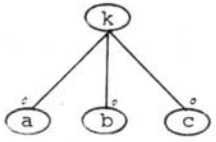
---

* ⟨xp⟩ ist eine Variable, die generiert werden muß.

$P_4(\langle p \rangle) \rightarrow$ '$\langle p \rangle$'
$P_5(\langle p \rangle) \rightarrow$, '$\langle p \rangle$' $P_5(\langle next(p) \rangle)$
$P_5(\langle nil \rangle) \rightarrow$
$P_6(\langle p \rangle) \rightarrow \langle p \rangle := \langle xv \rangle =$ '$\langle p \rangle$' ; $P_6(\langle next(p) \rangle)$
$P_6(\langle nil \rangle) \rightarrow$

Die Anzahl der Elemente der For-Liste ist hier N(v)+1, wobei das letzte Element der Liste die Möglichkeit 'falsch' darstellt.

Beispiel: (v={...,k,...}):




```
P2(k) → for <xv> := 'a','b','c', <xp> do
        begin
        a:=<xv>='a';
        b:=<xv>='b';
        c:=<xv>='c';
        P2(<next(v)>)
        end
```

2) Jedes nicht-primitive Datenobjekt x, $x \in L_0 - P$, wird durch eine boolesche Funktion dargestellt. (Diese Abbildung zeigen wir an Beispielen.)

Der Algorithmus zum Auffinden der 'toten' Datenobjekte benötigt zwei Listen, die gebraucht werden, um diese toten Datenobjekte zu suchen. Die Liste der invarianten Objekte enthält nach Beendigung des Algorithmus alle 'toten' Datenobjekte; für jeden Zyklus des Algorithmus enthält die Liste der varianten Objekte (LVO) diejenigen Datenobjekte, die 'true' geworden sind.

Bevor der Algorithmus mit der Ausführung anfängt, wird die Liste der invarianten Objekte (LIO) mit x initialisiert.

Wenn die boolesche Funktion, die für jedes Datenobjekt abgeleitet wird, für das Datenobjekt weder 'true' noch 'false' liefern kann - da z.B. die Funktion mit der Definition eines konjunktiven Datenobjekts nicht zusammenpasst - bekommt die globale boolesche Variable 'Fehler' den Wert 'true'.

Jede abgleitete Funktion hat das folgende Muster:

```
Funktion P;
        1. P berechnen;
        2. Fehler überprüfen;
        3. Fehler berechnen;
        4. P in LVO einfügen
end P
```

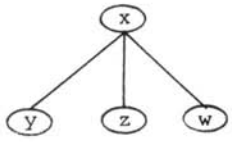
Für jedes x ϵ K$\cup$I$\cup$D gilt

a) für x ϵI

→

```
Funktion x;
        1. x:=y;
        2. if Fehler then return;
        3.
        4. if x then
              x in LVO einfügen

end x
```

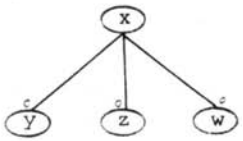
b) für x ϵK



→

```
Funktion x;
        1. x:=y ∧z ∧w;
        2. if Fehler then return;
        3. if ¬x then
              if y ∨ z ∨ w then
                 Fehler:=true,
                 return;
        4. if x then
              x in LVO einfügen
end x
```

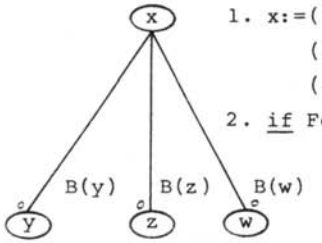
c) für x ϵTD



→

```
Funktion x;
    1. x:=(y ∧~z ∧~w) ∨( ~y ∧z ∧~w) ∨
          ( ~y ∧~z ∧w)
    2. if Fehler then return;
    3. if ~x then
          if y ∨ z ∨ w then
             Fehler:=true,
             return;
    4. if x then
          x in LVO einfügen
end x
```

e) für x ϵTDF




```
Funktion x;
1. x:=((y ∧(B(y))) ∧~(z ∧(B(z))) ∧~(w ∧(B(w)))) ∨
      (~(y ∧(B(y))) ∧(z ∧(B(z))) ∧~(w ∧(B(w)))) ∨
      (~(y ∧(B(y))) ∧~(z ∧(B(z))) ∧(w ∧(B(w)))))
2. if Fehler then return;
3. if ~x then
      if y ∨ (B(y)) ∨ z ∨ (B(z)) ∨
         w ∨ (B(w)) then
                  Fehler:=true,
                  return;
4. if x then
      x in LVO einfügem
end x
```

Die Notation B(<n>) bezeichnet in diesem Falle die Bedingung von n, wobei n ein Nachfolger von x ist, z.B. n=y → B(y).

Allgemein gilt:
Sei 1. y ein TDF-Datenobjekt
2. $n \in N(y)$
3. $x \in S(y)$
4. $e \in C(n,x,y)$ ein Nachfolger von x; e entspricht dem Nachfolger n von y

Nehmen wir an, daß
- die Nachfolger n von N(y) eine Liste bilden,
- die Datenobjekte x aus S(y) eine Liste bilden, und
- die Nachfolger e aus C(j,x,y) für jedes x auch eine Liste bilden;

Dann wird $B(\langle n_i \rangle)$, die Bedingung des i-ten Nachfolges von y, wie folgt abgeleitet. (s. Punkt 11.2.5)

$B(\langle n_i \rangle) \rightarrow (B_1(\langle x \rangle)) B_2(\langle next(x) \rangle)$
$B_1(\langle x \rangle) \rightarrow \langle e \rangle B_3(\langle next(e) \rangle)$
$B_2(\langle x \rangle) \rightarrow \wedge (B_1(\langle x \rangle)) B_2(\langle next(x) \rangle)$
$B_2(\langle nil \rangle) \rightarrow$
$B_3(\langle e \rangle) \rightarrow \vee \langle e \rangle B_3(\langle next(e) \rangle)$
$B_3(\langle nil \rangle) \rightarrow$

Das Ergebnis läßt sich damit wie folgt zusammenfassen:
Die Funktion x ist 'true', wenn nur <u>ein</u> Nachfolger von x und seine Bedingung 'true' sind.

Die Funktion ist 'false', wenn alle Nachfolger und Bedingungen 'false' sind. In allen anderen Fällen gibt es einen Fehler.

Durch P_1 und P_2 haben wir eine Reihe von For-Anweisungen abgeleitet:

<u>for</u>...<u>for</u>...<u>for</u>...<u>for</u>...$P_2(\langle nil \rangle)$ <u>end</u>...<u>end</u>...<u>end</u>...<u>end</u>

Die For-Anweisungen haben die Aufgabe, den P-Datenobjekten Werte

zuzuweisen. Es ist jetzt noch das Verfahren für die Funktion, die der Wurzel der Datenstruktur entspricht, herzuleiten:

```
P2(<nil>) → begin
            Fehler:= false;
            if   <D0>
            then if ~ Fehler
                 then  Variante Datenobjekte löschen,
                       Lösung := Lösung + 1;
            LVO löschen
            end
```

Bemerkung: 'Variante Datenobjekte löschen' ist eine Prozedur, die diejenigen Datenobjekte aus der Liste der invarianten Objekte löscht, die auch auf der Liste der varianten Objekte stehen.

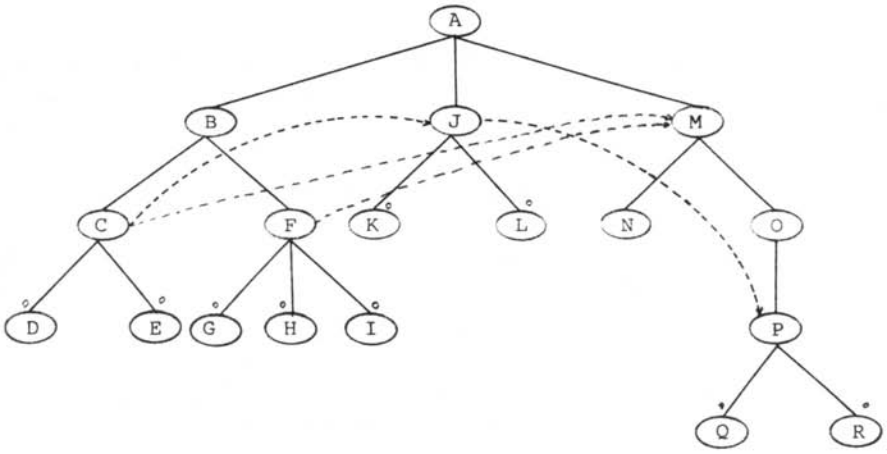
Der Algorithmus ist abgeschlossen, wenn nach dem letzten end die Liste der invarianten Objekte und die Anzahl der Lösungen ausgedruckt ist: Dafür sind zwei Anweisungen vorgesehen:

```
'Invariante Liste ausdrucken';
'Lösung ausdrucken'
```

Der Algorithmus wird im folgenden Diagramm zusammenfassend dargestellt:

For
For
Fehler := false
Do
else
then
Fehler
else
then
Variante Datenobjekte loeschen
Loesung := Loesung + 1
Variante Liste loeschen
Invariante Liste ausdrucken
Loesung ausdrucken

Beispiel.
Die folgende Datenstruktur



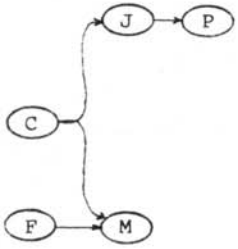
ist so aufgebaut, daß sie die Abhängigkeits-Bedingungen (einschließlich RB.1-5) erfüllt.

Die Abhängigkeit wird wie folgt spezifiziert (s. Punkt 12.5):

<dependency assignment> :

```
J C=D:K;      M F=G:          P J=K:Q;
  C=E:L end     C=D:N;          J=L:R end
                C=E:
                F=H:I:O end
```

Daraus folgt die Darstellung:



1) Die folgenden For-Anweisungen werden wie folgt abgeleitet:

a) Ableitung der For-Anweisungen von $p \epsilon P$ so, daß

$$N(V(p)) \cap P \neq N(V(p))$$

for N := true,false do P_1(<next(p)>)

b) Ableitung der For-Anweisungen für $v \epsilon L_0$ so, daß

$$N(v) \cap P = N(v)$$

Fall 1 - v ϵ K ∨ v ϵ I: entfällt.

Fall 2 - v ϵ D: Es gibt die Datenobjekte C,F,J,P; für jedes wird nun eine For-Anweisung hergestellt:

```
P2(C)  →  for XC* ='D','E','F'** do
          begin
          D:=XC='D';
          E:=XC='E';
          P2( <next(v)> )
          end

P2(F)  →  for XF:= 'G','H','I','F' do
          begin
          G:=XF='G';
          H:=XF='H';
          I:=XF='I';
          P2( <next(v)> )
          end
```

---

* Angenommen: <xv> := X+ v; X+v ist kein Datenobjekt.

** Angenommen: <xp> = 'F'

```
P2(J)  →  for XJ:= 'K','L','F' do
          begin
          K:=XJ='K';
          L:=XJ='L';
          P2( <next(v)> )
          end

P2(P)  →  for XP:='Q','R','F' do
          begin
          Q:=XP='Q';
          R:=XP='R';
          P2( <next(p)> )
          end
```

2) Die Ableitung der Funktionen liefert für die verschiedenen Typen von Datenobjekten das folgende:

a) $x \in I$, $I = \{O\}$

```
Function O;
        1. O:=P;
        2. if Fehler then return;
        3.
        4. if O then
              O in LVO einfügen;
end O
```

b) $x \in K$, $K = \{A,B\}$

```
Function A;
        1. A:=B ∧J ∧M;
        2. if Fehler then return;
        3. if ¬A then
              if B ∨J ∨M then
                 Fehler:= true,
                 return;
        4. if A then
              A in LVO einfügen;
end A
```

```
Function B;
        1. B:=C ∧F;
        2. if Fehler then return;
        3. if ~B then
             if C ∨F then
                Fehler :=true,
                return;
        4. if B then
             B in LVO einfügen
end B
```

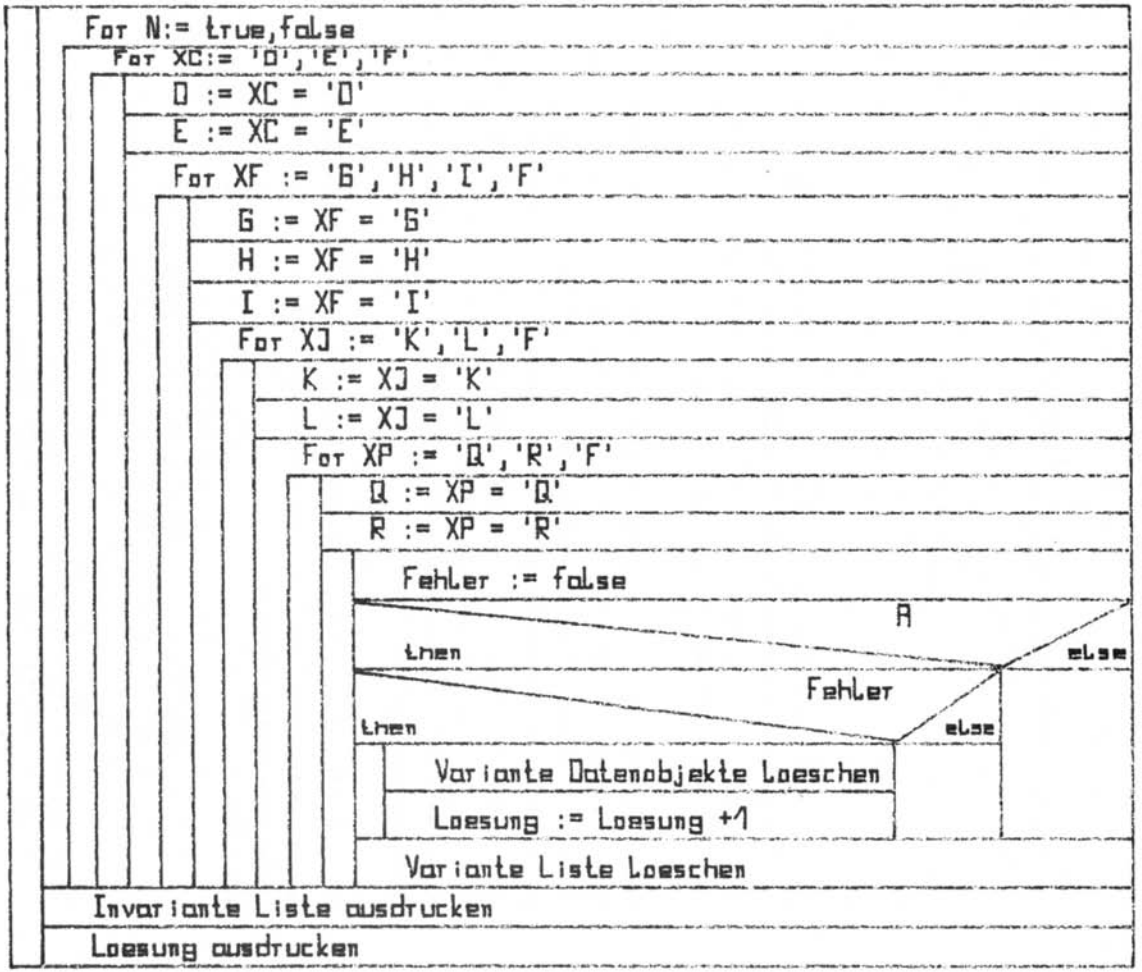
c) $x \in TD$, $TD = \{C, F\}$

```
Function C;
        1. C:= (D ∧~E) ∨ (~D ∧E);
        2. if Fehler then retun;
        3. if ~C then
             if D ∨E then
                Fehler:=true,
                return;
        4. if C then
             C in LVO einfügen
end C

Function F;
        1. F:=(G ∧~H ∧~I) ∨( ~G ∧H ∧~I) ∨( ~G ∧~H ∧I)
        2. if Fehler then return;
        3. if ~F then
             if G ∨H ∨I then
                Fehler:=true,
                return;
        4. if F then
             F in LVO einfügen
end F
```

d) x ∉TDF, TDF= {J,P,M}

```
Function J;
      1. J:=((K∧((D)))∧~(L∧((E))))∨
            (~(K∧((D)))∧(E∧((L))));
      2. if Fehler then retun;
      3. if ~J then
            if K∨((D))∨L∨((E))
               then Fehler:= true,
               return;
      4. if J then
            J in LVO einfügen
end J

Function P;
      1. P:= ((Q∧((K)))∧~(R∧((L))))∨
            (~(Q∧((K)))∧(R∧((L))))
      2. if Fehler then return;
      3. if ~P then
            if Q ∨ (K) ∨ R ∨ (L) then
               Fehler:=true,
               return;
      4. if P then
            P in LVO einfügen
end P

Function M;
      1. M:= ((N∧((D)∧(G)))∧~(O∧((E)∧(H∨I))))∨
            (~(N∧((D)∧(G)))∧(O∧((E)∧(H∨I))))
      2. if Fehler then return;
      3. if ~M then
            if N∨((D)∧(G))∨O ∨((E)∧(H∨I)) then
            Fehler:= true,
            return;
      4. if M then
            M in LVO einfügen
end M
```

Der Algorithmus wird im folgenden Diagramm zusamenfassend gezeigt.


For N:= true,false
For XC:= 'D','E','F'
D := XC = 'D'
E := XC = 'E'
For XF := 'G','H','I','F'
G := XF = 'G'
H := XF = 'H'
I := XF = 'I'
For XJ := 'K','L','F'
K := XJ = 'K'
L := XJ = 'L'
For XP := 'Q','R','F'
Q := XP = 'Q'
R := XP = 'R'
Fehler := false
A
then
else
Fehler
then
else
Variante Datenobjekte Loeschen
Loesung := Loesung +1
Variante Liste Loeschen
Invariante Liste ausdrucken
Loesung ausdrucken

11.2.8 - Kommentare über die Abhängigkeit von disjunktiv Datenobjekten

1. Wenn ein Datenobjekt x das Datenobjekt y steuert, fordern wir in Punkt 11.2.1.1, daß

$$M(x,p) \cap K = M(x,p)$$

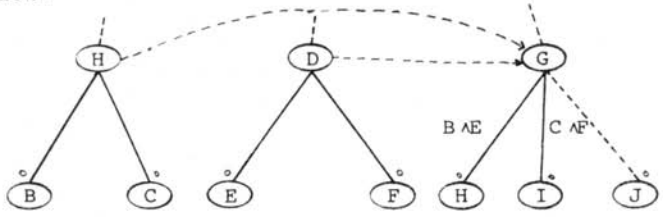
wobei

$$p=TP(x,y).$$

Wenn y verbraucht wird, ist es möglich alle Bedingungen berechnen, dann ist x sicher (s. RB.5) bereits verbraucht. Das heißt, daß x als 'tag' (s. Punkt 11.1.2) einen 'Wert', der dem ausgewählten Nachfolger entspricht, besitzt.

Wenn wir erlauben, daß auf dem Weg von x nach p mindestens ein D-Datenobjekte liegt, könnte es geschehen, daß, wenn y verbraucht wird, die Bedingungen für die Nachfolger von y entweder nicht berechnet werden können, (denn x ist unbestimmt) oder ein alter Wert von x wieder verwendet wird. Die Lösung mit der Initialisierung des tag ermöglicht es immer noch, daß ein alter Wert des Tag verwendet wird. An dieser Einschränkung soll festgehalten werden.

2. Wenn der Benutzer bei der Erstellung der Datenstruktur die Abhängigkeit festlegt, kann es geschehen, daß diese Abhängigkeit nicht alle Fälle abdeckt.
Beispiel:

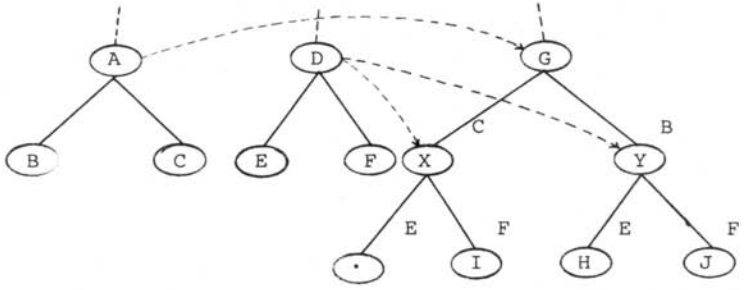


Stellen wir uns vor, daß auch B ∧F richtig wäre, dann ermöglicht es diese Datenstruktur nicht, diesen Fall abzudecken.

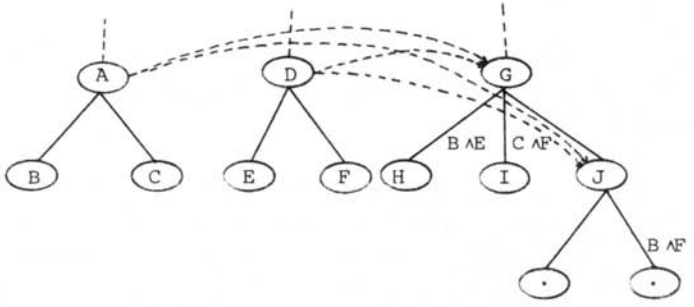
Eine Lösung wäre hier, einen Nachfolger von G ohne Bedingung zu erlauben. Dies würde bedeuten, daß J auch für alle anderen Fälle gilt. (Bei Programmiersprachen entspricht das einem Case mit otherwise.)

Wir könnten zusätzlich die Abhängigkeitsbeschreibung ändern, um diese Eigenschaft einzuführen. Dies soll aber nicht geschehen, weil es immer eine Lösung gibt:

a) Der Benutzer kann die Datenstruktur immer so ändern, daß die anderen Fälle ausgedrückt werden können, (wie unten gezeigt)



b) Die Datenstruktur wäre schlecht zu interpretieren und zu korrigieren: Damit der Benutzer die Bedingung B ∧F darstellen kann, sollte er die Datenstruktur so erstellen:



G und J werden von A und D gesteuert. Sie sind also Klassen von Datenobjekten, die beide von A und D abgeleitet werden und dieselbe hierarchische Ebene besetzen müssen. J als Nachfolger

von G bringt daher Unverständnis bei der Interpretation der Datenstruktur.

3. Bei einer Datenstruktur ohne Abhängigkeit ist es für die Nachfolger von K-Datenobjekten gleichgültig, welche Reihenfolge sie auf der Datei haben. Jeder Nachfolger ist von den anderen unabhängig. Bei einer Datenstruktur mit Abhängigkeit fordern wir, daß die P-Datenobjekte, die auf der Datei stehen, eine bestimmte Reihenfolge haben, so daß die steuernden Datenobjekte davor kommmen.

4. Wenn die Bedingungen der Nachfolger eines Datenobjektes berechnet werden und keine 'true' ist, so gehen wir davon aus, daß die Datenstruktur nicht richtig aufgebaut wurde, sondern daß die Datei FEHLER enthält.

Wir legen fest, daß eine Datenstruktur richtig aufgebaut wurde, wenn der Algorithmus gezeigt hat, daß sie wenigstens eine Lösung und kein 'totes' Datenobjekt hat.

12 - Die Iterativ-Datenobjekten

Auf dem Datenträger werden die Nachfolger eines I-Datums, die eine Liste bilden, sequentiell abgelegt. Wir brauchen damit ein Kennzeichen, das das Ende der Liste signalisiert.

Der Bereich eines I-Datenobjekts wird durch MIN und MAX spezifiziert, wobei MIN die untere Grenze des Bereichs und MAX die obere Grenze des Bereichs ist*.

MIN und MAX können Einschränkungen aus der realen Welt oder der Technik sein. Wenn man beispielsweise aussagt, daß ein Student in einem Semester nicht mehr als 10 Fächer hören darf, ist das

* Diese Bereiche werden in der mathematischen Logik Quantoren genannt /Rap-82/. Wir verwenden hier den Ausdruck Bereich mit oberer Grenze (MAX) und unterer Grenze (MIN), da das Quantor-Konzept zu allgemein ist.

eine Einschränkung aus der realen Welt. Wenn man aber aussagt, daß ein Datenträger für ein bestimmtes Problem nicht mehr als 1000 Blöcke enthalten kann, ist das eine technische Einschränkung.

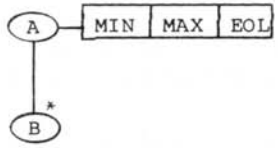
Im Zusammenhang mit Ausprägungsbaum und Lösung ist die untere bzw. obere Grenze des Bereichs die minimale bzw. maximale Anzahl der Nachfolger eines I-Datums, die 'true' sein müssen, wenn das I-Datum verbraucht wird.

Ein I-Datenobjekt mit unterer Grenze des Bereichs gleich 0 und oberer gleich 1 ist wird 'optional' genannt. Daß heißt, daß dann ein Datum x_i einen oder keinen Nachfolger hat. Das 'optionale' Datenobjekt kann dann auch durch ein D-Datenobjekt spezifiziert werden.

12.1 - Darstellung des EOL und der Bereiche

EOL ist eine Konstante, die das Ende der Liste signalisiert. Dadurch wird die Erstellung des Dateilesers vereinfacht.

Um die Bereiche und EOL besser darstellen zu können, dient die folgende Darstellung.



A ist ein I-Datenobjekt; MIN und MAX sind die minimale bzw. maximale Anzahl von B, und EOL ist das Ende der Liste.

Das Datum a_i ($a_i \in A$), wird verbraucht, wenn alle Nachfolger $\{b_j,\ldots,b_m\}$ bereits verbraucht wurden und nach b_m EOL erkannt wurde.

12.2 - Beziehungen zwischen den Bereichsgrenzen

Es seien die Funktionen MIN und MAX definiert duch

$$MIN : L_0 \rightarrow \mathbb{N} \cup \{'*'\}$$
$$MAX : L_0 \rightarrow \mathbb{N} \cup \{'*'\}$$

Der Ausdruck MIN(x) = '*' bzw. MAX(x) = '*' soll bedeuten, daß die Bereichsgrenzen keinen festen Wert haben.

Die folgende Tabelle zeigt die richtigen Angaben für die Bereichsgrenzen:

$x \in I$, $n \in \mathbb{N}$

MIN \ MAX	n	'*'
n	if MIN(x) ≤ MAX(x) then true else false	true
'*'	true	true

Wenn MIN(x)='*' ∧ MAX(x) $\in \mathbb{N}$
dann $0 \leq MIN(x) \leq MAX(x)$

Wenn MIN(x) $\in \mathbb{N}$ ∧ MAX(x) = '*'
dann $MAX(x) \geq MIN(x)$

Wenn MIN(x) = '*' ∧ MAX(x) = '*'
dann $MIN(x) = MAX(x) \geq 0$

12.3 - Beziehungen zwischen den Bereichsgrenzen und dem EOL

Es sei die Funktion EOL wie folgt gegeben:

$EOL: L_0 \rightarrow$ Konstante $\cup$ {leer}

Die folgende Tabelle zeigt dann die Beziehungen zwischen den Bereichsgrenzen und dem EOL.

$x \in I,\ n \in N$

MAX / MIN	n	'*'
n	if MAX(x)=MIN(x) then EOL(x) = leer else EOL(x) = Konstante	EOL(x) = Konstante
'*'	EOL(x) = Konstante	EOL(x) = Konstante

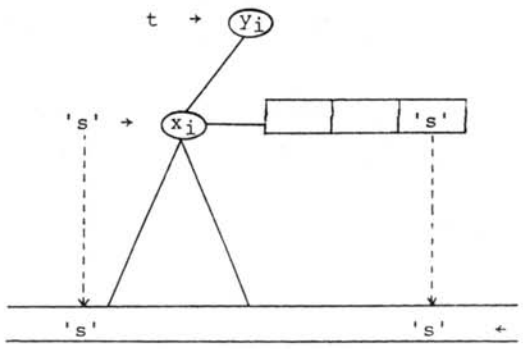
12.4 - Zweideutigkeit bei der Interpretation des EOL

Die Konstante, die ein I-Datenobjekt als EOL erhalten kann, kann explizit oder implizit angegeben werden. Das EOL wird dann implizit angegeben , wenn das I-Datenobjekt selbst als EOL verwendet wird.

Nehmen wir an, daß

1) $x \in I$
2) $y \in TD,\ [y,x] \in F$
3) $[t,y] \in F_T\ ,\ [s,x] \in F_1$
4) EOL(x) = 's', s='s'

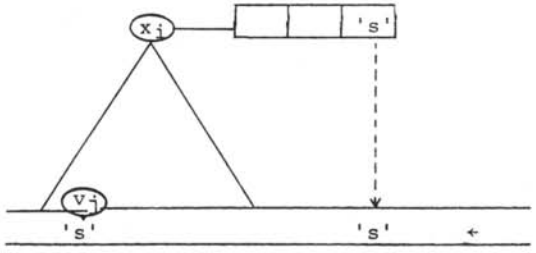
dann kann die Konstante 's' zweimal auf der Datei erscheinen. Es gibt aber keine Gefahr für eine Zweideutigkeit, da 's' als tag-Wert zuerst vorkommt.



