Vorlesungsskripte INFORMATIK

Software Engineering
Prof. Dr. Jürgen Ebert

Judith Joch Christoph Albrecht Bernhard Stüber

8. Dezember 1994

Inhaltsverzeichnis

Teil I

GRUNDLAGEN

Kapitel 1

Grundbegriffe

Definitionen

Software-Engineering (= ingenieurmäßige Erstellung von Software) ist dasjenige Teilgebiet der Informatik, das sich mit der Erstellung und Nutzung von Software (insbesondere großer Softwaresysteme im industriellen Rahmen) befaßt.

Software ist eine Menge von Programmen zusammen mit den begleitenden Dokumenten, die für ihre Nutzung notwendig oder hilfreich sind.

Feststellung

Zur Software gehören viele einzelne Dokumente (artifacts), z.B. Programmtext, Benutzerhandbuch, Entwurfsbeschreibung, Objektmodul und vieles andere mehr.

Übersicht

P. Freeman unterscheidet verschiedene Wissenschaftsparadigmen :

- mathematisches Paradigma
  1. Schritt: Modellklasse definieren
  2. Schritt: Sätze formulieren
  3. Schritt: Diese Sätze beweisen
  4. Schritt: Die bewiesenen Ergebnisse interpretieren
- naturwissenschaftliches Paradigma
  1. Schritt: Eine Hypothese aufstellen
  2. Schritt: Ein Modell erstellen und gewisse Vorhersagen wagen

3. Schritt: Ein Experiment entwerfen und Daten bei der Ausführung des Experimentes sammeln
4. Schritt: Ergebnisse des Experiments analysieren

- ingeneurwissenschaftliches Paradigma

1. Schritt: Anforderungen feststellen
2. Schritt: Ein Modell spezifizieren
3. Schritt: Das Modell entwerfen und anschließend implementieren
4. Schritt: Das Modell prüfen

Anmerkung

Software–Engineering folgt dem ingenieurwissenschaftlichem Paradigma.

1.1 Historie

Name

Der Begriff “Software–Engineering“ entstand 1968/69 bei zwei NATO–Tagungen in Garmisch–Partenkirchen, die diesen Namen trugen.

Vereinigungen

Als Vereinigungen im Bereich “Software–Engineering“ sind zu nennen:

- ACM darin die Gruppe SIGSOFT (=Special Interest Group on SOFTware–Engineering)
- GI darin die Fachgruppe “Software–Engineering“

Zeitschriften

Zeitschriften die sich speziell mit Software–Engineering befassen:

- von SIGSOFT: “Software Engineering Notes“
- von IEEE : “Transactions on Software Engineering“, “Software“
- von GI : “Softwaretechnik–Trends“

1.2 Beteiligte

Kontext

Ein **Auftraggeber** beauftragt einen **Hersteller** mit der Erstellung eines Software–Systems.

Skizze

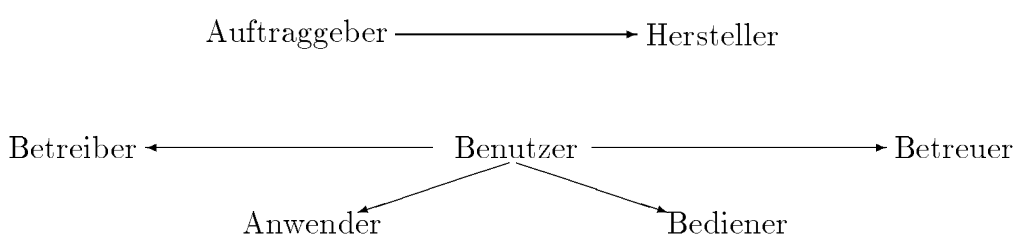


Abbildung 1.1: Beteiligte am Software–System

Anmerkung

Wir beschäftigen uns hier im Wesentlichen mit den Problemen des Herstellers.

1.3 Produktion

Feststellung

Der Produktionsprozeß von Software unterscheidet sich in zwei wesentlichen Punkte von anderen technischen Prozessen:

1. Der Entwurf ist der eigentliche Produktionsprozeß
2. Die Produktion besteht aus einer reinen Vervielfältigung

Definitionen

Ein **Projekt** ist die zeitlich begrenzte, einmalige Verfolgung eines vorgegebenen Zieles, die zu einem definierten Ergebnis führen soll.

Ein **Produkt** ist ein in sich abgeschlossenes (im allgemeinen für einen Auftraggeber bestimmtes) Ergebnis eines erfolgreich durchgeführten Projektes.

Ein **System** ist dasjenige Produkt, das die vorgegebenen Ziele eines Projektes in ihrer Gesamtheit realisiert.

Feststellung

- Ein Hersteller erstellt ein Software–System im Rahmen eines Projektes
- Ein System besteht i.a. aus vielen Dokumenten

Feststellung

Man kann die Produktion von Software entweder mehr **prozeßorientiert** oder mehr **produktorientiert** betrachten.

Definition

Ein **Baustein** ist ein abgeschlossener, begrifflich zusammengehöriger Teil eines Systems.

Feststellung

Systeme bestehen im allgemeinen aus mehreren Bausteinen. Bausteine können in unterschiedlicher Form vorliegen, z.B. definiert, implementiert, getestet usw.. Zu einem Baustein kann es mehrere Dokumente geben.

Kapitel 2

Phasenmodell

Problem

- Gegeben sei eine beliebige Problemstellung.
- Gesucht sei ein System, welches das Problem löst.

2.1 Modellübersicht

Anmerkung

Das nachfolgende Phasenmodell ist eines von mehreren möglichen. Es soll vor allem als Rahmen für diese Vorlesung dienen und weniger als obligatorisches Modell.

Phasenmodell (Software–Lebens–Zyklus, Software Life Cycle)

Entwicklung – Verwendung

↓	Problem–/Aufgabenstellung
analysieren	
↓	Problemanalyse (=Istanalyse, Studie)
definieren	
↓	Anforderungsdefinition (=Pflichtenheft, Lastenheft)
entwerfen und spezifizieren	
↓	Entwurf und Spezifikation
implementieren und integrieren	
↓	fertiges System
installieren	
↓	System in seiner Zielumgebung
betreiben und betreuen	

Anmerkung

- Es gibt viele Life-Cycle-Modelle
- dient uns nur als Grundlage der Diskussion
- Phasen können evtl. verschränkt sein

Feststellung

Jede dieser Phasen besitzt eine Feinstruktur. Zu jedem Phasenmodell gehören auch phasenbegleitende Tätigkeiten:

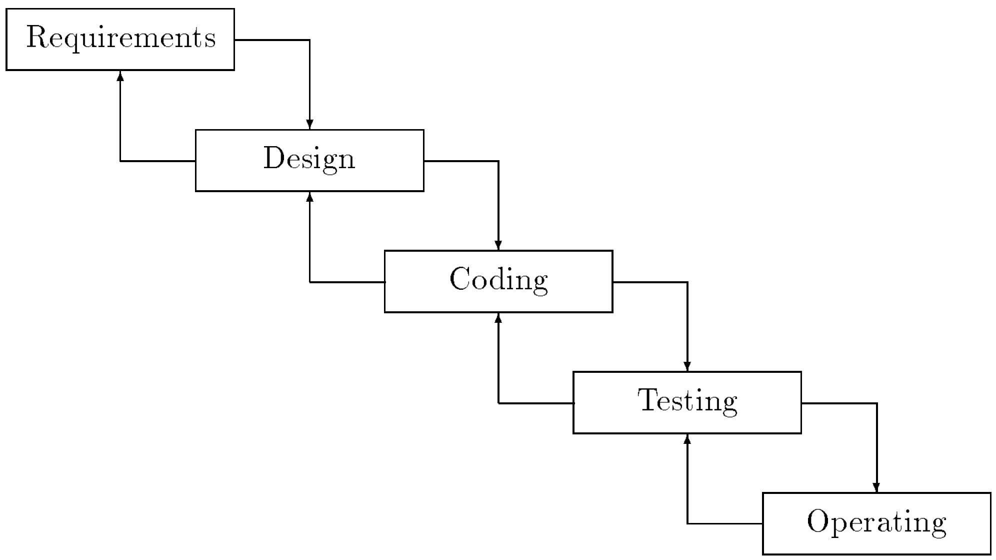
- Phasenfeinstruktur:
  - Planen
  - Realisieren (Durchführen)
  - Dokumentieren
  - Qualität sichern
- Phasenbegleitendes:
  - Planen und Überwachen
  - Dokumentieren
  - Qualität sichern

Anmerkung

Phasenmodelle sind **logische** Modelle und **nicht unbedingt zeitliche** Modelle, obwohl sie zeitlich interpretiert werden können. In diesem Fall bilden die Zwischenprodukte **Meilensteine**, die zur Fortschrittskontrolle verwendet werden können.

Ergänzung

Natürlich sind weitere Varianten von Phasenmodellen denkbar, wie etwa das folgende aus der Literatur entnommene:



Aufgrund dieser Abbildung spricht man im Zusammenhang mit Phasenmodellen auch oft von sog. **Wasserfallmodellen**.

2.2 Tätigkeiten

Anmerkung

Im folgenden werden die Einzeltätigkeiten im Software-Phasenmodell im einzelnen beschrieben.

Definitionen

Analysieren ist die Tätigkeit, die auf das Verstehen und eine Erfassung der in einer Aufgabenstellung angesprochenen Sachverhalte, sowie auf eine Entscheidung über die Durchführbarkeit und die Erfolgsaussichten eines künftigen Projektes abzielt.

Definieren ist die Tätigkeit, die darauf gerichtet ist, Anforderungen an ein zur Lösung einer Aufgabenstellung erforderliches Produkt zu stellen.

Entwerfen (im engeren Sinne) ist das Konkretisieren eines Bausteins durch weitere Bausteine und deren Beziehungen.

Spezifizieren ist das Präzisieren von Anforderungen, die an einen Baustein für dessen Benutzung gestellt werden.

Implementieren umfaßt Entwerfen (im weiteren Sinne), Programmieren und Validieren eines (im allgemeinen nicht weiter zerlegten) Bausteins.

Integrieren umfaßt Montieren und gemeinsames Validieren von implementierten und/oder bereits integrierten Bausteinen.

Installieren ist das Einrichten eines Systems in dessen Zielumgebung zum Zwecke des Betriebs.

Betreiben (=Nutzen) ist die Verwendung eines Systems in seiner Zielumgebung.

Betreuen umfaßt Pflege und Wartung.

Plegen ist das Anpassen und Weiterentwickeln von Produkten an neue Anforderungen.

Warten ist das Beheben von Fehlern in Produkten nach der Installation.

Definitionen

Qualität sichern ist die Zusammenfassung aller geplanten und systematisch durchgeführten Tätigkeiten, die dazu geeignet sind, ein angemessenes Vertrauen zu erreichen, daß etwas (z.B. Produkt, Baustein, Herstellungsprozeß etc.) vorgegebenen Anforderungen genügt.

Dokumentieren ist das schriftliche Festhalten des Ergebnisses einer Tätigkeit.

Management umfaßt das Planen, Führen und Überwachen von Abläufen.

Aufwandsverteilung der Kosten (nach Zellkowitz, 1978):

- 11% für das Analysieren und Definieren
- 5% für das Entwerfen und Spezifizieren
- 15% für das Implementieren
- 7% für das Integrieren und Installieren
- 62% für Betreibung und Betreuung

Aus dieser Aufwandsverteilung folgt, daß 2/3 aller Kosten für die letzte Phase in unserem Phasenmodell gebraucht werden. Deswegen bezeichnet man diese Kostenaufteilung auch 1/3 **:** 2/3–Regel.

Anmerkung

Da diese Aufwandsverteilung über zehn Jahre alt ist, kann heute wohl eher vermutet werden, daß die Kosten für das Entwerfen und Spezifizieren etwas größer und bei der Betreibung und Betreuung etwas geringer sind.

Konsequenz

Die **Wartbarkeit** des Softwaresystems spielt daher eine große Rolle.

Kapitel 3

Beurteilungskriterien

Anmerkung

In diesem Abschnitt werden die Beurteilungskriterien für Softwaresysteme betrachtet. Beurteilungskriterien sind:

1. Qualität
2. Kosten
3. Zeit

Diese Beurteilungskriterien konkurrieren miteinander. Die Aufgabe der Praxis ist es, zwischen ihnen einen Kompromiß zu finden.

3.1 Qualität

Definitionen

Qualität ist die Gesamtheit von Eigenschaften, die sich auf die Eignung zur Erfüllung vorgegebener Qualitätsanforderungen beziehen.

Qualitätsanforderungen sind Anforderungen an die Ausprägung von Qualitätsmerkmalen.

Definitionen (Qualitätsmerkmale)

Zuverlässigkeit ist der Umfang, in dem von einem System erwartet werden kann, daß es die beabsichtigte Funktion mit der erforderlichen Genauigkeit ausführt. Sie umfaßt Korrektheit, Robustheit und Ausfallsicherheit.

Korrektheit (= funktionale Korrektheit) ist der Umfang, in dem die Systemausgaben die Anforderungen für den gewünschten Gebrauch erfüllen.

Robustheit ist der Umfang, in dem das System bei nicht vorgesehener Verwendung ein sinnvolles und sicheres Verhalten zeigt.

Ausfallsicherheit ist der Umfang, in dem ein System gegen Störungen unempfindlich ist.

Benutzerfreundlichkeit ist der Umfang, in dem ein System den Anforderungen der Benutzer nach einer einfachen, sicheren und leicht erlernbaren Handhabung nachkommt.

Wartbarkeit ist der Umfang, in dem das Lokalisieren und Beheben von Fehlern, sowie das Anpassen an veränderte Anforderungen in einem System erleichtert wird. Sie umfaßt Validierbarkeit, Flexibilität und Verständlichkeit.

Validierbarkeit ist der Umfang, in dem ein System erlaubt festzustellen, ob es die Anforderungen erfüllt.

Flexibilität ist der Umfang, in dem Veränderungen eines Systems erleichtert werden.

Verständlichkeit ist der Umfang, in dem es das System anderen Personen als den Entwicklern erlaubt, seine Struktur, Annahmen und Einschränkungen zu erfahren.

Portabilität ist der Umfang, in dem das Überführen eines Systems in eine andere Hard-/Software-Umgebung erleichtert wird.

Effizienz ist der Umfang, in dem ein System mit Betriebsmitteln sparsam umgeht.

Anmerkung
Die Zuverlässigkeit ist das wichtigste Qualitätsmerkmal.

3.2 Kosten und Zeit

Feststellung
Die Beurteilungskriterien bezüglich Kosten und Zeit gehen in zwei Richtungen. Einmal aus der Sicht des Herstellers und zum anderen aus der Sicht des Kunden.

Kostenaspekte sind:

1. Entwicklungskosten
2. Wartungskosten

3. Einsatzkosten

Zeitaspekte sind:

1. Entwicklungszeit
2. Einsatzdauer

Kapitel 4

Prinzipien

Definitionen

Prinzipien sind Grundsätze, die man seinem Handeln zugrunde legt.

Methoden sind planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen.

Hilfsmittel dienen der Unterstützung konkreter Anwendungen und Methoden.

Anmerkung

Der Schwerpunkt dieser Vorlesung liegt auf der Darstellung von Methoden.

4.1 Hauptprinzipien

Definitionen

Zerlegungsprinzip: Auf jeder Ebene ist das Produkt sinnvoll in Bausteine und deren Beziehungen zu zerlegen.

Abstrahierungsprinzip: Elemente, die einheitliche Eigenschaften haben, sind zu identifizieren, selbst wenn sie an verschiedenen Stellen verwendet werden.

Lokalisierungsprinzip: Zusammengehörige Dinge sind auch physisch zusammen zu bringen.

Prinzip der Trennung der Belange: Dinge, die nichts miteinander zu tun haben, sind auch getrennt unterzubringen.

Prinzip des Verbergen von Informationen: Nur lokal benötigte Details sind nach außen hin unzugänglich zu machen.

Einheitlichkeitsprinzip: Dinge, die ähnlich sind, sind auch ähnlich zu behandeln.

Vollständigkeitsprinzip: Es sind stets alle möglichen Fälle zu erfassen, selbst wenn nur einige explizit benötigt werden.

Nachweisbarkeitsprinzip: Es ist stets nachprüfbar zu machen, ob gestellte Ziele erreicht wurden.

Prinzip der Überraschungsarmut: ("Prinzip der kleinsten Verwunderung"): Das Verhalten des Produktes soll immer möglichst das nächstliegende sein.

Teil II

Architektur

Kapitel 5

Zerlegung

Feststellung

Systeme werden in **Bausteine** und **Beziehungen** zerlegt. Die (graphische oder textuelle) Darstellung dieser Zerlegung heißt **Architektur** des Systems.

Feststellung

Ein Baustein wird idealerweise in nicht mehr als sieben weitere Bausteine zerlegt.

5.1 Beziehungen

Beispiele für Bausteine

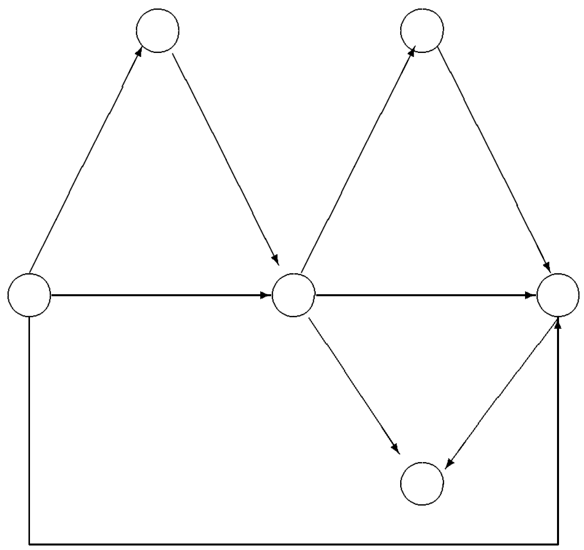
- Prozeduren
- Anweisungsblöcke
- Module, Packages, Pakete

Beispiele für Beziehungen

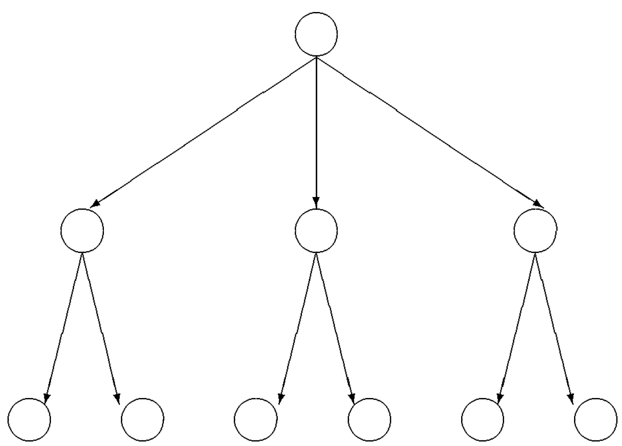
- A benutzt B
- A ruft B auf
- A liefert Daten an B
- A folgt vor B
- A enthält B
- etc.

Beispiele für Architekturgraphen

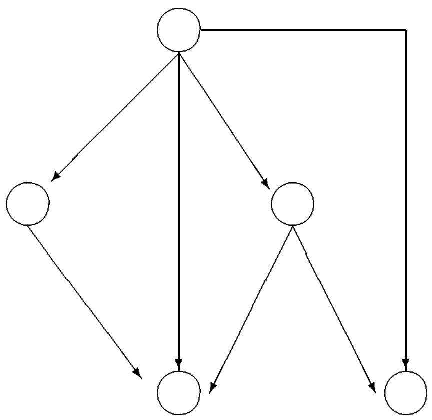
Die Knoten entsprechen Bausteine und die Kanten entsprechen Beziehungen



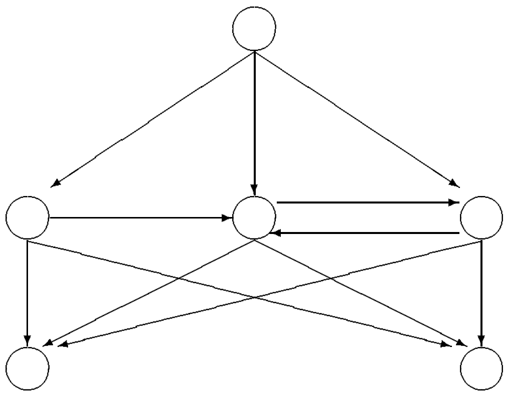
allg. Struktur



baumartige Struktur



azyklische Struktur
(Hierarchie)



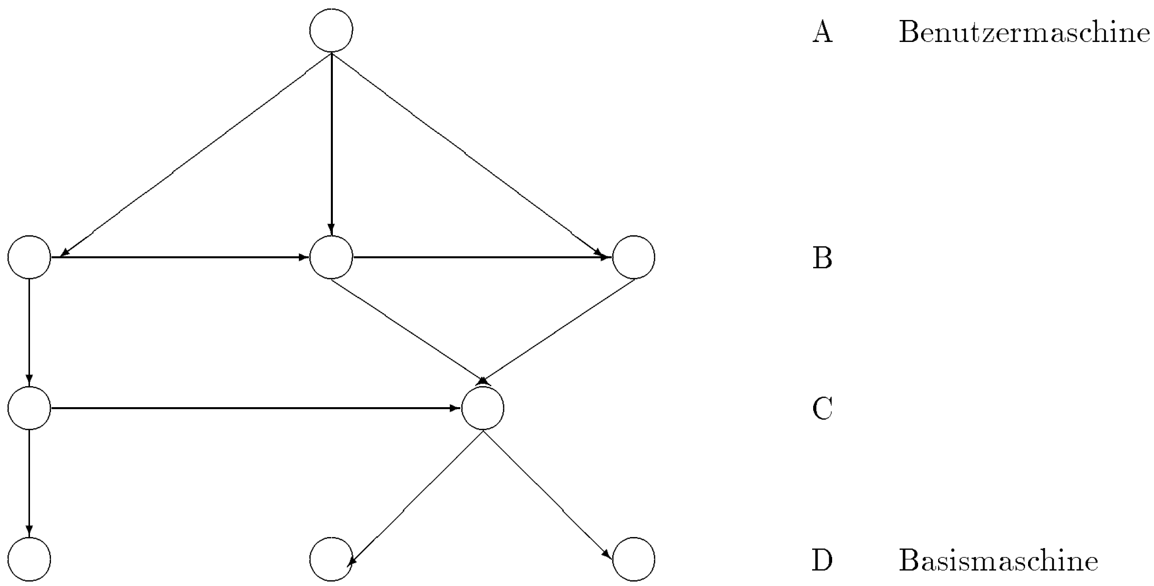
Pseudohierarchie

Feststellung

"Gute Architekturen" (z.B. Pseudohierarchie) lassen eine Ebeneneinteilung erkennen.

Feststellung

Ebenen von Architekturen können als abstrakte Maschinen gesehen werden.



Feststellung

Bei geschichteten Strukturen kann man grundsätzlich

- von oben nach unten (top-down) oder
- von unten nach oben (bottom-up) arbeiten.

5.2 Module

Feststellung

Bausteine werden häufig Module genannt, wenn sie (idealerweise)

- benannt
- getrennt übersetzbar
- logisch abgeschlossen
- physisch zusammenhängend
- deutlich begrenzt
- faßbar und manipulierbar

- für sich allein verstehbar
- von Systemteilen benutzbar
- austauschbar durch ein anderes mit gleicher Schnittstelle

sind.

Feststellungen

- Ein Modul hat eine klare Schnittstelle zu seiner Umgebung.
- Ein Modul ist durch ein anderes mit der gleichen Schnittstelle, aber einer anderen Implementation ersetzbar.
- Die Korrektheit eines Moduls ist unabhängig von seiner Umgebung nachweisbar.

Definition

Die **Schnittstelle** eines Bausteins ist die Menge der Informationen, die er bzw. seine Umgebung benötigen, um miteinander in Beziehung zu treten.

Anmerkung

Eine **Schnittstelle** ist nach *Parnas* die Menge der Annahmen, die man über ein Modul macht.

Feststellung

Man kann die Beschreibung von Modulen aufteilen in:

1. Beschreibung der "Syntax"
2. Beschreibung der "Semantik"

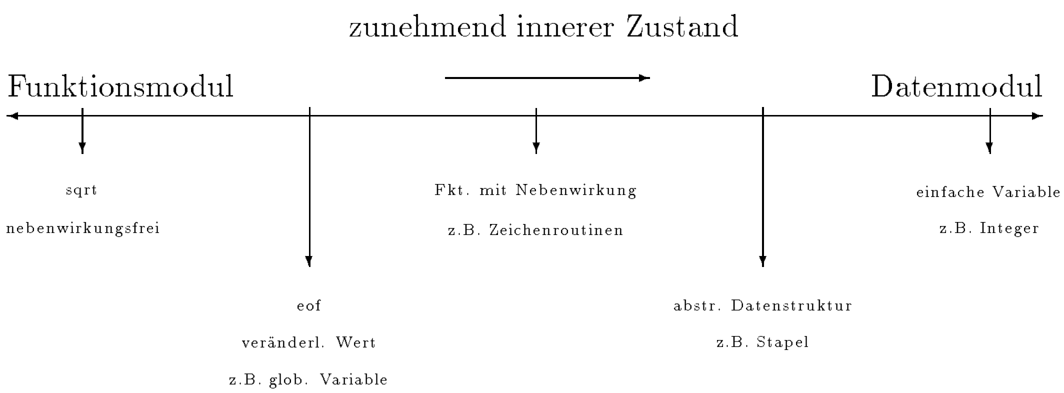
Feststellung

Es gibt folgende Hauptgruppen von Modulen:

- **Funktionsmodul**: Dies ist ein Baustein, der durch sein Ein-/Ausgabeverhalten ohne inneren Zustand charakterisierbar ist, z.B. Sinusfunktion, lineare Optimierung, Statistik.
- **Datenmodul**: Dies ist ein Baustein, der durch mehrere (miteinander zusammenhängende) Operationen mit innerem Zustand charakterisierbar ist, z.B. Datenbank, Stapel, Schlange, Baum.

Skizze

Zwischen den Hauptgruppen gibt es zahlreiche Mischformen.



Feststellung

Die **abstrakten Daten(-strukturen, -typen)** stellen den allgemeinsten Modultyp dar, der je nach Ausprägung des Zustandsanteils, als Funktions- oder Datenmodul aufgefaßt werden kann.

Feststellung

Durch die Verwendung von Modulen werden mehrere wichtige Prinzipien unterstützt:

- Zerlegungsprinzip
- Geheimnisprinzip
- Abstraktionsprinzip

5.3 Abstrakte Daten

Definitionen

Eine **abstrakte Datenstruktur** ist ein Objekt, das vollständig durch die hierauf verfügbaren Operationen charakterisiert ist.

Ein **abstrakter Datentyp** ist eine Klasse von Objekten, die vollständig durch die hierauf verfügbaren Operationen charakterisiert sind.

Anmerkung

Häufig werden in mancher Literatur “abstrakte Datentypen” im engeren Sinne nur für algebraische spezifizierte Datentypen verwendet (siehe Kapitel 7).

Beispiel (Stapel)

init	schafft einen leeren Stapel
isEmpty	testet, ob der Stapel leer ist
push	legt Element auf Stapel
pop	entfernt Element vom Stapel
top	liefert oberstes Element
ungenau bzgl. Syntax	*ungenau bzgl. Semantik*

Ungenau bzgl. Syntax, weil z.B. *pop* entweder das entfernte Element als Ergebnis mitliefern kann oder nicht.

Ungenau bzgl. Semantik, weil z.B. implizit vorausgesetzt wurde, daß man weiß, was ein Stapel ist.

Syntaxbeschreibung in MODULA–2:

1. Stapel als **abstrakte Datenstruktur**:

   definition module $STACK$;
       procedure $init(\ )$;
       procedure $isEmpty(\ ) : BOOLEAN$;
       procedure $push(x : INTEGER)$;
       procedure $pop(\ )$;
       procedure $top(\ ) : INTEGER$
   end $STACK$.

2. Stapel als **abstrakter Datentyp**:

   definition module $STACK$;
       type $stack$ = **record** ... **end**; (Das Prinzip des Verbergens von Informationen wird hier verletzt.)
       procedure $init($**var**$s : stack)$;
       procedure $isEmpty(s : stack) : BOOLEAN$;
       procedure $push(s : stack; x : INTEGER)$;
       procedure $pop(s : stack)$;
       procedure $top(s : stack) : INTEGER$
   end $STACK$.

Anmerkung

Stapel als abstrakter Datentyp in PASCAL:

program ...
const ...
  $nmax = 100;$
type ...
  $stack =$ **array** $[1 \ldots 100]$ **of** ...
  ...
var ...
  $s1, s2 : stack;$
procedure $push(\ldots)$
  ...
begin
end. $(*\ Programm\ *)$

In PASCAL wird das Lokalitätsprinzip verletzt, weil die Datentypbeschreibung auf die **const**–, **type**–, **var**– und **proc**–Klauseln verteilt werden muß.

Feststellung

Essentiell sind bei der Syntax die Angaben der

- Typen
- Operationen und
- deren Ein-/Ausgabeparameter.

Beispiel (mathematische Notation)

Typen (= **sorts**): *stack*

Operationen (= **opns**): $init : stack$ (= Konstante);
$isEmpty : stack \to BOOLEAN;$
$push : stack \times INTEGER \to stack;$
$pop : stack \to stack;$
$top : stack \to INTEGER;$

Kapitel 6

Architekturbeschreibung

Dieses Kapitel behandelt die Sprache **EML** (= elementary module language), mit der man die Möglichkeit hat, die Struktur von Software-Systemen zu bescreiben.

6.1 Signaturen

Definition

Eine **Signatur** $\Sigma = (S, B, c)$ besteht aus:

1. einer Menge $S \neq \emptyset$ von Sortenbezeichnern,
2. einer Menge B von Operationsbezeichnern und
3. einer Abbildung $c : B \to C(T)$.

mit $S \subset T$ und $T \cap B = \emptyset$.

Dabei ist für eine Sorten(-bezeichner)menge T $C(T)$, die **Menge der Charakteristiken** über T, rekursiv definiert gemäß:

1. $a \in T \Rightarrow a \in C(T)$.
2. $a_1, \ldots, a_n \in C(T) \Rightarrow (a_1 \times \ldots \times a_n) \in C(T)$.
3. $a, b \in C(T) \Rightarrow (a \to b) \in C(T)$.

Anmerkung

- $C(T) \subset \left(T \cup \{\text{“(”}, \text{“)”}, \text{“}\times\text{”}, \text{“} \to \text{”}\}\right)^*$, d.h. Charakteristiken sind nur Symbolfolgen!

- Zur Einsparung von Klammern erhält "$\times$" eine höhere Priorität als "$\rightarrow$"

Bemerkung

- Mit $sort(c)$ wird die Menge der Sorten bezeichnet, die in den Charakteristiken des Bildbereichs von c vorkommen.
- Bezeichner in $sort(c) \setminus S$ heißen **freie Bezeichner**.

Bem.

Beispiel

$S = \{stack\}$

$B = \{init, isEmpty, push, pop, top\}$

c	$init$	$isEmpty$	$push$
	$stack$	$stack \rightarrow bool$	$stack \times int \rightarrow stack$

pop	top
$stack \rightarrow stack$	$stack \rightarrow int$

$sort(c) = \{stack, bool, int\}$ bool und int sind freie Bezeichner.

Anmerkung

Charakteristiken der Art

$$\Big((a \rightarrow b) \times (b \rightarrow c)\Big) \rightarrow (a \rightarrow c)$$

sind in von Neumann–Sprachen nicht möglich.
Eine "eingeschränkte" Signatur enthält nur Charakteristiken der Gestalt

$$a_1 \times \ldots \times a_n \rightarrow a_{n+1}.$$

6.2 Umbenennungen

Definition

Ist $\Sigma = (S, B, c)$ eine Signatur und ist $h : S \cup B \cup sort(c) \rightarrow X$ bijektiv, so ist $\Sigma' = (S', B', c')$ mit

1. $S' = h(S)$,
2. $B' = h(B)$ und
3. $c'\Big(h(f)\Big) = \tilde{h}\Big(c(f)\Big)$

mit

1. $\tilde{h}(a) = h(a)$,
2. $\tilde{h}\Big((a_1, \ldots, a_n)\Big) = \Big(\tilde{h}(a_1) \times \ldots \times \tilde{h}(a_n)\Big)$ und
3. $\tilde{h}\Big((a \rightarrow b)\Big) = \Big(\tilde{h}(a) \rightarrow \tilde{h}(b)\Big)$

eine **Umbenennung** von Σ.

Beispiel

$$
\begin{array}{lll}
\mathit{KOMPLEX} := \mathit{COMPLEX} & & \\
 & \textbf{rename} & complex \rightarrow komplex \\
 & & realPart \rightarrow Realteil \\
 & & imagPart \rightarrow Imagin\ddot{a}rteil \\
 & \textbf{end} &
\end{array}
$$

ist Umbenennung mit

h	$complex$	$realPart$	$imagPart$
	$komplex$	$Realteil$	$Imaginärteil$

Anmerkung

Die Beziehung zwischen **COMPLEX** und **KOMPLEX** wird graphisch wie folgt dargestellt:

$$
\begin{array}{c}
\boxed{\text{COMPLEX}} \\
\downarrow \\
\boxed{\text{KOMPLEX}}
\end{array}
$$

6.3 Einschränkung

Definition

Ist $\Sigma = (\ S\ ,\ B\ ,\ c)$ eine Signatur und
$D \subset S \cup B$, so ist $\Sigma' = (\ S'\ ,\ B'\ ,\ c'\ )$ mit

$S' = S \cap D$
$B' = B \cap D$
$c' = c\ |_{B \cap D}$

eine **Einschränkung** von Σ, falls

$\mathrm{sort}(c') \cap S \subset S'$

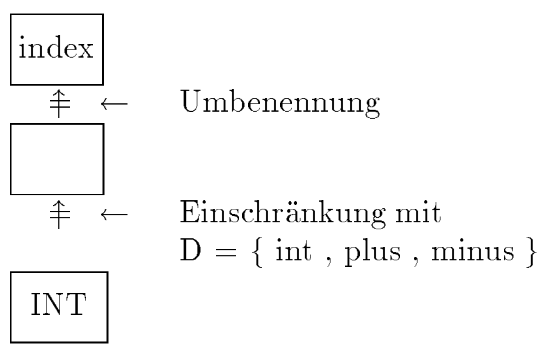
d.h. es kann kein Bezeichner frei werden, der nicht schon vorher frei war.

Beispiel

INDEX := *INT*
restricted to *int* , *plus* , *minus*
end
rename *int* → *index*
end;

Anmerkung

Verdeutlicht wird der Zusammenhang durch folgende Skizze



Grammatik

module ::= **module** *[sorts_clause] [opns_clause]* **end**
| *module*
rename

```
                    id → id { , id → id }
                end
              | module
                restricted to
                    id { , id }
                end
sorts-clause ::= sorts sort_id { , sort_id } ;
opns-clause  ::= opns { opn_id { , opn_id } : char; }
char         ::= product [ product ]
product      ::= term { × term }
term         ::= sort_id | ( char )
```

6.4 Sichtbarkeit

Beispiel **BUFFER** ::= **module**

```
BUFFER ::= module
              knows INTEGER,BOOLEAN;
              sorts  buffer;
              opns
                 empty : buffer;
                 isEmpty : buffer → boolean;
                 isFull : buffer → boolean;
                 put : buffer × integer → buffer;
                 get : buffer → integer;
           end.
```

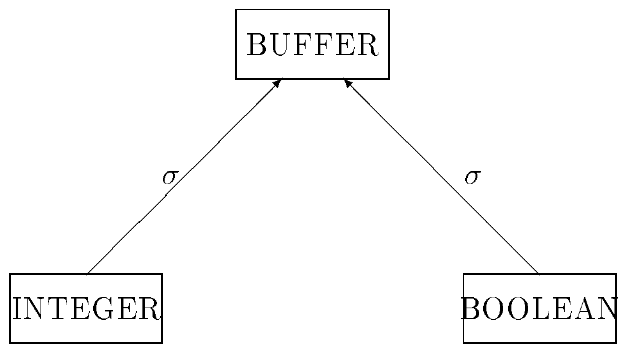
Feststellung
Bezeichner, die in einem Modul verwendet werden und dort frei sind, können durch eine **knows**-Klausel explizit gebunden werden.