Nehmen wir aber an, daß

1. $x \in I$
2. $\exists v$, so daß $x \in M(v, D_0)$ mit $v \in V$ (V ist die Menge der P-Datenobjekte, die Variable sind)
3. $\exists\ s \in v$, so daß s='s' *)
4. EOL(x)='s'

dann kann 's' zweimal auf der Datei erscheinen; wenn x verbraucht wird, wird nach EOL abgefragt. EOL wird also sofort erkannt und es wird so interpretiert, als ob die Liste leer wäre, weil 's' das erste Datum ist, das verarbeitet werden muß.



Bei Überprüfung der Datenstruktur kann dieser Fall festgestellt und der Benutzer gewarnt werden.

*) stellt eine Menge von Zeichenketten dar, die ausgehend von einem Alphabet erzeugt werden.

Das ist nicht die einzige Möglichkeit, wie das EOL mit anderen Zeichenketten verwechselt werden kann.

Allgemein gilt: Sei - $x \in I$
- y, so daß
$x \notin M(y, D_0)$;

dann kann y Zweideutigkeit bringen, wenn

$$(y \in I \wedge EOL(y)=EOL(x)) \vee$$
$$(y \in TD \wedge \exists\, s \in N([t,y]),\ [t,y] \in F_T, \text{ so daß, } s=EOL(x)) \vee$$
$$(y \in V \wedge \exists\, s \in y \text{ so daß, } s=EOL(x))$$

Ein solches y liefert Zweideutigkeit, wenn

$$M(y,x) \cap TD = \emptyset$$
$$\text{und } \forall n \in M(y,x) \cap K$$
$$l \notin M(y,n) \cap N(n)$$
$$\text{und } LINKS(l) = \emptyset.$$

12.5 - Spezifikation der Bereichsgrenzen und EOL

Wir möchten die Syntax der Bereichsgrenzen und des EOL so festlegen zeigen,wie sie der Software-Entwickler spezifizieren kann.

```
<domain-assignment> ::= <iteration-dataobject>
                        [min = <integer> , ]
                        [max = <integer> , ]
                        [eol = <string> ] end
```

Das '[' Symbol bedeutet 'optional'. Wenn über die Bereichsgrenzen und über das EOL eines Datenobjektes x, $x \in I$, nichts ausgesagt wird, dann gilt:

$$MIN(x) = '*'$$
$$MAX(x) = '*'$$
$$EOL(x) = 'x'$$

IV - KONTROLLSTRUKTUREN

Ziel dieses Kapitels ist, aus der Datenstruktur ein Programm, das als Dateileser bezeichnet wird, abzuleiten. Das Programm holt (liest) die Daten für Datenobjekte aus einer Datei, die der Datenstruktur entspricht.

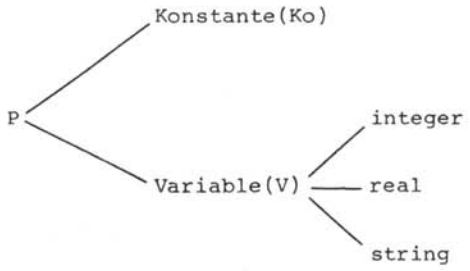
Ergänzend werden noch zwei weitere Programme erzeugt:

- Ein 'Report-Program' (R-Programm); es hat die Aufgabe, ein Formular und seine Daten (letztere liegen auf der Datei) auf das Ausgabemedium zu bringen.

- Ein 'Up-date Program' (Up-Programm); es hat die Aufgabe, die Daten für D_0, die Wurzel der Datenstruktur, auf den neuesten Stand zu bringen.

Im folgenden empfiehlt sich als Vereinfachung, nur mit expliziten Kontrolldaten zu arbeiten, das heißt, die TD-Datenobjekte als 'tags' und ihre Nachfolger als tag-Werte (Konstante) zu verwenden.

Für die P-Datenobjekte gilt zusammenfassend:



$P = Ko \cup V$

Für die Ableitung der genannten Programme ist folgende Notation zweckmäßig:

Definition: R-Datenobjekte sind P-Datenobjekte, deren 'true' P-Daten (Ausprägung) auf der Datei stehen. Das sind

R_1 : alle Datenobjekte x, so daß
$x \in V \wedge V(x) \notin TD$

R_2 : alle Datenobjekte x, so daß
$x \in V \wedge V(x) \in TD$

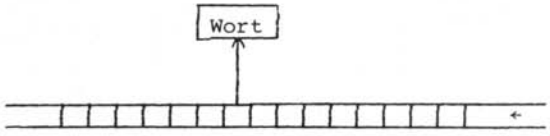
R_3 : alle Datenobjekte x, so daß
$x \in Ko \wedge V(x) \in TD$

Definiton: W-Datenobjekte sind P-Datenobjekte, derer 'true' P-Daten (Ausprägung) nicht auf der Datei stehen müssen. (Sie können abgeleitet werden). Für W-Datenobjekte gilt damit

$$x \in Ko \wedge V(x) \notin TD.$$

Im Text der Programme treten folgende Operationen auf:

a) Read(Wort): Diese Operation liest die Datei Einheit für Einheit und übergibt den Inhalt des gelesenen Elements an die String-Variable Wort.



b) <x>:=Wort: Der Inhalt der Variablen Wort wird in x gespeichert.

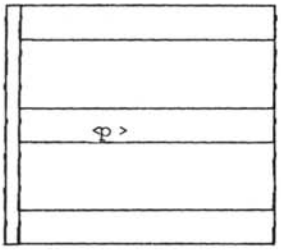
c)ENDE(<s>) ist eine boolesche Funktion; Sie wird 'true', wenn das Ende der Liste eines I-Datums erreicht ist. d.h, wenn Wort das 'end-of-list' s enthält:

ENDE := <s>=Wort;

Für die Syntax des abzuleitende Programms <p> soll folgende Darstellung, die sich am Struktogramm von Nassi-Schneidermann anlehnt, verwendet werden:

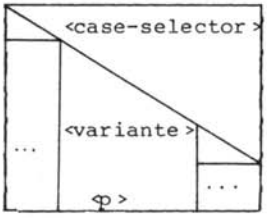
<p> ::= <seq> | <case-K> | <case-A> | <while> | <prim>

<seq> ::=



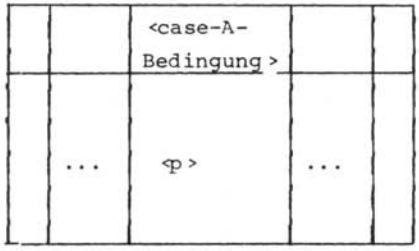
Die Programmanweisungen bilden eine Sequenz.

<case-K> ::=



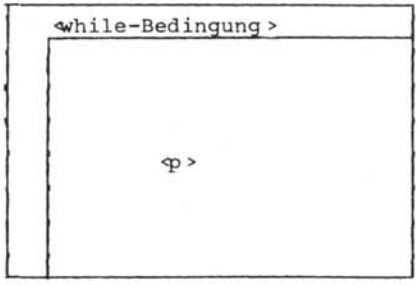
Die Programmanweisungen bilden eine mehrfache Verzweigung.

<case-A> ::=



Die Programmanweisungen bilden eine mehrfache Verzweigung mit expliziter Bedingung.

<while> ::=



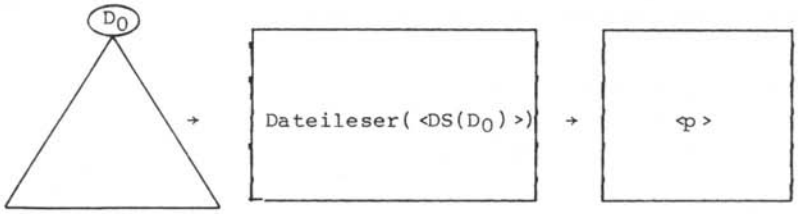
Das Programmanweisung bildet eine Wiederholung.

<prim> ::= <op>

<op> eine primitive Operation

1. Dateileser

Die Herleitung des Dateilesers stellt sich dann wie folgt dar:



Datenstruktur Ableitungsfunktion Programm

$DS(D_0)$ sei eine Datenstruktur. Im einzelnen erfolgt die Ableitung in Abhängigkeit des Typs der Datenobjekte.

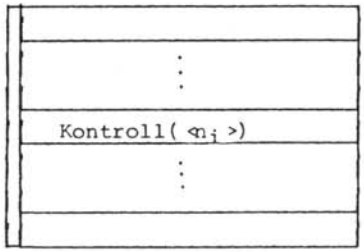
a) Die Funktion Dateileser erzeugt folgende Kontrollstruktur:

Dateileser(<DS(D_0)>) →

Read(Wort)
Kontroll(<D_0>)

b) Die Funktion Kontroll(<x>) wird dann rekursiv verfeinert :

b.1) Für konjunktive Datenobjekte:

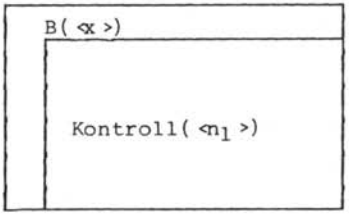
Kontroll(⟨x⟩) → 

, wenn x ∈K

Wenn die Nachfolger von x eine Liste bilden, bezeichnet n_i den i-ten Nachfolger von x.

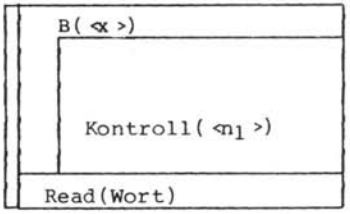
b.2) Für iterative Datenobjekte:

b.2.1) mit fehlender 'end-of-list':

Kontroll(⟨x⟩) → 

, wenn x ∈I ∧ EOL(x)=leer

b.2.2) mit 'end-of-list':

Kontroll(⟨x⟩) → 

, wenn x ∈I ∧ EOL(x) ≠leer

B(⟨x⟩) wird für den allgemeinen Fall wie folgt abgeleitet:

$$B(\langle x\rangle) \rightarrow \langle x\rangle \leq B_1(\langle x\rangle) \vee (\langle x\rangle \leq B_2(\langle x\rangle) \wedge \sim \text{ENDE}('B_3\langle x\rangle'))$$

$B_1(\langle x \rangle)$, $B_2(\langle x \rangle)$ und $B_3(\langle x \rangle)$ bezeichnen MIN(x),MAX(x) bzw. EOL(x).

Das Datenobjekt x wird hier selbst als Zähler verwendet. Bei der Codegenerierung sollte diese Variable generiert werden, da x auch als tag verwendet werden kann. Wenn die Schleife zum Ende kommt, weil die maximale Anzahl von Wiedeholungen ($\langle x \rangle > B(\langle x \rangle)$) erreicht ist, muß dann die Variable Wort 'end-of-list' enthalten, sonst sind die Daten falsch.

B_1,B_2 und B_3 erzeugen, wie im folgenden dargestellt, die Bereichsgrenzen MIN(x),MAX(x) und EOL(x), die eine geordnete Liste mit den Elementen p bilden.

```
B1(<x>) → B4(<p>);                    /*  MIN(x)  */
B2(<x>) → B4(<next(p)>);              /*  MAX(x)  */
B3(<x>) → B4(<next(next(p))>);        /*  EOL(x)  */
B4(<p>) → <p>
```

Beispiel: Die entsprechende Anweisung für die While-Kontrollstruktur lautet (für MIN=5, MAX=10 und EOL=x) in Algol wie folgt:

```
for x:=1 step 1 while x≤5 ∨ ( x ≤ 10 ∧ ~ ENDE('x'))
```

Bei iterativen Datenobjekten kann man 5 Fälle feststellen:

1 - (EOL(x) ≠ leer) ∧ (MIN(x) ∈ N) ∧ (MAX(x) ∈ N),

dann ergibt sich der allgemeine Fall.

2 - (MIN(x) ∈ N) ∧ (MAX(x) = '*'), dann ergibt sich

$B(\langle x \rangle) \rightarrow \langle x \rangle \leq B_1(\langle x \rangle) \vee \sim ENDE('B_3(\langle x \rangle)')$

3 - (MIN(x) = '*') $\wedge$ (MAX(x) ϵ N), dann ergibt sich

$B(\langle x \rangle) \rightarrow \langle x \rangle \leq B_2(\langle x \rangle) \wedge \sim ENDE('B_3(\langle x \rangle)')$

4 - (MIN(x) = '*') $\wedge$ (MAX(x) = '*'), dann ergibt sich

$B(\langle x \rangle) \rightarrow \sim ENDE('B_3(\langle x \rangle)')$

5 - EOL(x) = leer, dann ergibt sich

$B(\langle x \rangle) \rightarrow \langle x \rangle \leq B_1(\langle x \rangle)$

b.3) für die disjunktive Datenobjekte,

b.3.1) die abhängig sind:

Kontroll($\langle x \rangle$) $\rightarrow$

...	$B(\langle n_i \rangle)$	...
...	Kontroll($\langle n_i \rangle$)	...

,

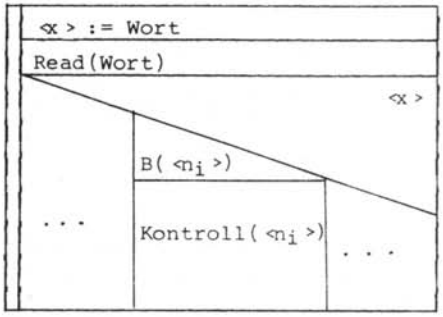
wenn $x \epsilon TDF$

$B(\langle n_i \rangle)$ bezeichnet die Bedingung von n_i, wobei n_i der i-te Nachfolger von x ist. Wenn keine oder mehrere Bedingung(en) 'true' sind, sind die Daten falsch.

Für $B(\langle n_i \rangle)$ gilt die Ableitung, die wir in Kapitel III, Punkt 11.2.7 entwickelt haben.

b.3.2) die unabhängig sind:

Kontroll(<x>) →



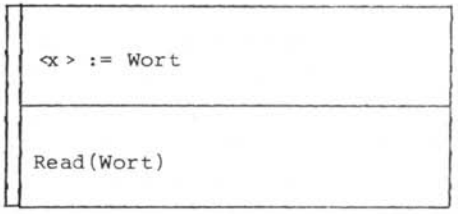
, wenn x ∈ TD

B(<n_i>) → 'n_i'

Wenn das tag <x> keinen Wert 'n_i' enthält, sind die Daten falsch.

b.4) Für die primitiven Datenobjekte gilt:

Kontroll(<x>) →



,

wenn x ∈ R_1 ∨ x ∈ R_2

Kontroll(<x>) →

*

,

wenn x ∈ R_3 ∨ x ∈ W

---

* + bedeutet eine unspezifizierte Operation. Das zeigt, daß der Dateileser allein noch kein Anwenderprogramm ist.

1.2 - W-Datenobjekt Bedingung

Bei der Ableitung eines Programms kann der Fall auftreten, daß ein Programm entsteht, das nicht frei von unerwünschten (unendlichen) Schleifen ist. Dieser Fall tritt dann ein, wenn

a) es ein iteratives Datenobjekt x mit 'end-of-list' gibt;

b) es eine Lösung der Datenstruktur gibt, für die Z(x) den Wert 'true' liefert und alle primitive Datenobjekte, die im Baum mit der Wurzel x stehen und für die Z(y)='true' gilt, W-Datenobjekte sind.

c) es auf dem Weg von y nach x kein TD-Datenobjekt gibt, daß also gilt

$$M(x,y) \cap TD = \emptyset$$

Der Grund für diese letzte Bedingung ist, daß aus den TD-Datenobjekten READ-Operationen abgeleitet werden. Wäre ein TD-Datenobjekt auf dem Weg von y nach x, dann wird eine READ-Operation ausgeführt und der Datenträger geht vorwärts.

Kann für die unter den genannten Bedingungen primitiven Datenobjekte y keine READ-Operation abgeleitet werden, bleibt der Datenträger stehen und das 'End-of-list' kann nicht gelesen werden.

1.3 - Beispiel für Dateileser:

Gegeben sei die folgende Datenstruktur:

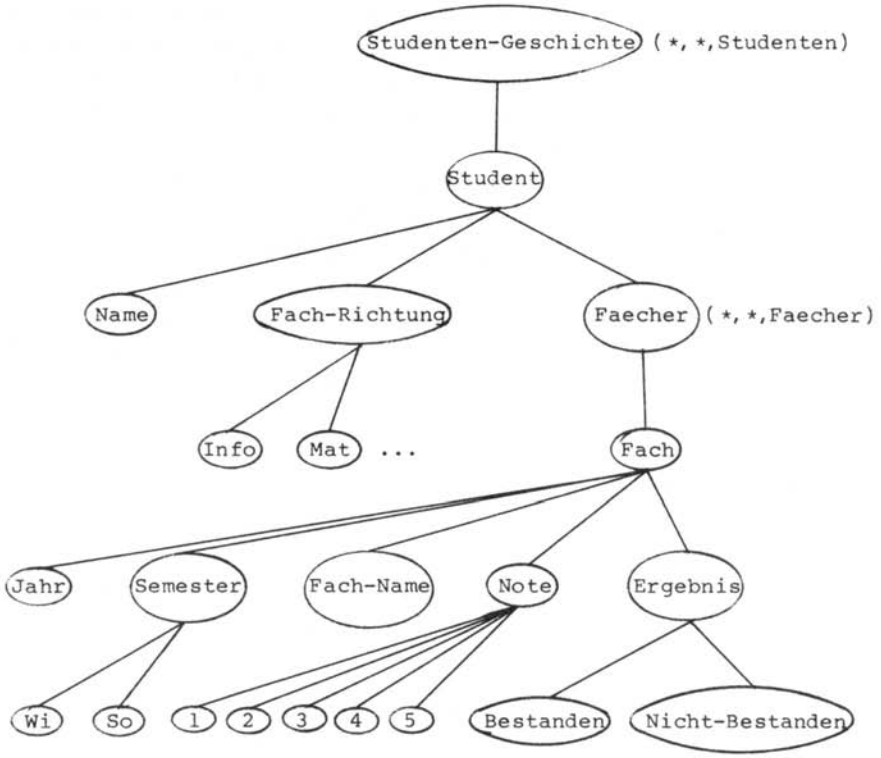


Bild 4.1: Die Datenstruktur der Studenten-Geschichte.

Name ist eine Variable (vom Typ String).
Jahr ist eine Variable (vom Typ Integer).
Fach-Name ist eine Variable (vom Typ String).
Wi, So, 1, 2, 3, 4, 5, Bestanden und Nicht-Bestanden sind Konstanten.

Spezifikation der Abhängigkeit:

<Dependency-assignment>:

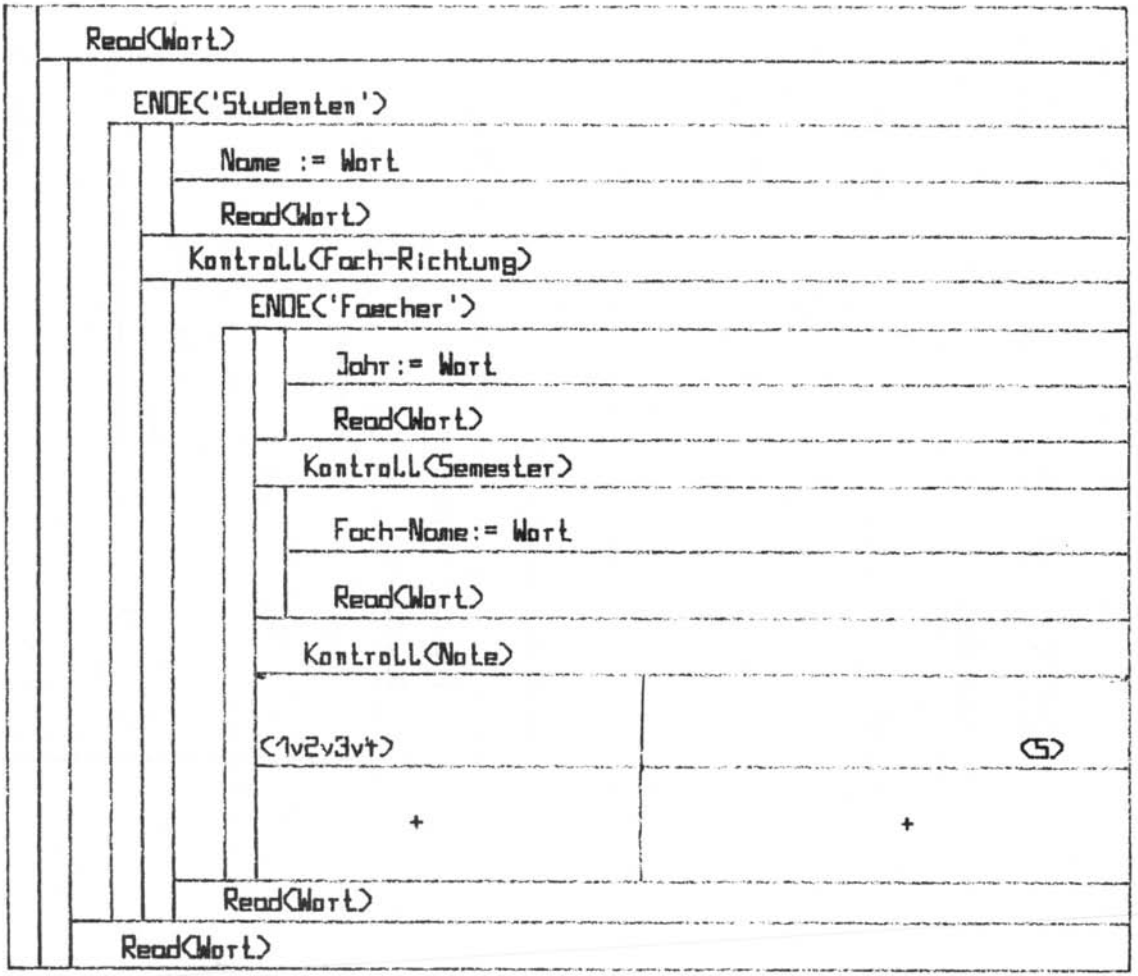
Ergebnis Note = 1 : 2 : 3 : 4 : Bestanden ;
Note = 5 : Nicht-Bestanden ;end

Spezifikation der Bereichsgrenzen:

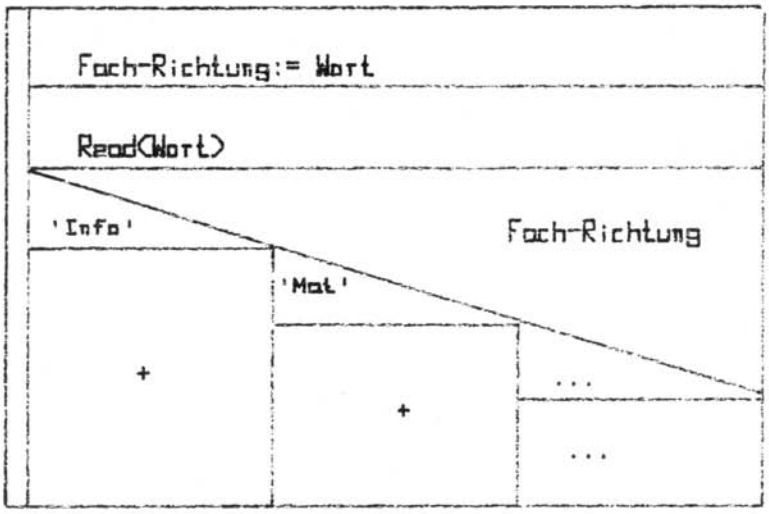
<domain-assignment>:

Studenten-Geschichte
eol = Studenten end

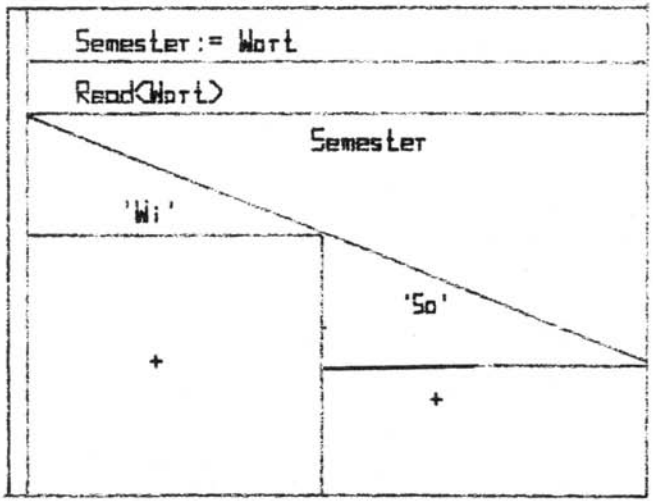
Dateileser(DS(Studenten-Geschichte)) = s. Seite 101


ReadCWort)
ENDEC'Studenten')
Name := Wort
ReadCWort)
KontrollCFach-Richtung)
ENDEC'Faecher')
Jahr := Wort
ReadCWort)
KontrollCSemester)
Fach-Name:= Wort
ReadCWort)
KontrollCNote)
<1v2v3v4>
<5>
+
+
ReadCWort)
ReadCWort)

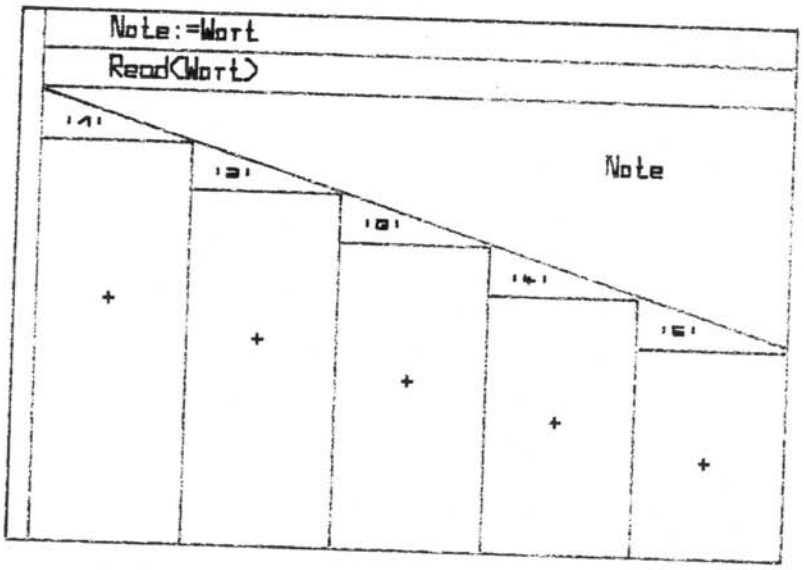
kontroll(Fach-Richtung) =



Kontroll(Semester) =



Kontroll(Note) =



2 - Report-Program-(R-Programm)

Das R-Programm wird aus der Datenstruktur abgeleitet; es hat die Aufgabe, Daten und zu jedem Datenobjekte x einen vom Benutzer angegebenen erläuternden Text auszugeben.

Bei der Jackson-Methode gehören solche erläuternden Texte zur Datenstruktur selbst. Allgemein ist aber eine Datenstruktur eine abstrakte Beschreibung eines Objektes der realen Welt und solche 'Texte' gehören nicht dazu.

Alle Texte bilden ein Formular, dessen Struktur ähnlich der Struktur der Datenstruktur ist. In dieses Formular werden dann auch die Daten eingetragen. Für das Formular spezifiziert der Software-Entwickler zu jedem Datenobjekt die erläuternde Texte.

Wir gehen davon aus, daß alle Datenobjekte, die iterativ, konjunktiv, disjunktiv oder primitiv (und Variable) sind, einen erläuternden Text haben können und alle Datenobjekte, die primitiv und konstant sind, einen erläuternden Text haben müssen.

Das Bild 4.2 zeigt die Erstellung eines R-Programms:

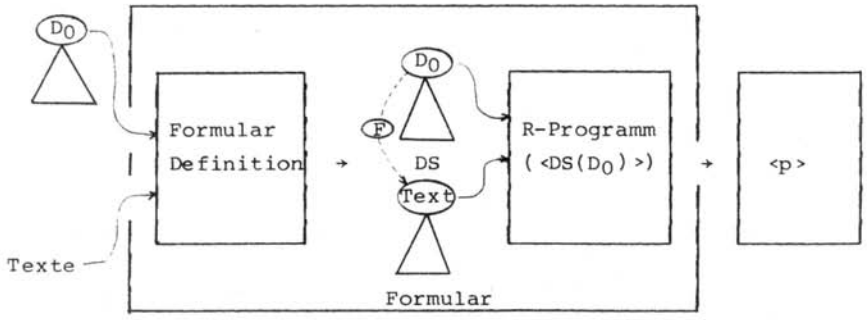


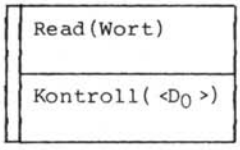
Bild 4.2: Die Erstellung eines R-Programms

Bei 'Formular Definition' gibt der Software-Entwickler zu jedem Datenobjekt einen entsprechenden Text und bei R-Programm(<DS(D_0) >) wird ein R-Programm generiert.

Die Ableitung des R-Programms erfolgt analog zur Ableitung des Programms Dateileser: Die Funktion F liefert für jedes Datenobjekt L_0 einen Text, der auch leer sein kann.

a) - Die Funktion R-Programm erzeugt folgende Kontrollstruktur:

R-Programm(<DS(D_0) >) →

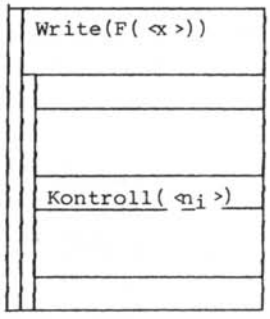


b) - Die Funktion Kontroll(<x >) wird dann rekursiv verfeinert:

b.1) - Für die konjunktiven Datenobjekte:

b.1.1) ohne Texte:

Kontroll(<x >) →

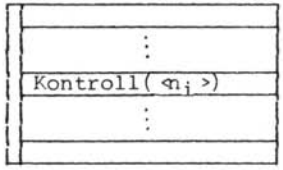


, wenn x ∈ K ∧ F(x) ≠ leer.

F(<x >) → '<Text >'

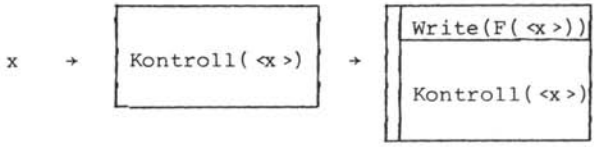
Da wir davon ausgehen, daß zuerst die Bedeutung der Daten und dann erst die Daten selbst ausgedruckt werden, wird die Write-Operation zuerst durchgeführt.

b.1.2) mit Texten:

Kontroll($\langle x \rangle$) → 

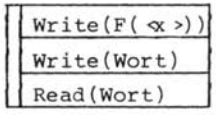
, wenn $x \in K \vee F(x) = \text{leer}$.

Wenn F(x) ≠ leer, ist damit die Kontrollstruktur von x eine Sequenz, deren erste Unterkomponente eine Write-Operation und deren zweite die Kontrollstruktur von x ist, die wir beim Dateileser abgeleitet haben:

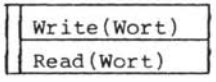


Wenn dagegen F(x) = leer ist , so ist die Kontrollstruktur von x genau gleich wie derjenigen, die beim Dateileser abgeleitet wurde. Das läßt sich als allgemeine Regel für die Ableitung von x, $x \in TD \cup TDF \cup I$, verwenden.

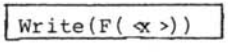
b.2) Für die primitiven Datenobjekte gilt dann:

Kontroll($\langle x \rangle$) → 

, wenn $(x \in R_1 \vee x \in R_2) \wedge F(x) \neq \text{leer}$

Kontroll($\langle x \rangle$) → 

, wenn $(x \in R_1 \vee x \in R_2) \wedge F(x) = \text{leer}$

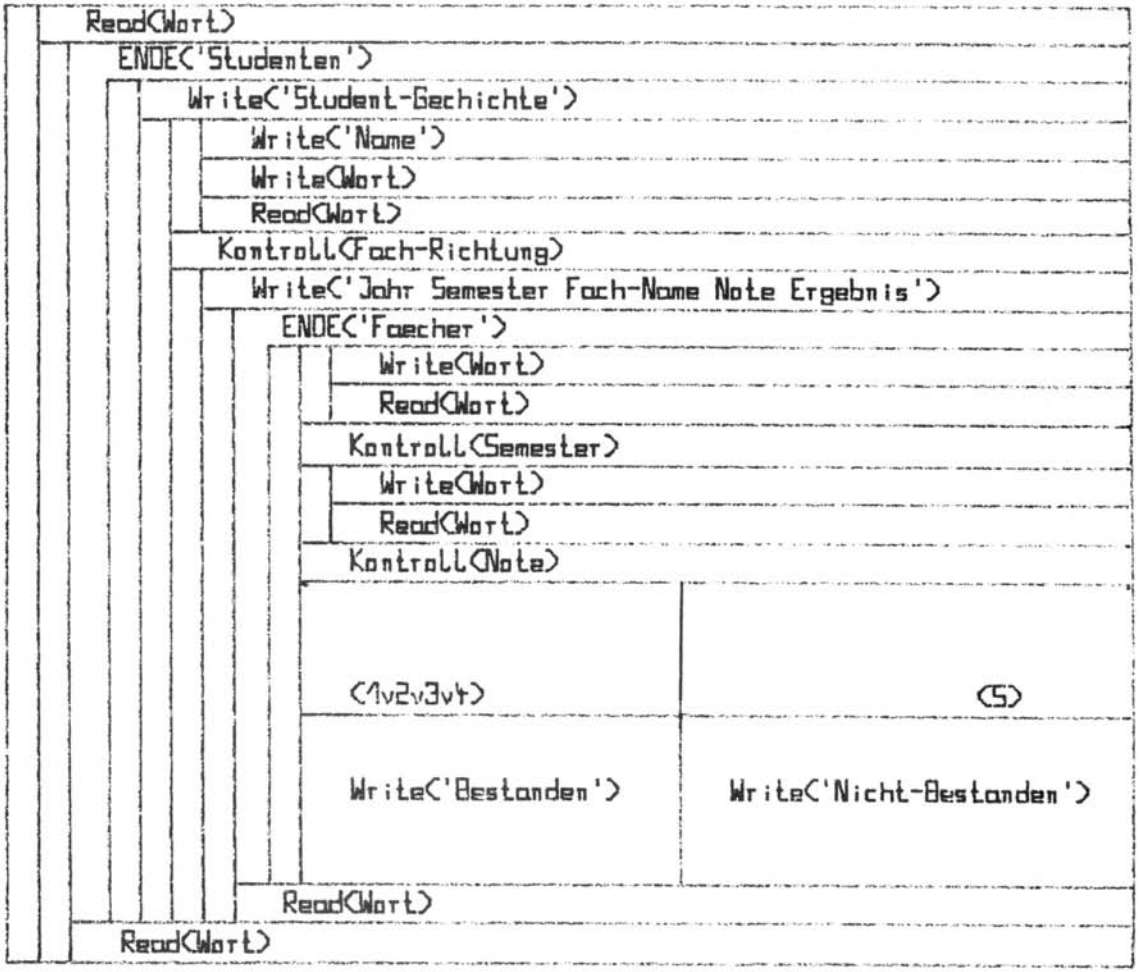
Kontroll($\langle x \rangle$) → 

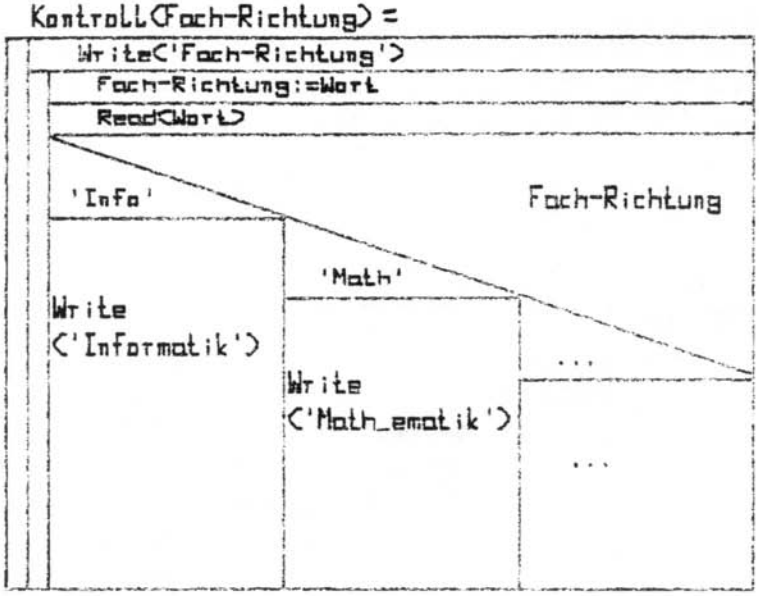
, wenn $x \in R_3 \vee x \in W$

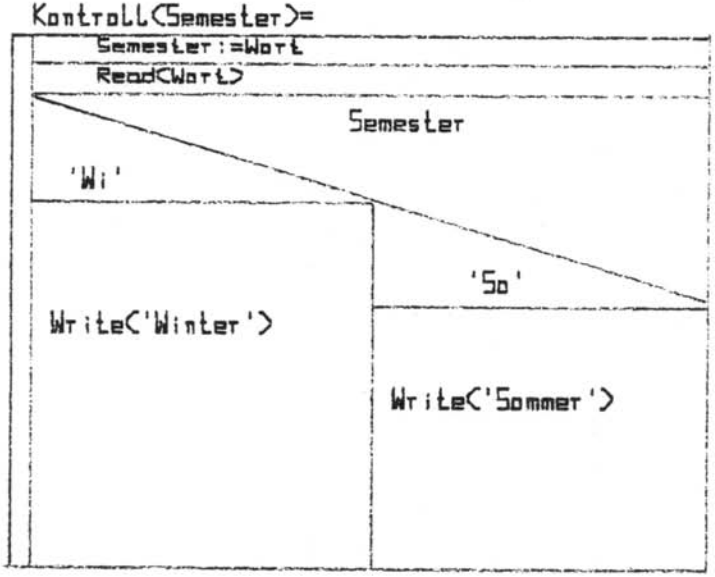
2.1 - Beispiel. In der Datenstruktur 'Studenten-Geschichte' (Bild 4.1) soll gelten:

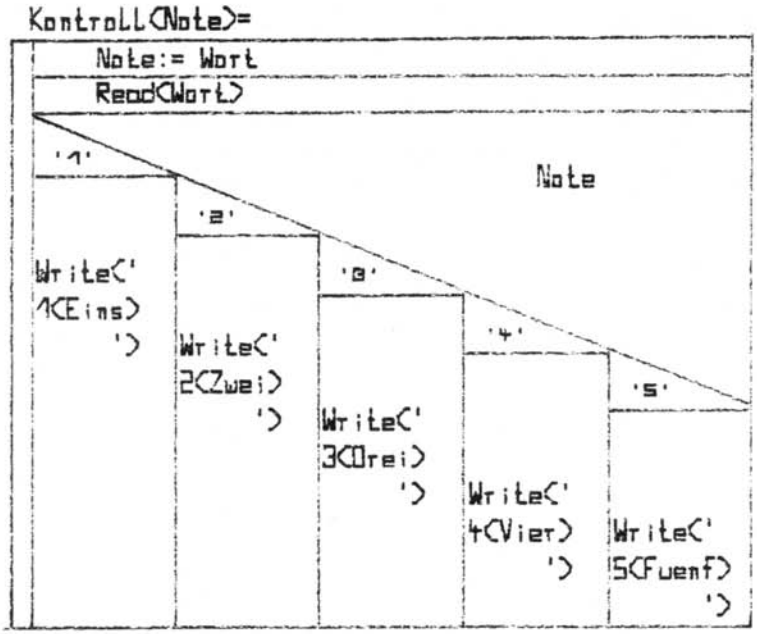
```
F(Student)          = Student-Geschichte
F(Name)             = Name
F(Fach-Richtung)    = Fach-Richtung
F(Info)             = Informatik
F(Mat)              = Mathematik ...
F(Faecher)          = Jahr Semester Fach-Name Note Ergebnis
F(Wi)               = Winter
F(So)               = Sommer
F(1)                = 1(Eins)
F(2)                = 2(Zwei) ...
F(Bestanden)        = Bestanden
F(Nich-Bestanden)   = Nicht Bestanden
```

Das R-Programm(DS(Studenten-Gechichte)) stellt sich dann wie folgt dar:


ReadCWort)
ENDE('Studenten')
Write('Student-Gechichte')
Write('Name')
WriteCWort)
ReadCWort)
KontrollCFach-Richtung)
Write('Jahr Semester Fach-Name Note Ergebnis')
ENDE('Faecher')
WriteCWort)
ReadCWort)
KontrollCSemester)
WriteCWort)
ReadCWort)
KontrollCNote)
<1v2v3v4>
<5>
Write('Bestanden')
Write('Nicht-Bestanden')
ReadCWort)
ReadCWort)


Kontroll(Fach-Richtung) =
Write('Fach-Richtung')
Fach-Richtung:=Wort
Read(Wort)
'Info'
Fach-Richtung
'Math'
Write
('Informatik')
Write
('Math_ematik')
...
...

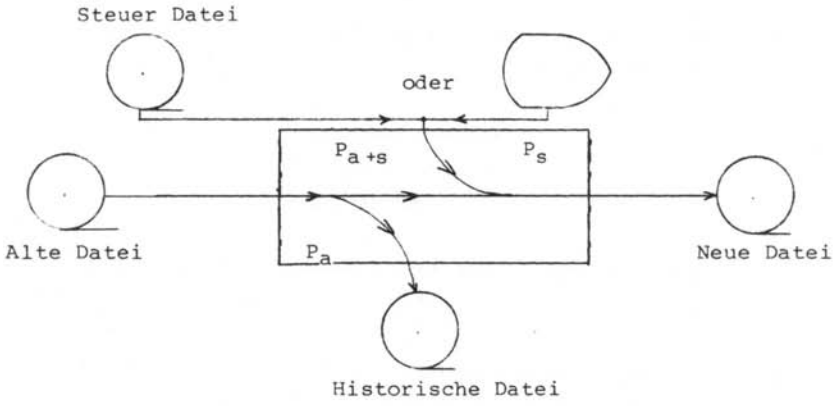

Kontroll(Semester)=
Semester:=Wort
Read(Wort)
Semester
'Wi'
'So'
Write('Winter')
Write('Sommer')


KontroLL(Note)=
Note:= Wort
Read(Wort)
'1'
'2'
'3'
'4'
'5'
Note
Write('
1(Eins)
')
Write('
2(Zwei)
')
Write('
3(Drei)
')
Write('
4(Vier)
')
Write('
5(Fuenf)
')

3. - Update Programm(Up-Programm)

Ein Up-Programm bringt die Datei auf den jeweils neuesten Stand. (Wenn das Objekt in der realen Welt geändert wird, führt das zu einer neuen Ausprägung.)

Die folgende Abbildung zeigt die Komponenten des Up-Programms:



Die sogenannte 'Alte Datei' enthält die Daten der alten Ausprägung. Die 'Steuer-Datei' enthält sämtliche Änderungen (neuer Zustand der Datenobjekte) und Kontrolldaten*. Die sogenannte 'Neue-Datei' enthält die Daten der neuen Ausprägung. Die 'Historische Datei' enthält sämtliche Änderungen (alter Zustand der Datenobjekte)

Das Program P_a liest die Alte Datei (Read(Worta)) und erzeugt die Historische Datei (Write(Worta)). (Wir gehen aber davon aus, daß die Historische Datei noch für andere Anwendungen gebraucht wird.)

Das Programm P_s liest die Steuer-Datei (Read(Worts)) und erzeugt die Neue Datei (Write(Worts)).

* Diese Kontrolldaten zeigen an, was mit den Nachfolgern der I-Daten der alten Ausprägung geschehen soll.

Das Programm P_{a+s} (das Up-Programm) hat zwei Aufgaben:

- es steuert P_a und P_s
- es liest die Alte Datei (Read(Worta)) und erzeugt die Neue Datei (Write(Worta)).

Aus der Datenstruktur $DS(D_0)$ erzeugt man folgende drei Datenstrukturen: $DS(D_a)$, $DS(D_s)$ und $DS(D_{a+s})$.

Diese Datenstrukturen sollen von derselben Struktur sein, dh. x bei $DS(D_0)$ entspricht

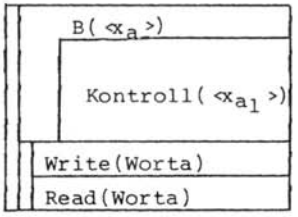
x_a bei $DS(D_a)$,
x_s bei $DS(D_s)$ und
x_{a+s} bei $DS(D_{a+s})$.

Das Programmstück von P_{a+s}, das durch Kontroll($\ll x_{a+s} \gg$) generiert wird, überprüft zuerst, ob die Werte von x_a (Worta) und x_s (Worts) übereinstimmen. Wenn dies der Fall ist, bleibt der Kontrollfluß bei P_{a+s}, sonst werden das durch Kontroll(x_a) generierte Programmstück von P_a und das durch Kontroll(x_s) generierte Programmstück von P_s aufgerufen.

3.1 - Ableitung der Kontrollstrukturen von P_a

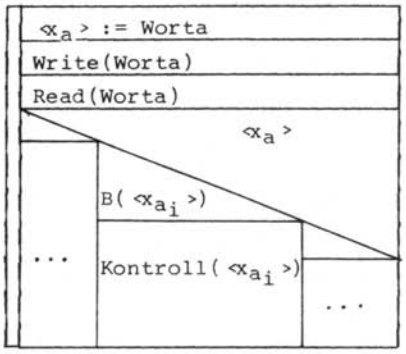
Diese Ableitung erfolgt analog zur Ableitung des Programms Dateileser.

Kontroll($\ll x_a \gg$) →



, wenn $x_a \in I_a \wedge EOL(x_a) \neq$ leer

Kontroll($\langle x_a \rangle$) →



, wenn $x_a \in TD_a$

Kontroll($\langle x_a \rangle$) → Write(Worta) / Read(Worta) , wenn $x_a \in R_{a_1} \vee x_a \in R_{a_2}$

Kontroll($\langle x_a \rangle$) → leer , wenn $x_a \in R_{a_3} \vee x_a \in W_a$

Die Ableitung der Kontrollstrukturen der anderen Typen von Datenobjekten entsprechen ebenso den Ableitungen für den Dateileser.

3.2 - Ableitung der Kontrollstrukturen von P_s

Die Entwicklung entspricht der für die Funktion Kontroll($\langle x_a \rangle$).

3.3 - Ableitung der Kontrollsturen von P_{a+s}

Diese Entwicklung erfolgt analog zur Entwicklung des Dateilesers.

Kontroll ($\ll x_{a+s} \gg$) →

Worta = Worts			Worta ≠ Worts
$\ll x_a \gg$:= Worta			
Write(Worta)			
Read(Worts)			
Read(Worta)			Kontroll($\ll x_a \gg$)
	$\ll x_a \gg$		
	B($\ll n_{(a+s)_i} \gg$)		
...	Kontroll ($\ll n_{(a+s)_i} \gg$)	...	Kontroll($\ll x_s \gg$)

wenn $x_{a+s} \in TD_{a+s}$

Der i-te Nachfolger von x_{a+s} wird mit $n_{(a+s)_i}$ bezeichnet.

Kontroll ($\ll x_{a+s} \gg$) →

M			~ M
...	B($\ll n_{a_i} \gg$) ∧ B($\ll n_{s_i} \gg$)	...	
			Kontroll($\ll x_a \gg$)
...	Kontroll($\ll n_{(a+s)_i} \gg$)	...	
			Kontroll($\ll x_s \gg$)

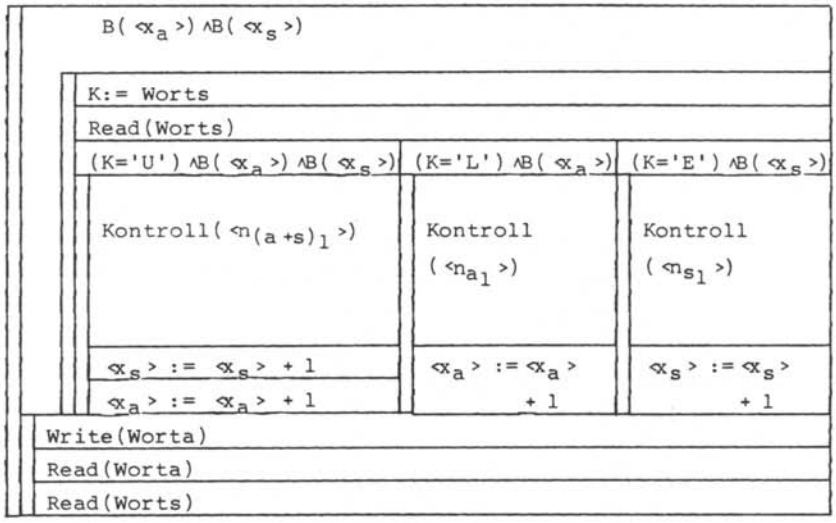
wenn $x_{a+s} \in TDF_{a+s}$.

Dabei bedeutet die Bedingung M, daß die Daten der 'Alte Datei' und der 'Steuer-Datei' mehrfach übereinstimmen können. Formal gilt:

$$M \rightarrow \ldots \vee(B(\langle n_{a_i} \rangle) \wedge B(\langle n_{s_i} \rangle)) \vee \ldots$$

n_{a_i} bezeichnet den i-ten Nachfolger von x_a.
n_{s_i} bezeichnet den i-ten Nachfolger von x_s.

Kontroll($\langle x_{a+s} \rangle$) →



wenn $x_{a+s} \in I_{a+s}$

Der Ablauf ist wie folgt zu beschreiben:

- Die Steuer-Datei enthält die Information(k), ob
  - ein Nachfolger von x_{a_i}, $x_{a_i} \in x_a$, auf einen neuen Stand gebracht werden soll (K='U')
  - oder ein Nachfolger von x_{a_i} gelöscht werden soll (K='L')
  - oder x_{a_i} einen neuen Nachfolger erhalten muß (K='E', einfügen)

- Wir brauchen die Zähler nur, wenn
  $(MIN(x_{a+s}) = '*') \wedge (MAX(x_{a+s}) = '*')$
- Wir brauchen die zweite Unterkomponente der Konstrollstruktur von ($<x_{a+s}>$) nur, wenn $EOL(x_{a+s})$ = leer ist.
- Für die Kontrollstruktur von x_{a+s} sind hier noch keine 'Fehlerangaben' vorgesehen. Insbesondere bei der Aktualisierung der Daten über Bildschirmmenüs kann der Fall eintreten, daß der Benutzer

  - den Befehl k='U' gibt und es keinen Nachfolger mehr gibt;
  - oder den Befehl k='E' gibt und die Anzahl der Nachfolger von x_{s_i} gleich $MAX(x_s)$ ist
  - oder den Befehl k='L' gibt und x_{a_i}, $x_{a_i} \in x_a$, keinen Nachfolger zum Löschen hat.

  Zu diesen Fällen sollen Bedingungen vorgesehen werden, so daß der Benutzer seine Angabe wiederholt.

Die Kontrollstruktur von x_{a+s}, $x_{a+s} \in K$, hat wieder dieselbe Ableitung wie beim Dateileser.

Für variable primitive Datenobjekte gilt:

Kontroll($<x_{a+s}>$) → | Write(Worta) | , wenn $x_{a+s} \in R(a+s)_1 \vee x_{a+s} \in R(a+s)_2$
| Read(Worta) |

Für konstante primitive Datenobjekte gilt:

Kontroll($<x_{a+s}>$) → | leer | , wenn $x_{a+s} \in R(a+s)_3 \vee x_{a+s} \in W_{a+s}$

3.4 -Ableitung vom Up-Programm

Die Ableitung des Up-Programms stellt sich dann wie folgt dar:

Up-Programm(<DS(D_{a+s})>) → Read(Worta); Read(Worts); Kontroll(<D_{a+s}>)

Dieses Up-Programm bringt seinem Benutzer viel Aufwand, weil

- für jedes TD-Datum abgefragt wird, ob die alte Variante gehalten oder geändert werden soll und
- für jeden Nachfolger von I-Daten abgefragt wird, ob er auf den neuen Stand gebracht oder ob er gelöscht werden soll.

Eine Variante dieses Programms wäre, mit den V-Datenobjekten selbst zu arbeiten, das heißt, daß der Benutzer ein V-Datenobjekt ausgibt und ihm direkt einen Wert zuweist. Das Programm sucht dann das betreffende Datum; Wenn es gefunden wird, läuft das Programm, wie gezeigt, weiter. Da nur auf diese Weise solche Datenobjekte bearbeitet werden, die der Benutzer ändern will, wird der Dialog verkürzt.

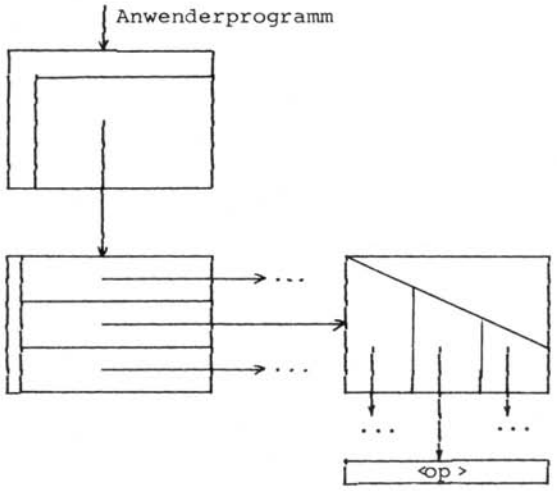
Das Problem dabei ist, daß auch Daten, die von den geänderten Daten abhängen, auf den neuesten Stand gebracht werden müssen (als 'Seiten-Effekt').

V - OPERATIONEN

Wie in der Einleitung dargestellt wird ein Anwenderprogramm aus dem Dateileser der Datenstruktur und aus Operationen, die sich aus der Anforderungsbeschreibung ergeben, erzeugt.

Jedes Anwenderprogramm ist damit darstellbar als Baum von Komponenten*, deren Blätter primitive Komponente (P-Komponenten) sind, die eine Operation enthalten.

Ein Beispiel dafür ist



Die verwendeten Operationen sollen sich beschränken auf Read, Write und arithmetische Operationen, die durch arithmetische Ausdrücke (Funktionen) dargestellt werden.

Da keine Softwarespezifikationsprache vorgestellt werden soll, wurde keine einheitliche Syntax für die Operationen festgelegt.

* Wir verwenden in dieser Arbeit die Begiffe Komponente und Struktogramm synonym.

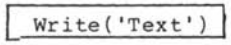
1 - Operationen: Informelle Beschreibung

Für die informelle Beschreibung von Operationen sind folgende Aussagen wichtig.

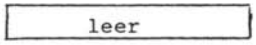
1.1 - Eine Operation läßt sich durch eine Liste von Ein- bzw. Ausgabedatenobjekten und eine Funktion beschreiben. Die Funkion der Operation ist Read, Write oder ein arithmetischer Ausdruck und wird informelle durch Texte beschrieben.

1.2 - Eine Operation kann dabei:

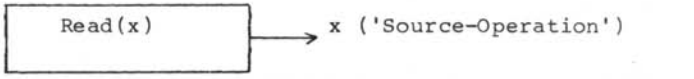
- keine Eingabe- bzw. Ausgabedatenobjekte haben, wie z.B.

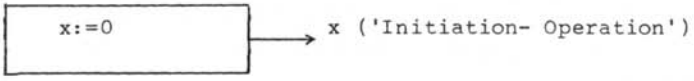


oder die leere Operation ('dummy Operation')* sein

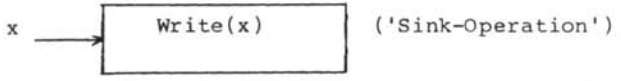


- nur Ausgabedatenobjekte haben, wie z.B.





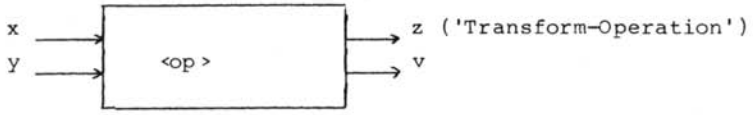
- nur Eingabedatenobjekte haben, wie z.B.



---

* s. /Mye-76/

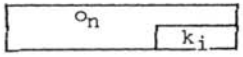
- Ein- und Ausgabedatenobjekte haben, wie z.B.



1.3 - Eine Operation kann auch eine Liste von arithmetischen Ausdrücken zusammenfassen.

1.4 - Die primitiven Komponenten eines Anwenderprogramms bilden eine Menge.

1.5 - Die primitiven Komponenten enthalten jeweils nur eine Operation:



Die Komponente k_i enthält die Operation o_n.

1.6 - Eine Operation kann in verschiedenen P-Komponenten auftreten.

1.7 - Wenn eine Operation ausgeführt wird, werden alle Eingabedatenobjekte benutzt und alle Ausgabedatenobjekte erzeugt. (Eine Operation wird ausgeführt, wenn der Kontrollfluß (Kf) sie erreicht.)

1.8 - Ein Datenobjekt kann durch unterschiedliche Operationen erzeugt werden.