Beispiel (MODULA – 2)

```
definition module BUFFER;
    const n_max    = 100;
    type  buffer   = record
                        buf: array[1..n_max-1]of INTEGER;
                        k:[0..n_max-1];
                     end;
    procedure isEmpty (b : buffer) : BOOLEAN;
    procedure isFull (b : buffer) : BOOLEAN;
```

procedure *put* (**var** *b : buffer ;* x : *INTEGER) ;*
procedure *get* (**var** *b : buffer ;* **var** x : *INTEGER) ;*
procedure *empty* (**var** *b : buffer) ;*
end *BUFFER.*

Skizze



Beispiel (MODULA–artig)

definition module *A;*
from *INTEGER* **import** *integer;*
oder
definition module *A;*
import *INTEGER;*
⋮
x : *INTEGER.integer;*

Grammatikregel

knows-clause ::= **knows**
[module_id =] module
{, [module_id =] module } ;

Feststellung

Ein Modul ist **vollständig**, falls gilt :

$\mathrm{sort(c)} \subset S \cup \bigcup_{\Sigma' \underset{\sigma}{\rightarrow} \Sigma} S'$

Definition

Σ' **benutzt direkt syntaktisch** Σ (in Zeichen: $\Sigma \underset{\gamma}{\rightarrow} \Sigma'$), falls gilt :

$\Sigma = \Sigma'$

oder

$\Sigma \underset{\sigma}{\rightarrow} \Sigma'$ und $(sort(c')\backslash S') \cap S \neq \emptyset$

d.h. in Σ' kommt auch mindestens eine Sorte von Σ vor

Anmerkung

Die Beziehung “Modul A benutzt Modul B“ kann heißen:

$$B \underset{\sigma}{\rightarrow} A$$

oder

$$B \underset{\gamma}{\rightarrow} A$$

oder

$$B \overset{*}{\underset{\sigma}{\rightarrow}} A$$

oder

$$B(\nrightarrow \cup \nVrightarrow \cup \underset{\sigma}{\rightarrow} \cup \underset{\gamma}{\rightarrow})^* A$$

6.5 Parameterisierung

Beispiel

PQUEUE := **parms**
    ITEM = **module**
        sorts *item;*
        opns *le:item* × *item* → *bool;*
    end
body
    module
        sorts *pqueue;*
        opns
            init:pqueue;
            isEmpty:pqueue → *bool;*
            insert:pqueue × *item* → *pqueue;*
            extractMin:pqueue → *pqueue* × *item;*
            min:pqueue → *item;*
    end.
INTPQUEUE := *PQUEUE* **rename** *item* → *int* **end** (*INT*).

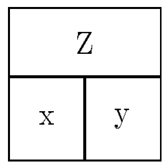
Feststellung

Module können evtl. durch andere Module parametrisiert sein. Ein parametrisiertes Modul kann durch ein “passendes“ aktuelles Argumentmodul konkretisiert sein.

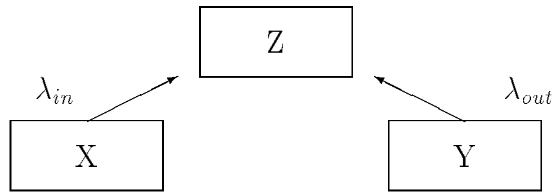
A “paßt“ zu B bedeutet soviel wie: A ist syntaktisch kompatibel zu B, d.h. B ist eine Einschränkung von A.

Skizze

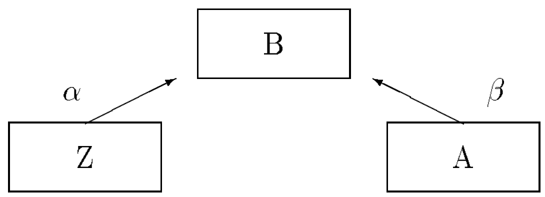
Z := **parms** X **body** Y



oder



B := Z(A)



Teil III

Spezifikation

Kapitel 7

Algebraische Spezifikation

Literatur

E. Ehrig, B. Mahr
Fundamentals of algebraic specification
Springer, EATCS-Serie, Bd. 6, 1985

7.1 Grundlagen

Historie

Erste Ansätze:

$$\left.\begin{array}{l} Liskov/Zilles \\ Guttag \\ ADJ-Group \end{array}\right\} \sim 1975$$

Beispiel

SEMIGROUP = **module**
sorts *S;*
opns * : $S \times S \rightarrow S$;
eqns *a,b,c : S ;*
*(a*b)*c = a*(b*c)*
end

Zugehörige konkrete Halbgruppenmodelle:
(N,+) oder (N,*) oder $(\Sigma^*, \|)$

Beispiel

$BOOL :=$ **module**
  sorts: *bool;*
  opns: *true, false : bool;*
    not : bool $\to$ bool;
    and,or : bool $\times$ bool $\to$ bool;
  eqns: *b : bool;*
    not(true) = FALSE;
    not(false) = TRUE;
    and(b,true) = b;
    and(b,false) = FALSE;
    or(b,true) = TRUE;
    or(b,false) = b;
end

Die bei **eqns** *aufgeführten Zeilen stellen keine Zuweisungen, sondern Gleichungen dar.*

Feststellung

Spezifikation = Signatur + Gleichungen
Für eine Spezifikation gibt es im allgemeinen mehrere Modelle im Sinne der Logik.

Beispiele

- $(BOOLEAN, TRUE, FALSE, \mathbf{not}, \mathbf{and}, \mathbf{or})$ ist Modell für BOOL.
- $(2^M, M, \emptyset, \lambda x.M \setminus x, \cap\ , \cup\ )$ ist Modell für BOOL.

Beispiel

$NAT :=$ **module**
  sorts: *nat;*
  opns: *0 : nat;*
    succ: nat $\to$ nat;
    add : nat $\times nat \to nat$;
  eqns: *n, m : nat;*
    add(n,0) = n;
    add $\Big(n, succ(m)\Big) = succ\Big(add(n,m)\Big)$;
end

Beispiel

Auch $(\{1\}, 1, \lambda x.1, \lambda[x,y].1)$ ist Modell für NAT.

7.2 Signatur–Algebra

Einschränkung

Alle Charakteristiken sind im folgenden einfach, d.h. sie haben die Gestalt:

$$sort\_id$$
$$|sort\_id\{\times sort\_id\} \rightarrow sort\_id.$$

Charakteristiken bestehen aufgrund dieser Einschränkung entweder nur aus einer Sorte oder besitzen auf der linken Seite mindestens ein Kreuz.

Notation (bei Ehrig/Mahr) am Beispiel der Signatur von NAT:

bisherige Notation	Notation bei Ehrig/Mahr
$0 : nat$	$0 :\rightarrow nat$
$add : nat \times nat \rightarrow nat$	$add : nat\ nat \rightarrow nat$

D.h. bei Ehrig/Mahr haben alle Charakteristiken die Gestalt:

$$\{sort\_id\} \rightarrow sort\_id.$$

Definition

Ist $\Sigma = (S, B, c)$ eine Signatur, so ist

$$A = (S_A, B_A)$$

eine Σ–**Algebra**, falls gilt:

$S_A = (A_s)_{s \in S}$ ist eine Familie von Basismengen, und

$B_A = (N_A)_{N \in B}$ ist eine Familie von Funktionen

mit

$$N_A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$$

falls

$$c(N) = s_1 \ldots s_n \rightarrow s.$$

Feststellung

$S_A = (A_s)_{s \in S} = \{A_s | s \in S\}$

$B_A = (N_A)_{N \in B} = \{N_A | N \in B\}.$

Beispiel

NAT–Σ–Algebra	:	N
Basismenge	:	$N_{nat} = \mathbb{N}$
Funktionen	:	$0_N = 0$
	:	$succ_N = \lambda x.x + 1$
	:	$add_N = +$

oder

NAT–Σ–Algebra	:	$= U$
Basismenge	:	$U_{nat} = \{1\}$
Funktionen	:	$0_U = 1$
	:	$succ_U = \lambda x.1$
	:	$add_U = \lambda[x, y].1$

Definition

Ist $\Sigma = (S, B, c)$ eine Signatur und sind A, B Σ-Algebren, dann ist ein **Σ-Homomorphismus** eine Familie

$$f : (f_s : A_s \to B_s)_{s \in S}$$

von Abbildungen mit

$$f_s\Big(N_A(a_1, \ldots, a_n)\Big) = N_B\Big(f_{s_1}(a_1), \ldots, f_{s_n}(a_n)\Big)$$

falls

$$c(N) = s_1 \ldots s_n \to s.$$

Skizze

$$\begin{tikzcd} A_{s_1} \arrow[d, "f_{s_1}"'] & \times \ldots \times & A_{s_k} \arrow[d, "f_{s_k}"] \arrow[r, "N_A"'] & A_s \arrow[d, "f_s"] \\ B_{s_1} & \times \ldots \times & B_{s_k} \arrow[r, "N_B"] & B_s \end{tikzcd}$$

Definition

Sind alle f_s bijektiv, so heißt f **Isomorphismus**.

Zwei Σ–Algebren A,B heißen **isomorph**, falls es einen Isomorphismus zwischen A und B gibt.

Beispiel

$f : N \to \{1\}$ ist Homomorphismus

zu zeigen:
$f(0) = 1 \ \surd$
$f(\lambda x.x+1) = 1 = (\lambda x.1)\Big(f(n)\Big) \ \surd$
$f(n+m) = 1 = (\lambda[x,y].1)\Big(f(n), f(m)\Big) \ \surd$

7.3 Terme

Beispiel (Terme über NAT)

$0, succ(0), succ\Big(succ(0)\Big),$
$add(0,0), add\Big(succ(0), 0\Big), add\Big(0, succ(0)\Big),$
$add\Big(succ(0), succ(0)\Big), succ\Big(add(0,0)\Big), \ldots$

falls x, y Variablen der Sorte *nat* sin auch:
$succ(x), succ(y), x, y, add(x,0), \ldots$

Definition

Sei $\Sigma = (S, B, c)$ eine Signatur und $X = (X_s)_{s \in S}$ eine Familie von Variablenmengen, dann ist

$$T_{B,s}(X),$$

die **Menge der Terme der Sorte s** mit Variablen aus X folgendermaßen definiert:

1. $X_s \cup \{N \in B | c(N) = s\} \subset T_{B,s}(X)$.
2. Für $N \in B$ mit $c(N) = s_1 \times \ldots \times s_k \to s$ und $t_1 \in T_{B,s_1}(X) \ldots$ und $t_k \in T_{B,s_k}(X)$ ist $N(t_1, \ldots, t_k) \in T_{B,s}(X)$.

Verabredung

•$T_{B,s} := T_{B,s}()$ ist die Menge der Grundterme der Sorte s
•$T_B(X) := \bigcup_{s \in S} T_{B,s}(X)$
•$T_B := T_B()$ ist die Menge aller Grundterme

Satz und Definition

Ist Σ eine Signatur. Dann ist

$$A = \Big((T_{B,s})_{s \in S},\ (N_A)_{N \in B}\Big)$$

mit

$$N_A : T_{B,s_1} \times \ldots \times T_{B,s_n} \to T_{B,s}$$

gemäß

$$(t_1, \ldots, t_n) \mapsto N(t_1, \ldots, t_n)$$

falls

$$c(N) = s_1 \ldots s_n \to s$$

eine Σ - Algebra (genannt **Termalgebra**)

Feststellung

Die Elemente der Termalgebra sind **Symbolketten** in

$$\Big(B \cup \{'(',')',','\}\Big)^* ,$$

die **gleichzeitig** als Auswertungsvorschrift gelesen werden können.

Definition

Sei A eine Σ - Algebra. Dann ist eine **Auswertung** (evaluation) von Termen über Σ in A eine Familie von Abbildungen

$$eval = (eval_s : T_{B,s} \to A_s)_{s \in S}$$

mit

$$eval_s\Big(N(t_1, \ldots, t_n)\Big) = N_A\Big(eval_{s1}(t_1), \ldots, eval_{sn}(t_n)\Big)$$

für alle $N \in B$ mit $c(n) = s_1 \ldots s_n \to s$.

Beispiel (zu NAT)

Algebra $N = (\ I\!N,\ 0,\ \lambda x.x+1,\ +)$

$$\begin{aligned}
& eval\bigg(succ\Big(add\Big(succ(0), 0\Big)\Big)\bigg) \\
&= (\lambda x.x+1)\bigg(eval\Big(add\Big(succ(0), 0\Big)\Big)\bigg) \\
&= (\lambda x.x+1)\Big(eval\Big(succ(0)\Big) + eval(0)\Big) \\
&= (\lambda x.x+1)\Big(\Big(\lambda x.x+1\Big)\Big(eval(0)\Big) + eval(0)\Big) \\
&= (\lambda x.x+1)\Big((\lambda x.x+1)(0) + 0\Big)
\end{aligned}$$

$$
\begin{aligned}
&= (\lambda x.x + 1)(1 + 0) \\
&= (\lambda x.x + 1)(1) \\
&= 2
\end{aligned}
$$

Feststellung

eval ist ein Σ - Homomorphismus von der Termalgebra in A.

Es gibt von der Termalgebra aus einen Σ - Homomorphismus in **jede** Σ - Algebra.

7.4 Spezifikationsalgebren

Beispiel

$STACK =$ **module**

knows *INT, BOOL;*

sorts *stack;*

opns *empty : stack;*

top : stack $\rightarrow$ *int;*

pop : stack $\rightarrow$ *stack;*

push : stack $\times$ *int* $\rightarrow$ *stack;*

isEmpty : stack $\rightarrow$ *bool;*

eqns

x : int;

s : stack;

pop(empty) = empty

*(*Fehlerbehandlung lassen wird aus*)*

$pop\Big(\ push(s,x)\ \Big) = s$

isEmpty(empty) = true

$isEmpty\ \Big(\ push(s,x)\ \Big) = false$

top(empty) = 0

*(*Fehlerbehandlung lassen wir aus*)*

$top\ \Big(\ push(s,x)\ \Big) = x$

end *STACK*

Anmerkung

'=' ist ein Meta-Symbol, um Paare von Termen zu formulieren.

Beispiel

STRING := **module**

knows	char;
sorts	string;
opns	
	empty : → string;
	make : char → string;
	ladd : char × string → string;
	radd : string × char → string;
	concat : string× string → string;
eqns	a:char; s,s1,s2,s3:string;
	concat(s,empty) = s;
	concat(empty,s) = s;
	concat(concat(s1,s2),s3) =
	concat(s1,concat(s2,s3))
	ladd(a,s) = concat(make(a),s)
	radd(s,a) = concat(s,make(a))

end.

Idee

Die Gleichungen besagen, daß in jeder zu der Spezifikation gehörenden Algebra die Auswertung der linken Seite jeweils dasselbe ergibt, wie die Auswertung der rechten Seite.

Definition

Ist A eine Σ-Algebra und ist $X = (X_s)_{s \in S}$ eine Familie von Variablenmengen, so ist

$$ass = (ass_s : X_s \rightarrow A_s)_{s \rightarrow S}$$

eine Belegung (assignment).

Die **erweiterte Belegung**

$$\overline{ass} = (\overline{ass}_s : T_{B,s}(X) \rightarrow A_s)_{s \in S}$$

ist definiert als

$$\overline{ass}_s(X) = (ass_s(x), falls\ x \in X_s$$

$$\overline{ass}_s\Big(N(t_1, \ldots, t_n\Big) = N_A\Big(ass_{s1}(t1), \ldots, ass_{sn}(tn)\Big)$$

falls $N \in B$ mit $c(N) = s_1, \ldots, s_n$.

Skizze

$$\begin{tikzcd}
T_B \arrow[r] \arrow[dr, "\text{eval}"'] & T_B(X) \arrow[d, "\overline{ass}"'] & X \arrow[l] \arrow[dl, "\text{ass}"] \\
 & A &
\end{tikzcd}$$

Definition

Ist Σ eine Signatur und $X = (X_s)_{s\in S}$ eine Familie von Variablenmengen. Ein Tripel (X, L, R) mit $L, R \in T_{B,s}(X)$ heißt **Gleichung** der Sorte s.

Definition

Eine **(Gleichungs-) Spezifikation** SPEC $=(\Sigma, E)$ besteht aus einer Signatur Σ und einer Menge E von Gleichungen bezüglich Σ.

Definition

Eine Gleichung e=(X, L, R) der Sorte s ist **gültig** (synonym: A **erfüllt** e) in einer Σ–Algebra A, falls für alle Belegungen $ass = (ass_s : X_s \rightarrow A_s)_{s\in S}$ gilt:

$$\overline{ass}_s(L) = \overline{ass}_s(R)$$

Definition

Sei SPEC=(Σ, E) eine Spezifikation. Eine Σ–Algebra A heißt SPEC–**Algebra**, falls sie alle Gleichungen in E erfüllt.

Beispiel

Es gibt im allgemeinen viele SPEC–Algebren

$$S\left(\underbrace{\{0,1\}}_{\text{char}} , \underbrace{\{\varepsilon\}}_{\text{string}} , \underbrace{\varepsilon}_{\text{empty}} , \underbrace{\begin{array}{c|cc} & 0 & 1 \\ \hline & \varepsilon & \varepsilon \end{array}}_{\text{make}} , \underbrace{\begin{array}{c|cc} & 0 & 1 \\ \hline \varepsilon & \varepsilon & \varepsilon \end{array}}_{\text{ladd/radd}} , \underbrace{\begin{array}{c|c} & \varepsilon \\ \hline \varepsilon & \varepsilon \end{array}}_{\text{concatenate}} \right)$$

7.5 Initiale Semantik

Feststellung

Man kann einer Spezifikation als Semantik zuordnen

1. alle SPEC–Algebren (loose semantics)
2. eine ausgezeichnete Algebra (initial semantics)
3. alle Algebren mit dem “gleichen” Verhalten (behavioural semantics)

Definition

Sei SPEC=(Σ, E) eine Spezifikation. Dann sei

$$\equiv \; = (\equiv_s)_{s \in S}$$

eine Familie von Äquivalenzrelationen auf $T_{B,s}$ gemäß
¿ $t_1 \equiv_s t_2$ gdw. $eval_s^A(t_1) = eval_s^A(t_2)$ in **jeder** SPEC–Algebra A.

Feststellung

Die $\equiv_s$ sind Äquivalenzrelationen, d.h. sie sind symmetrisch, reflexiv und transitiv.