1.9 - Jedes Eingabedatenobjekt für eine Operation wird geliefert von Operationen aus bestimmten primitiven Komponenten. Diese Komponenten bezeichnet man auch als die Lieferanten des Datenobjekts.

1.10 - Ein Anwenderprogramm, das besteht aus

- Bedingungs-Komponenten ('conditional statement'). Sie stellen die Bedingung für die Operationen dar;
- primitiven Komponenten (Sie enthalten die Operationen) und
- Beziehungen, die die primitiven Komponenten zueinander haben (dargestellt durch die Lieferanten der Datenobjekte),

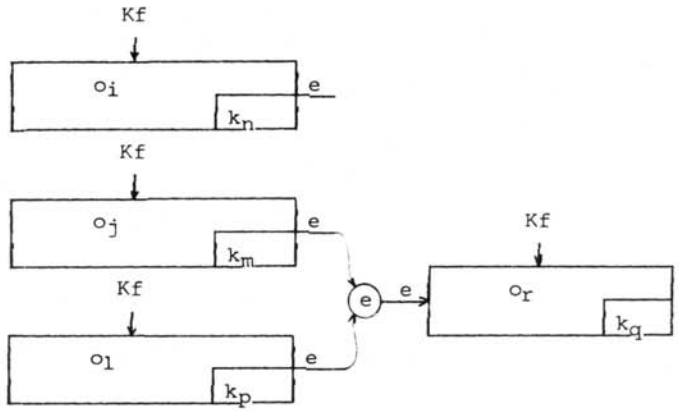
stellt ein Software-System dar. Die Lieferanten der Datenobjekte stellen den Datenfluß und die Bedingungs-Komponenten den Kontrollfluß dar.

2 - Operationen: Syntax Beschreibung

Es werden folgende Bezeichnungen eingeführt:

- Op für die Menge von Operationen:
  $|Op| > 0$,
- E(o), $o \in Op$, für die Menge von Eingabedatenobjekten von o:
  $|E(o)| \geq 0$,
- Pk für die Menge von primitiven Komponenten,
- K(o), $o \in Op$, für die Menge von P-Komponenten, die die Operation o enthalten:
  $|K(o)| \geq 1$,
- L(e,k), $k \in K(o)$, $o \in Op$, $e \in E(o)$, für die Menge von P-Komponenten, die Operationen enthalten, die das Eingabedatenobjekt e für die Operation o in der Komponente k liefern. (L(e,k) stellt die Lieferanten von e dar):
  $|L(e,k)| \geq 1$.

  Dies werde für $L(e,k) = \{ k_m, k_p \}$ am Beispiel dargestellt.



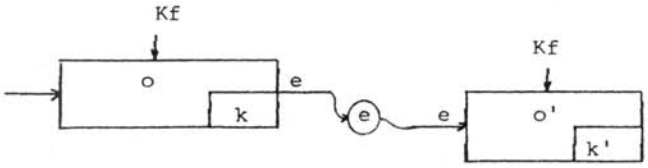
2.1 - Bedingungen

1 - Die Lieferanten eines Datenobjekts müssen Operationen enthalten, die das Datenobjekt als Ausgabedatenobjekt haben. Das kann man folgendermaßen gewährleisten:

Sei $o \in Op$,
$e \in E(o)$,
$k \in K(o)$ dann gilt:

$$\forall k' \in L(e,k),\ \exists o' \in Op, \text{ so daß } k' \in K(o') \wedge e \in A(o').$$

2 - Zu jedem Ausgabedatenobjekt einer Operation des Typs 'Transformation' gibt es wenigstens eine Operation, die dieses Datenobjekt verwendet.



Das kann man folgendermaßen gewährleisten.

Sei $o \in Op$ (angenommen, o sei vom Typ 'Transformation')
$e \in A(o)$,
$k \in K(o)$ dann gilt:

$\exists (o' \in Op \wedge k' \in K(o'))$, so daß
$e \in E(o') \wedge k \in L(e,k')$.

3 - Um eine Operation, die in einer Komponente steht, immer ausführen zu können, müssen sämtliche Eingabedatenobjekte zur Verfügung stehen. Das heißt, daß alle Operationen, die in den Lieferanten der Eingabedatenobjekte stehen, bereits mindestens einmal ausgeführt worden sind. Das kann man folgendermaßen gewährleisten:

Es seien $o \in Op$,
$e \in E(o)$,
$k \in K(o)$,
$L(e,k)$ die Lieferanten von e,
Sk die Menge von Sequenz-Komponenten,
Wk die Menge von While-Komponenten,
Ck die Menge von Case-Komponenten,
$TP(k_1,k_2)$ der Treffpunkt von k_1 und k_2 (s. die entsprechende Funktion (Kapitel III, Punkt 11.2.1.1), die für die Datenstruktur definiert wurde.), wobei k_1 und k_2 Komponenten sind,
$M(k_1,k_2)$ die Menge von Komponenten, ausschließlich k_1 und einschließlich k_2, die zwischen k_1 und k_2 liegen (s. Kapitel III, Punkt 11.2.1.1),
U(s) die Liste der direkten Unterkomponente von der Komponente s.

Dann gibt es wenigstens ein k' ϵ L(e,k) und ein s ϵ Sk, so daß

a) TP(k',k) = s (Bild 5.1),
b) $u_{k'}$ muß <u>vor</u> u_k kommen, wobei
   $u_{k'} \epsilon U(s) \cap (M(k',s) \cup \{k'\})$ und
   $u_k \epsilon U(s) \cap (M(k,s) \cup \{k\})$ (Bild 5.1) und
c) Wenn k' ϵ U(s), dann $\sim \exists u_x$, $u_x \epsilon$ Pk, $u_x \epsilon$ L(e,k), so daß der Treffpunkt
   TP(k',u_x) eine Case-Komponente ist.

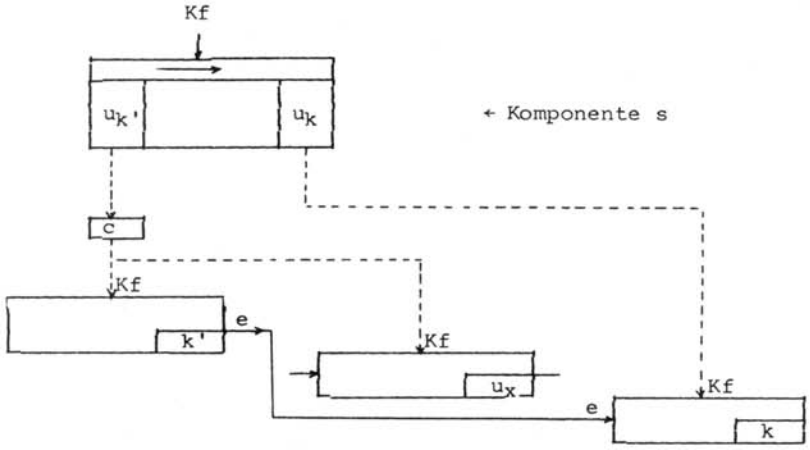


Bild 5.1: Der Treffpunkt der Komponente k' und k

Zur Erläuterung sei ausgeführt:

- Wenn der Treffpunkt TP(k',k) für jedes k' ϵ L(e,k) eine Case-Komponente wäre, dann kann der Kontrollfluß bei der Case-Komponente den Weg zur Komponente k nehmen;

- Wenn es eine Sequenz-komponente s gibt, s = TP(k',k), s ϵ Sk, muß zusätzlich gewährleistet werden, daß es keine beliebige Operation u_x gibt, zu der der Kontrollfluß bei c (Bild 5.1), c=TP(k',u_x), c ϵ Ck, geht

4 - Es sei k' ϵ L(e,k).

Wenn a) der Treffpunkt von k' und k eine Sequenz-Komponente ist und
b) k' rechts von k im Baum steht,

dann gibt es wenigstens eine While-komponente, die zwischen der Komponente TP(k',k) und der Wurzel des Anwenderprogramms steht. Diese Bedingung gewährleistet, daß das Datenobjekt e, das durch k' geliefert wird, für k verwendbar ist.

2.3 - Abchließende Bemerkungen

Wenn man zuläßt, daß ein Datenobjekt durch verschiedene Operationen erzeugt wird, wird das Operationen-Netz noch komplizierter; dies hat zur Folge, daß die Codegenerierung schwieriger wird.

Da die bekannten Programmiersprachen das Datenfluß-Konzept nicht unterstützten, wird die 'Code-Generierung' für ein Anwenderprogramm auch dadurch noch erschwert.

VI - Anwenderprogramm

Ein Anwenderprogramm besteht, wie bereits dargestellt, aus dem Dateileser und den Operationen. Wie die beiden Komponenten zusammengefügt werden, wird im folgenden dargestellt.

Das Diagramm zeigt die einzelnen Schritte, die durchzuführen sind, um ein Anwenderprogramm zu erzeugen:

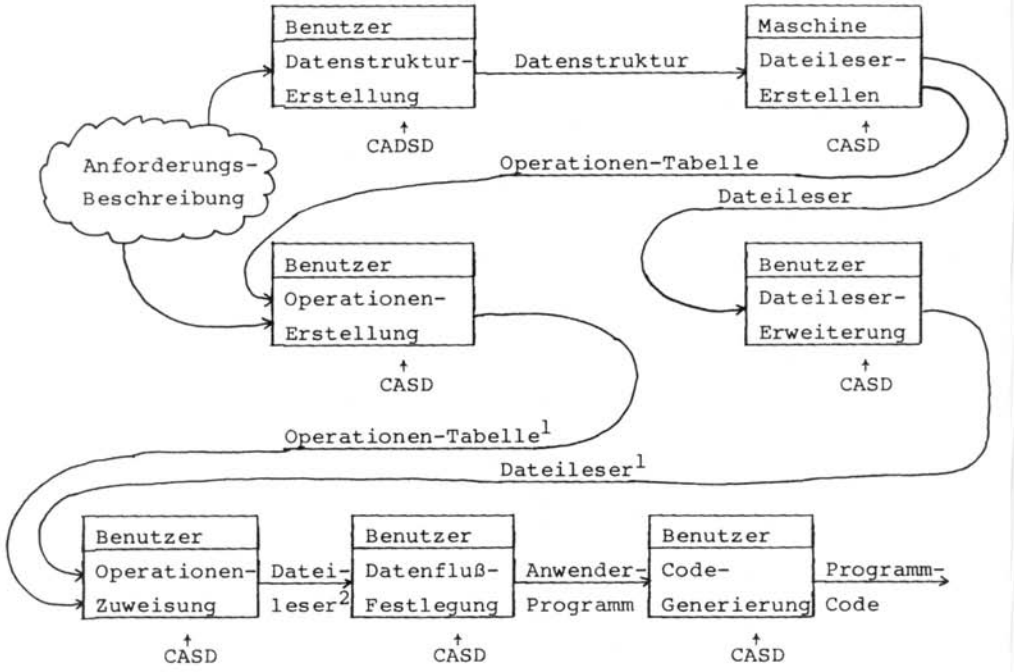


Bild 6.1: Die Erstellung eines Anwenderprogramms

Jedes Rechteck trägt schon die Bezeichnung eines CAD-Systems, das den Benutzer bei der Durchführung der verschiedenen Tätigkeiten unterstützt: CADSD für 'Computer Aided Data Structure Design' und CASD für 'Computer Aided Software Design'.

Der Index bei der Angabe der Tätigkeit zeigt die verschiedenen Zustände, die die Ausgabe annimmt.

Die einzelnen Aktivitäten sind wie folgt beschreiben:

- Datenstruktur-Erstellung - Der Software-Entwickler erstellt mit Hilfe von CADSD die Datenstruktur.

- Dateileser-Erstellung - Nach den Ableitungsregeln, die wir in dem Kapitel IV vorgestellt haben, wird ein Programm, das als Dateileser bezeichnet wird, abgeleitet.

- Operationen-Erstellung - Der Software-Entwickler erzeugt die auszuführenden Operationen.

- Dateileser-Erweiterung - Hier erweitert der Software-Entwickler den Dateileser, um die Operationen zu erhalten.

- Operationen-Zuweisung - Der Software-Entwickler weist die Operationen den Komponenten zu.

- Datenfluß-Festlegen - Der Benutzer legt den Datenfluß fest; damit ist das Anwenderprogramm fertig.

- Code-Generierung - Hier erzeugt der Software-Entwickler Code für das Anwenderprogramm.

1 - Die Operationentabelle.

Die Tabelle von Operationen enthält Namen, Typ, Eingabe-Liste E(o), Ausgabe-Liste A(o), die Funktion F(o) und die Kennzeichnungen der Komponenten K(o) der Operationen:

Name(o)	Typ(o)	E(o)	A(o)	F(o)	K(o)
R(i)	P	leer	<x>	<x>:=Wort	Kn

Bild 6.2: Die Operationentabelle

Ihr Name dient dazu, die Operation zu identifizieren. Der Typ der Operation stellt den Typ der Komponente dar, die für die Operation vorgesehen werden soll. Die Eingabe-Liste und die Ausgabe-Liste wurden bereits erwähnt. Die Funktion der Operation ist ein Text. Die Kennzeichnungen identifizieren die Komponenten, denen die Operation zugewiesen wird.

Die Belegung der Operationentabelle wird durch die Aktivität 'Dateileser-Ableitung' begonnen: Immer wenn die Funktion Kontroll(<x>), $x \in R_1 \vee x \in R_2$ aufgerufen wird, wird eine Zeile in diese Tabelle eingetragen. Im Beispiel von Bild 6.2 gilt: R(i) ist der Name der Operation (sequentiell zugeordnet); P, das für primitiv steht, ist der Typ der Operation (Typ(R(i))=P); die Operation hat keine Eingabedatenobjekte(E(R(i))=$\emptyset$); <x> ist ein Ausgabedatenobjekt der Operation (A(R(i))={<x>}); <x>:=Wort ist die Funktion der Operation (F(R(i))=<x>:=Wort) und Kn die Kennzeichnung der Komponente, der die Operation zugewiesen wurde (K(R(i))={Kn}).

1.1 - Typ der Operation

Zusätzlich zu Operationen vom Typ primitiv (P-Operation) kann der Benutzer noch Operationen vom Typ Test (T-Operation) einfügen. Diese verlangen eine Komponente vom Typ if (if-Komponente).

Wir haben deshalb die Operation vom Typ Test vorgesehen, da in der Tabelle eingetragene Operationen nur ausgeführt werden dürfen, wenn sie gewisse Bedingungen erfüllen. Andernfalls

dürfte der Software-Entwickler in die Tabelle nur P-Operationen eintragen, was in dieser Beschränkung die Realität nicht widerspiegeln würde.

1.2 - Die If-Komponente

Die if-Komponente wird daher wie folgt beschrieben:

wobei die Bedingung B die Funktion F(o) der Operation ist.

1.2.1 - Die Bedingung B

Die Bedingung ist ein boolescher Ausdruck, in dem

- Datenobjekte der Datenstruktur, die Nachfolger von D-Datenobjekte sind,
- Datenobjekte, die durch die Operationen erzeugt werden,
- Konstante vom Typ String, Integer und Real,
- logische Operatoren ($\wedge$ $\vee$ $\sim$),
- arithmetische Operatoren (- + x / ...) und
- relationale Operatoren ($>$ $\geq$ $<$ $\leq$ $\neq$ $=$)

auftreten.

Die Datenobjekte der Datenstruktur werden als boolesche Variable und die Datenobjekte, die durch die Operationen erzeugt werden, als Real-, Integer- bzw. String-Variable, verwendet.

Die Datenobjekte, die als Real-, Integer- bzw. String-Variable in dem booleschen Ausdruck auftreten, müssen als Eingabedatenobjekte in die Tabelle eingetragen werden, um bei der Code- Generierung den Lieferanten des Datenobjekts deutilich zu machen.

Die Eingabedatenobjekte des Typs Test müssen die Bedingungen, die wir im letzten Kapitel vorgestellt haben, ebenso erfüllen; das heißt, daß die Datenobjekte sämtlich zur Verfügung stehen müssen, wenn die T-Operation ausgeführt werden soll.

1.2.2 - Beispiel:

Gegeben sei die Datenstruktur von Studenten-Geschichte (Bild 4.1). Ein Anwenderprogramm soll diejenigen Studenten ausgeben, die Informatik studieren, die im Jahr 1981 die Prüfung Informationssysteme abgelegt haben und die in dieser Prüfung durchgefallen sind.

Die Operationen sind:

a)

Typ(o) = T

F(o) = Inf ∧ Jahr = 1981 ∧ Fach-Name = 'Informationssysteme' ∧ Nicht-Bestanden

E(o) = {Jahr,Fach-Name }

b)

Typ(o) = P

F(o) = Write(Name)

E(o) = {Name }

2 - Dateileser-Erweiterung

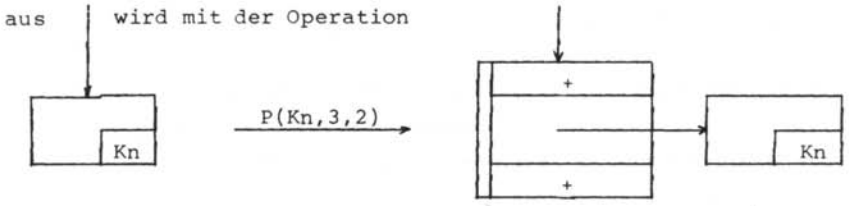
Um die Operationen, die in die Tabelle eintragen wurden, in den Dateileser zu bringen, muß dieser erweitert werden. Die Frage ist, wie man im Dateileser neue Komponenten hinzufügen kann, ohne daß seine Logik betroffen wird. Es gibt dafür zwei Voraussetzungen: Erstens darf der Software-Entwickler keinesfalls eine Read-Operation in die Tabelle eintragen und zweitens kann der Dateileser-Baum nur aufgebrochen werden, um eine neue Komponente einzufügen, wenn die neue Komponente eine Sequenz ist.

Um den Dateileser zu erweitern, haben wir daher die folgenden beiden Operationen vorgesehen:

a) Eine Operation, um eine Sequenz in den Dateileser einzufügen.

P(Kn,#Uk,i): - Kn ist die Komponentenkennzeichnung,
- #Uk ist die Anzahl von Unterkomponten der Sequenz, die eingefügt werden soll.
$\#Uk \geq 2$
- i ist die i-te Unterkomponente der Sequenz, die Kn sein soll.

Beispiel:

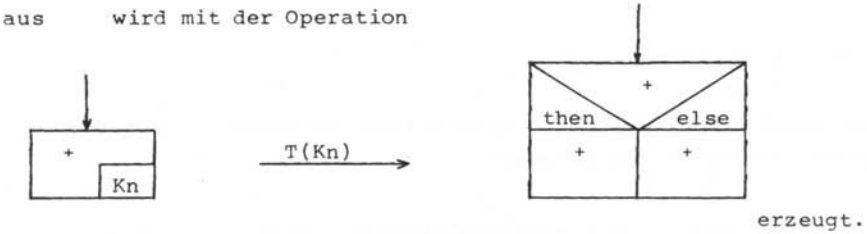


Bemerkung: Die Unterkomponenten der Sequenz, die nicht spezifiziert wurden, sind unspezifizierte Komponenten(+).

b) Eine Operation, um einen Test in den Dateileser einzufügen.

T(Kn): - Kn muß eine unspezifizierte Komponente sein, die sich in eine If-Komponente verwandelt.

Beispiel:



Die beiden Unterkomponenten der If-Komponente entstehen als unspezifizierte Komponenten.

3 - Operationen-Zuweisen

Um die in der Operationentabelle eingetragenen Operationen unspezifizierten Komponenten zuzuweisen, wird folgende Operation vorgesehen.

Z(Name,Kn): Name ist der Name der Operation;
Kn ist die Komponentenkennzeichnung.

Wenn Kn eine If-Komponente ist, muß die Operation vom Typ T sein; wenn Kn eine unspezifizierte Komponente ist, muß die Operation vom Typ P sein.

Als Ergebnis dieser Operation wird F(o) in das Bedingungs-Feld(B) kopiert, wenn die Operation vom Typ T ist; F(o) wird in die unspezifizierte Komponente kopiert, wenn die Operation vom Typ P ist.

Bemerkung: Die Tätigkeiten des Software-Entwicklers wie Operationen-Erstellung, Dateileser-Erweiterung und Operationen-Zuweisung werden bei CASD nicht sequentiell durchgeführt, wie unser Diagramm zeigt, sondern iterativ. Das heißt, daß der Software-Entwickler, sobald er eine neue Operation in der Tabelle eintragen möchte, zuerst den Dateileser erweitert; bei der Operationen-Zuweisung werden sowohl die neue Operation in die Tabelle eingetragen als auch ihre Funktion in die Komponente kopiert. Der Typ der Operation wird automatisch abgeleitet, weil er vom Typ der Komponente abhängt.

4 - Datenfluß-Festlegen

Wir haben schließlich eine zusätzliche Operation F vorgesehen, um den Datenfluß festzulegen:

F(Kn,e,L): Kn ist die Komponentenkennzeichnung;
e ist ein Eingabedatenobjekt;
L ist die Liste von Lieferanten von e.

VII - Zwei CAD-Systeme zur Unterstützung beim Entwurf von Datenstrukturen und von Anwenderprogrammen

Die Beschreibung von Datenstrukturen oder eines Anwenderprogramms bei der Programmentwicklung durch graphische Darstellungen ist meist verständlicher als durch die übliche verbale Form. In der vorliegenden Arbeit werden daher zwei Prototypen von CAD Systemen dargestellt, nämlich CADSD und CASD, die den Entwurf von Datenstrukturen bzw. Anwenderprogrammen unterstützen.

1 - CAD-Computerunterstützung beim Entwurf von Datenstrukturen (CADSD)

Eine Datenstruktur ist, wie im vorangehenden dargestellt, eine abstrakte Beschreibung eines Objekts der realen Welt. Dazu kann bei der Programmentwicklung ein Dateileser erzeugt werden, der die der Datenstruktur entsprechende Datei liest. Was geschieht, wenn das Objekt und damit als Folge die Beschreibung der Datenstruktur geändert wird? Ist dann der neue Dateileser noch in der Lage, die alte Datei zu lesen? Dies ist dann der Fall, wenn in der alten Datenstruktur

a) - kein Datenobjekt eingefügt oder dort gelöscht wird, das im Dateileser eine READ-Operation erzeugt oder löscht oder das auf der Datei steht. Das bedeutet dann: In die Menge R

$R = \{x \mid x \in T \cup L \cup R_1\}$,
mit $T = TD \cup (\cup N(t),\ t \in TD)$, und L die Menge von Datenobjekten i mit $i \in I \wedge EOL(i) \neq$ leer,

darf kein neues Datenobjekt x eingefügt oder eines der vorhandenen gelöscht werden;

b) - kein Datenobjekt y gelöscht wird, das eines der unter a) genannten Datenobjekte x als direkten oder indirekten Nachfolger hat, dh.

$y \in (M(x,D_0) \cup \{D_0\}), \forall x \in R.$

In diesem Falle würde auch das Datenobjekt x gelöscht.

Beispiel: Gegeben sei die Datenstruktur der Studenten-Geschichte (Bild 4.1). Daraus ergeben sich

R = {Fach-Richtung, Semester, Note, Studenten-Geschichte, Faecher, Name, Jahr, Fach-Name, Info, Mat, ..., Wi, So, 1, 2, ...}.

a) Das Datenobjekt Ergebnis ist löschbar, da es nicht in der Menge R enthalten ist und es kein x gibt, so daß

$Ergebnis \in M(x,Studenten\text{-}Geschichte), \forall x \in R$ wird.

b) Nehmen wir aber an, daß die Beziehung zwischen den Datenobjekten Note und Ergebnis die folgende ist:

```
<Dependency-assignment> :
                    Ergebnis Note = 1 : 2 : 3 : Bestanden;
                             Note = 4 : 5: Nicht-Bestanden; end
```

dann ist das Datenobjekt Ergebnis einfügbar, da es ein TDF-Datenobjekt ist und seine Nachfolger W-Datenobjekte sind.

Aus dieser durch b) geänderten neuen Beschreibung der Datenstruktur Studenten-Geschichte wird ein neuer Dateileser erzeugt, der auch die alte Datei noch lesen kann. Dies ist ein Bespiel dafür, daß eine Datenstruktur an eine Veränderung der realen Welt angepaßt werden kann, ohne daß die vorhandenen Daten geändert werden müssen.

Im folgenden wird das System CADSD näher beschrieben.

Das CADSD-System verwaltet eine 'Library' von Datenstrukturen, die jeweils durch ihre Wurzel identifiziert werden. Jede Datenstruktur ist, wie schon dargestellt, in einer Datei abgelegt.

Wenn der Benutzer des CADSD-System aus einer alten Datenstruktur eine neue erzeugen möchte, genügt es, dann die Datei zu lesen, den Namen der Wurzel der Datenstruktur zu ändern, die gewünschten Änderungen in der Datenstruktur zu machen und diese wiederum in eine Datei abzuspeichern.

1.1 - CADSD-Operationen

Das CADSD-System /Wac-84/ besitzt eine Reihe von Operationen, die den Benutzer bei der Erstellung der Datenstruktur unterstützen. Die Datenstruktur wird mit dem derzeitigen CADSD-Prototyp noch in verbaler Form dargestellt. (s. Kapitel III, Punkt 4.)

Zunächst wird der Benutzer gefragt, ob er eine neue Datenstruktur erstellen oder mit einer abgespeicherten Datenstruktur weiter arbeiten möchte. Das dem Benutzer angebotene Grundmenü besteht aus den folgenden Operationen:

1.1.1 - Erstellung von Datenstrukturen

Bei dieser Operation erstellt der Benutzer rekursiv eine Datenstruktur. Von oben nach unten und von links nach rechts werden für jedes Datenobjekt einer Datenstruktur dessen Typ (konjuktiv, disjuktiv, iterativ, primitiv) und dessen Nachfolger spezifiziert. Es wird vom System übergeprüft, daß der Name jedes Nachfolgers noch nicht verwendet wurde. Die vom Benuzer gewählte Anzahl von Nachfolgern wird hier nicht überprüft. Bei den P-Datenobjekten wird zusätzlich abgefragt, ob es sich um eine Konstante oder Variable handelt. Das Verfahren wird durch Wahl von 'Quit' unterbrochen.

1.1.2 - Fortsetzen von Datenstrukturen

Bei dieser Operation wird eine begonnene Datenstruktur fortgesetzt und nach einem Datenobjekt gefragt, das noch nicht spezifiziert wurde. Dazu wird ebenfalls das Menü 'Erstellen von Datenstrukturen' verwendet.

1.1.3 - Korrekturen von Datenstrukturen

Es gibt drei Typen von Korrekturen: Ersetzen, Verändern und Löschen von Datenobjekten.

1.1.3.1 - Ersetzen

Bei dieser Operation werden die Kanten zu den Nachfolgern eines Datenobjekts aufgebrochen, und der Benutzer kann das Datenobjekt neu spezifizieren. Auch dazu wird das Menü 'Erstellen von Datenstrukturen' verwendet. Für die Wiederverwendung von Datenobjekten muß gewährleistet werden, daß alle Datenobjekte, die keinen Vorgänger haben, die Wurzel eines Baumes darstellen. Ein Datenobjekt kann damit mehreren Bäumen angehören, wie das Bild 7.1 zeigt.

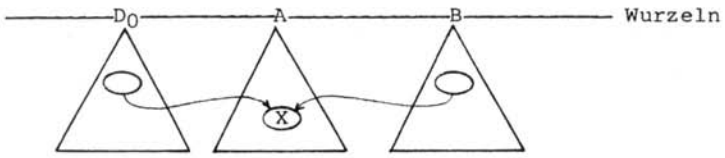


Bild 7.1: Wiederverwendung von Datenobjekten

1.1.3.2 - Verändern

Beim Verändern wird ein Menü verwendet, das die Operationen Einfügen, Ändern und Löschen für jeden Nachfolger anbietet.

Einfügen

Durch Einfügen ist es möglich, einen neuen Nachfolger einzufügen. Für die Wiederverwendung von Datenobjekten (Bild 7.1), muß gewährleistet werden, daß die Datenobjekte, die eine Wurzel sind, einen Baum darstellen. Wenn es sich dabei um einen neuen Nachfolger handelt, wird, auf Wunsch des Benutzers, das Menü 'Erstellen von Datenstrukturen' eingeblendet.

Ändern

Durch Ändern ist es möglich, den Namen des Nachfolgers zu ändern. Für die Wiederverwendung von Datenobjekten (Bild 7.1), muß gewährleistet werden, daß die Datenobjekte, die Wurzel sind, einen Baum darstellen. Wenn es sich dabei um einen neuen Nachfolger handelt, wird, auf Wunsch des Benutzers, das Menü 'Erstellen von Datenstrukturen' eingeblendet.

Löschen eines Nachfolgers

Beim Löschen eines Nachfolgers wird die Kante zu ihm aufgebrochen.