Beispiel

Aufgrund von $\equiv_{nat}$ kann man die Terme nun in die folgenden $\equiv_{nat}$–Klassen einordnen (a=add, s=succ):

$$\begin{array}{c|c|c|c|c}
\mathsf{0} & \mathsf{s(0)} & \mathsf{s}\Big(\mathsf{s(0)}\Big) & \mathsf{s}\Big(\mathsf{s}\Big(\mathsf{s(0)}\ \Big)\Big) & \cdots \\
\mathsf{a(0,0)} & \mathsf{a}\Big(\mathsf{s(0)},\mathsf{0}\Big) & \mathsf{a}\Big(\mathsf{s}\Big(\mathsf{s(0)}\ \Big),\mathsf{0}\Big) & \mathsf{a}\Big(\mathsf{s}\Big(\mathsf{s}\Big(\mathsf{s(0)}\Big)\Big),\mathsf{0}\Big) & \vdots \\
\vdots & \mathsf{a}\Big(\mathsf{0},\mathsf{s(0)}\Big) & \mathsf{a}\Big(\mathsf{s(0)},\mathsf{s(0)}\Big) & \vdots & \\
 & \vdots & \vdots & &
\end{array}$$

Beispielsweise liegen $\mathsf{add}\Big(\mathsf{succ(0)},\mathsf{0}\Big)$ und $\mathsf{succ(0)}$ in derselben $\equiv_{nat}$–Klasse, wegen:

$$\mathsf{add}\Big(\mathsf{succ(0)},\mathsf{0}\Big) \equiv_{nat} \mathsf{succ}\Big(\mathsf{add(0,0)}\Big) \equiv_{nat} \mathsf{succ(0)}$$

Feststellung

$T_{B,s}$ wird durch $\equiv_s$ in Äquivalenzklassen zerlegt.

Ist $t \in T_{B,s}$, so ist $[t]_{\equiv_s} = \{t' | t' \equiv_s t\}$ die Äquivalenzklasse zu t.

Satz

Die Relationen $\equiv_s$ sind verträglich mit den Operationen, d.h.

$$t_1 \equiv_s t'_1, \ldots, t_k \equiv_s t'_k \quad \Rightarrow \quad N(t_1, \ldots, t_k) \equiv_s N(t'_1, \ldots, t'_k)$$

falls

$$c(N) = s_1 \times \ldots \times s_k \to s$$

($\equiv_s$ ist Kongruenzrelation)

Satz und Definition

Ist SPEC=(Σ, E) eine Spezifikation, so ist

$$Q_{SPEC} = \Big((Q_s)_{s \in S}, (N_Q)_{N \in B}\Big)$$

mit

$$Q_s = \{[t]_{\equiv_s} | t \in T_{B,s}\}$$

und

$$N_Q : Q_{s_1} \times \ldots \times Q_{s_k} \to Q_s$$

$$([t_1]_{\equiv_{s_1}}, \ldots, [t_k]_{\equiv_{s_k}}) \mapsto \Big(N(t_1, \ldots, t_k)\Big)_{\equiv_s}$$

mit

$$c(N) = s_1 \ldots s_k \to s$$

eine SPEC–Algebra (**Quotiententermalgebra**)

Anmerkung

Eine **initiale Semantik** einer Spezifikation ist die Menge aller Algebren, die mit der Quotiententermalgebra isomorph sind.

7.6 Abschließende Bemerkung

Faustregel zur Konstruktion algebraischer Strukturen

1. Man suche konstruierende Operationen (und eventuel zugehörige Gleichungen, die die Äquivalenzklassen der Quotientenalgebra erzeugen.

2. Man beschreibe die anderen Operationen in Bezug auf die konstruierenden.

Beispiel

$$
\begin{array}{lll}
\mathit{SET} := \textbf{module} & & \\
& \textbf{knows}: & \mathit{INT,BOOL;} \\
& \textbf{sorts}: & \mathit{set;} \\
& \textbf{opns}: & \mathit{empty : set;} \\
& & \mathit{isEmpty : set \to BOOL;} \\
& & \mathit{insert : set \times int \to set;} \\
& & \mathit{delete : set \times int \to set;} \\
& & \mathit{if\text{-}then\text{-}else : bool \times set \times set \to set;} \\
& \textbf{eqns}: & \mathit{s : set;\ x,y : int;} \\
& & \mathit{isEmpty(empty) = true} \\
& & \mathit{isEmpty\Big(insert(s,x)\Big) = false} \\
& & \mathit{delete(empty,x) = empty} \\
& & \mathit{delete\Big(insert(s,x),y\Big) = if\ x{=}y\ then\ delete(s,y)} \\
& & \qquad\qquad\qquad\qquad \mathit{else\ insert\Big(delete(s,y),x\Big)} \\
\quad \textbf{end} & &
\end{array}
$$

Terme: $\equiv_{set}$–Klassen (e=*empty*, i=*insert*)

$$
\begin{array}{c|c|c|c|c}
e & i(e,1) & i\Big(i(e,1),2\Big) & i\Big(i\Big(i(e,1),2\Big),3\Big) & \cdots \\
 & i\Big(i(e,1),1\Big) & \vdots & \vdots & \vdots \\
 & i\Big(i\Big(i(e,1),1\Big),1\Big) & & & \\
 & \vdots & & & \\
\emptyset & \{1\} & \{1,2\} & \{1,2,3\} &
\end{array}
$$

Anmerkung

- **Vorteile**
  - präzise Darstellungsform
  - größtmögliche Allgemeinheit
  - Unabhängigkeit von einer Implementation
- **Nachteile**
  - Unanschaulichkeit
  - keine isolierte Spezifikationsmöglichkeit für Operationen

Kapitel 8

Konstruktive Spezifikation

Literatur

C.B. Jones
Systematic Software Development Using VDM
Englewood Cliffs, Prentice Hall 1986.

8.1 Grundlagen

Name

VDM = "Vienna Definition Method"
entwickelt von C. Jones, D. Björner, W. Henhapel, P. Lucas
an den Wiener Laboratorien der IBM

Idee

Spezifikation erfolgt **modell-orientiert**.
Man beschreibt einen **Zustandsraum**.

Beispiel

$$
\begin{array}{lll}
STACK = \textbf{module} & & \\
& \textbf{knows} & INT,\ BOOL \\
& \textbf{sorts} & stack := \textbf{list}\ of\ int \\
& \textbf{opns} & \\
& & empty : stack \\
& & \quad := <> \\
& & push : stack \times int \to stack \\
& & \quad := \lambda[s,i].i :: s
\end{array}
$$

$pop : stack \rightarrow stack$
$:= tail$
...
end

Feststellung

Der Datentyp wird mathematisch modelliert. Die Operationen werden

- als Konstante
- als Funktionen oder
- durch Angabe von Vor- und Nachbedingungen

beschrieben.
Zur Beschreibung werden Listen, Mengen und Abbildungen (Mappings) verwendet.

Anmerkung

Ein Mapping ist eine partielle Abbildung mit endlichem Urbildbereich, d.h. falls m ein Mapping vom Typ
map X **to** Y
ist, so ist
m ⊂ X × Y
mit
$\{x \in X \mid \exists y \in Y : (x, y) \in m \}$ endlich
and $(x, y_1) \in m, (x, y_2) \in m \Rightarrow y_1 = y_2$

Operationen

Grundtypen

- (z.B.) int, bool, string

Sortenkonstruktionen

- **list of** t
- **set of** t
- **map** t_1 **to** t_2
- $t_1 \times \ldots \times t_n$

Mengenoperationen

- a $\in$ A
- A $\subset$ B
- A $\cup$ B
- A $\cap$ B
- A $\setminus$ B
- $\mid$ A $\mid$
- { x **in** A $\mid$ p(x)}
- $a_1, \ldots, a_k$

Listenoperationen

• <>	leere Liste
• $< x_1, \ldots, x_k >$	Liste bestehend aus $x_1, \ldots, x_k$
• x :: l	"cons" d.h. ist l $\hat{=} < x_1, \ldots, x_k >$, so ist x :: l = $< x, x_1, \ldots, x_k >$
• head(l)	erstes Element der Liste
• tail(l)	Rest der Liste, d.h. ist l = $< x_1, \ldots, x_k >$, so ist tail(l) = $< x_2, \ldots, x_k >$
• lenlist(l)	Länge der Liste
• l(i)	i - tes Element der Liste
• $l_1 \parallel l_2$	Konkatenation zweier Listen
• elems(l)	Menge aller Elemente der Liste
• rev(l)	Umkehrung der Liste, d.h. ist l = $< x_1, \ldots, x_k >$, so ist rev(l) = $< x_k, \ldots, x_1 >$

Mappingoperationen

• [* *]	leeres Mapping
• [* $d_1 \to r_1, \ldots, d_k \to r_k$ *]	
• [* $d \to r \mid p(d, r)$ *]	
• dom(m)	Urbildbereich, d.h. { $x \in X \mid \exists y \in Y : (x, y) \in m$}
• rng(m)	Bildbereich, d.h. { $y \in Y \mid \exists x \in X : (x, y) \in m$}
• m(d)	Anwendung (Applikation) für d $\in$ dom(m)
• m#n	Überschreiben (Overrite) { $(x, y) \mid (x, y) \in n$ **or** $(x \notin dom(n)$**and**$(x, y) \in m)$ }
• m$S	Einschränkung $(x, y) \mid (x, y) \in m$ **and** $x \in S$

8.2 Vor- / Nachbedingungsspezifikation

Beispiel

function *sqrt (x : real) : real;*
  pre - *sqrt(x) == x* $\geq$ *0*
  post - *sqrt(x;y')* $==$ $y'^2 = x$ *and y'* ≥ 0

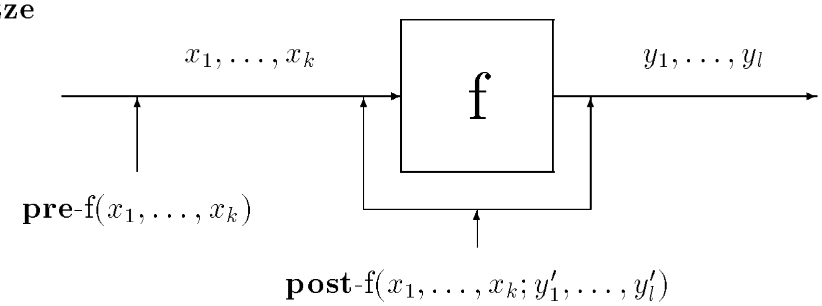
function *max (x,y : integer) : integer*
  pre - *max(x,y)* == **true** d.h. keine Vorbedingung
  post - *max(x,y;z')* ==
    $z' \geq$ x **and** *z'* $\geq$ *y* **and** *(z' = x* **or** *z' = y)*

Feststellung

Eine Funktion f mit den Eingabeparametern $x_1, \ldots, x_k$ und den Ausgabeparametern $y_1, \ldots, y_l$ kann spezifiziert werden durch zwei Prädikate:

1. **pre**-f($x_1 \ldots, x_k$), beschreibt die Anwendbarkeitsbedingungen.
2. **post**-f($x_1, \ldots, x_k; y'_1, \ldots, y'_k$) beschreibt
   die Beziehung zwischen den Ein– und Ausgabewerten.

Skizze



Beispiel

FILE = **module**
  knows *INFO, KEY*
  sorts *file := **map** key **to** info*
  opns
    init : file := [*]*
    insert : file $\times$ *key* $\times$ *info* $\rightarrow$ *file*
      pre - *insert(f,k,i)* == $k \notin$ *dom(f)*

post - *insert(f,k,i;f')* $==$ *f'* $=$ *f* $\#$ *[* k* $\rightarrow$ *i *]*
delete : file $\times$ *key* $\rightarrow$ *file*
pre - *delete(f,k)* $==$ $k \in dom(f)$
post - *delete(f,k,f')* $==$

$$f' = f \ \$ \left(\ dom(f) \setminus \{k\} \ \right)$$

end

8.3 Bemerkungen

Beispiel

SET := **module**
  knows *INT;*
  sorts *set* = **set of** *INT;*
  opns *empty : set;*
    $:= \emptyset$
      oder **post**-*empty(s')* == $s' = \emptyset$;
    insert : set $\times$ *int* $\rightarrow$ *set;*
    $:= \lambda[s,i].s \cup \{i\}$
      oder **post**-*insert(s,i;s')* == $s' = s \cup \{i\}$;
    delete : set $\times$ *int* $\rightarrow$ *set;*
    $:= \lambda[s,i],\ s \setminus \{i\}$
      oder **post**-*delete(s,i;s')* == $s' = s \setminus \{i\}$;

Beispiel

procedure *insert* (**var** *s:set; i:int)*

hier:

c(insert) = *set* $\times$ *int* $\rightarrow$ *set* *und*
post-*insert(s,i;s')* == $s' = s \cup \{i\}$

Anmerkung
Häufig ist es sinnvoll, Spezifikationen durch zusätzliche Definitionen zu ergänzen (mit dem Ziel der Schreibabkürzung), wie z.B.:

X := **module**
  knows ...
  sorts ...
  defs
    $rhead := \lambda\ l.head\left(rev(l)\right);$
    $nmax := 1000;$
  opns
    $\vdots$
end

Anmerkung
Die Sprachebene der Spezifikation kann je nach Anwendungsbereich auch höher gewählt werden, z.B. Matrizen, Automaten, Graphen.

Beispiel

Eine Funktion, welche aus einem regulärer Ausdruck einen DFA erzeugt:

Es sei

$rexpr$ = Menge der regulären Ausdrücke
$auto$ = Menge der deterministischen endlichen Automaten

Dann sei

$RtoDFA : rexpr \rightarrow auto$
post-*RtoDFA(R;U')* == *L(R)* = *L(U')*

Anmerkung

Die Sprache muß nur **rigoros** verwendet werden, d.h man muß im Zweifel auch weiter formal verfeinern können.

Anmerkung

Man kann den Modellraum durch Invarianten einschränken, z.B.

$$\textbf{sorts}\ \text{queue} := \{l \in \textbf{list of}\ int \mid \underbrace{lenlist(l) \leq 100}_{Invariante}\}$$

Wenn die Invariante zu groß wird, sollte man die gesamte Modellbildung überdenken!

Anmerkung

Record–artige Strukturen können als Tupel oder als Abbild von den Selektoren in die Wortbereiche aufgefaßt werden.

Beispiel

type *datum* = **record**;
  tag : {1 ... 31};
  monat : {1 ... 12};
  jahr : {00 ... 99};
 end;

entspricht

$$\{\ 1 \ldots 31\ \} \times \{\ 1 \ldots 12\ \} \times \{\ 00 \ldots 99\ \}$$

oder

$$d\colon \{\ tag, monat, jahr\ \} \rightarrow \{\ 0 \ldots 99\ \}$$

wobei

$d(tag) \in \{ 1 \ldots 31 \}$ **and**
$d(monat) \in \{ 1 \ldots 12 \}$ **and**
$d(jahr) \in \{ 0 \ldots 99 \}$

Man kann dann alle Mapping–Operationen auch für Records verwenden.

$d' = d \# [* \text{ jahr} \rightarrow 90 *]$
d.h. wenn d = 10.5.87 ist, so ist d'= 10.5.90

Anmerkung

Probleme der Speicherverwaltung innerhalb von Programmen werden durch konstruktive Spezifikation **nicht** erfaßt, z.B. (alloc = Allokation)

rec_alloc(p);
*rec *p;*
post-$rec\_alloc\left(*p;(*p)'\right)$ == **true**

Ein-/Ausgabe innerhalb von Operationen sind ebenfalls nicht beschreibbar, z.B.

function *readx( ) : int;*
post-*readx(i')* == **true**

Man kann aber den Speicher bzw. die Datei mitmodellieren, z.B.

sorts *file* = **list of** *char;*
sorts *readx : file* $\rightarrow$ *file* $\times$ *int;*
post-*readx(f;f',x')* == ...

Anmerkung

Man kann auch abstrakte Daten**strukturen** spezifizieren, wenn man eine **states**–Klausel verwendet, z.B.

STAPEL := **module**
knows *INT, BOOL;*
states *s :* **list of** *int* :=<>;

opns
isEmpty $:\rightarrow$ *bool;*
post-*isEmpty(s;s',b')* ==
b'= (s=<>);
⋮
end

Man muß den Zustandsbezeichner dann stets bei den Ein- und Ausgabeparametern mit angeben.

Anmerkung
Durch die Modellbildung wird auch eine Realisierung suggeriert. Eigentlich soll durch Spezifikationen aber nur ein Ein–/Ausgabeverhalten festgelegt werden.

Beispiel (Prioritätsschange)

PQUEUE := **module**
knows *INT, BOOL;*
sorts *pqueue* := { $l \in$ **list of** *int* | *isSorted(l)* }
defs *isSorted(l)* :=
if *lenlist(l)* <= *1* **then** *true*
else $head(l) \leq head\Big(tail(l)\Big)$
and $isSorted\Big(tail(l)\Big)$
⋮
end

Diese Spezifikation suggeriert die Verwendung von Listen, obwohl die Verwendung von Bäumen (Heaps) besser wäre.

Anmerkung
Man kann die Vor– und Nachbedingungen auch innerhalb einer Spezifikation verwenden, z.B.

$f : X \rightarrow Y$
post-*f(x;y')* $== \exists\, z \in Y \Big($ **pre-post**-*g(x,z)* **and** $y' = 5 * z \Big)$

pre-post-*g(x,z)* sei hier die Konjunktion der Vor– und Nachbedingung von g.

Anmerkung

Vorteile:

- präzise Darstellung möglich
- von Implementation unabhängig
- jede Operation einzeln spezifizierbar
- anschaulich (-er als algebraische Spezifikation)

Nachteil:

- eher Ungenauigkeit möglich

8.4 Spezifikation nach Parnas

Literatur

D. L. Parnas
A technique for software module specification with examples.
Comm. ACM 15 (1972), S. 330-336

Idee

Die Module werden mit Operationen beschrieben. Dabei unterscheidet man die Operationen in:

- **V–Operationen** (= wert–liefernde Operationen)
- **O–Operationen** (= verändernde Operationen)

Beispiel (Stapel als Datenobjekt)

V–Operationen:

function *isEmpty( ) : boolean;*
function *top( ) : item;*
hidden function *element(i:natural) : item;*
hidden function *pos( ) : natural;*

O–Operationen (werden mit Hilfe der V–Funktionen erklärt):

procedure *init( );*
effect: *isEmpty'( )* = *true*
and *top'( )* = **undefined**
and $\forall i \in$ *natural*: *element'(i)* = **undefined**
and *pos'( )* = *0*

procedure *push(x:item);*
effect: *isEmpty'( )* = *false*
and *top'( )* = *x*
and $element'\Big(pos'(\ )\Big) = x$
and *pos'( )* = *pos( )+1*
procedure *pop( );*
effect: *isEmpty'( )* = $\Big(pos(\ ) <= 1\Big)$
and *top'( )* = $element\Big(pos'(\ )\Big)$
and $element'\Big(pos(\ )\Big)$ = **undefined**
and *pos'( )* = $max\Big(pos(\ ) - 1, 0\Big)$

Anmerkungen

- Durch die V–Funktionen wird der Zustand des beschriebenen Datenobjektes konstruktiv angegeben.
- Alle **nicht** in der **effect**–Klausel angegebenen Größen bleiben unverändert.
- Warnung: Parnas verwendet in der Originalarbeit den Strich für den **Vor**–Zustand!