1.1.3.3 - Löschen aller Nachfolger

Beim Löschen aller Nachfolger werden die Kanten zu allen Nachfolgern aufgebrochen.

1.1.4 - Löschen von Datenstrukturen

Bei Löschen einer Datenstruktur wird alles gelöscht.

1.1.5 - Tag-assignment

Bei dieser Operation wird nach dem Namen des D-Datenobjekts, seinem tag und dem tag-Wert jedes seiner Nachfolger gefragt. Sowohl das tag als auch die tag-Werte sind 'optional' . Es wird

vom System geprüft, ob das D-Datenobjekt ein TD-Datenobjekt ist und ob die tag-Werte (N([t,x]) s. Kapitel III, Punkt 10.1.3) eine Menge verschiedener Namen bilden. Die Operation 'Ändern' wird ebenfalls zur Verfügung gestellt.

1.1.6 - Dependency-assignment

Bei dieser Operation wird vom System nach dem abhängigen Datenobjekt und dem steuernden Datenobjekt gefragt. Für jeden Nachfolger des abhängigen Datenobjekts wird eine Liste von Nachfolgern des steuernden Datenobjekts erfragt. Es wird geprüft, ob das abhängige Datenobjekt ein TDF-Datenobjekt ist. Auch hier wird die Operation 'Ändern' ebenfalls zur Verfügung gestellt.

1.1.7 - Domain-assignment

Bei dieser Operation fragt das System nach dem Datenobjekt, dem MIN, dem MAX und dem EOL. Diese drei Parameter sind 'optional'. Es wird geprüft, ob das Datenobjekt ein I-Datenobjekt ist. Die Operation 'Ändern' wird zur Verfügung gestellt.

Zur den bisher genannten Operationen soll folgendes bemerkt werden:

- Alle Blätter der Datenstruktur, für die nichts weiteres ausgesagt wurde, werden als primitive und konstante Datenobjekte voreingestellt.

- Die D-Datenobjekte, über die nichts weiteres ausgesagt wurde, werden als TD-Datenobjekte voreingestellt. Ihre tags sind sie selbst (implizite Kontrolldaten) und ihre tag-Werte sind ihre eigenen Nachfolger.

- Bei 'Tag-assignment' werden die 'optional'-Parameter voreingestellt.

- Die I-Datenobjekte, über die nichts weiteres ausgesagt wurde, werden wie in Kapitel III, Punkt 12.5, voreingestellt.

- Bei 'Domain-assignment' werden die 'optional'-Parameter wie in Kapitel III, Punkt 12.5, voreingestellt.

1.1.8 - CADSD-Überprüfungsoperationen

Wenn der Benutzer seine Datenstruktur überprüfen möchte, wählt er im Grundmenü die Option 'Überprüfen'. Folgende Punkte werden dann überprüft:

- die Anzahl der Nachfolger der verschiedenen Datenobjekte(s. Kapitel III, Punkt 7),

- Die Bedingung $\bigcap C(a,x,y) = \emptyset$ (s. Kapitel III, Punkt 11.2.3 - 4.b),

- die Reihenfolge-Bedingungen: RB.1-6 (s. Kapitel III, Punkt 11.2.1.1),

- die Beziehungen zwischen den Bereichsgrenzen. (s. Kapitel III, Punkt 12.2) und

- Die Beziehungen zwischen den Bereichsgrenzen und dem EOL (s. Kapitel III, Punkt 12.3.).

Die entsprechende Fehler werden gemeldet.

Außer diesen Operationen bietet das CADSD noch weitere Operationen an, wie z.B. Drucken der Datenstruktur, Löschen der überflüssigen Datenobjekte, die nicht zu D_0 gehören (s. Bild 7.1) usw.

In folgenden zeigen wir die Darstellung der Datenstruktur Studenten-Geschichte, die mit Hilfe des CADSD-Systems erzeugt wurde.

*** DATEI-BESCHREIBUNG ***

Datei-Name : STUDENTEN-GESCHICHTE
Datei-Art : Input

```
STUDENTEN-GESCHICHTE ::= { STUDENT }   Min *   Max *
                         Ende der Liste: STUDENTEN

             STUDENT ::= NAME + FACH-RICHTUNG + FAECHER

                NAME ::= << Variable >>

       FACH-RICHTUNG ::= INFO | MATH | USW.

             FAECHER ::= { FACH }   Min *   Max *
                         Ende der Liste: FAECHER

                INFO ::= << Konstante >>

                MATH ::= << Konstante >>

                USW. ::= << Konstante >>

                FACH ::= JAHR + SEMESTER + FACH-NAME + NOTE +
                         ERGEBNIS

                JAHR ::= << Variable >>

            SEMESTER ::= WI | SO

           FACH-NAME ::= << Variable >>

                NOTE ::= 1 | 2 | 3 | 4 | 5

           ERGEBNIS ::= BESTANDEN | NICHT-BESTANDEN

ERGEBNIS wird kontrolliert von NOTE
          BESTANDEN:1
                    2
                    3
                    4
    NICHT-BESTANDEN:5


                  WI ::= << Konstante >>

                  SO ::= << Konstante >>

                   1 ::= << Konstante >>

                   2 ::= << Konstante >>

                   3 ::= << Konstante >>
```

4 ::= << Konstante >>

5 ::= << Konstante >>

BESTANDEN ::= << Konstante >>

NICHT-BESTANDEN ::= << Konstante >>

*** ENDE DATEI-BESCHREIBUNG ***

2 - CAD-Unterstützung beim Entwurf von Software Systemen (CASD)

Das CASD-System, wie es 1982 in /Nun-82/ spezifiziert wurde, ist ein Prototyp für ein weiteres Konzept eines CAD-Systems, das alle wesentlichen Eigenschaften des Prototyps enthält, und für eine erweiterte Aufgabenstellung einsetzbar ist.

a) Das neue System ist ein allgemeines Werkzeugt, das dem Benutzer, ausgehend von der Anforderungsbeschreibung, bei der Entwurf der Struktur eines Anwenderprogramms unterstützt und

b) das Konzept des Prozeduraufrufs verwendet, um dem Benutzer primitive Operationen, die gleich oder ähnlich ablaufen, als Prozeduren darzustellen.

Der erwähnte implementierte Prototyp (1982) war notwendig, um für des neue Konzept grundsätliche Erfahrungen zu sammeln. Das neue System ist noch nicht vollständig implementiert, obwohl zahlreiche Module des früheren Prototyps verwendet werden konnten.

Im folgenden wird das erweiterte Konzept des CASD-Systems dargestellt; später wird dann gezeigt, wie es eingesetzt werden kann, um ein Anwenderprogramm auf dem in Kapitel VI geschildeten Weg zu erzeugen.

2.1 - CAD-Unterstützung beim Prozeß der Software-Spezifikation

Unter Software-Spezifikation wird der Prozeß vestanden, der aus der Anforderungsbeschreibung eine Programmstruktur entwickelt. Diese Programmstruktur stellt am Anfang die Anforderungsbeschreibung selbst dar; im Verlauf eines Verfeinerungsprozesses werden Bestandteile der Programmstruktur rekursiv strukturiert.

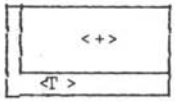
Das Ergebnis des Verfahrens ist ein Programm- oder ein Prozedur-Ableitungsbaum. Da das CASD-System für verschiedene Aufgaben verwendet werden kann, enthält und verwaltet es eine 'Library'

der Ableitungsbäume. Durch die Operationen des CASD-Systems erzeugt der Benutzer selbst einen solchen Ableitungsbaum.

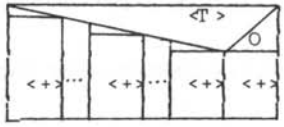
Die in folgenden dargestellte Grammatik beschreibt die genannte 'Library' der Ableitungsbäume.

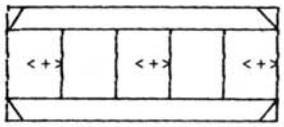
```
<Library>                          ::=  { <Ableitungsbaum> }
<Ableitungsbaum>                   ::=  <Programm-Ableitungsbaum> |
                                        <Prozedur-Ableitungsbaum>
<Programm-Ableitungsbaum>          ::=  Programm
                                        <Programm-Ableitungsbaumname>
                                        <Anforderungsbeschreibung>
                                        [ <Diagramm> ]
<Programm-Ableitungsbaumname>      ::=  <Diagrammname>
<Anforderungsbeschreibung>         ::=  <prim>
<Prozedur-Ableitungsbaum>          ::=  Prozedur
                                        <Prozedur-Ableitungsbaumname>
                                        <Anforderungsbeschreibung>
                                        [ <Diagramm> ]
<Prozedur-Ableitungsbaumname>      ::=  <Diagrammname>
<prim>                             ::=  { s. Kapitel VI   }
<Diagramm>                         ::=  <if> | <while> |
                                        <repeat-until> | <case-K> |
                                        <case-O> | <paralell> | <seq>
<if>                               ::=  { s. Kapitel IV }
<while>                            ::=  { s. Kapitel IV }
<case-K>                           ::=  { s. Kapitel IV }
<seq>                              ::=  { s. Kapitel IV }
```

<repeat-until> ::=

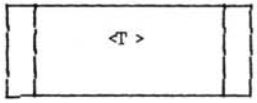


<case-O> ::=



<parallel> ::= 

<T> ::= <Text>
<+> ::= <Diagramm> |
<Aktivität> |
<Aktivität> ::= <prim> [<Verfeinerung>] |
<Procedure-call>
<Verfeinerung> ::= <Diagrammname> <Diagramm>
<Procedure-call> ::= <Prozedur-Ableitungsbaumname>
<Procedure-call>

<Procedure-call> ::= 

In dieser Grammatik achtet das CASD-System auf Einhaltung folgender Regeln:

a) Die 'Library' ist eine Menge von Ableitungsbäumen, die durch ihre Wurzel identifiziert werden. Verschiedene Ableitungsbäume haben verschiedene Namen.

b) Die Diagramme eines Ableitungsbaums bilden eine Menge. In einem Ableitungsbaum hat jedes Diagramm einen unterschiedlichen Namen.

Ein Ableitungsbaum ist damit ein Baum von Diagrammen und ein Diagramm ein Baum von Komponenten.

Durch das CASD-System kann der Benutzer, wie schon gesagt, die

Anforderungsbeschreibung rekursiv strukturieren. Mit Hilfe des CASD-System übersetzt also der Benutzer eine Anforderungsbeschreibung in ein Diagramm, dessen Syntax oben gezeigt wurde.

Ein Diagramm stellt eine Aktivität dar und soll aus nur wenigen Komponenten bestehen, damit die Übersicht über die Aktivität nicht verloren geht. (Wenn ein Diagramm ein Modul sein sollte, sollte sein Code nach Myers /Mye-76/ zwischen 10 und 100 'high-level' Anweisungen haben. Nach IBM-Regeln soll ein Modul nicht mehr als 45 Zeilen haben.)

Wenn jede Aktivität, die verfeinert wurde, durch ihr Diagramm ersetzt wird, bekommt man die Struktur des Programms. Für das Programm muß noch der Code generiert werden.

Die Anforderungsbeschreibung stellt die Wurzel des Ableitungsbaums dar und wird in einem Diagramm dargestellt, dessen Blätter nur Primitiv- oder Prozeduraufruf-Aktivitäten (die als primitive Operationen betrachtet werden) sind. Primitive Aktivitäten, die noch zu komplex sind, werden weiter strukturiert und somit der Baum von oben nach unten (Top-down) erzeugt (Bild 7.2).

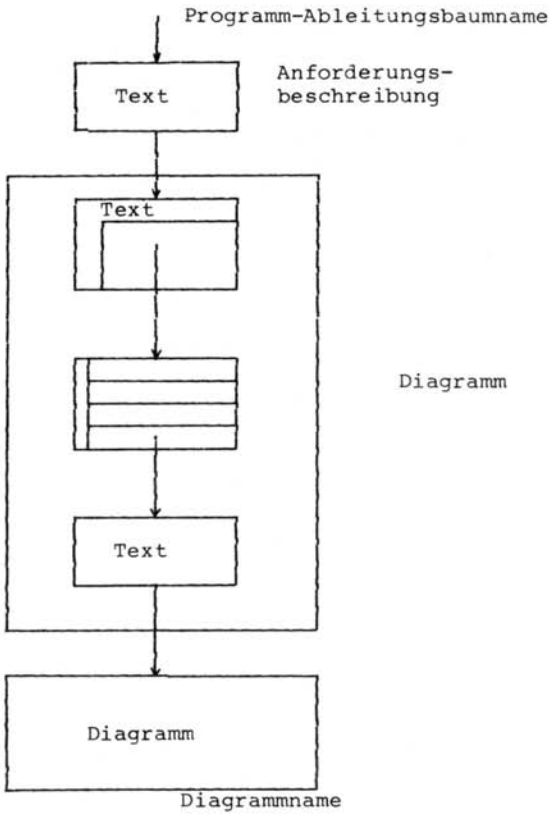


Bild 7.2: Der Ableitungsbaum.

Beim Entwurf des Ableitungsbaums tauchen Aktivitäten auf, die identisch (sie verarbeiten dieselben Datenobjekte) oder die änhlich sind (sie verarbeiten unterschiedliche Datenobjekte, die aber von demselben Typ sind.). Um solche Aktivitäten identifizieren zu können, soll der Benutzer ein gewisses Abstraktionsvermögen haben. Beim ersten Typ von Aktivitäten entstehen Prozeduren, die keine Parameter haben; beim zweiten entstehen solche mit Parameter. Bei beiden Fällen wird die

Aktivität separat als Prozedur-Ableitungsbaum entwickelt. Die Aktivitäten sollen auf dem Programm-Abbildungsbaum durch einen Prozeduraufruf ersetzt werden. Um diese Ersetzung möglich zu machen, muß der Prozedur-Ableitungsbaum schon vorliegen. Da die Prozeduren schon vorhanden sein müssen, wenn ein Aufruf auf dem Programm-Ableitungsbaum gemacht wird, kann der Benutzer ein Software-System auch 'bottom-up' entwickeln. Das heißt, daß Teile der Anforderungsbeschreibung herangezogen werden, um Prozeduren zu spezifizieren.

Es ist auch möglich, unabhängig von konkreten Problemen (Anforderungsbeschreibungen) Prozeduren zu entwickeln, die für eine Klasse von Problemen verwendet werden können. Sie stehen zur Verfügung; wenn der Benutzer sich auskennt, kann er eine Aktivität in einen Prozeduraufruf verwandeln.

Dazu sollen noch folgende Bemerkungen angebracht werden:

- In abstrakter Weise existiert also kein Unterschied zwischen einem Programm und einer Prozedur. Normalerweise kommunizieren die Programme miteinander über eine Datei (Schnittstelle) und die Prozeduren über Parameter.

- Wenn die Komponenten einer Programmstruktur durch ihre lineare Form (z.B. Die if-Komponente durch if (Text) then (...) else (...)) ersetzt werden, erhält man einen Text wie in der PDL-Sprache /Ram-83/.

- Der Benutzer soll eine Aktivität dann nicht mehr verfeinern, wenn er sich den Code unmittelbar vorstellen kann /Mye-76/.

- Bei der Entwicklung eines Programms geht der Benutzer nicht immer fehlerfrei vor. Fehler bei der Interpretation der Anforderungsbeschreibung, Unvollständigkeiten der Anforderungsbeschreibung, Optimierungsmöglichkeit usw. führen zu einer Änderung der Strukturierung der Anforderungsbeschreibung.

Das CASD-System muß so flexibel sein, daß der Benutzer Änderungen einfach vornehmen kann.

2.1.1 - Der CASD-Bildschirm

Der Benutzer kommuniziert mit dem CASD-System durch Menüs (Bild 7.3). Die Menüs enthalten die Operationen oder Daten für die Operationen, die auszuwählen sind. Das Feld 'Diagrammbereich' wird auch dazu verwendet, um Parameter für die Operationen zu sammeln. Die System-Zeile wird verwendet:

- um den Benutzer zu informieren, was er als nächsten Schritt tun muß,
- um Daten einzutragen, die nicht auf dem Menü stehen können und
- um Fehler zu melden.

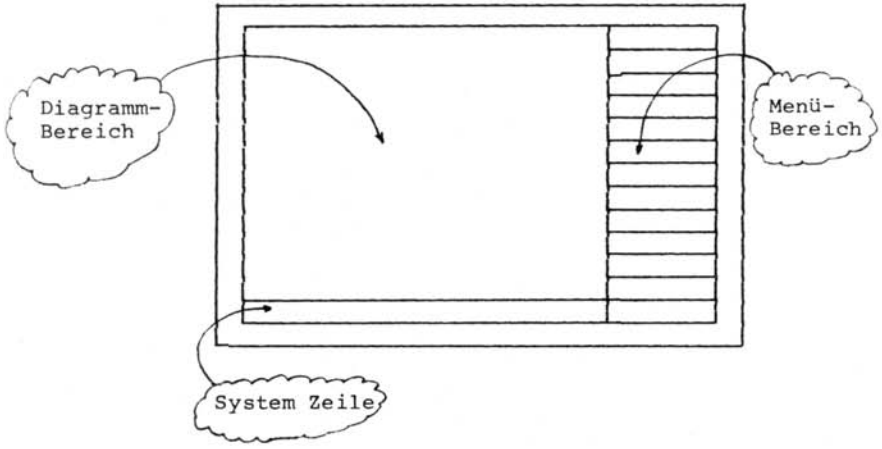


Bild 7.3: Das CASD-Bildschirm

2.1.2- Die CASD-Operationen

Bei der Beschreibung der Operationen gilt: Ein Menü enthält eine Liste von Operationen, von denen jeweils eine durch den Benutzer

ausgewählt werden soll. Das Menü kann auch eine Liste von Parametern für Operationen enthalten, wenn ein Parameter für eine Operation ein Datenobjekt darstellt, das vom Typ 'disjunktiv' ist und die Anzahl der Nachfolger klein ist.

Aus der Grammatik werden Datenobjekte (zum Beispiel Ableitungsbaum) verwendet, zu den es eine zugehörige Menge von Operationen gibt. Ein Datenobjekt und seine Operationen bilden dann ein Menü, dessen Name dem des Datenobjekts entspricht. Wenn die Ausführung einer Operation abgeschlossen ist, wird ein neues Aufrufmenü eingeblendet.

Die CASD-Operationen beziehen sich auf verschiedene Typen von Datenobjekten:

Datenobjekt - Library
Datenobjekt - Ableitungsbaum
Datenobjekt - Diagramm
Datenobjekt - 'Dump-Library'

Die Operationen auf dem Datenobjekt Library erlauben es dem Benutzer, Programme und Prozeduren in die Library einzufügen, zu löschen oder einen bestimmten Ableitungsbaum zu holen, um ihn zu untersuchen oder weiter zu verarbeiten. Wenn der Programm-Ableitungsbaum fertig ist, kann der Benutzer den Code generieren.

Die Operationen auf einem Ableitungsbaum erlauben es dem Benutzer, durch Nachfolge-Diagramm-Holen, übergeordnetes--Diagramm-Holen oder beliebiges-Diagramm-Holen auf ihm zu navigieren. Das aktuelle Diagramm wird gezeigt und als solches gekennzeichnet. Das heißt, daß der Benutzer durch Diagramm-Erzeugen alle Operationen auf dem Diagramm anwenden kann.

Die Operationen auf einem Diagramm erlauben es dem Benutzer, den Diagrammbaum aufzubauen. Da dabei eine unspezifizierte Kom-

ponente durch eine reale Komponente ersetzt wird, erscheint der Diagrammbaum nicht explizit im Diagrammbereich des Bildschirms.

Die Operationen des Diagrammmenüs können in vier Gruppen aufgeteilt werden:

- Zuerst kommen die Operationen(Seq-Einfügen...Text Editor Aufruf und Komponenten-Format-Ändern), womit der Benutzer sein Diagramm aufbaut. Durch die Komponente Format-Ändern kann der Benutzer die Größe der verschiedenen Felder, woraus eine Komponente besteht, ändern. Das Ziel ist, den Platz eines Feldes zu vergrößern. Da eine Komponente einen Knoten des Diagrammbaums dargestellt, sind von der Änderung ihrer Felder alle direkten und indirekten Unterkomponenten betroffen.

- Die zweite Reihe von Operationen (Diagramm-Transformation bis Komponente-Löschen) erlaubt dem Benutzer, Änderungen zu machen und sogar eine neue Komponente in eine Liste von Unterkomponenten einzufügen wie z.B. eine neue Komponente in eine Sequenz, in einen Case usw.

- Die Operation Diagramm-Transformation erlaubt es dem Benutzer, eine Komponente zu löschen und in einer Dump-Library abzulegen, damit er sie gegebenenfalls wiederverwenden, dh. in ein beliebiges Diagramm kopieren kann.

- Die letzte Operation, Verfeinerungs-Erzeugen, erlaubt es dem Benutzer einer primitiven Komponente, die einen Text enthalte, eine Struktur zu geben, das heißt ein Diagramm für sie zu erzeugen. Auf diese Weise wird der Ableitungsbaum erzeugt.

Eine Operation wird dem Benutzer nur dann angeboten, wenn ihre Parameter vorliegen. So wird z.B. die Operation Programm-, Prozedur-Holen nur angeboten, wenn die Library nicht leer ist. Bevor ein Menü eingeblendet wird, wird zunächst überprüft, welche Menüoperationen dem Benutzer angeboten werden können.

Dadurch wird vermieden, daß der Benutzer eine Operation auswählt, die nicht ausgeführt werden kann.

Im folgenden zeigen wir alle Menüs und die Semantik der Operationen und in Anhang A wird das CASD-System kurz beschrieben.

Menü: Library		
Operation/ Parameter	Semantik	Aufrufmenü
Programm-/ Prozedur-Holen Programm-/ Prozedur-Ableitungsbaumname	Die Anforderungsbeschreibung wird auf dem Diagramm-Bereich eingeblendet.	Ableitungsbaum
Programm-/ Prozedur-Einfügen Programm-/ Prozedur-/ Ableitungsbaumname	Ein neuer Programm-, Prozedur-Ableitungsbaum wird in die Libray eingefügt.Auf dem Diagramm-Bereich wird eine leere primitive komponente eingeblendet, und der Text Editor wird automatisch zur Verfügung gestellt,um es dem Benutzer zu ermöglichen, seine Anforderungsbeschreibung einzutragen. Am Ende (stop mit dem Text Editor) dieser Benutzertätigkeit wird ein leeres Diagramm eingeblendet.	Diagramm
Programm-/ Prozedur-Löschen Programm-/ Prozedur-Ableitungsbaumname	Der Programm-, Prozedur-Ableitungsbaumname wird auf der Library gelöscht.	Library
Library-Listing	Eine Liste aller Progamm-, Prozedur-Ableitungsbaumnamen wird gezeigt.	Library
Code-Generierung Programm-/ Prozedur-Ableitungsbaumname	Es wird für den Ableitungsbaum mit Hilfe des Benutzers ein Programmtext in Pascal erzeugt /Gai-84/. Es wird ein Menü angeboten, wo der Benutzer den Code für die Texte angibt und die Deklarationen macht.	Library
Stop	Das CASD-System wird ausgeschaltet.	Betriebsystem

Menü: Ableitungsbaum		
Operation/ Parameter	Semantik	Aufrufmenü
Diagramm-Erzeugen Bildschirmdiagramm	Es wird das Menü Diagramm angeboten.	Diagramm
Diagramm-Löschen Bildschirmdiagramm	Es wird das Diagramm, das auf dem Bildschirm steht, gelöscht und der Vorgängerdiagramm oder die Anforderungsbeshreibung auf dem Bildschirm eingeblendet.	Ableitungsbaum
Diagramm-Holen Diagrammname	Es wird das Diagramm, dessen Name auf der System-Zeile angegeben wird, auf dem Bildschirm-Bereich gezeigt.	Ableitungsbaum
Nachfolger-Holen Aktivität mit Verfeinerung	Das Diagramm, das eine Struktur eines Texts darstellt, wird auf dem Bildschirm-Bereich gezeigt.	Ableitungsbaum
Diagramme Listing	Es wird eine Liste aller Diagrammnamen dieses Ableitungsbaums ausgedruckt.	Ableitungsbaum
Ableitungsbaum-Plotten.	Es wird der ganze Ableitungsbaum oder, nach Wunsch des Benutzers, ein Stück von ihm, auf den Plotter gebracht. Ein Menü wird angeboten, um den Benutzer die Bäume auswählen zu lassen./Str-84/	Ableitungsbaum
Stop	Es wird das Menü Library eingeblendet.	Library

Menü: Diagramm		
Operation/ Parameter	Semantik	Aufrufmenü
Seq-Einfügen Unspezifizierte Komponente, Anzahl von Unterkomponenten.	Die unspezifierte Komponente wird durch eine Sequenz ersetzt. Der Benutzer zeigt dazu auf die unspezifizierte Komponente. Die möglichen Anzahlen von Unterkomponenten der Sequenz werden auf dem Menü-Bereich angegeben. Die maximale Anzahl von Unterkomponenten wird so berechnet, daß der Benutzer in jede Unterkomponente wenigstens eine Zeile eintragen kann. /Ren-83/	Diagramm
If-Einfügen Unspezifizierte Komponente	Die unspezifizierte Komponente wird durch ein If ersetzt.	Diagramm
While-Einfügen Unspezifizierte Komponente	Die unspezifizierte Komponente wird durch ein While ersetzt.	Diagramm
Repeat-until-Einfügen Unspezifizierte Komponente	Die unspezifizierte Komponente wird durch ein Repeat-until ersetzt.	Diagramm
Case-Einfügen Unspezifizierte Komponente, Anzahl von Unterkomponenten.	Die unspezifizierte Komponente wird durch ein Case ersetzt. Die Anzahl von Unterkomponenten wird wie bei der Sequenz berechnet.	Diagramm
Case-O-Einfügen Unspezifizierte Komponente, Anzahl von Unterkomponenten.	Die unspezifizierte Komponente wird durch ein Case-O (Case-Otherwhile) ersetzt. Die Anzahl von Unterkomponenten wird wie bei der Sequenz berechnet.	Diagramm
Parallel-Einfügen Unspezifizierte Komponente, Anzahl von Unterkomponenten.	Die unspezifizierte Komponente wird durch eine Parallelismus-Komponente (s. Punkt 2.1) ersetzt. Die Anzahl von Unterkomponenten wird wie bei Sequenz berechnet.	Diagramm

Diagramm-menü: Fortsetzung

Menü: Diagramm		
Operation/ Parameter	Semantik	Aufrufmenü
Procedure-call-Einfügen Unspezifizierte Komponente, Prozedur-Ableitungsbaumname	Es wird die unspezifizierte Komponente durch eine Procedure-call-Komponente ersetzt, wenn der Prozedur-Ableitungsbaumname auf der Library steht. Der Texteditor wird automatisch zur Verfügung gestellt.	Diagramm
Aktivität-Einfügen Unspezifizierte Komponente.	Es wird eine unspezifizierte Komponente (kein leeres Diagramm) in eine Primitiv-Komponente verwandelt. Der Texteditor wird automatisch zur Verfügung gestellt.	Diagramm
Texteditor Aufruf Feld	Es wird das Menü des Texteditors eingeblendet, um einen Text in die möglichen ausgewählten Felder einzutragen oder zu korrigieren. /Pre-83/	Diagramm
Komponente-Format-Ändern Komponente	Es wird die graphische Darstellung der Komponente geändert, ohne daß ihre graphische Kennzeichnung verändert wird. Es wird ein neues Menü angeboten./Sch-83/	Diagramm
Diagramm-Transformieren Bildschirmdiagramm	Es wird das Menü Dump-Library angeboten /Som-84/	Dump-Library
Komponente-Einfügen Komponente, Unterkomponentenname.	Durch diese Operation ist es möglich, eine neue Unterkomponente in eine Sequenz, in ein Case, in ein Case-O oder in einen Parallelismus einzufügen. Dazu wird ein neues Menü eingeblendet./Mor-83/	Diagramm
Komponente-Löschen Komponente, Komponentenname, Stelle in der Liste	Durch diese Operation ist es möglich, eine Komponente zu löschen, oder besser, die Komponente durch eine unspezifizierte Komponente zu ersetzen. Wenn die Komponente einer Liste von Unterkomponenten angehört, dann wird die Komponente aus der Liste gelöscht, wenn die minimale Anzahl(≥ 2) von Unterkomponenten nicht mehr erreicht ist.	Diagramm
Verfeinerungs-Erzeugung Primitiv-Komponente, Diagrammname	Ein leeres Diagramm wird eingeblendet, wenn der Diagrammname neu ist und wenn es für die primitive Komponente noch keine Verfeinerung gibt.	Diagramm
Stop	Es wird das Menü Ableitungsbaum eingeblendet.	Ableitungsbaum

Menü: Dump-Library		
Operation/ Paramenter	Semantik	Aufrufmenü
Dump-Erzeugung Komponente des Aktuell-Diagramms	Die ausgewählte Komponente wird durch eine unspezifierte Komponente ersetzt und in einer Library von Komponenten abgelegt. Dazu wird der Komponente eine Nummer (Dump-Nummer) automatisch zugewiesen, um sie zu identifizieren. Diese Komponente wird auch als Aktuell-Dump bezeichnet.	Dump-Library
Dump-Kopieren Aktuell-Dump, Unspezifizierte Komponente des Aktuell-Diagramms	Die Komponente, die als 'actual' Dump bezeichnet ist, ersetzt die ausgewählte unspezifizierte Komponente. Es wird überprüft, ob die Integrität des Programm-, Prozedur-Ableitungsbaums erhalten bleibt. Das heißt, daß der 'actual' Dump nicht in die unspezifizierte Komponente kopiert wird, wenn ein bereits im Ableitungsbaum vorhandenes Diagramm mitkopiert wird.	Dump-Library
Aktuell-Dump Ändern Dump-Nummer	Durch diese Operation wird ein neuer Dump als Aktuell-Dump ausgewählt.	Dump-Library
Show-Dumps Aktuell-Dump	Es wird ein neues Menü eingeblendet, das es ermöglicht, die Dump-Library zu untersuchen. Mit nächster Dump, voriger Dump, erster Dump, letzter Dump und Dump-Nummer wird die Komponente auf dem Bildschirm gezeigt und automatisch als Aktuell-Dump bezeichnet.	Dump-Library
Komponente-Kopieren Komponente des Aktuell-Diagramms, Unspezifizierte Komponente des Aktuell-Diagramms	Die ausgewählte Komponente wird in die unspezifizierte Komponente kopiert. Die Komponente kann nachher, auf Wunsch des Benutzers, durch eine unspezifierte Komponente ersetzt werden.	Dump-Library
Dump-Text-Kopieren Quelltext des Aktuell-Dumps, Feld	Der Benutzer soll den gewünschten Text in dem 'actual' Dump und das Feld in den Aktuell-Diagramm auswählen. Der gewünschte Text wird in das Feld kopiert.	Dump-Library
Diagramm-Text-Kopieren Diagramm, Quelltext des Diagramms, Feld des Aktuell-Diagramms	Es wird zuerst ein Menü angeboten, wodurch der Benutzer die Möglichkeit hat, ein Diagramm auszuwählen, in dem den gewünschte Text steht. Der Quelltext wird in das Feld des Aktuell-Diagramms kopiert.	Dump-Library
Stop	Es wird das Diagrammmenü eingeblendet.	Diagramm

2.1.3 - Beispiel für die Anwendung des CASD-Systems beim Prozeß der Software-Spezifikation

Problembereich: In einer Universität werden jedes Semester

- die Fächer, die angeboten werden,
- die Gruppen jedes Fachs (z.B. Fach X: Gruppen A,B,C),
- die maximale Anzahl von Besuchern jedes Tupels (Fach,Gruppe) und
- die Termine jedes Tupels

veröffentlich. Die Studenten sollen eine Anzahl solcher (Fach,Gruppe)-Tupel auswählen.