8.5 Sprache Z

Historie

Z ist eine Notation. Sie wurde an der Oxford University in der Gruppe von Jean–Raymond Abrial für die konstruktive Spezifikation entwickelt (ab 1980).

Literatur

Ian Hayes (Ed.)
Specification Case Studies.
Prentice Hall 1987

Anforderungen an Spezifikationssprachen:

- rigoros
- reich im Formulierungsvorrat
- offen für Erweiterungen

Festlegung

Ein **Schema** ist eine Zusammenfassung von **Variablen–Deklarationen** (Signatur) und einem **Prädikat** (Bedingung) hierüber.

Beispiel

S

$$\left.\begin{array}{l} x : \mathbb{N} \\[1ex] y : \mathrm{seq}\ \mathbb{N} \end{array}\right\}\ \text{Tupel (Signatur) von S}$$

$$\left.\begin{array}{l} x = \#y \end{array}\right\}\ \text{Prädikat von S}$$

Symbolerklärung: seq = Liste (Sequenz)
\# = Länge

bzw. andere Notation (die wir nicht verwenden):

$S \widehat{=} [x : \mathbb{N}; y : seq \;\; \mathbb{N} | x = \# y]$

Anmerkung

Man kann Schemata als Typen auffassen, nämlich als die Menge aller Objekte, die dadurch beschrieben werden.

Beispiel

ST

st : SYM $\rightarrow$ VAL

Operationenbeschreibung

Wird ein Schema zur Beschreibung von Operationen verwendet, so gilt für einen Bezeichner x:

x	beschreibt den Zustand vorher
x'	beschreibt den Zustand nachher
$x?$	beschreibt die Eingabe
$x!$	beschreibt die Ausgabe

Beispiel

Update

st , st' : SYM $\rightarrow$ VAL

s? : SYM

v? : VAL

---

st' = st $\oplus$\{ s $\mapsto$ v \}

Symbolerklärung:

SYM	= Menge der Symbole
$\rightarrow$	= partielle Abbildung
VAL	= Menge der Werte
$\oplus$	= Überschreiben (overriding)
$\mapsto$	= s wird an v gebunden (Mapping-Konstruktion)

Konventionen (Schreibweisen in Z)

- $\triangle$ bezeichnet Schemata, die die Vor- und Nachzustände beschreiben.

$\triangle$ ST

$$st, st' : SYM \to VAL$$

- $\equiv$ bezeichnet Schemata, die Vor- und Nachzustände beschreiben und gleichsetzen.

$\equiv$ ST

$$st, st' : SYM \to VAL$$

$$st = st'$$

Beispiel

Look Up

$\equiv$ ST

s? : SYM

v! : VAL

---

s? $\in$ dom (st)

v! = st (s?)

Konventionen

! (in nachgestellter Form) bezeichnet Schemata, die Fehler erkennen.

Beispiel

NotPresent!
$\equiv$ ST

s? : SYM

rep! : REPORT

---

s? $\notin$ dom (st)

rep! = “Symbol not Present“

Symbolerklärung: *REPORT* = Menge der Fehlermeldungen

Schemaausdrücke

Man kann mit Schemata auch Ausdrücke bilden. Es werden die Deklarationsteile vereinigt und die Prädikate entsprechend verknüpft.

Beispiele

Mit

Success

rep! : REPORT

---

rep! = “OK“

entspricht dann

STLookUp = (LookUp $\wedge$ Success) $\vee$ NotPresent!

dem Schema

STLookUp

$$st\ ,\ st' : SYM \rightarrow VAL$$

$$s? : SYM$$

$$v! : VAL$$

$$rep! : REPORT$$

$$st = st'$$

$$(s? \in dom\ (st) \wedge v! = st\ (s?) \wedge rep! = \text{“OK“}) \vee$$

$$(s? \notin dom\ (st) \wedge rep! = \text{“Symbol not Present“})$$

Literatur

Caroll Morgan, Bernard Suffrin
Specification of the UNIX-file-system
in Ian Hayes (ed.)
Specification Case Studies
Prentice Hall, 1987

Beispiele

Sei

$$FILE \mathrel{\widehat{=}} seq\ BYTE$$

dabei bedeutet seq X:

$$seq\ X = \ s : N^+ \rightarrow X \mid (\exists n : N \bullet\ dom\ (s) = 1 \ldots n)$$

read File

file , file' : FILE

offset? : $I\!N$

length? : $I\!N$

data! : seq BYTE

---

file' = $file$

data! $= (1 \ldots length?) \lhd (file \bullet succ^{offset?})$

write File

file , file' : FILE

offset? : N

data? : seq BYTE

---

file' $= zero_{offset?} \oplus file \oplus (data? \circ pred^{offset?})$

Symbolerklärung:

$zero_{offset}$	=	Sequenz der Länge 'offset' aus Nullwerten
$\lhd$	=	Bereichseinschränkung
$\bullet$	=	Komposition

Kapitel 9

Pflichtenheft

Feststellung

Im Pflichtenheft werden die Anforderungen an das Produkt niedergeschrieben. Es wird vom Hersteller und Auftraggeber gemeinsam und im allgemeinen in natürlicher Sprache erstellt, enthält eventuell Diagramme und konkretere technische Angaben (im Anhang). Es beschreibt das **Was** und nicht das **Wie**.

Beispiel (mögliche Gliederung eines Pflichtenheftes)

Vorbemerkungen

- Sinn des Systems
- abzulösendes System
- Vorabreden

Randbedingungen

- existierende Systeme, die relevant sind
  - Erfassungssoftware
  - Informationssysteme
  - Buchhaltung
  - ...
- Festlegungen in Soft - / Hardware
  - Datenbanksystem
  - Betriebssystem
  - Rechnerklasse

Grundsätzliches

- Oberflächenverhalten
- Hilfesystem
- Datensicherung
- Menüverhalten

Aufgaben des Systems

- Auflistung der Einzelfunktionen → Hierarchiebäume
- Auflistung des Datenmodells → E–R–Diagramme
- Auflistung des Datenflusses → Datenflußdiagramme

Die Funktionen im einzelnen

- **Alle** verfügbaren Informationen über die Funktionen
  - Abrechnungsformeln
  - Plausibilitätsprüfung
  - interaktive Korrekturmöglichkeiten
  - Fehlererkennung

Die Daten im einzelnen

- **Alle** verfügbaren Informationen über die Daten
  - Relationenschemata
  - Dateiformate
  - Zugriffsart

Abläufe

- Benutzerinteraktion
  - Interaktionsdiagramme
  - Menübaum
- Operatoraufgaben

Abnahmebedingungen

- Zeitplan
- Performanz (= Leistung des Systems)
- Dokumentation
- Festlegung der Abnahmetests

⋮

viele Anlagen

Kapitel 10

Entity - Relationship - Modellierung

10.1 Grundlagen

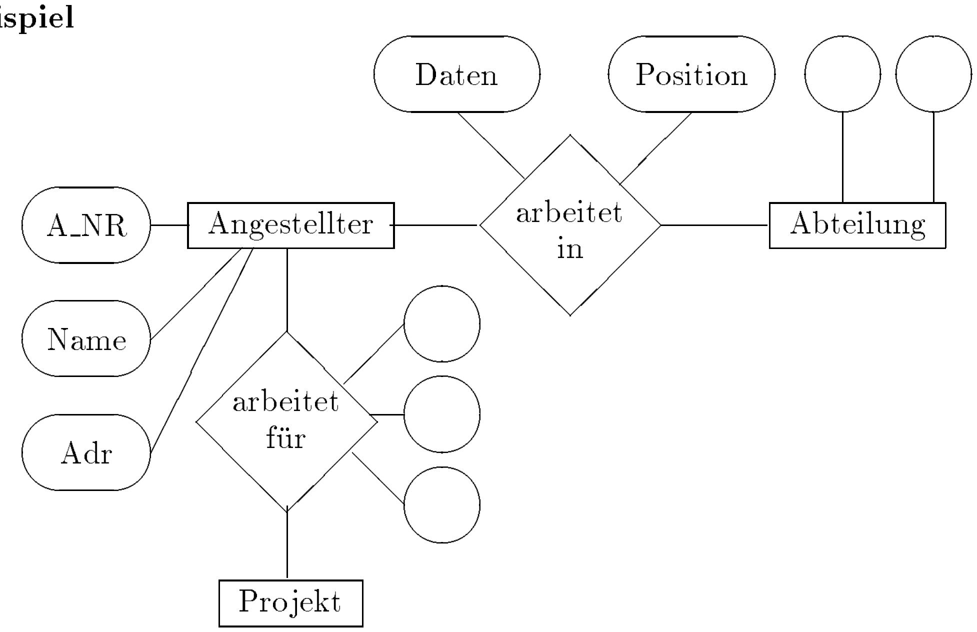
Historie

E — R - Modellierung wird C.P.Chen zugeschrieben, 1976

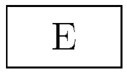
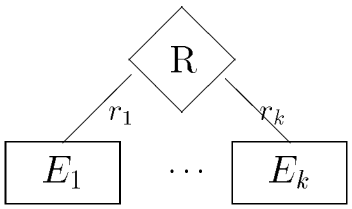
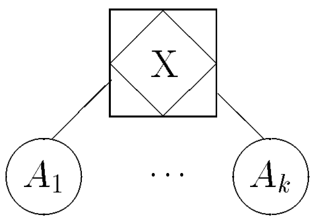
Feststellung

E - R - Modelle bestehen aus Objektklassen ("entity types") und Beziehungsklassen ("relationship types") mit zugehörigen Attributen.

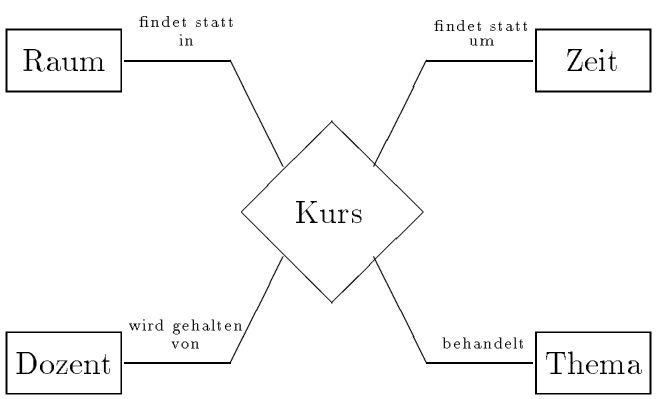
Beispiel



Bestandteile

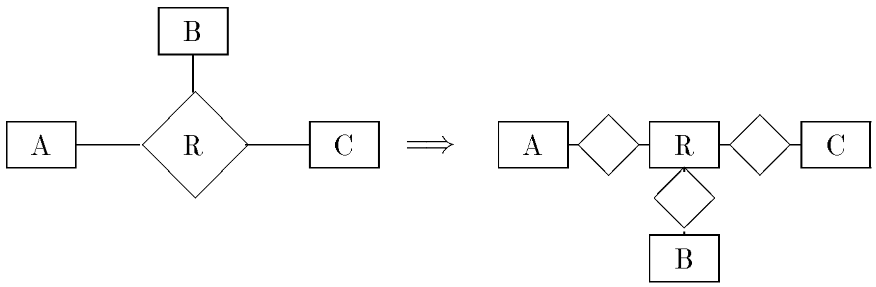
Schema		Instanz
	Entitytypen	Menge $\hat{E}$ (evtl. von Surrogaten)
	Beziehungstypen	$\{f : \{r_1, \ldots, r_k\} \to \cup_{i=1}^{k} \hat{E}_i$ $f(r_i) \in \hat{E}_i, 1 \le i \le k\}$
	Attribute mit Wertebereichen	$\{f : \{A_1, \ldots, A_k\} \to \cup_{i=1}^{k} dom(A_i) \mid$ $f(A_i) \in dom(A_i), 1 \le i \le k\}$

Beispiel



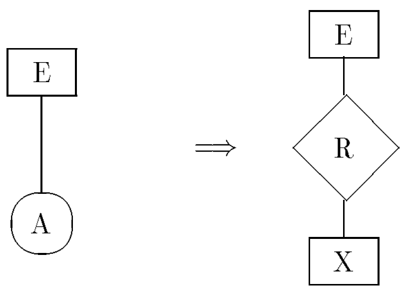
Bemerkung

Man kann n-stellige Relationen immer mit Hilfe von n binären Relationen und einem “Kett“ - Entitytyp darstellen.

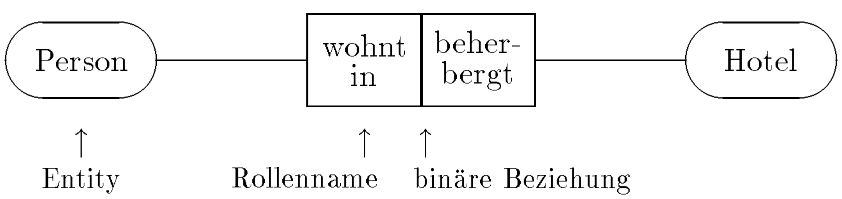


Bemerkung

Auch Attributzuordnungen können über binäre Relationen simuliert werden.

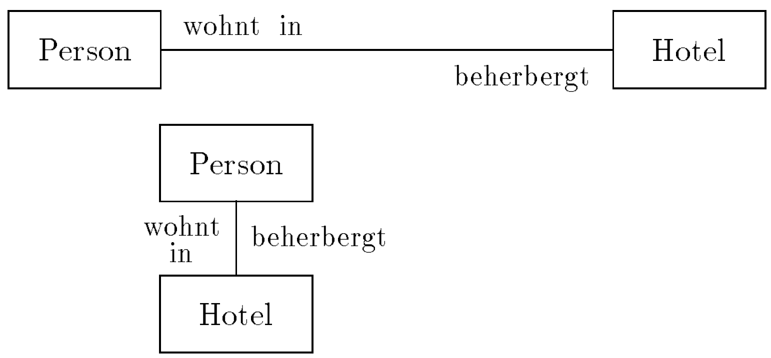


Beispiel (Nijssen)



NIAM-Netz (Nijssen Information Analysis Modell)

Beispiel (James Martin)

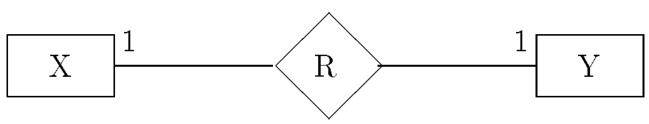


10.2 Integritätsbedingungen

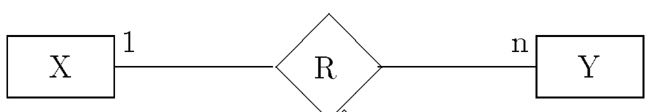
Feststellung

Manche E - R - Ansätze lassen zusätzliche Angaben zu, die die Menge der zugehörigen Instanzen einschränken.

Beispiele (Komplexität, Chen)

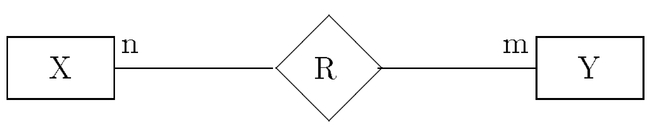


$(x_1, y_2), (x_2, y_2) \in \hat{R} \Rightarrow (x_1 = x_2 \wedge y_1 = y_2) \vee (x_1 \neq x_2 \wedge y_1 \neq y_2)$d.h. jedem x ist genau ein y zugeordnet und umgekehrt



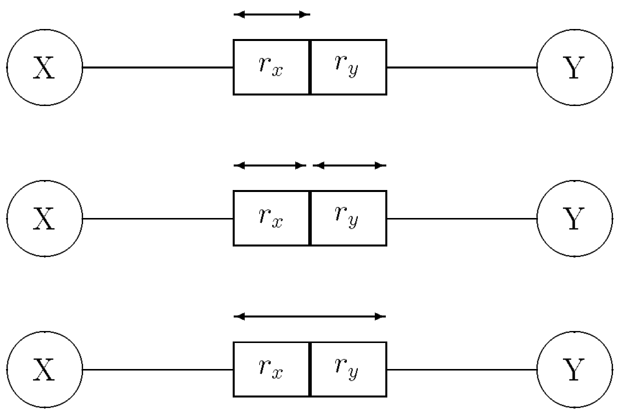
$(x_1, y), (x_2, y) \in \hat{R} \Rightarrow x_1 = x_2$

d.h. jedem y ist höchstens ein x zugeordnet.



keine Einschränkung

Beispiele (Komplexität, Nijssen)

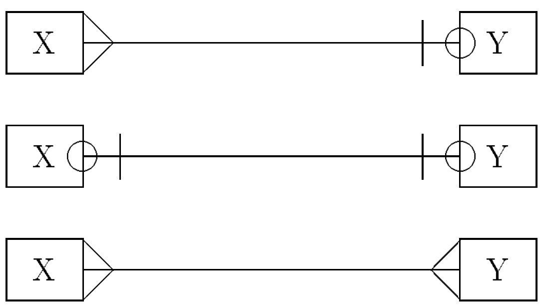


$\leftrightarrow$ = n:1

$\leftrightarrow\leftrightarrow$ = 1:1

$\longleftrightarrow$ = n:m

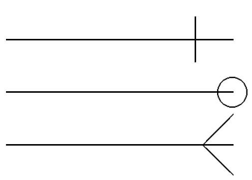
Beispiel (Komplexität, Martin)



n : 1

1 : 1

n : m

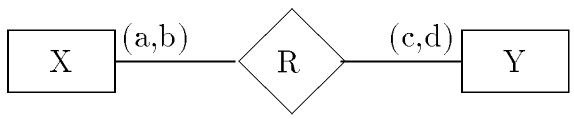


“1“

“0“

“n“

Beispiel ((min,max) - Notation)

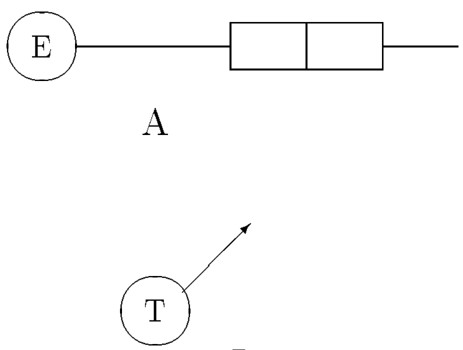


$\forall y \in \hat{Y} : a \leq$
$\mid \{x \in \hat{X} \mid (x,y) \in \hat{R}\} \mid \leq b$

(0,n) $\hat{=}$ (0,∞)

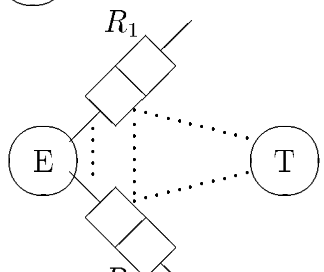
Weitere Integrationsbedingungen (Nijssen)

1.) Totalität (total constraint)



Π ($\hat{\mathrm{R}}$) = E

$\{\ x \in \hat{E} \mid \exists y : (x,y) \in \hat{R}\}$

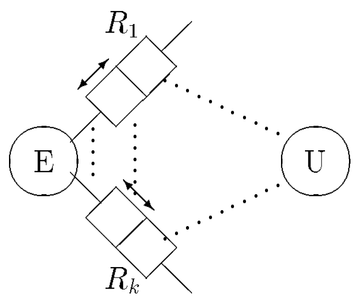


$\Pi(\hat{R} \cup \ldots \cup \hat{R}) = \hat{E}$

oder nach James Martin

1 oder n, aber nicht Null

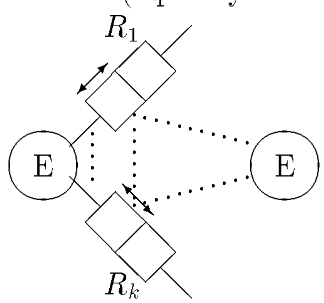
2.) Eindeutigkeit (uniqueness constraint)



$$e, e' \in \hat{E} \wedge \exists_{i,e_i} : ((e, e_i) \in R_i \wedge (e', e_i) \in R_i)$$

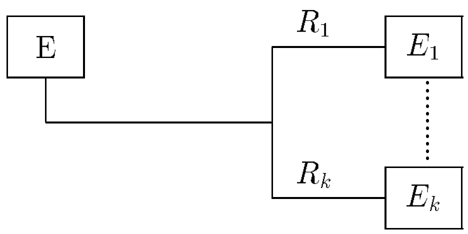
$$\Longrightarrow e = e'$$

3.) Gleichheit (equality constraint)

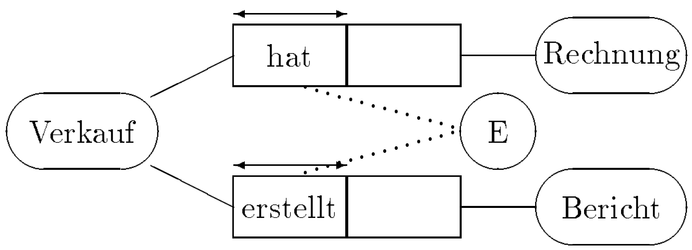


$$\forall i, j\ 1 \leq i, j \leq n, i \neq j : \Pi_1(R_i) = \Pi_1(R_j)$$

oder nach James Martin

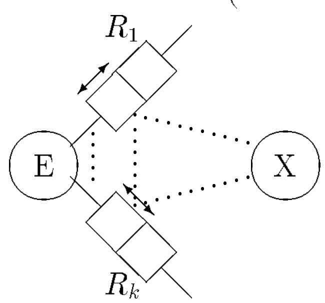


zum Beispiel:



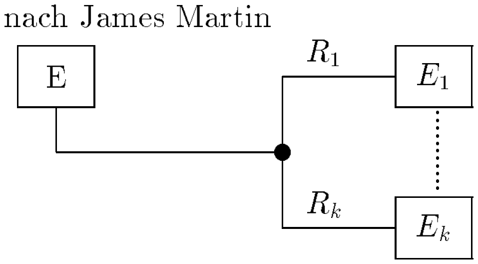
Das eine nicht ohne das andere

4.) Ausschließlichkeit (exclusive constraint)

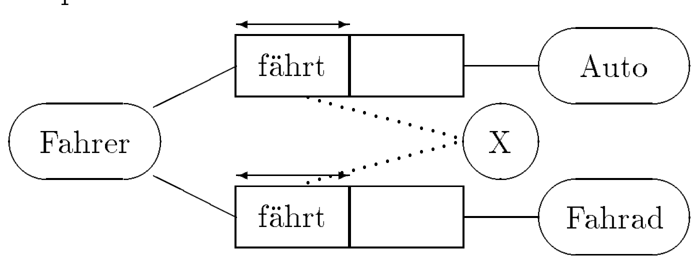


$\forall i, j \; 1 \leq i, j \leq k, i \neq j : \Pi_1(R_i) \wedge \Pi_1(R_j) =$

oder nach James Martin



zum Beispiel:



entweder / oder

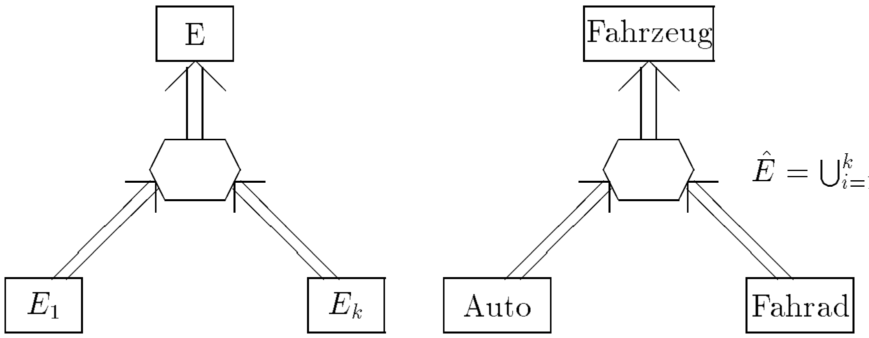
10.3 Erweiterungen

Feststellung

Weitere Integritätsbedingungen werden durch Einführung neuer Knotenformen ausgedrückt.

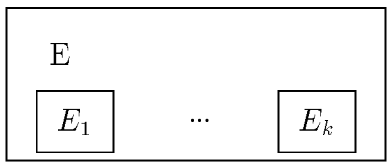
Beispiel

Generalisierung

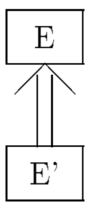


$\hat{E} = \bigcup_{i=1}^{k} \hat{E}_i$

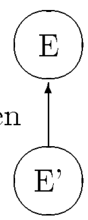
andere Notation



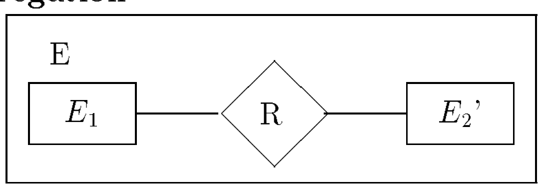
Subtyp



$\hat{E}' \subset \hat{E}$ bzw. bei Nijssen

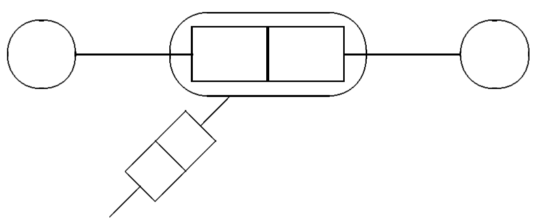


Aggregation

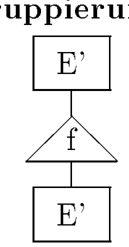


Verwendung von $\hat{\mathrm{R}}$ als Entitymenge

bzw. bei Nijssen:



Gruppierung (Assoziation)



$\hat{\mathrm{f}} : \hat{\mathrm{E}} \rightarrow \hat{\mathrm{E}}$' *surjektiv, total*

bzw.

E' ist Partition von E

Kapitel 11

Datenflußdiagramme

11.1 Yourdon / de Marco

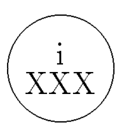
Literatur

T. de Marco
Structured Analysis and System Specification
Englewood Cliffs, Prentice Hall, 1978

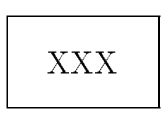
Feststellung

Datenflußdiagramme (DFD's) sind gerichtete Graphen.

Zeichen

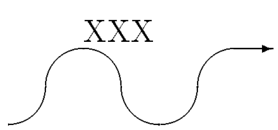


Prozeß
i = Prozeßnummer, XXX = Name (Verb + Objekt)



Schnittstelle (Interface) (→ z.B. Kunden, Waren etc.)
(→ *Teile der realen Welt*)
XXX = Bezeichner

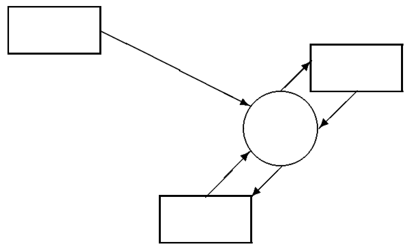
Dateie (interner Speicher)



Datenfluß

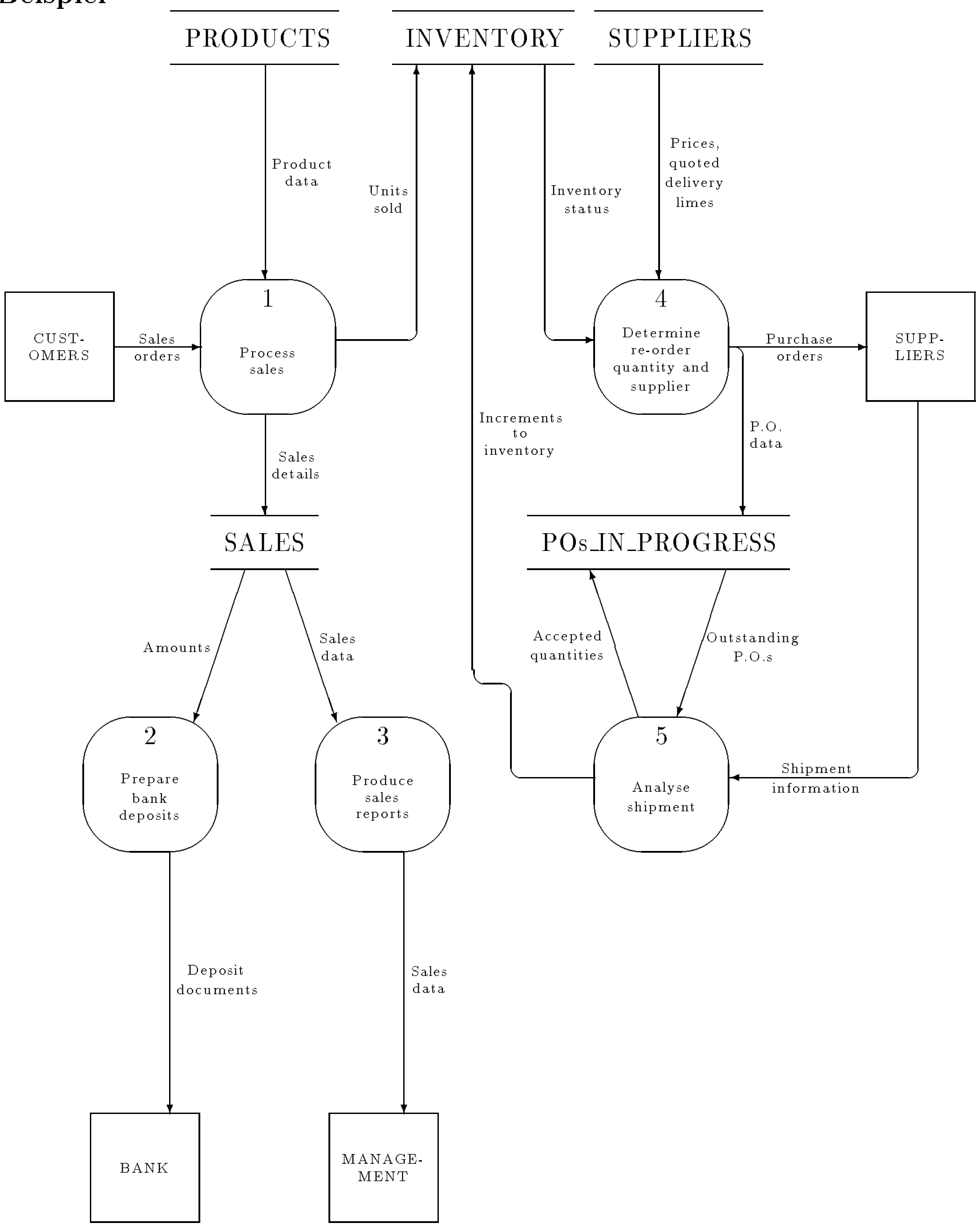
Verfeinerung

- Zuerst wird das System durch ein “Kontextdiagramm“ beschrieben.



- Prozesse werden durch eigene “Detaildiagramme“ mit maximal 7 Prozeßknoten verfeinert (dabei müssen die Kanten am Rande mit denen des Grobknotens übereinstimmen $\Longrightarrow$ “Balancierung“).

Beispiel


PRODUCTS
INVENTORY
SUPPLIERS
Product data
Units sold
Inventory status
Prices, quoted delivery limes
CUST-OMERS
Sales orders
1
Process sales
4
Determine re-order quantity and supplier
Purchase orders
SUPP-LIERS
Increments to inventory
P.O. data
Sales details
SALES
POs_IN_PROGRESS
Amounts
Sales data
Accepted quantities
Outstanding P.O.s
2
Prepare bank deposits
3
Produce sales reports
5
Analyse shipment
Shipment information
Deposit documents
Sales data
BANK
MANAGE-MENT

Datenstrukturen

Dateiknoten werden durch Namen identifiziert und in einem **Data Dictionary** (=DD) beschrieben.

Die Struktur der Daten ist regulär

		in Pascal
+	Sequenz (=Konkatenation)	*Record*
[|]	Alternative	*variantes Record*
( )	Option	
{ }	Iteration	*Array, File*

Beispiel

Invoice = *Invoice Number*
+ *Customer Name*
+ *Customer Adress*
+ (*Special Instructions*)
+ { *Product* + *Quantity* + *Unit Price* }

Minispecs

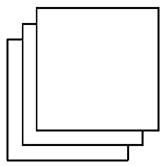
Prozeßknoten werden durch Namen identifiziert und durch **Minispecs** annotiert (z.B. durch Pseudocode, Entscheidungstabellen).

11.2 Gane/Sarson

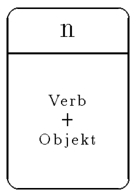
Literatur

C. Gane, T. Sarson:
Structured System Analysis: Tools and Techniques
Englewood Cliffs, New York. Prentice Hall, 1979.

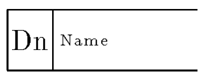
Symbole



Schnittstelle, externes Entity



Prozeß (n = Nummer)

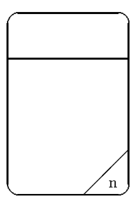


Datei (n = Nummer)

↖ achsenparallel

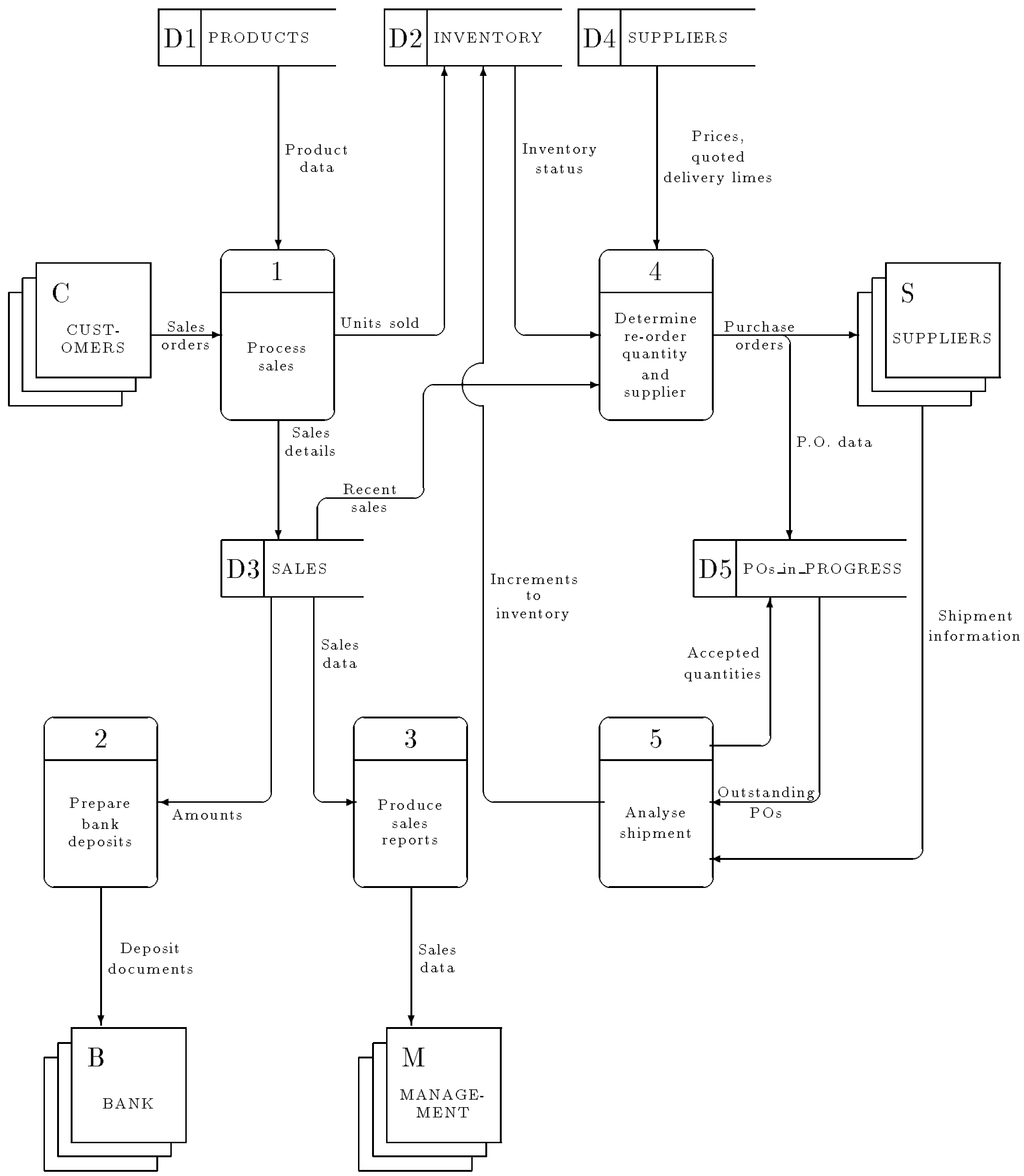
Datenfluß

Knoten können auch mehrfach gezeichnet werden:



n = Anzahl der Kopien

Beispiel



D1 PRODUCTS
D2 INVENTORY
D4 SUPPLIERS
Product data
Inventory status
Prices, quoted delivery limes
C CUST-OMERS
Sales orders
1
Process sales
Units sold
4
Determine re-order quantity and supplier
Purchase orders
S SUPPLIERS
Sales details
Recent sales
P.O. data
D3 SALES
Increments to inventory
D5 POs_in_PROGRESS
Shipment information
Sales data
Accepted quantities
2
Prepare bank deposits
Amounts
3
Produce sales reports
5
Analyse shipment
Outstanding POs
Deposit documents
Sales data
B BANK
M MANAGE-MENT

Datenstrukturen

A B C	entspricht	Sequenz
[ ]	entspricht	Option
$*$	entspricht	Iteration

Beispiel

Invoice
  Invoice Number
  Customer Name
  Customer Adress
  [*Special Instructions*]
  Itemline $*$
    Product Code
    Unit Price
    $\vdots$

11.3 Datenflußpläne

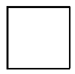
Feststellung

Datenflußabläufe in Organisationen werden nach der DIN 66001 durch **Datenflußpläne** beschrieben.