Anforderungsbeschreibung: Es soll ein Programm entwickelt werden, das einen Immatrikulationsantrag (IA) annimmt und überprüft, ob

- der Student die Voraussetzungen jedes ausgewählten Fachs hat,
- die Termine der (Fach,Gruppe)-Tupel sich überschneiden,
- noch freie Plätze in jedem Tupel zu belegen sind.

Wenn ein Fehler auftritt, hat der Student die Möglichkeit, seinen IA zu korrigieren. Der neue IA muß dann wieder überprüft werden. Dieses iterative Verfahren endet, wenn der IA keinen Fehler mehr enthält.

Drei Dateien stehen zur Verfügung:

- Fach-Gruppe-Datei (FG-Datei), die die Anzahl von freien Plätze und die Termine jedes (Fach,Gruppe)-Tupels enthält.
- Studenten-Geschichte-Datei (SG-Datei), die die Fächer, die jeder Student besucht hat, und ihre Ergebnisse enthält.
- Voraussetzungen-Datei (V-Datei), die die Vorausetzungen für jedes Fach (die Fächer, die die Studenten mit Erfolg besucht haben müssen), enthält.

Wenn der IA keinen Fehler mehr enthält, soll die FG-Datei auf den neuen Stand gebracht und der IA in einer IA-Datei abgelegt werden.

Die einzelnen Schritte der aus dieser Anforderungsbeschreibung resultierenden Softwarespezifikation werden im Anhang B dokumentiert.

2.2 - CAD-Unterstützung bei der Erzeugung von Anwenderprogrammen

Mit Hilfe von CADSD kann der Benutzer, wie schon gesagt, eine Datenstruktur entwickeln. Daraus wird ein Programm, das durch ein Diagramm dargestellt und als Dateileser bezeichnet wird, erzeugt. Um ein Anwenderprogramm zu erstellen, muß der Benutzer den Dateileser gegebenfalls erweitern, die Operationen dem Dateileser zuweisen (ohne daß seine Logik geändert wird) und den Datenfluß festlegen. Wie soll nun das CASD-System verwendet werden, um das Anwenderprogramm zu erstellen?

2.2.1 - Dateileser-Erweitern

In Kapitel VI, Punkt 2, haben wir beispielsweise zwei Operationen vorgesehen, um den Dateileser zu erweitern:

1) P(Kn,#Uk,i) und
2) T(Kn)

Um die Operation P durch das CASD-System durchführen zu können, muß der Benutzer folgende Schritte ausführen:

1. - Die Operation 'Diagramm-Transformieren' im Diagramm-menü auswählen;

2. - Die Operation 'Dump-Erzeugung' im Dump-Library-Menü auswählen. Dabei wird gefragt, welche Komponente (erster Parameter der Operation P) zur Dump-Library gebracht werden soll. Es

sei E die unspezifizierte Komponente, die die Komponente ersetzt hat;

3. - Die Operation 'Stop' im Dump-Librarymenü auswählen;

4. - Die Operation 'Seq-Einfügen' im Diagramm-Menü auswählen. Auf die Frage, wo die Sequenz-Komponente eingefügt werden soll, muß der Benutzer die unspezifierte Komponente E angeben. Dabei wird gefragt, wieviele Unterkomponenten (zweiter Parameter der Operation P) die Sequenz-Komponente haben soll. Es sei S die Sequenz-Komponente, die eingefügt wurde.

5. - Die Operation 'Diagramm-Transformation' im Diagramm-Menü auswählen;

6. - Die Operation 'Dump-Kopieren' in dem Dump-Library-Menü auswählen. Auf die Frage, wohin der aktuelle Dump kopiert werden soll, soll der Benutzer eine Unterkomponente (dritter Paramenter der Operation P) der Komponente S angeben;

Um die Operation T durch das CASD-System durchführen zu können, muß der Benutzer die Operation 'If-Einfügen' im Diagramm-Menü und für diese Operation eine unspezifizierte Komponente auf dem Diagramm auswählen.

2.2.2 - Operation-Zuweisen

In Kapitel IV, Punkt 2, wurde eine einzige Operation vorgesehen, mit der der Benutzer eine Operationen spezifizieren und zuweisen kann:

Z(Name,Kn [, F(o),E(o),A(o)])

Mit den Operationen des CASD-Systems, die auf den Menüs stehen, kann man diese Operation Z nicht ausführen. Also fügen wir eine solche Operation Z auf dem Diagramm-Menü ein:

Operation: Operation-Zuweisen

Parameter: - der Name der Operation,
- eine If-Komponente, deren Bedingungs-Feld B (s. Kapitel VI, Punkt 1.2) leer oder eine unspezifizierte Komponente ist,
- ein Text, der die Funktion der Operation F(o) darstellt,
- eine Liste der Eingabedatenobjekten E(o) und
- eine Liste der Ausgabedatenobjekten A(o).

Semantik: Wenn der Name der Operation auf der Operationstabelle bereits existiert, dann wird die Funktion der Operation in die unspezifierte Komponente kopiert, wenn die Operation vom Typ P ist (s. Kapitel VI, Punkt 1.1) oder auf des Bedingungs-Feld B, wenn die Operation vom Typ T ist.

Wenn der Name der Operation nicht auf der Tabelle steht, dann wird automatisch der Texteditor zur Verfügung gestellt, um die Funktion der Operation in die ausgewählte Komponente einzutragen (dritter Parameter der Operation Z). Anschließend wird nach der Liste der Ein- bzw. Ausgabedatenobjekte gefragt (vierter und fünfter Parameter der Funktion Z). Der Name, der Typ , der von der ausgewählten Komponente abhängt, E(o), A(o) der Operation und die Komponentenkennzeichnung werden in die Operationstabelle eingetragen.

2.2.3 - Datenfluß-Festlegen

In Kapitel VI, Punkt 4, wurde eine Operation vorgesehen, nämlich

F(Kn,e,L),

mit der der Benutzer den Datenfluß festlegen kann.

Wiederum kann man mit den Operationen des CASD-Systems den Datenfluß nicht festlegen. Die Operation wird in das Library-Menü eingefügt.

Operation: Datenfluß-Festlegen.

Parameter: - Kn ist eine Primitiv-Komponente oder eine If-Komponente,
- e ist ein Eingabedatenobjekt und
- L ist eine Liste von primitiven Komponenten, (die Lieferanten)

Semantik: Es wird eine neue Zeile und eine neue Spalte in die Datenflußtabelle (Bild 7.4) eingetragen:

Komp. / Ein.	. . . . .	Kn	
. . . .			
e		L	

Bild 7.4: Die Datenflußtabelle.

2.2.3.1 - Datenfluß-Überprüfung

Diese Operation, die auf dem Library-Menü des CASD-Systems angeboten werden soll, überprüft die Reihe von Bedingungen für die Operationen, die wir in Kapitel V, Punkt 2.1 vorschlagen haben.

2.2.4 - Code-Generierung

Nachdem die Operationen dem Dateileser zugewiesen und der Datenfluß festgelegt wurde, ist das Anwenderprogramm fertig. Der nächste Schritt (s. Kapitel VI) ist, den Code für das Anwenderprogramm zu generieren.

Um diese Tätigkeit durchführen zu können, muß der Benutzer eine Programmiersprache aus einem Menü auswählen.

Da die die Gruppe der sogenannten Prozeduralsprachen (Algol, Pascal, PL/I ...) die Kontrollstrukturen wie if-then-else, begin-end, ... haben, wird die Übersetzung eines Anwenderprogramms in eine dieser Sprache leichter als z.B. in Prolog /Clo-81/ oder Snobol /Ros-67/.

Da das CASD-System selbst die Operation 'Code-Generierung' für ein Anwenderprogramm nicht enthält, wird im folgenden eine Vorgehensweise vorgeschlagen, um ein Anwenderprogramm in Pascal zu übersetzen.

2.2.4.1 - Spezifikation der Daten

a) - Da der Datenträger Zeichenketten (P-Datenobjekte, die Variable sind, und tag-Werte) enthält, soll die Variable 'Wort' vom Typ String sein (s. Kapitel IV). Dabei wird der Benutzer nach der Länge n des Strings gefragt.

b) - Jedes tag soll eine Variable vom Typ String sein.

c) - Jeder Zähler des I-Datenobjektes (s. Kapitel IV, Punkt 1) soll eine Variable vom Typ Integer sein.

d) - Zu jedem Ausgabedatenobjekt jeder Operation, die auf der Operationstabelle steht, muß der Benutzer den Typ (Integer, Real oder String) angeben.

Bemerkungen: - Da insgesamt nur Read, Write und arithmetische Operationen vorkommen, darf kein Ausgabedatenobjekt von 'Transform-Operation' (s. Kapitel V) vom Typ String sein.

- Alle Ausgabedatenobjekte e der Operationen der Lieferanten von e müssen von demselben Typ sein. In Bild 7.5 sind Ki, Kj und Km die Lieferanten von e und Op_1, Op_2 und Op_3 die Operationen der Lieferanten von e.

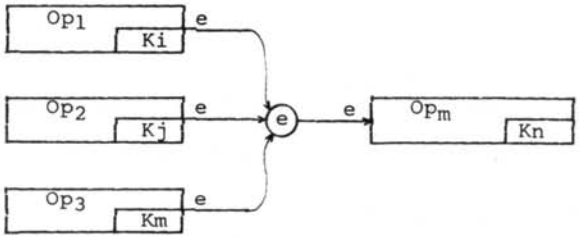


Bild 7.5: Der Datentyp der Lieferanten von e.

2.2.4.2 - Generierung der Deklarationen

a) - für Wort wird

'Var Wort : String [<n>] ; '

generiert.

b) - für jedes tag t, $[t,x] \in F_T$, $x \in TD$ wird

'Var <t> : String [<n>] ; '

generiert.

c) - Für jedes Paar [e,Kn], Bild 7.5, wird

'Var $<V_i>$: <Typ> ; '

generiert, wobei <Typ> der Typ des Ausgabedatenobjekts der Operationen der Lieferanten von e (Bild 7.5) ist und $<V_i>$ ein 'Label', das sequentiell generiert werden soll(V1,V2,V3...)

d) - für jedes Datenobjekt i, so daß $i \in I$ und MAX(i) $\neq$ '*' $\vee$ MIN(i) $\neq$ *, wird

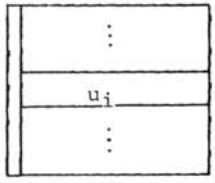
'Var <Z_j> : Integer ; '

generiert, wobei <Z_j> ein 'Label' ist, das sequentiell generiert werden soll (Z1,Z2,Z3...).

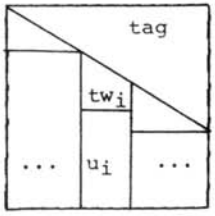
2.2.4.3 - Generierung der Anweisungen

Die Funktion Übersetzung wird in einer Algol-ähnlichen Sprache spezifiziert; Sie generiert den Pascal-Code für ein Anwenderprogramm. Die Funktion erhält als Parameter eine Komponente, für die sie Anweisungen erzeugt. Die Funktionen, die die Funktion Übersetzung aufrufen, und die verwendeten Notationen werden erst nach der Darstellung der Funktion erklärt.

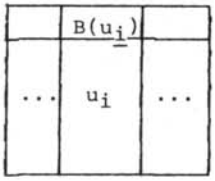
```
Funktion Übersetzung (k:Komponente):String;
Case k Of

Sequenz:

Übersetzung :=
                'Begin' +
                . . . +
                Übersetzung(ui) +
                . . . +
                'End ;' ;
```




```
Case-K:

Übersetzung :=
                'Case' + tag    +   'Of' +
                . . . +
                twi    + ':' +
                Übersetzung(ui)   + ';' +
                . . . +
                'End ;' ;
```



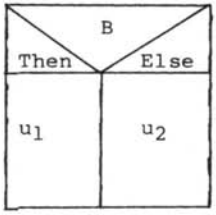
Case-A:



Übersetzung :=

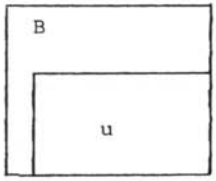
```
. . . +
'If' + T1(B(ui)) +
'Then' + Übersetzung(ui) +
'Else' +
. . . + ';' ;
```

If:




```
Begin
Write (B);
Read (Bedingung);
Übersetzung :=
              'If' + T2(Bedingung)  +
              'Then' + Übersetzung(u1) +
              'Else' + Übersetzung(u2)+';'
End ;
```

While:

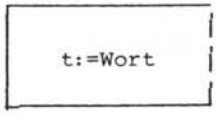


Übersetzung :=

```
          T3(B) + Übersetzung(u)  + ';' ;
```

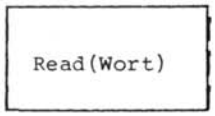
Primitiv:

Case P-Typ Of



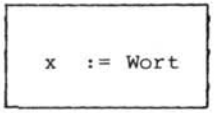
Typ-A:

Übersetzung := t + ':= Wort ;' ;



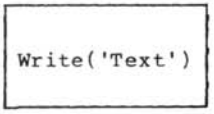
Typ-B:

Übersetzung := 'Read(Wort) ;' ;



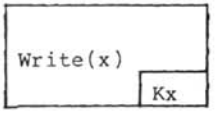
Typ-C:

Übersetzung:= T_4(x) + ';' ;



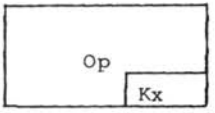
Typ-D:

Übersetzung:= 'Write('Text') ;' ;



Typ-E:

Übersetzung := 'Write(' + T_5(x,Kx) + ') ;';



Typ-F:

```
Begin
Write(F(Op))
While x ∈ A(Op) existiert  Do /*Pseudo-Code*/
   Begin
   Write (x)
   Read (Pascal-Op)
   Übersetzung := T6 (x,Kx,Pascal-Op) + ';' ;
   End;
End;
End.
```

Wir werden nun die Notation und die aufrufenden Funktionen erklären:

a) - Notation:

Bei Sequenz : u_i ist die i-te Unterkomponente der Sequenz.

Bei Case-K : tag ist der 'Case-Selector'(s. Pascal-Syntax), tw_i ist das i-te 'Case-Label' von Case-K und u_i ist die i-te Unterkomponente von Case-K.

Bei Case-A : $B(u_i)$ bezeichnet die Bedingung der i-ten Unterkomponente von Case-A.

Bei If : u_1 ist die Unterkomponente, die 'then' entspricht und u_2 ist die Unterkomponente, die 'else' entspricht.

Bei While : B ist die Bedingung (s. Kapitel IV, Punkt 1) und u ist die Unterkomponente.

Bei Primitiv :

Typ-A : Diese Operation wird bei 'Dateileser-Ableitung' dann erzeugt, wenn ein Datenobjekt x der Datenstruktur ein TD-Datenobjekt ist und es das tag 'tag' hat ($[tag,x] \in F_T$, s. Kapitel IV, Punkt 1). Für sie ist automatisch (ohne Hilfe des Benutzers) Code zu generieren.

Typ-B : Diese Operation wird bei 'Dateileser-Ableitung' erzeugt. Für sie ist es möglich, den Code automatisch zu generieren.

Typ-C : Diese Operation wird bei 'Dateileser-Ableitung' erzeugt, wenn das Datenobjekt x eine Variable ist. Für sie ist es möglich, den Code automatisch zu generieren.

Typ-D : Diese Operation ist erkennbar, wenn der Benutzer

sie durch ihre Syntax spezifiziert, und so ist es möglich, für sie den Code automatisch zu generieren.

Typ-E : x ist ein Datenobjekt, das Ausgabe einer Operation ist. Es ist erkennbar, wenn der Benutzer sie durch ihre Syntax spezifiziert, und so ist es möglich, den Code automatisch zu generieren.

Typ-F : Op ist eine Operation, deren Funktion, F(o), vom Typ-A bis E nicht zu erkennen ist.

b) - Aufruffunktionen:

$T_1(B(u_i))$: Diese Funktion soll aus der Bedingung der Unterkomponente u_i (s. Kapitel IV, Punkt 1) einen korrekten Pascal Text erzeugen. Nehmen wir als Beispiel an, daß

$$(A) \wedge (B \vee C)$$

eine Bedingung $B(u_i)$ ist, wobei A,B und C Datenobjekte sind. Die Funktion $T_1(B(u_i))$ soll jedes Datenobjekt durch seine Bedingung ersetzen. Dieser Ersetzungsvorgang hört auf, wenn alle Datenobjekte keine Bedingung mehr haben. Diese Datenobjekte sind Nachfolger von TD-Datentobjekten und müssen durch

$$t = tw_i$$

ersetzt werden, wobei t das tag des Vorgängers und tw_i der tag-Wert des Datenobjekts ist.

Beispiel: Für den Dateileser von DS(Studenten-Geschichte) und die Bedingung

$$(1 \vee 2 \vee 3 \vee 4)$$

erzeugt die Funktion T_1 den Code

(Note=1 v Note=2 v Note=3 v Note=4),

wenn Note selbst das tag des Datenobjekts Note und die tag-Werte selbst die Nachfolger sind (Es handelt sich hier um implizite Kontrolldaten.)

T_2(Bedingung) : Die T-Operation, die einer If-Komponente zugewiesen wurde, wird vom Benutzer spezifiziert und steht auf der Operationstabelle. B ist die Funktion der Operation. Dieser Ausdruck wird dem Benutzer gezeigt (Write(B)); dafür soll er einen korrekten Pascal-Text angeben (Read(Bedingung)). Die Bedingung ist ein boolescher Ausdruck (s. Kapitel VI, Punkt 1.2.1) und enthält Datenobjekte, die Eingabedatenobjekte sind. Sie stehen auf der Operationstabelle. Nehmen wir an, x sei ein solches Datenobjekt. Dann soll x durch V_i ersetzt werden, wobei V_i die Variable ist, die für das Paar [x,if] deklariert wurde (s. Generierung der Deklarationen). Die Datenobjekte, die im Pascal-Text stehen und zur Datenstruktur gehören, müssen durch ihre Bedingung ersetzt werden (s. Vorgang bei T_1).

T_3(B): B ist eine 'while' Bedingung. Da wir fünf Fälle von Bedingungen festgestellt haben (sh. Kapitel IV, Punkt 1), soll die Funktion einen Pascal-Text für jeden dieser Fälle erzeugen. Wir haben gezeigt, daß die richtige Anweisung für die While-Komponente das For-Step-While von Algol wäre. Pascal hat aber eine solche Anweisung nicht. Wir erzeugen aber trotzdem For-Step-While, da die kompakte Darstellung des Codes die Übersetzung in Pascal immer ermöglicht. Für jeden in Kapitel IV, Punkt 1 geschildeten Fall erzeugen wir folgenden Code:

Fall 1. For Z_i := 1 Step 1 While B Do

Fälle 2, 3 und 5 sind gleich Fall 1.

Fall 4. While B Do

$T_4(x)$: Die Operation x:=Wort wurde durch 'Dateileser-Ableitung' erzeugt. Die Operation liefert das Datenobjekt x. Eine Frage ist, welche Operation das Datenobjekt x benutzt. $T_4(x)$ soll den Code so generieren, daß jede Operation, die x benutzt, es ,wie in Bild 7.6 gezeigt, bekommt.

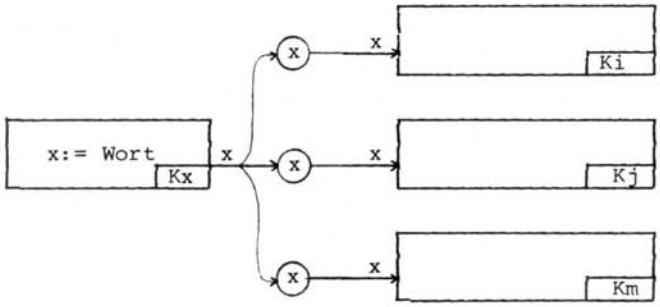


Bild 7.6: Die Benutzer von x

In Bild 7.6, ist Kx ein Lieferant von x:

$$L(x,Ki) = \{ \ldots,Kx, \ldots \}$$
$$L(x,Kj) = \{ \ldots,Kx, \ldots \}$$
$$L(x,Km) = \{ \ldots,Kx, \ldots \}$$

Damit jeder Benutzer von x auch x bekommen kann, soll T_4 den folgenden Code generieren:

$$V_i := V_j := V_m := \text{Wort} ,$$

wobei V_i,V_j und V_m diejenigen Variablen sind, die respektive für die Paare

$[x,Ki]$,
$[x,Kj]$ bzw.
$[x,Km]$

deklariert wurden.

$T_5(x,Kx)$: Die Operation Write(x) wird durch den Benutzer spezifiziert und steht auf der Operationstabelle. x ist das Eingabedatenobjekt für die Operation. T_5 generiert den Code:

$$V_k \ ,$$

wobei V_k die Variable ist, die für $[x,Kx]$ deklariert wird und Kx die Komponente ist, der die Operation Write(x) zugewiesen wird.

$T_6(x,Ki,Pascal\text{-}Op)$: Die Operation Op wird durch den Benutzer spezifiziert und der Komponente Ki zugewiesen. Wir gehen davon aus, daß

- diese Operation vom Typ 'Transform' ist (s. Kapitel V, Punkt 1) und eine Liste von Operationen zusammenfassen kann. (Deshalb wird für jedes Ausgabedatenobjekt Code generiert. (s. Kapitel V, Punkt 2), und daß

- Pascal-OP ein korrekter arithmetischer Pascal-Ausdruck ist.

T_6 ist eine Kombination von T_4 und T_5. T_6 generiert (Bild 7.6):

$$V_i:=V_j:=V_m \ \ldots \ :=Pascal\text{-}Op',$$

wobei V_i,V_j,V_m diejenigen Variablen sind, für die

$[x,Ki]$,
$[x,Kj]$ bzw.
$[x,Km]$

deklariert wird; Pascal-Op' ist die Funktion der Operation Pascal-Op, in der jedes Eingabedatenobjekt e durch V_n ersetzt wird. V_n ist dabei diejenige Variable, für die das Paar

$[e,Kx]$

deklariert wird.

VIII - ABSCHLUSS

In folgenden werden Vorschläge zur thematischen Erweiterung der Arbeit gemacht.

1. Datenstrukturen

1.1 - Mehrfach-Datenstrukturen

Wir haben eine Datenstruktur eingeführt, aus der ein Dateileser erzeugt werden kann, der nur eine entsprechende Datei lesen kann. In der Praxis liest und verarbeitet ein Anwenderprogramm aber Daten, die auf verschiedene Dateien verteilt sind. Jede Datei enthält dabei die Daten eines bestimmten Datenobjekts . Es ist noch zu untersuchen, wie sich aus den verschiedenen Datenstrukturen eine neue Datenstruktur erstellen läßt, woraus der entsprechende Dateileser dann jede der Dateien richtig lesen kann. Dabei müssen sortierte Dateien sowie Datenobjekte, die zu verschiedenen Datenstrukturen gehören und sich wiederholen bzw. sich ergänzen, betrachtet werden.

1.2 - Zeitliche Zustandsänderung

Wie schon gesagt, dient die Datenstruktur dazu, um ein Objekt der realen Welt zu beschreiben. Wenn sich das Objekt in der realen Welt ändert, entspricht das einer neuen Ausprägung der Datenstruktur.

Diese neue Ausprägung wird durch des Up-Programm (s. Kapitel IV, Punkt 3) erzeugt. Dabei wird der Benutzer bei jedem unabhängigen Datum gefragt, ob das Nachfolgerdatum verändert werden muß.

Der Benutzer wird vom System in der Wahl dieses Nachfolgers nicht eingeschränkt. Dies entspricht oft nicht der Realität. Deshalb sollte die Datenstruktur so erweitert werden, daß schon

der Software-Entwickler beim Erstellen der Datenstruktur festlegt, welchen Nachfolger ein unabhängiges Datenobjekt am Anfang haben darf und welche weiteren Nachfolger dann zulässig sind.

Beispiel: Der Familienstand einer Person kann ledig, verheiratet, verwitwet oder geschieden sein. Wenn eine Person in eine Datei neu eingefügt wird, sind für ihren Familienstand alle vier Möglichkeiten offen. Bei einer Änderung des Familienstands sind die neuen Möglichkeiten jedoch eingeschränkt. Eine verheiratete Person kann den Status 'geschieden' oder 'verwitwet' bekommen, nicht jedoch 'ledig'.

2 - Die CASD-Systeme

2.1 - CAD-Unterstützung beim Entwurf von Datenstrukturen

Wenn ein Benutzer eine abgelegte Datenstruktur holt und abändert, so ist nicht sicher, ob der neue daraus abgeleitete Dateileser die Daten der alten Datenstruktur noch lesen kann. Das CADSD-System sollte daher eine neue Operation anbieten, die zwei Datenstrukturen vergleicht und dabei feststellt, ob die Änderungen des Benutzers so beschaffen sind, daß der neue Dateileser die Daten der alte Datenstruktur noch lesen kann.

2.2 - CAD-Unterstützung beim Erstellen von Anwenderprogrammen

In einen bestehenden Dateileser kann ein Benutzer mit Hilfe des CASD-Systems die interne Operationen einfügen und damit ein Anwenderprogramm erstellen. Dabei kann er Fehler machen und, da alle CASD-Operationen zur Fehler Korrektur zur Verfügung stehen, die Logik des gegebenen Dateilesers verletzen. Das CASD-System sollte daher eine neue Operation anbieten, die feststellt, ob bei Erstellung des Anwenderprogramms die Logik des Dateilesers betroffen wird.