Symbole

Prozesse (Tätigkeiten)

 allgemein

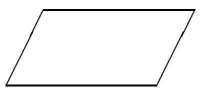
 Hilfsfunktion

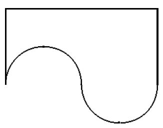
 Eingreifen per Hand

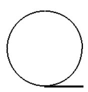
 Eingeben per Hand

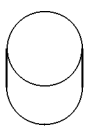
⋮

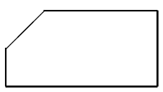
Dateien (Datenträger)

 allgemein

 Schriftstück (lesbar / Papier)

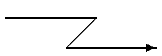
 Magnetband

 Magnetplatte

 Lochkarte

Datenflüsse

 allgemein

 Datenfernübertragung

 Datentransport per Hand

11.4 Datenstrukturbeschreibungen

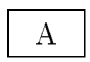
Feststellung

Datenstrukturen entsprechen regulären Ausdrücken. Sie sind darstellbar z.B.

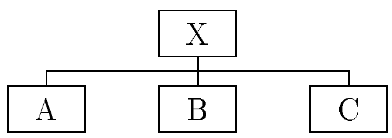
- nach Yourdon/de Marco (siehe oben)
- nach Gane/Sarson (siehe oben)
- nach Jackson
- nach Warnier/Orr.

Jackson - Bäume

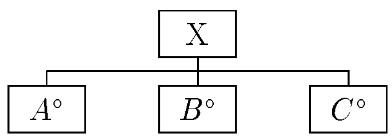
atomare Komponenten



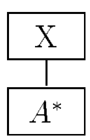
Sequenz



Alternative



Iteration



Jackson–Pseudocode (schematische Logik)

```
atomare Komponente: A
Sequenz:            x: seq
                          A;
                          B;
                          C
                    x: end
Alternative:        x: select
                          A
                       or
                          B
                       or
                          C
                    x: end
Iteration:          x: iter
                          A
                    x: end
```

Warnier/Orr–Diagramme

Sequenz $\quad X \begin{cases} A \\ B \\ C \end{cases}$

Alternative $\quad X \begin{cases} A \\ \oplus \\ B \\ \oplus \\ C \end{cases}$

Iteration $\quad X \left\{ {}_{*}A \right.$

11.5 Kontrollflußbeschreibungen

Feststellung

In den Minispecs wird das Ein–/Ausgabeverhalten der Prozesse festgelegt, z.B.

- natürlichsprachlich
- durch Vor–/Nachbedingungen (siehe oben)
- durch Entscheidungstabellen

- durch Pseudocode
- durch Struktogramme
- durch Jackson–Bäume (siehe oben)
- durch Flußdiagramme

⋮

Entscheidungstabellen

Entscheidungstabellen nach DIN 66261 sind Darstellungsmittel für Fallunterscheidungen.

Name								
Bedingung 1	j	.	.	.	.	.	.	.
Bedingung 2	j	.	.	.	.	.	.	.
Bedingung 3	n	.	.	.	.	.	.	.
Aktion 1	X	.	.	.	.	.	.	.
Aktion 2	-	.	.	.	.	.	.	.
Aktion 3	X	.	.	.	.	.	.	.

Mögliche Markierungen

Bedingungen

j	$\widehat{=}$	ja
n	$\widehat{=}$	nein
#	$\widehat{=}$	nicht sinnvoll
-	$\widehat{=}$	irrelevant

Aktionen

X	$\widehat{=}$	ausführen
-	$\widehat{=}$	nicht ausführen

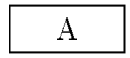
Anmerkung

Durch die disziplinierte Bearbeitung von Fallunterscheidungen ist eine Überprüfung von Vollständigkeit, Redundanzfreiheit und Widerspruchsfreiheit leicht möglich.

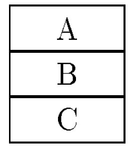
Struktogramme

auch: Nassi-Shneiderman-Diagramme
beschreiben strukturierte Programme (reguläre Programmstrukturen).

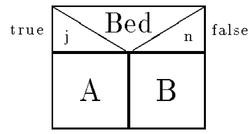
atomare Aktionen



Sequenz von Aktionen



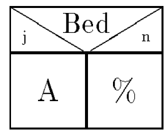
Alternative



if then else

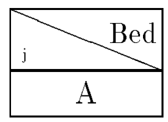
Option (Variante der Alternative)

richtig:

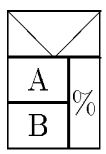


if then

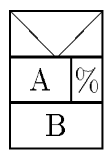
falsch:



denn so kann sowohl

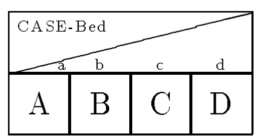


als auch

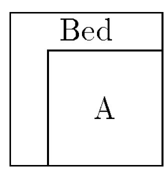


entstehen

Fallunterscheidung (CASE)

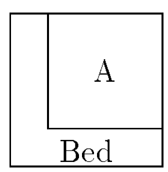


Iteration



while

bzw.



repeat

Flußdiagramme (=Programmablaufpläne)
Flußdiagramme sind eine nicht strukturierte Kontrollflußdarstellung nach DIN 66001.

11.6 SADT

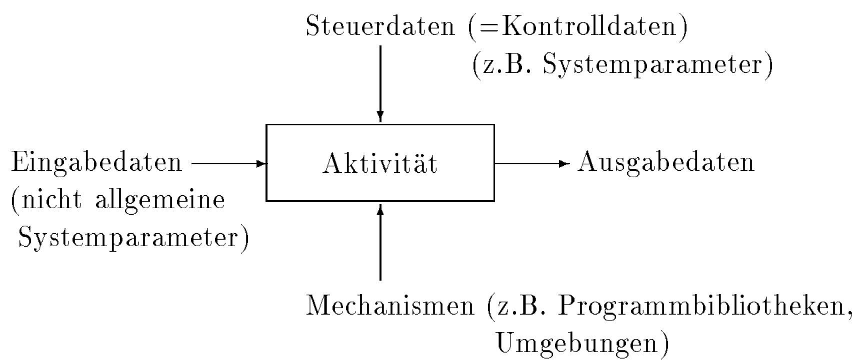
Literatur
D. T. Ross: *Structured Analysis: a language to communicate ideas*, IEEE-SE 3 (1977,1), S.16-34.

Anmerkung
SADT (= "Structured Analysis and Design Technique") ist zu unterscheiden von **Structured Analysis** bei Yourdon/de Marco.

Ansatz
Unterscheidung zwischen

- Aktivitätsdiagrammen (Prozesse als Knoten) und
- Datendiagrammen (Daten als Knoten)

Aktivitätsbox
Ein Aktivitätsdiagramm besteht aus Aktivitätsboxen:

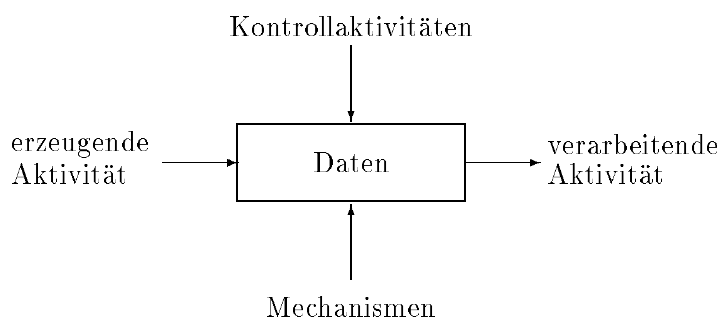


Eingabe– und Systemdaten unterscheiden sich mehr oder weniger willkürlich.

Feststellung
Aktivitätsboxen können durch weitere (Aktivitäts-) Diagramme verfeinert werden. Die Ränder müssen zueinander passen (Balancierung). Diagramme (außer dem Kontextdiagramm=äußerstes Diagramm) müssen 3 bis 6 Boxen enthalten.

Datenbox

Datendiagramme bestehen aus Datenboxen:



Feststellung

Idealerweise erfolgt eine Beschreibung durch Aktivitäts– und Datendiagramme.

11.7 HIPO

Literatur

J. F. Stay: *HIPO and integrated program design*, IBM Systems Journal, 1976

Feststellung

HIPO = "Hierachy + Input–Prozess–Output"

Beschreibung besteht aus:

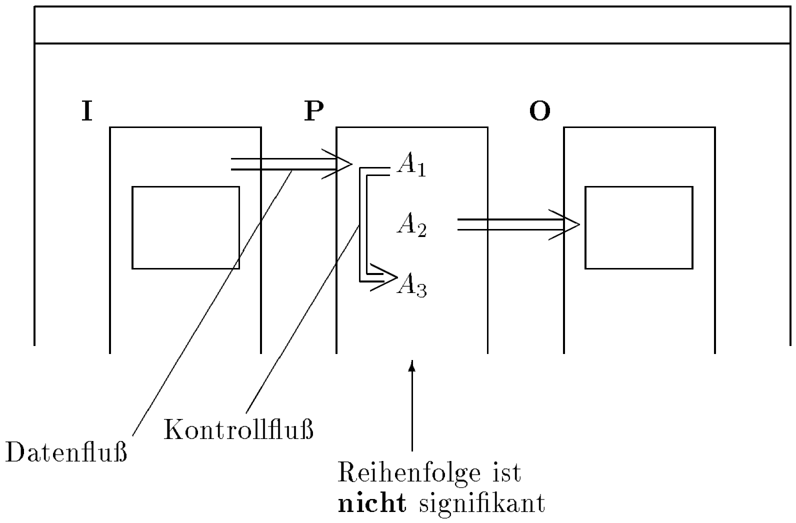
1. Hierarchiediagramm (für die Enthaltensbeziehung) (**VTOC** = **v**isual **t**able **o**f **c**ontents)
2. I–P–O–Diagramm (= Input–Prozeß–Output–Diagramm)

Ansatz

Aufteilung in:

- Eingabedaten (= **I**nput) (links)

- Verarbeitung (= **P**rozess) (Mitte)
- Ausgabedaten (= **O**utput) (rechts)



Anmerkung

Ein– und Ausgabedaten kommen in den O–Teil des Diagramms.

$\boxed{\text{Anm.}}$

Kapitel 12

Automatenmodelle

12.1 Grundlagen

Anmerkung

(Endliche) Automaten sind hilfreich zur Beschreibung von ereignisgetriebenen Systemen mit innerem Zustand.

Varianten

DFA, NFA, ε–NFA,
Mealy–Automat (DFA mit Ausgabe bei den Zustandsübergängen (Kanten))
Moore–Automat (DFA mit Ausgabe in den Zuständen (=Knoten))

Diese Varianten sind ergänzbar um beliebige Aktionen (z.B. Zuweisungen) an den Kanten.

Darstellungsformen

- Zustandsübergangsdiagramme
- Zustandsübergangsmatrix

Beispiel

Knoten : Zustände
Kanten : Zustandsübergänge

Kantenbeschriftungen:

- Eingabegrößen (Eingabetoken, –signale, –ereignisse)
- Ausgabegrößen (Ausgabetoken, –signale, –ereignisse)

Anmerkung

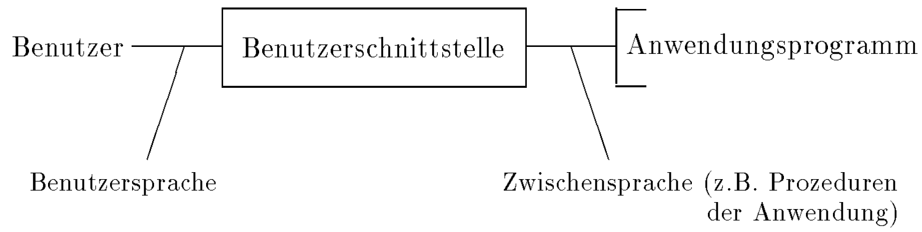
Es wird nicht immer ein Start–/Endzustand ausgezeichnet.

12.2 Interaktionsdiagramme nach Jacob

Literatur

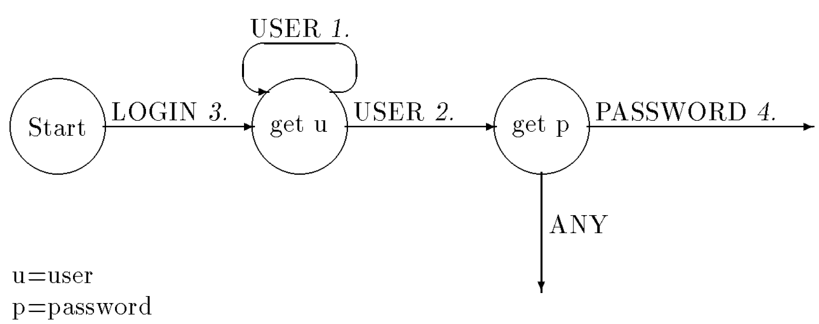
Robert J. K. Jacob,
Using formal specifications in the design of a human–computer interface,
Communications ACM 26 (1983,4), S.259–264

Kontext



Man beschreibt “Kommandos” der Benutzersprache mit Hilfe der Zwischensprache.

Beispiel



1. **cond** *:* **not** *EXISTS_USER($USER)*
   resp *: "incorrect user number – reenter"*

   (cond=condition, resp=response, $USER:Wert des Tokens EXISTS_USER: Prozedur der Zwischensprache)
2. **cond** *: EXISTS_USER($USER)*
   resp *: "enter password"*
3. **resp** *: "enter user number"*
4. **cond** *: $PASSWORD=GETPASSWORD($USER)*
   act *: CREATE_SESSION($USER, $PASSWORD)*

Übersicht

Knoten : Zustände, charakterisiert durch Namen
Kanten : Operationen, ausgelöst durch die Benutzer, beschrieben durch

- Input–Token (vorgegeben durch Token–Klassen–Beschreibungen)
- **cond** : Bedingung (in Zwischensprache formuliert)
- **resp** : Antwort (Text)
- **act** : Aktion (in Zwischensprache formuliert)

Ergänzungen

Es können globale Variablen verabredet werden. Die Werte der Token werden durch $ angesprochen.

ANY markiert Übergänge, die nur traversiert werden, wenn keine andere Möglichkeit besteht.

Ist kein Zustandsübergang definiert, so wird eine globale Fehlerroutine aufgerufen.

Teil IV

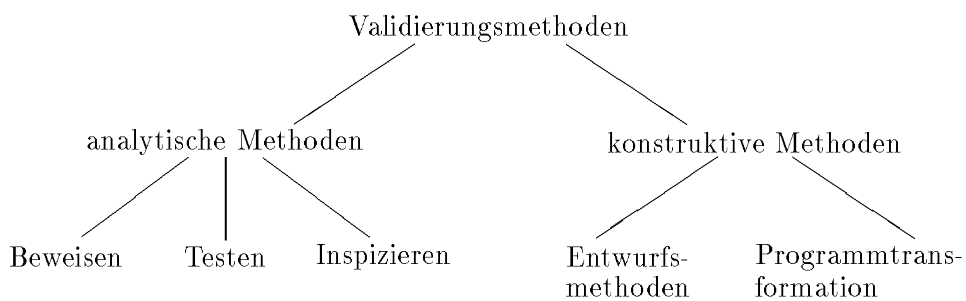
Implementation

Kapitel 13

Validieren

Feststellung: Unter **Validieren** versteht man den Nachweis, ob das von einer Programmiersprache implementierte Modul die geforderten Eigenschaften aufweist.

Übersicht



13.1 Verifizieren

13.1.1 Zusicherungsmethode von Hoare

Idee

Ergänze Programm durch Zusicherung, d.h. Prädikate über den Programmvariablen, und versuche durch Anwendung von Beweisregeln für die Anweisungskonstrukte der verwendeten Programmsprache aus der Eingangszusicherung die Ausgangszusicherung herzuleiten.

Schema: $\{P\}\ S\ \{Q\}$

Historie

R. W. Floyd:
Assigning Meanings to Programs
in: Proc. Symp. in Applied Math.
Americ. Math. Soc., 19 (1967), S.19–32.

C. A. R. Hoare:
An Axiomatic Basis for Computer Programming
CACM 12 (10,1971), S.576-580,583.

Literatur

S. Alagic, M. A. Arbib:
The Design of Well Structured and Correct Programs,
Springer, 1978.

D. Gries:
The Design of Programming, Springer, 1983.

$$
\begin{array}{llll}
S: & & x := l & \text{(Zuweisung)} \\
 & | & \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 & \text{(Alternative)} \\
 & | & \textbf{while } b \textbf{ do } s & \text{(Iteration)} \\
 & | & s_1 \; s_2 & \text{(Sequenz)}
\end{array}
$$

Beweisregeln

1. **Anweisungsaxiom**

   $\{P_{x \leftarrow e}\}\ x := e\ \{P\}$

   $P_{x \leftarrow e}$ ist das Prädikat, indem jedes freie Vorkommen von x in P durch e ersetzt wird.