2.3 - CASD-Unterstützung beim Prozeß der Software Spezifikation

Während der Arbeit mit dem CASD-System wurden von Benutzer zwei Probleme festgestellt:

1. Ein Diagramm stellt sich dar als Baum von Komponenten, dessen primitive Komponenten primitive Operationen oder Prozeduraufrufe sind. In einem ersten Schritt enthält ein Diagramm keinen Prozeduraufruf. In einem zweiten Schritte kann der Benutzer entscheiden, ob eine primitive Operation in einen Prozeduraufruf verwandelt werden kann. In diesem Fall muß der Benutzer zunächst die Prozedur spezifizieren. Der Text der Anforderungsbeschreibung der Prozedur muß aus den Beschreibungen der verschiedenen primitiven Operationen neu formuliert werden. Der Texteditor ist hier nicht verwendbar, weil er keine Funktion hat, die solche Texte aus unterschiedlichen primitiven Operationen kopieren und konkatenieren kann. Dann muß der Benutzer die primitiven Operationen durch die Operation 'Dump-erzeugen' des Menüs 'Dump-Library' löschen; sie werden durch unspezifizierte Komponenten ersetzt. Schließlich muß der Benutzer die unspezifizierten Komponenten mit der Operation 'Prozedur-Aufruf-einfügen' im Menü-Diagramm durch einen Prozeduraufruf ersetzen. Diese Schritte bringen für die Benutzer erfahrungsgemäß viel Aufwand. Das CASD-System sollte daher Operationen anbieten, die das Verfahren, primitive Operationen zu verwandeln, vereinfacht.

2. Ein zweites Problem betrifft die Diagrammgröße. Je grösser die Anzahl der Komponenten in einem Diagramm ist, desto kleiner werden die Felder zum Beschriften der Komponenten. Wenn mehr Platz in einem Feld benötigt wird, muß der Benutzer überlegen, wie er dafür Platz schaffen kann. Das Verfahren ist iterativ und meist zu langsam und lenkt den Benutzer von seinem eigentlichen Tun ab. Ein Ausweg ist hier, mit einer Fenstertechnik zu arbeiten oder wenigtens mit der Möglichkeit einer dynamischen Veränderung der Größe der Feldgrößen, zB. große Felder, die nur wenig Text enthalten, werden automatisch verkleinert.

2.4 - Trennung der Datenstruktur von den internen Operationen

Die Zusammenhänge der realen Welt sind in einer Datenstruktur dargestellt. Wenn der Software-Entwickler ein Anwenderprogramm erstellt, muß er dabei die internen Operationen festlegen. Die Datenstruktur bleibt unabhängig von der Wahl dieser Operationen.

Auch beim Einfügen neuer Datenobjekte in die Datenstruktur entstehen keine Beziehungen zu den internen Operationen.

Die Erstellung des Anwenderprogramms ist bisher 'halbautomatisch', da das Einfügen der (internen) Operationen in den Dateileser noch durch den Benutzer gemacht werden muß. Wünschenswert wäre, auch diesen Schritt zu automatisieren. Dies ist möglich, weil die Operationen vom Dateileser abhängen. Wenn zB. alle Eingabedaten einer Operation im Dateileser eingelesen sind, kann dieser die vom Benutzer festgelegte (interne) Operation ohne weiteres Zutun des Benutzers direkt ausführen.

Aus den Bedingungen, denen die Operationen in Anwenderprogramm unterliegen (s. Kapitel V, Punkt 2.1), lassen sich weitere Möglikeiten entwickeln, um den Vorgang der Erstellung von Anwenderprogrammen vollständig zu automatisieren.

3 - Erweiterung des Konzepts der Datenstruktur

Im Modell unserer Datenstruktur gelten folgende Bedingungen:

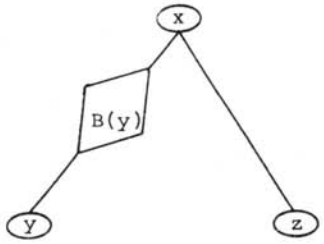
a) - die unabhängigen Datenobjekte besitzen keine Bedingung für ihre Nachfolger;

b) - die abhängigen Datenobjekte haben zwar eine Bedingung für ihre Nachfolger, diese aber wird nur logisch betrachtet;

c) - die Bedingungen für die iterativen Datenobjekte werden auf MIN (integer), MAX (integer) und EOL (Konstante) beschränkt.

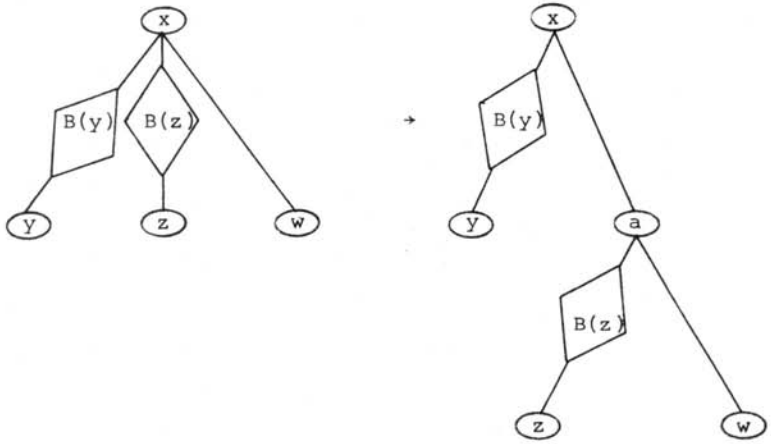
Eine Erweiterung dieses Modells wäre:

zu b) - Die abhängigen Datenobjekte haben eine Bedingung für ihre Nachfolger, die ein boolescher Ausdruck ist, der arithmetische, logische und relationale Operatoren enthält. Die graphische Darstellung für die unabhängigen Datenobjekte sei dann:



Das Datenobjekt x ist ein y Datenobjekt genau dann, wenn x die Bedingung B(y) erfüllt. Sonst ist das Datenobjekt x ein z Datenobjekt. Das Datenobjekt z kann explizit auftreten, wie die Abbildung zeigt, oder ein anonymes Datenobjekt darstellen.(Das Datenobjekt x hat dann nur einen Nachfolger.)

Allgemein gilt dann auch



Dabei ist die rechte Datenstruktur jeweils eine Expansion der linken Datenstruktur. Das Datenobjekt w kann ein anonymes Datenobjekt sein; bei der Expansion stellt dann das Datenobjekt a auch ein anonymes Datenobjekt dar.

zu c) - Die iterativen Datenobjekte besitzen für MIN und MAX je einen arithmetischen Ausdruck und für EOL einen booleschen Ausdruck.

Bei diesem neuen Modell von Datenstrukturen ist zu untersuchen:

a) - Welche Datenobjekte treten in den verschiedenen Ausdrücken auf? Anfangspunkte der Untersuchung könnten sein:

a.1 - Nachfolger der D-Datenobjekte, die als boolesche Variable betrachtet werden;

a.2 - P-Datenobjekte, die Variablen sind und vom Typ String, Integer oder Real sind und

a.3 - Symbole, die als Konstante betrachtet werden.

Da ein Datenobjekt mehrfach interpretiert werden kann (z.B. ein Nachfolger eines D-Datenobjkets kann auch ein P-Datenobjekt sein) sollte diese Zweideutigkeit untersucht werden;

b) - Welche Bedingung sollten die Datenobjekte erfüllen, um in einem Ausdruck auftreten zu dürfen? Die Reihenfolge-Bedingungen (s. Kapitel III, Punkt 11.2.1.1) könnten einen Anfang für diese Überlegungen sein.

c) - Wie kann man die 'toten' Datenobjekte feststellen?

d) - Wie kann man feststellen, ob die iterativen Datenobjekte unendliche Schleifen liefern?

e) - Wie kann man neue Datenobjekte in die Datenstruktur einfügen, ohne daß ihre Logik betroffen wird, und damit auch der neue Dateileser die alte Datei lesen kann?

ANHANG A - DAS CASD-SYSTEM

1 - Die Softwarearchitektur des CASD-Systems

Die Software des CASD-Systems ist hierarchisch strukturiert (Bild A.1).

Bild A.1: Die Softwarearchitektur des CASD-Systems

1.1 - Abstrakte Datenstruktur

Das Datenobjekt, das vom CASD-System verarbeitet wird, ist eine 'Library' der Ableitungsbäume (s. Kapitel VII, Punkt 2.1). Die abstrakte Datenstruktur der 'Library' ist komplex vom Typ 'record'.

1.2 - Prozedurale Schnittstelle

Zur Unterstützung der Verarbeitung des Datenobjekts 'Library' werden ca. 50 (interne) Operationen benötigt, die als Prozeduren

oder Funktionen implementiert wurden /Die-83/. Diese Operationen sind implementierungsabhängig und verwalten die Zeiger, die graphische Darstellung der Struktrogramme usw.. Die Programmiersprache ist Pascal.

1.3 Benutzeroperationen

Die Benutzeroperationen verarbeiten das Datenobjekt 'Library'. Sie wurden als Prozeduren implementiert und nach ihren Aufgaben in 6 Module gruppiert (Erzeugen, Modifizieren, Einfügen und Löschen, Dokumentation, Texteditor, Transformation)

1.4 Dialog-Schnittstelle

Ein weiteres Modul führt dem Dialog mit dem Benutzer. Der Dialog ist menügesteuert (Kapitel VII, Punkt 2.1.2); Durch die Auswahl des Benutzers wird das entsprechende Modul der Benutzeroperation aufgerufen. Das aufgerufene Modul übernimmt dann den Dialog.

1.5 Technische Angaben

Das Software-System umfaßt ca. 500 KByte. Die 'Library' jedes Benutzers wird in eine Datei abgelegt. Wegen der umfangreichen Anzahl von Diagrammen, die ein Ableitungsbaum besitzen kann, wird auch jedes Diagramm in eine Datei abgelegt. Das steigert zwar die Verarbeitungszeit (insbesondere derjenigen Operationen, die den Ableitungsbaum verarbeiten), ermöglicht aber, daß Ableitungsbäume mit einer beliebigen Anzahl von Diagrammen erzeugt werden können.

2 - Hardware Konfiguration

Zur Implementierung des CASD-Systems stand folgende Systemhardware (Hewlett-Packard*) zur Verfügung.

- CPU HP9834 mit: - 2 MByte Hauptspeicher,
  - 2 integrierte Laufwerke für 'mini Floppy', 5,25 Zoll, 250 KByte',
  - Tastatur Modell 36

- Bildschirm (24 Zeilen x 80 Zeichen),

- Graphik-Tablett HP 9111 A,

- Plotter HP 470 A,

- Thermodrucker HP 2671,

- Massenspeichereinheit HP 9895 A für Floppy-Disk, 8 Zoll, 1,15 Mbyte.

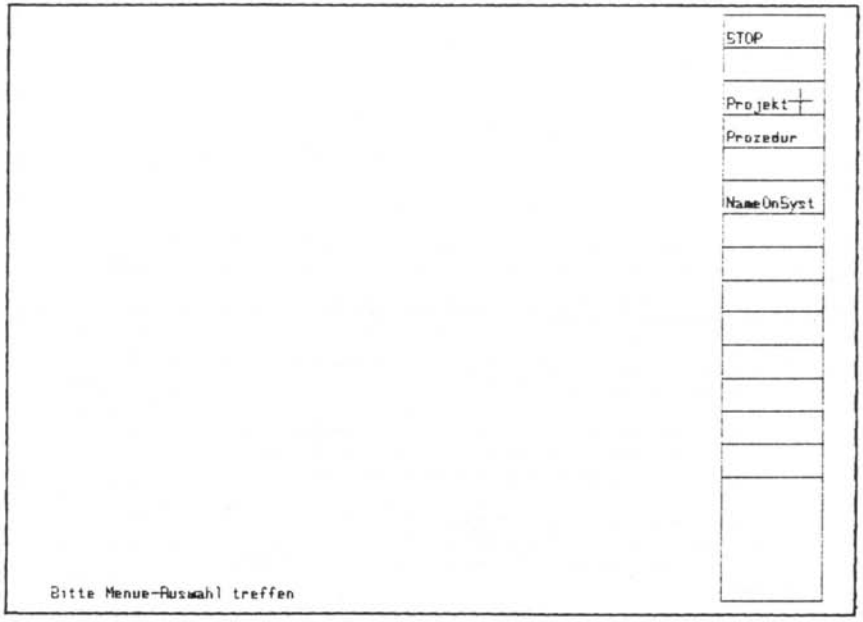
---

* Für die Bereitstellung der Geräte sei der Firma Hewlett-Packard GmbH in Böblingen herzlich gedankt.

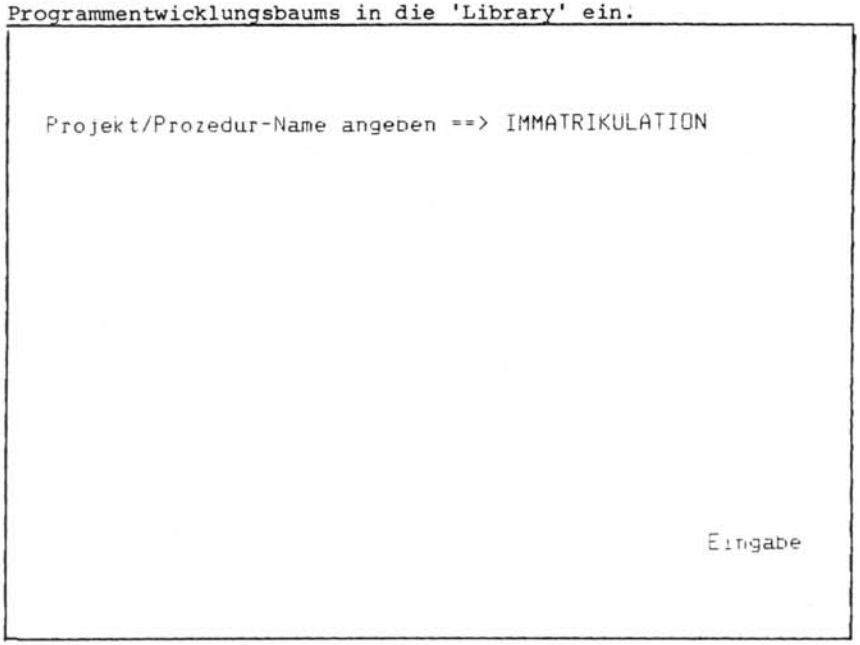
ANHANG B - ERSTELLUNG DER SOFTWARESPEZIFIKATION

Die Softwarespezifikation für die Anforderungsbeschreibung, die in Kapitel VII, Punkt 2.1.3, gegeben wurde, wird hier mit Hilfe des CASD-Systems am Beispiel einzelner Dialogschritte dargestellt.

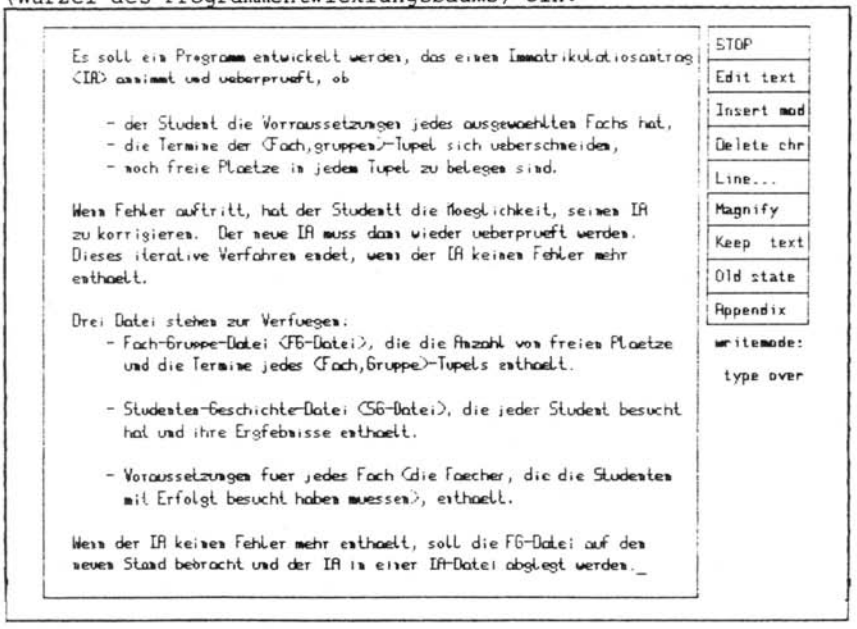
Beispiel 1: Der Benutzer teilt dem CASD-System mit, daß ein 'Programmentwicklungsbaum' (Projekt) zu erstellen ist. Das Fadenkreuz zeigt jeweils an, was der Benutzer auswählt.



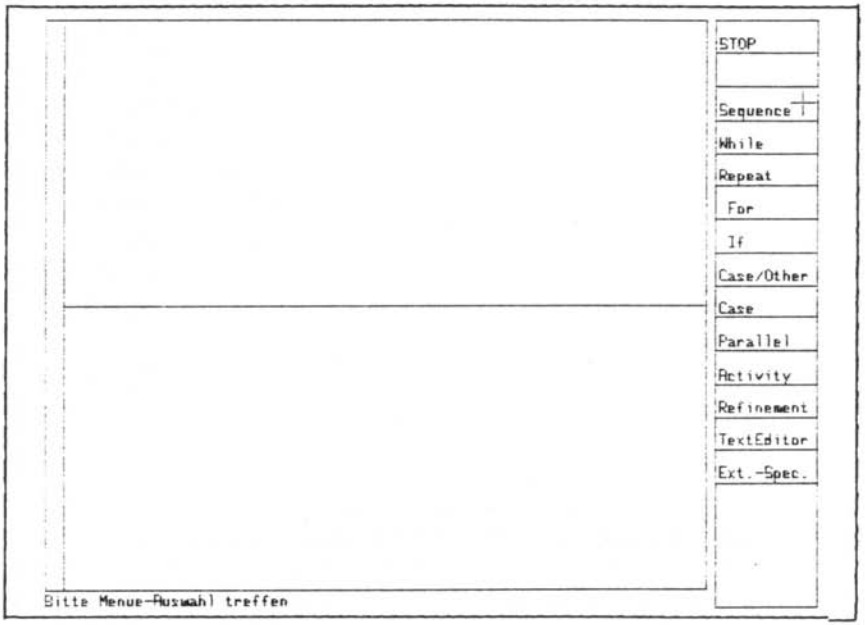
Beispiel 2: Der Benutzer trägt den Namen (IMMATRIKULATION) des Programmentwicklungsbaums in die 'Library' ein.



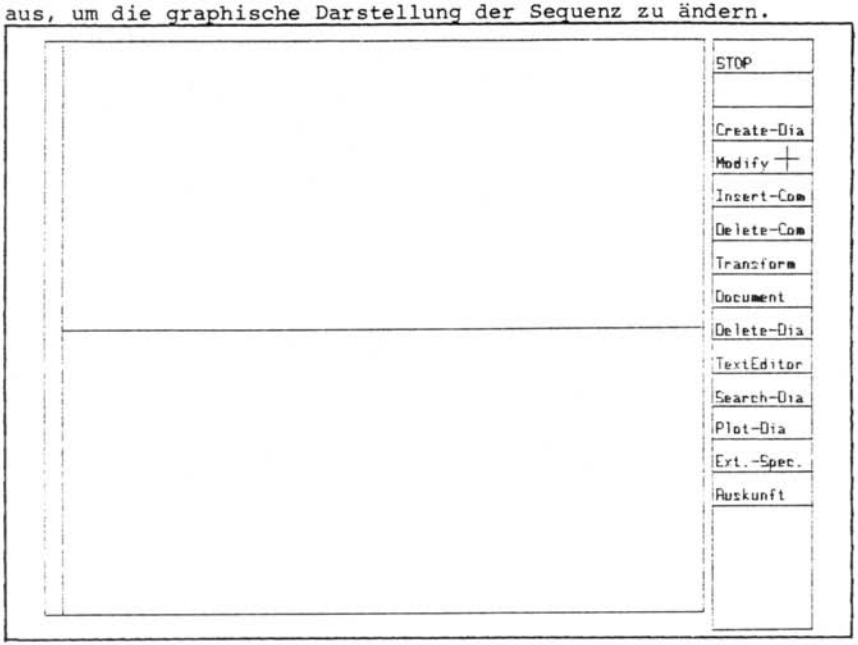
Beispiel 3: Der Benutzer trägt die Anforderungsbeschreibung (Wurzel des Programmentwicklungsbaums) ein.



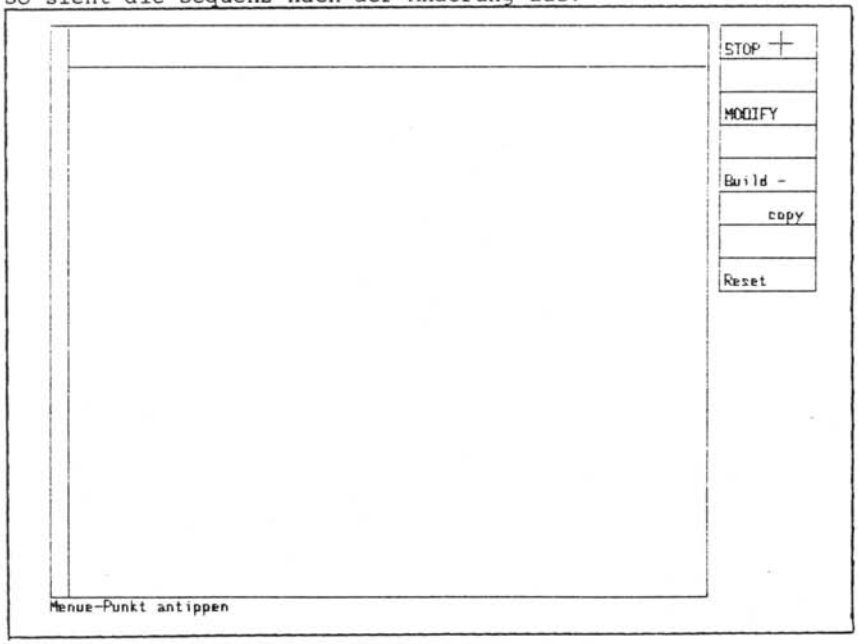
Beispiel 4: Das Diagramm, das der Anforderungsbeschreibung entspricht, fängt mit einer Sequenz an. Das Fadenkreuz zeigt wieder an, daß eine Sequenz ausgewählt wird.



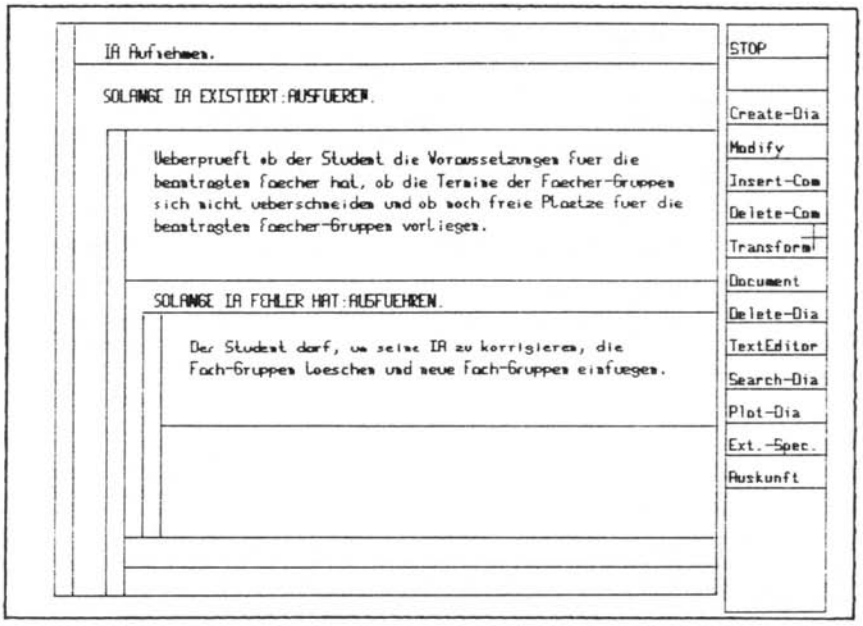
Beispiel 5: Der Benutzer wählt das Modul 'Modify' (Modifizieren) aus, um die graphische Darstellung der Sequenz zu ändern.



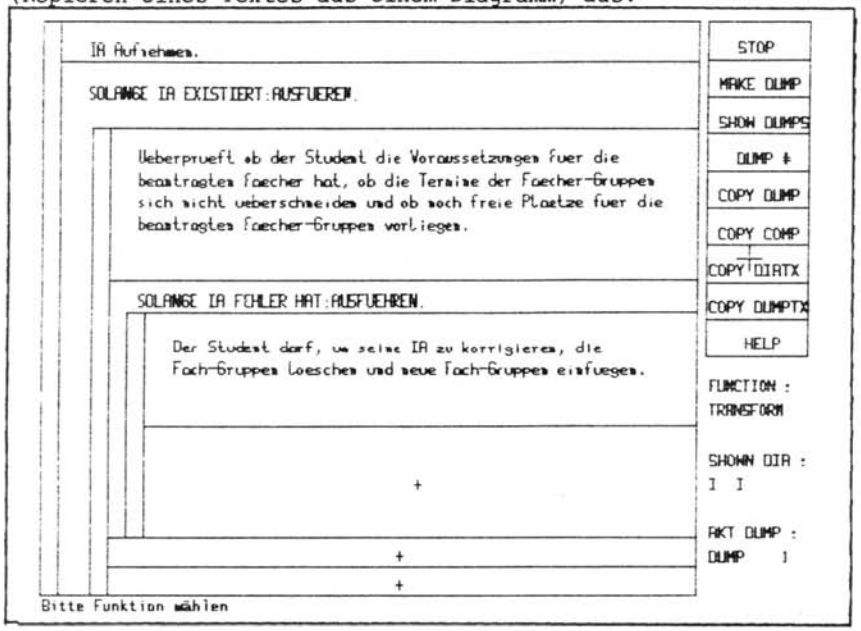
So sieht die Sequenz nach der Änderung aus:



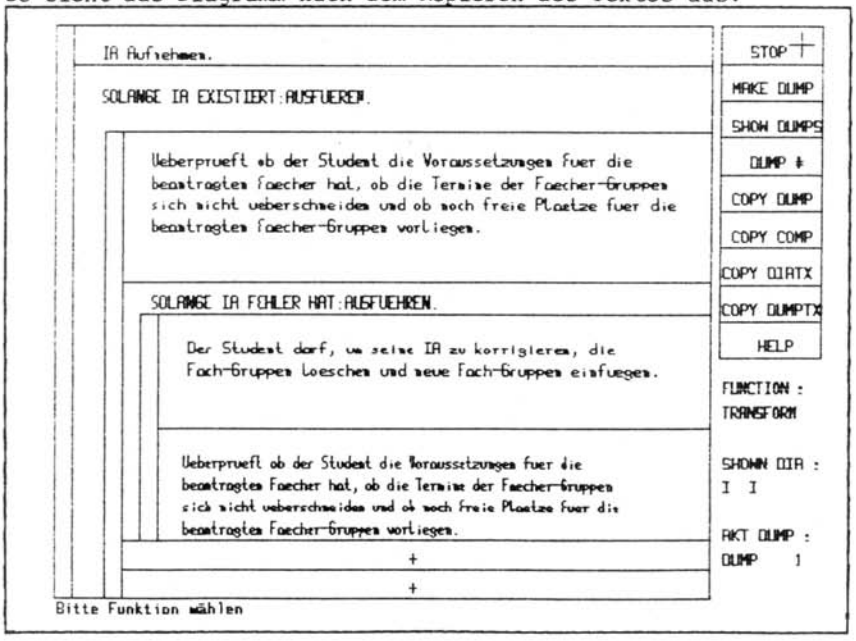
Beispiel 6: Der Benutzer wählt das Modul 'Transform' (Transformieren) aus, um ein Text zu kopieren.



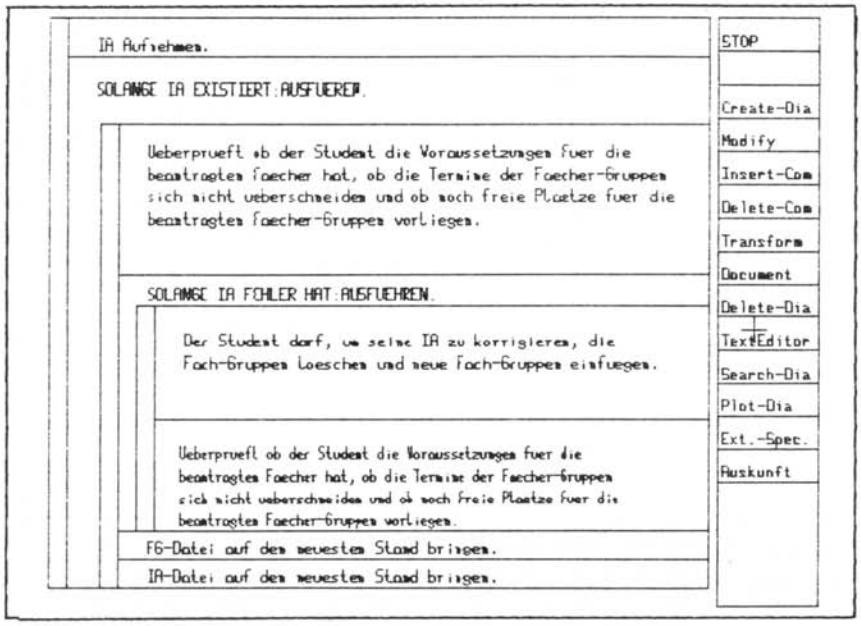
Beispiel 7: Der Benutzer wählt die Funktion 'Copy Diatx' (Kopieren eines Textes aus einem Diagramm) aus.



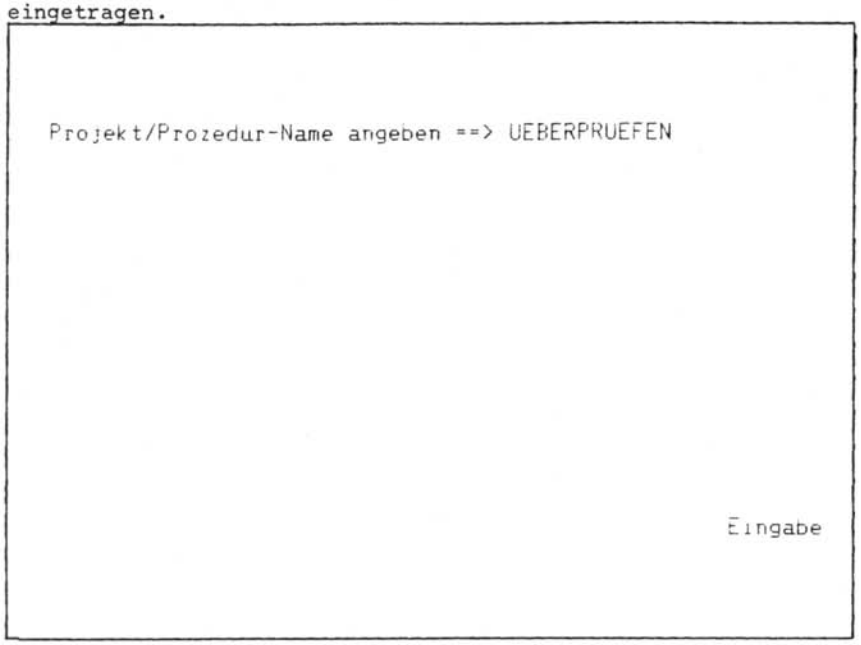
So sieht das Diagramm nach dem Kopieren des Textes aus.



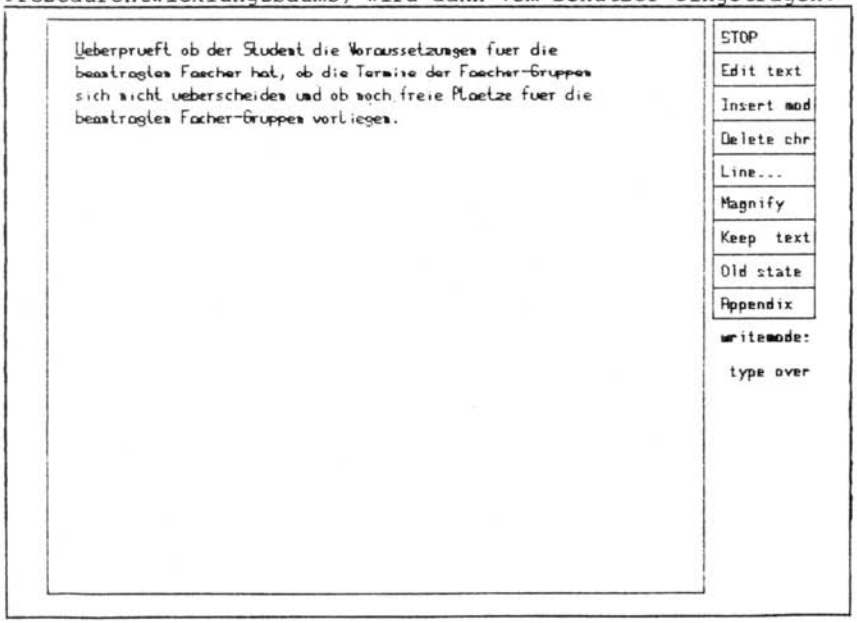
Nach mehreren weiteren Schritten sieht das fertige Diagramm so aus. Der Benutzer stellt fest, daß zwei Aktivitäten gleich sind und entschiedet, daß daraus ein Prozedurentwicklungsbaum spezifiziert werden kann.



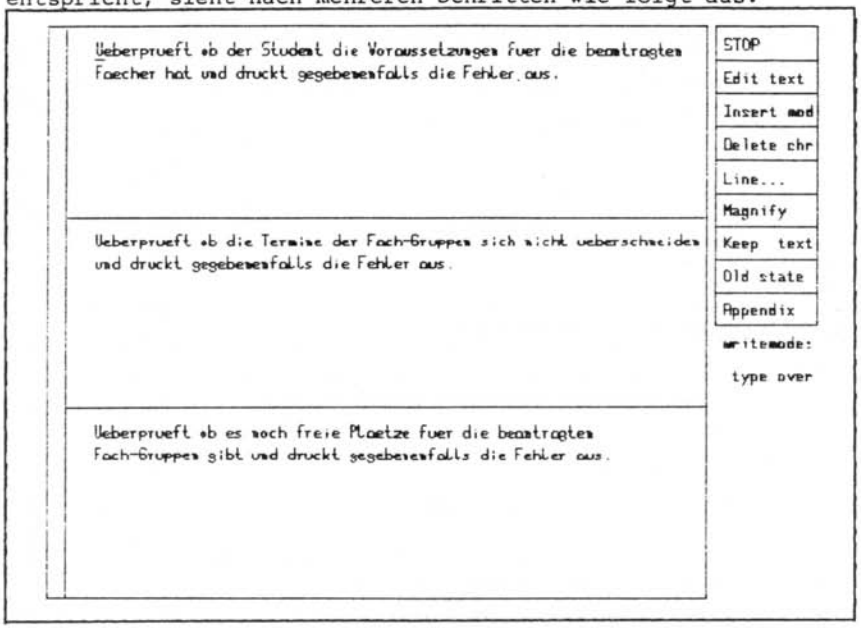
Beispiel 8: Der vom Benutzer angegebene Name des Prozedurentwicklungsbaums wird vom System in die 'Library' des Benutzers eingetragen.



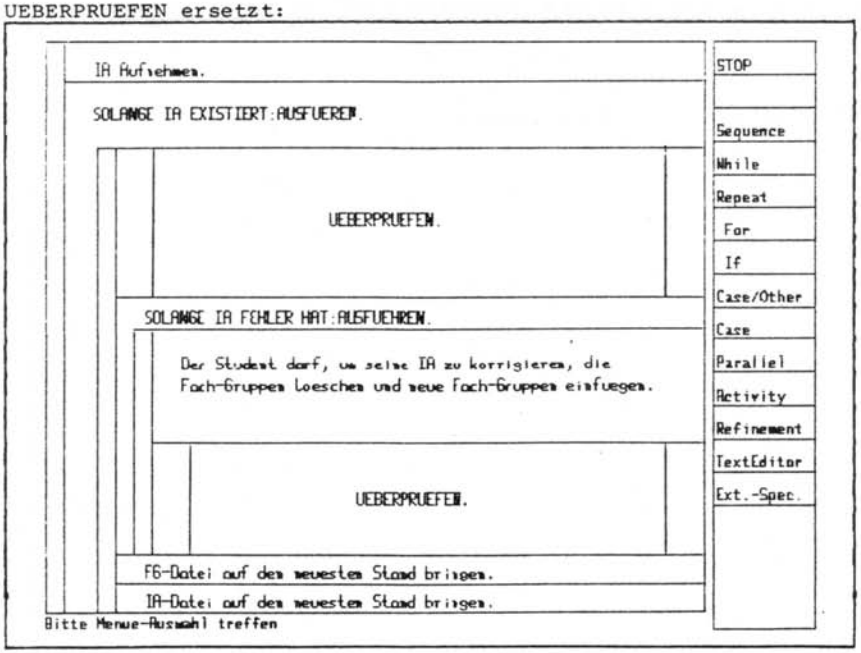
Beispiel 9: Die Anforderungsbeschreibung (Wurzel des Prozedurentwicklungsbaums) wird dann vom Benutzer eingetragen.



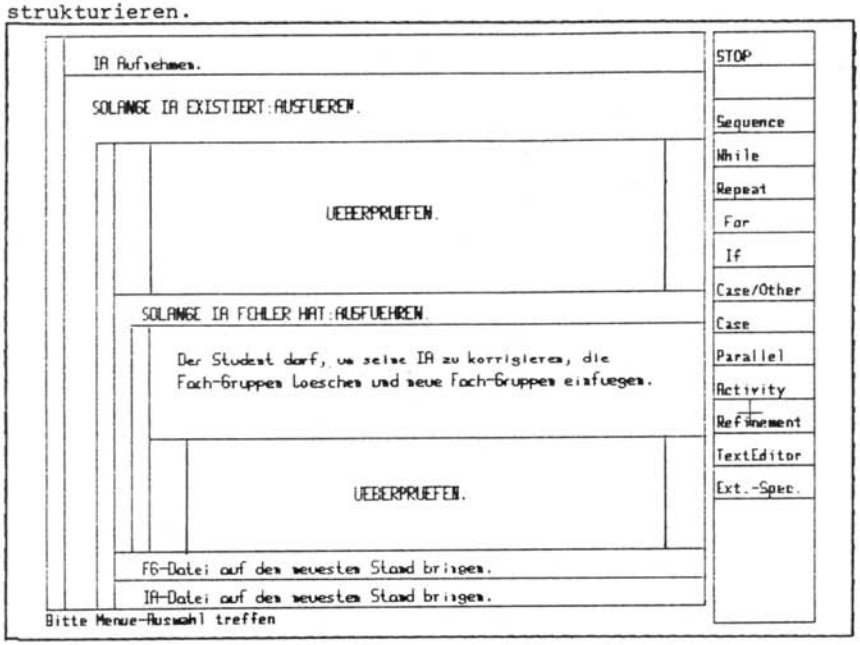
Das Diagramm, das der Anforderungsbeschreibung der Prozedur entspricht, sieht nach mehreren Schritten wie folgt aus.



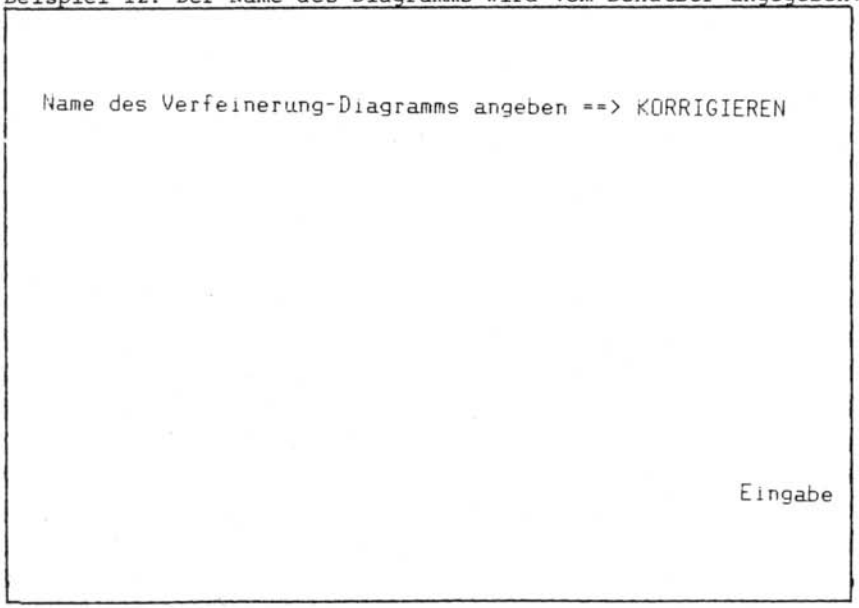
Beispiel 10: Die Aktivitäten werden durch die Prozedur UEBERPRUEFEN ersetzt:



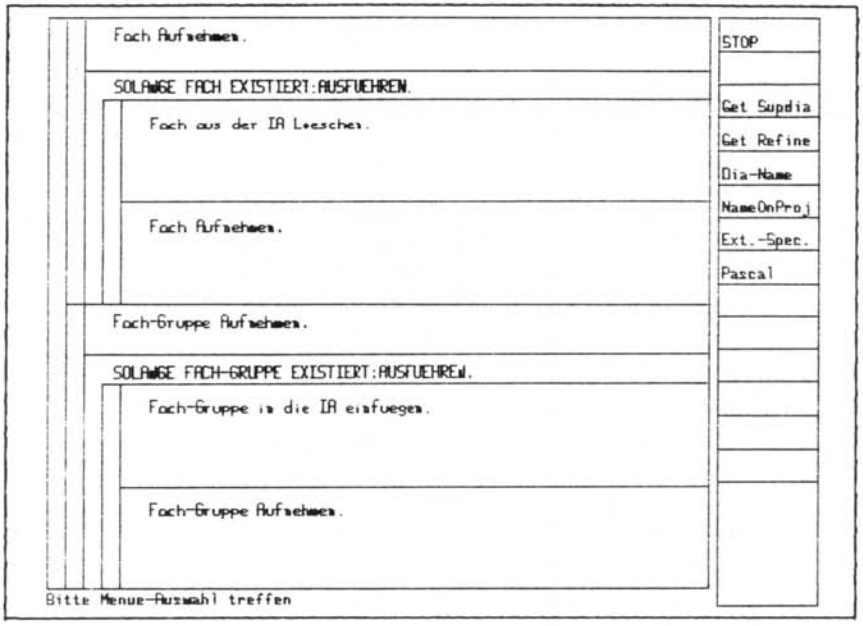
Beispiel 11: Der Benutzer wählt die Funktion 'Refinement' (Verfeinerung) aus, um eine Aktivität (Der Student ...) zu strukturieren.



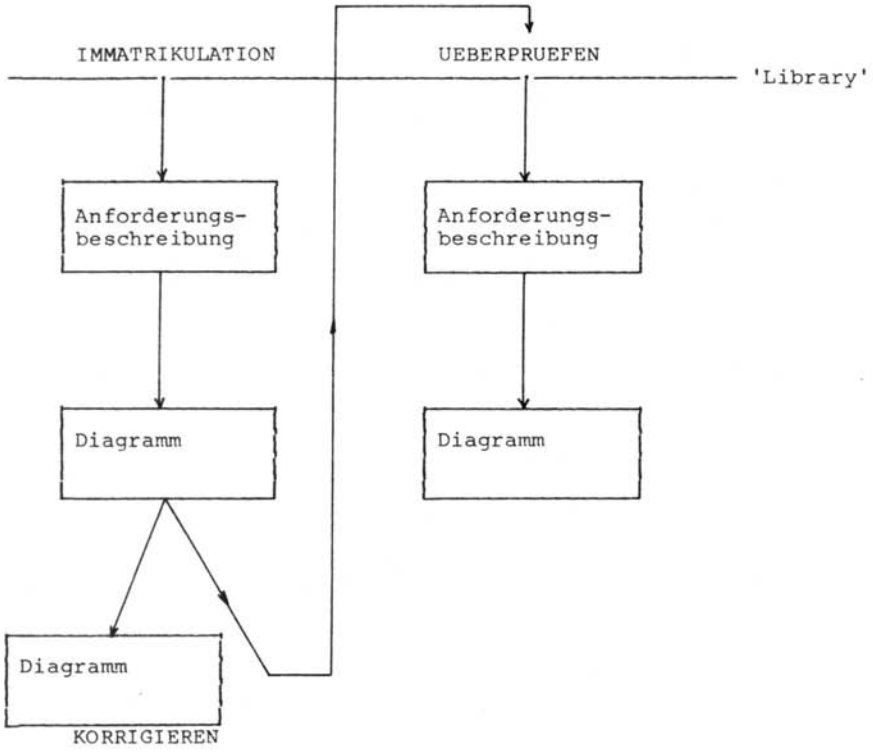
Beispiel 12: Der Name des Diagramms wird vom Benutzer angegeben.



Das entsprechende Diagramm sieht schließlich wie folgt aus.



Der geschilderte Dialog stellt sich in einer Übersicht wie folgt dar:



LITERATURVERZEICHNIS

/Bjo-78/ D. Bjorner und C.B. Jones(eds),'The Vienna Development Method: The Meta Language', Lecture Notes in Computer Science, Vol. 61, Springer Verlag, 1978.

/Boe-76/ B.W. Boehm, 'Software Engineering', IEEE Trans on Computer, Vol. C-25, Nr. 12, 1976.

/Boe-77/ B.W. Boehm, 'Software requirements and design aids', in Software Reliability, Vol. 2, Infotech State of the Art, 1877.

/Boe-66/ C.Böhm und G. Jacopini, 'Flow Diagrams, Turing Machines and Languages with only two Formation Rules', Commuminications of the ACM, Vol. 9, Nr. 5, 1966.

/Clo-81/ W.F. Clocksin und C.S. Mellisch, 'Programming in Prolog', Springer-Verlag Berlin New York, 1981.

/Cai-75/ S.H. Caine und E.K.Gordon, 'PDL-A tool for Software design', National Computer Conference, 1975, AFIPS Proc pp. 271-276.

/Con-78/ L.L Constantine, 'Structured Design', 2nd ed. New York: Yordon inc., 1978.

/Der-78/ N.Dershowitz und Zohar Manna, 'Inference Rules for Program Annotation', Proc of 3rd International Conference on Softwate Engineering, 1978.

/Dij68a/ E.W. Dijkstra, 'Go to Satement considered Harmful', Communications of the ACM, 11(3), 147-148. 1968.

/Dij68b/ E.W. Dijkstra, 'A constructive Aproach to the Problem of Program Correctness', BIT,8(3), 147-186.1968

/Dij-76/ E.W. Dijkstra, 'Programming Methodologies: Their objectives and their nature',in Structured Programming, Infotech State of the Art,1976.

/Die-83/ A. Dietrich und alii, 'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen:Prozedurale Schnittstelle', Softwarepraktikum 2, WS 1982/1983.

/Dic-78/ M.E.Dickovel et alii, 'Software Design using SADT', in Stuctured Analysis and Design, Infotech State of the Art,1978.

/Flo-67/ R.W. Floyd, 'Assigning Meanings to Programs', in J.T. Schwartz, Ed., Mathematical Aspects of Computer Science. Providence, R.I.: American Mathematical Society, 1967,pp. 19-32.

/Fre-78/ P. Freeman, 'Software life-cycle', in Structured Analysis and Design, Vol. 1, pp. 20-32, Infotech State of the Art, 1978.

/Flo-84/ C. Floyd, 'Eine Untersuchung von Software-Entwicklungs-methoden', Berichte des German Chapter of the ACM,1984.

/Gri-71/ D.Gries, 'Compiler Construction for Digital Computers', John Wiley & Sons, Inc. 1971.

/Gen-81/ H.J. Genrich und K. Lautenbach, 'System Modelling with High-Level Petri Netz', Theoretical Computer Science 13(1981),pp.109-136, North Holland Publisching Company.

/Gut-78/ J.V. Guttag und E. Horowitz, 'The design of data type specifications', In Current Trends in Programming Methodology IV, R.T. Yeh(ed.),pp.60-79,1978.

/Gai-84/ F. Gaisser, 'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen:Interaktive Programmerstellung' Studienarbeit, Universität Stuttgart, Institut für Informatik. 1984.

/IBM-74/ ------, 'HIPO-A Design Aid and Documentation Technique' IBM GC20-1851-1, IBM Corp., White Plains,1974.

/Jac-75/ M.A. Jackson, 'Principles of Program Design', Academic Press Inc. (London) LTD, 1975.

/Jac-76/ M.A. Jackson, 'Constructive Methods of Program Design', Proc of the First Conference of The European Cooperation in Informatics, 1976, Lecture Notes in Computer Science, Nr. 44, Springer-Verlag, pp. 236-262.

/Jac-83/ M.A. Jackson. 'System Development', Prentice-Hall International Series in Computer Science, 1983.

/Knu-68/ D.E. Knuth, 'The Art of Computer Programming', Vol. 1, Fundamental Algorithms. Reading, Mass.: Addison-Wesley, 1968.

/Kim-79/ R. Kimm et alii, 'Einführung in Software Engineering', Walter de Gruyter,Berlin-New York, 1979.

/Kre-77/ H.Kreibohn, 'Ein graphisches Programmierlabor zum schrittweisen Verfeinern von Flußdiagrammen', Dissertation, Universität Stuttgart, 1977.

/kre-83/ K-D. Kreplin, A. Schmidt, K.W. Wirtz, 'Erfahrungen beim Entwurf der Benutzerschnittstelle des mbp-tool-system unter Verwendung von Simulation und Prototyping', Software-Ergonomie, Tagung des German Chapter of the ACM in Nürnberg, 1983

/Lun-79/ M. Lundeberg, G. Godkuhl und A. Nilsson, 'A Systematic Approach to Systems Development', Information Systems 4 Nr. 151-12, 1979.

/Lew-77/ T.G.Lewis,'Blueprint Languages and chief architect concept', in Software Engineering Techniques, V. 2, Infotech International, 1977.

/Mye-75/ G.J.Myers, 'Reliable Software through Composite Design' Van Nostrand Reinhold Company, 1975.

/Mye-76/ G.J.Myers, 'Software Reliability, Priciples and Practices', A Wiley-Interscience Publication, 1976.

/McC-78/ C.L. McClure,'A Model for Program Complexity Analysis', Proc of 3rd Conference on Software Engineering, 1978.

/Mar-78/ T.Marco,'Structured Analysis and System Specification', Yourdon inc., 1978.

/Mek-80/ L.J.Mekly und S.S. Yau, 'Software Design Representation Using Abstract Process Networks', IEEE Trans on Software Engineering, Vol.SE-6, Nr. 5, 1980.

/Mat-82/ Y.Matsumoto, T. Tanaka und S. Kawakita, 'Specifications Transformations and A Requirements Specification of Real-Time Control', International Symposium on Current Issues of Requirements Engineering Enviromments, Ohmsha LTD,1982.

/Mar-82/ D.Marca und C. McGowan,'Static and Dynamic Data Modeling for Information System Design', CH1975-4/82/ 0000/0137§00.75, IEEE,1982.

/Mor-83/ O. Morar,'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen:Diagrammkorrektur durch Einfuegen und Loeschen', Studienarbeit 309, Universität Stuttgart, Institut für Informatik, 1983.

/Nun-82/ D.J.Nunes und U.Pletat, 'Ein CAD-System zur Software-Entwicklung basierend auf NS-Diagrammen', Universität Stuttgart, Institut für Informatik, 1982.

/Nun-85/ D.J.Nunes und R. Gunzenhäuser, 'Die SADt-Methode für die Software-Entwicklung', Erscheint in LOG-IN, 1985.

/Nau-66/ P. Naur 'Proof of Algorithmus by General Snapshots', BIT, 6(4),pp. 310-316, 1966.

/Nas-73/ I. Nassi und B. Shneiderman,'Flowchart Techniques for Structured Programming', SIGPLAN Notices, 1973.

/Neu-80/ E.J.Neuhold et alii,'Wie man mit Theoretischen Methoden in der EDV-Praxis Rationeller Arbeit', Angewandte Computertechnik für Informationssysteme GmbH, 1980.

/Neu-81/ E.J.Neuhold, 'Vergleichende Analyse von Software-Entwurfsmethoden', IHS-Jounal, Vol 5, pp.237-259, Physica Verlag, Wien, 1981.

/Neu-83/ E.J. Neuhold, 'Development Methodologies for Event and Message based Application Systems', Universität Stuttgart, Institut für Informatik, 1983.

/Pet-77/ J.L.Peterson, 'Petri Netz', ACM Computing Surveys, V.9, Nr. 3, pp. 223-252, 1977.

/Par-72/ D.L.Parna, 'On the criteria to be used in decomposition systems into Modules', CACM, Vol.15, Nr. 12, 1972.

/Pre-83/ T. Preuhs, ' Ein CAD-System zum Softwareentwurf mit Hilfe von N-S Diagrammen: Ein Editor für NS-Diagramme', Studienarbeit, Universtität Stuttgart, Institut für Informatik, 1983.

/Ram-83/ H.R.Ramsey, M.E.Atwood und R.U. Doven, 'Flowcharts Versus Programm Design Languages, An Experimental Comparation',Communications of the ACM,Vol. 26, Nr.6,1983

/Ros-67/ S.Rosen, 'Programming Systems & Languages', McGraw-Hill Book Company, 1966.

/Ros77a/ D.T.Ross und K.E. Schoman, 'Stuctured analysis for requirements definition', IEEE Tran on Software Enginneering, Vol. SE-3, Nr. 1, 1977.

/Ros77b/ D.T.Ross, 'Structured Analysis(SA): A Language for Communicating Ideas', IEEE Transactions on Software Engineering, Vol. SE-3, Nr. 1, 1977.

/Rap-82/ L-S. W. Rapperswil, 'Struktogramme als Programmiersprache', Angewandte Informatik 3/82. Friedr. Vieweg & Sohn Verlagsgesellschaft mbh. 1982.

/Ren-83/ I.Renz, 'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen: Die Diagrammerzeugung und das Verfeinerungskonzept', Diplomarbeit 235, Universtät Stuttgart, Institut für Informatik, 1983.

/Ric-82/ G. Richter und R. Durchholz, 'Information Systems Design Methodologies', North-Holland Publiching Company 1982.

/Ste-75/ T.B.Steel, 'Program chairm's introduction', Proc of lst IEEE Nat Conf. on Software Engineering, 1975.

/Som-84/ D.Sommer, 'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen:Die Transformation von Diagrammen', Studienarbeit 348, Universtität Stuttgart, Institut für Informatik, 1984.

/Sch-83/ T. Schacherer, 'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen: Änderung der graphischen Darstellung von Diagrammen', Studienarbeit 309, Universität Stuttgart, Institut für Informatik, 1983.

/Str-84/ M. Strobel, 'Ein CAD-System zum Softwareentwurf mit Hilfe von NS-Diagrammen: Realisierung eines Dokumentationsmoduls', Studienarbeit 370, Universität Stuttgart, Institut für Informatik, 1984.

/Tei-77/ P. Teichroew und E.A. Hersley III, 'PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems', IEEE Transation on Software Engineering, Vol. SE-3, Nr. 1, 1977.

/Tho-78/ M. Thomas, 'Functional Decomposition:SADT', in Structured Analysis and Design, Infotech State of the Art, 1978.

/Was-77/ I. Wasserman,'Aproaches to Software Design', Software Engineering Techniques, pp. 103-106, Infotech International Lmd. 1977.

/Wir-71/ N.Wirth, 'Program Development by Step-Wise Refinement', Communications of ACM, 14(4), pp. 221-227, 1971.

/Wir-74/ N.Wirth, 'On the Composition of Well-Structured Programs', Computing Surveys, 6(4), pp. 247-259, 1974.

/Wac-84/ G-V. Wachter, 'Data-Dictionary zur Beschreibung von Datei und Programmstruktur', Diplomarbeit 239, Universität Stuttgart, Institut für Informatik, 1984.

/You-82/ S.J. Young, 'Real Time Languages', Ellis Horwood Publishers-Computer and their Application, 1982.

/Zav-84/ P. Zave, 'The Operational versus the conventional Aproach to Software Development',Communications of ACM, Vol 27, Nr. 2, 1984.

/Zin-84/ G.D. Zinke, 'CAS System LITOR', Konzept und Realisierung einer Arbeitsumgebung für den iterativen, graphisch unterstützten Softwareentwurf', Berichte des German Chapter of the ACM, 18, 1984.

Lebenslauf

geboren: 28. November 1943 in Orleaes-Brasilien
Staatsangehörigkeit: brasilianisch

Schulbildung:

1951-1955	Grundschule in Rio Fortuna, Brasilien
1956-1964	Gymnasium in Tubarao, Brasilien
1964	Abitur in Porto Alegre, Brasilien

Studium:

1965-1969	Studium der Elektrotechnik, Schwerpunkt Eletronik an der 'Universidade Federal do Rio Grande do Sul (UFRGS)'
1971-1972	M.Sc. in Informatik an der Kath. Universität von Rio de Janeiro (PUC/RIO)
1981-1985	Doktorand am Institut für Informatik der Universität Stuttgart

Berufsausübung:

1968-1969	Programmierer am Rechenzentrum der UFRGS
1970	System-Analytiker am Rechenzentrum der UFRGS
1971-1972	'Professor Auxiliar de Ensino' an der UFRGS
1973-1978	'Professor Assistente' an der UFRGS
1979-1985	'Professor Adjunto' an der UFRGS