   Beispiel 1
   $\{x + y = 10\}$
   $z := x + y;$
   $\{z = 10\}$

   Beispiel 2
   $\{x - y \geq 0\}$
   $x := x - y;$
   $\{x \geq 10\}$

2. **Syntaxorientierte Inferenzregeln**

(a) **Alternative**

$$\frac{\{b \wedge P\}\ S_1\ \{Q\}\quad und\quad \{\neg b \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

(b) **Iteration**

$$\frac{\{b \wedge P\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{P \wedge \neg b\}}$$

(c) **Sequenz**

$$\frac{\{P\}\ S_1\ \{Q\}\quad und\quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1,\ S_2\ \{R\}}$$

3. **Logische Inferenzregel**

$$\frac{\{P\}\ S\ \{Q\}\quad und\quad Q \Rightarrow R}{\{P\}\ S\ \{R\}}$$

oder

$$\frac{\{P\}\ S\ \{Q\}\quad und\quad R \Rightarrow P}{\{R\}\ S\ \{Q\}}$$

Anmerkung (partielle Korrektheit)

{P} S {Q} heißt:
Ist P wahr, so ist nach der Ausführung von S, falls S terminiert auch Q wahr. Die Termination von S muß getrennt nachgewiesen werden.
(totale Korrektheit: partielle Korrektheit + Termination)

Beispiel (Vertauschen von zwei natürlichen Zahlen)

$\{x = a\ und\ y = b\}$
$x = x - y;$
$\{y = b\ und\ x = a - b\}$
$y := x + y;$
$\{x = a - b\ und\ y = a\}$
$x := y - x;$
$\{x = b\ und\ y = a\}$

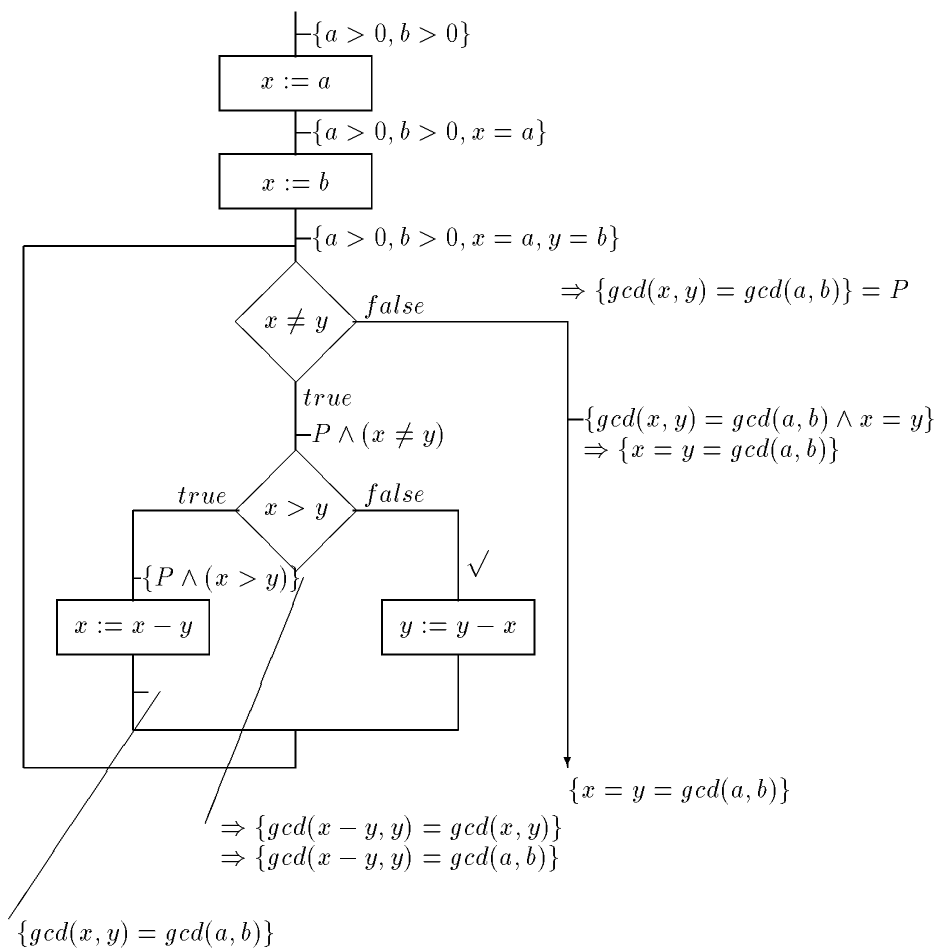
Beispiel (Größter gemeinsamer Teiler (= gcd) von zwei Zahlen)

Es gelten folgende Axiome:

1. $u = gcd(u, u)$
2. $gcd(u, v) = gcd(v, u)$
3. $u > v \Rightarrow gcd(u - v, v) = gcd(u, v)$
4. $u = n * v \Rightarrow gcd(u, v) = v$

Der nun folgende Algorithmus mit Vor- und Nachbedingung wird in diesem Beispiel graphisch verifiziert.

$\{a > 0, b > 0\}$
$x := a;$
$y := b;$
while $x \neq y$ **do**
  if $x > y$ **then** $x := x - y$
  else $y := y - x$
{ x=y=gcd(a,b)}

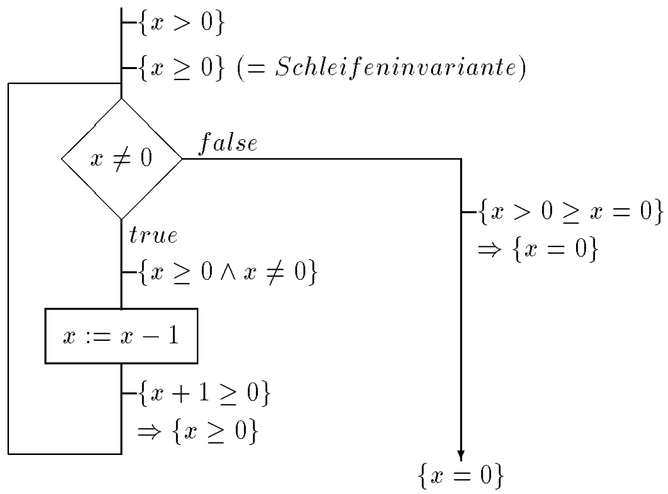


Anmerkung

Mit der Zusicherungsmethode wird nur "partielle Korrektheit", d.h. im Falle der Termination bewiesen.

Z.B.:

$\{x > 0\}$
while $x \neq 0$ **do**
    $x := x - 1;$
end;



Da der Algorithmus durch die Vorbedingung $\{x > 0\}$ terminiert, kann man die Zusicherungsmethode anwenden.
Bei der Vorbedingung $\{x = 0\}$ (=kein Durchlauf) oder $\{x < 0\}$ (=Endlosschleife) kann man die Zusicherungsmethode **nicht** anwenden.

Anm.

13.1.2 Symbolische Ausführung

Idee

- Die Werte von Variablen werden in symbolischer Form verwaltet.
- Ein Programm wird mit den symbolischen Werten durchgerechnet.
- Man versucht hier durch die symbolische Programmausführung die Ausgangszusicherung aus der Eingangszusicherung herzuleiten.
- Partielle Korrektheit.

Literatur

S. L. Hantler, J. C. King:
An Introduction to Proving the Correctness of Programs,
Computing Surveys 8 (3, 1976), S.331–353.

Vorgehensweise

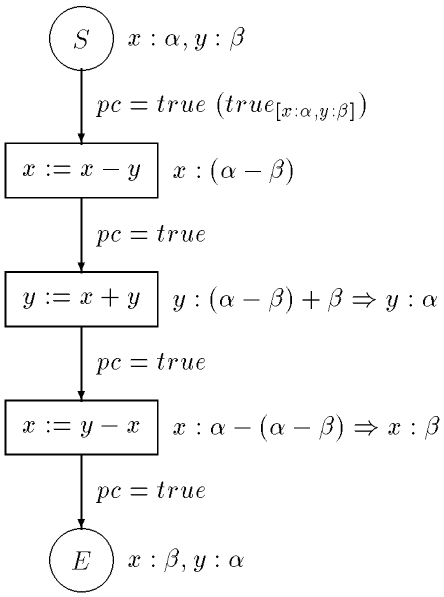
1. Erstellen eines Ausführungsbaumes (mit einem ausgesuchten-Wurzelknoten und einer Menge von zusätzlichen Endblättern).
2. Annotieren des Ausführungsbaumes
   - Knoten mit Wertgleichungen
   - Kanten mit Pfadbedingungen (pc=path condition)

   (An den Startknoten werden die Variablenbelegungen eingetragen; für die vom Startknoten ausgehende Kante geben wir "pc-=Eingangszusicherung" an)
3. Für alle Endblätter ist zu zeigen:

   Aus der Pfadbedingung auf der einlaufenden Kante folgt die Ausgangsbedingung.

Beispiel (Vertauschen von zwei Werten)

$ASSUME(true)$	Eingangszusicherung
$x := x - y;$	
$y := x + y;$	
$x := y - x;$	
$PROVE(x' = y \ und \ y' = x)$	Ausgangszusicherung

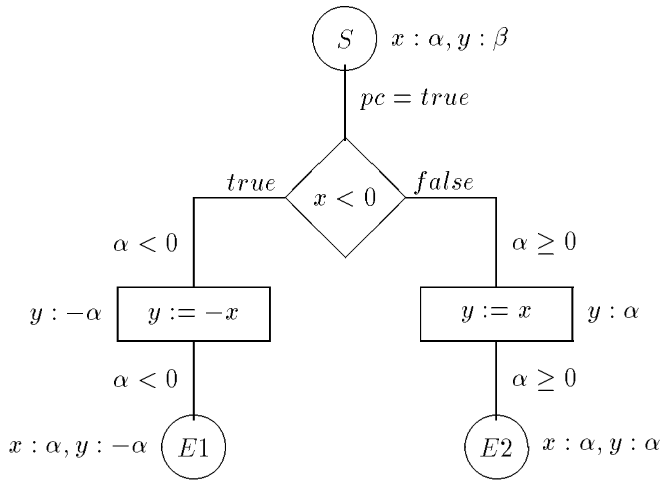


zu zeigen:

$$
\begin{aligned}
& true \;\Rightarrow\; x' = y \wedge y' = x_{[x':\beta, y':\alpha, x:\alpha, y:\beta]} \\
\Leftrightarrow\; & true \;\Rightarrow\; \beta = \beta \wedge \alpha = \alpha \\
\Leftrightarrow\; & true \;\Rightarrow\; true \\
\Leftrightarrow\; & true \surd
\end{aligned}
$$

Beispiel (Absolutbetrag von zwei Zahlen)

$ASSUME(true)$
if $x < 0$
  then $y := -x$
  else $y := x$
$PROVE\Big((y' = x \ oder\ y' = -x)\ und\ y' \geq 0\Big)$



$E1$ zu zeigen:

$$
\begin{array}{rl}
 & \alpha < 0 \;\Rightarrow\; (y' = x \vee y' = -x) \wedge y' \geq 0_{[x':\alpha, y':-\alpha, x:\alpha, y:\beta]} \\
\Leftrightarrow & \alpha < 0 \;\Rightarrow\; (-\alpha = \alpha \vee -\alpha = -\alpha) \wedge -\alpha \geq 0 \\
\Leftrightarrow & \alpha < 0 \;\Rightarrow\; -\alpha \geq 0 \\
\Leftrightarrow & true
\end{array}
$$

$E2$ zu zeigen:

$$
\begin{array}{rl}
 & \alpha \geq 0 \;\Rightarrow\; (y' = x \vee y' = -x) \wedge y' \geq 0_{[x':\alpha, y':\alpha, x:\alpha, y:\beta]} \\
\Leftrightarrow & \alpha \geq 0 \;\Rightarrow\; (\alpha = \alpha \vee \alpha = -\alpha) \wedge \alpha \geq 0 \\
\Leftrightarrow & \alpha \geq 0 \;\Rightarrow\; \alpha \geq 0 \\
\Leftrightarrow & true
\end{array}
$$

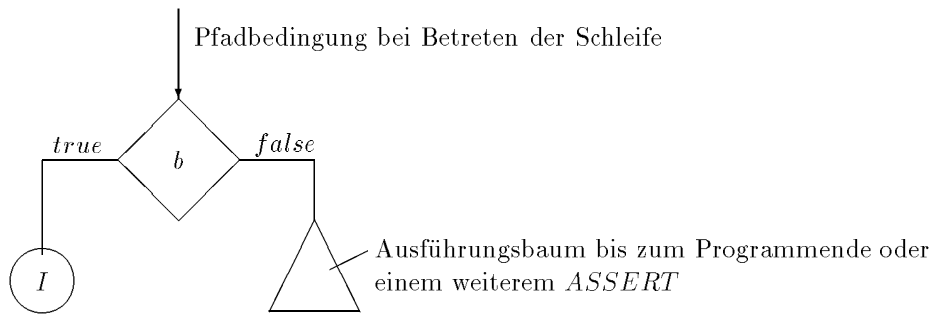
Hinweise

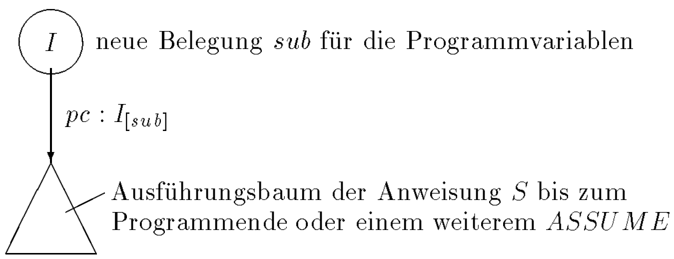
1. Die Pfadbedingung pc auf einer Kante des Ausführungsbaumes ist die Konjunktion aller Pfadbedingungen auf dem Weg von der Wurzel.
2. Der aktuelle Wert einer Variablen x ist die letzte Eintragung zu x auf diesem Weg.

Sonderbehandlung von Schleifen (wegen der unendlichen Ausführungsbäume)
Vorgehensweise:

Für eine Schleife **while** b **do** S werden:

1. eine Schleifeninvariante I bestimmt und mit $ASSERT(I)$ angegeben,
2. zwei Ausführungsbäume (getrennt voneinander) verifiziert.

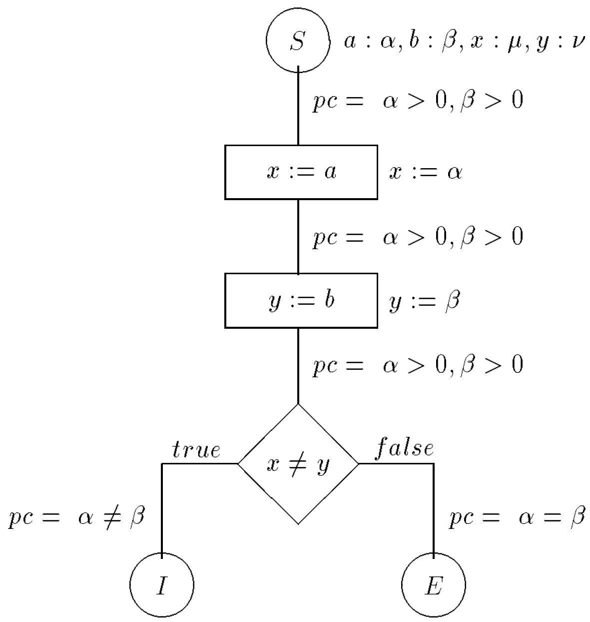




Beispiel (Größter gemeinsamer Teiler(=gcd))

$ASSUME(a > 0, b > 0)$
$x := a;$
$y := b;$
while $x \neq y$ **do**
  $ASSERT\Big(gcd(x, y) = gcd(a, b)\Big)$
  if $x > y$ **then** $x := x - y$
  else $y := y - x$
$PROVE\Big(x = y \wedge gcd(a, b)\Big)$

Ausführungsbaum 1



(I) zu zeigen:

$\alpha > 0 \wedge \beta > 0 \wedge \alpha \neq \beta$

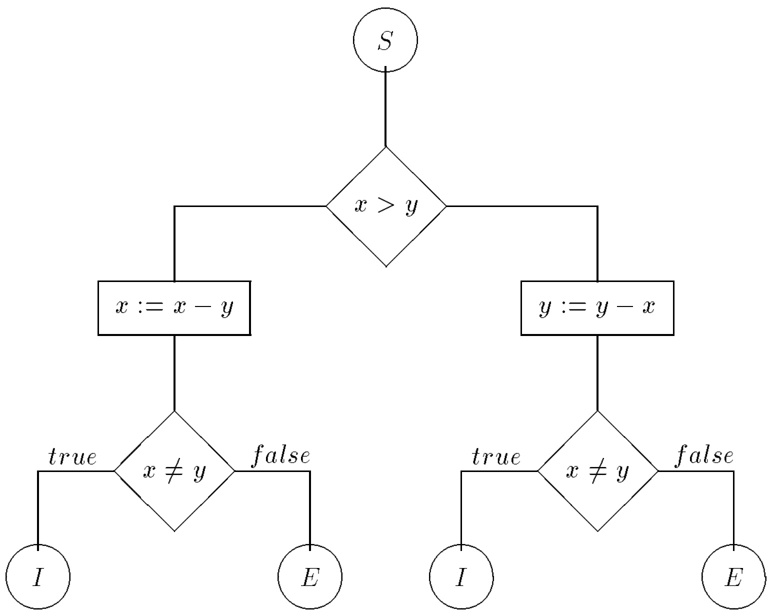
$\Rightarrow gcd(x, y) = gcd(a, b)_{[x:\alpha, y:\beta, a:\alpha, b:\beta]}$

(E) zu zeigen:

$\alpha > 0 \wedge \beta > 0 \wedge \alpha = \beta$

$\Rightarrow x = y = gcd(a, b)_{[x:\alpha, y:\beta, a:\alpha, b:\beta]}$

Ausführungsbaum 2



13.2 Testen

Skizze

Testen
- Stichprobentest
- ausschöpfender Test (≙Verifikation)

Testdaten–Erstellung
- Black–Box–Test
- strukturierter Test

Anmerkung

Testdaten müssen in einer Testbibliothek (PEB = Programmentwicklungsbibliothek) für “Regressionstest” (= Wiederholung eines Tests nach einer Änderung) aufbewahrt werden.

Auswahl von Testdaten (Black-Box-Test)

- zufallgesteuert
- wahrscheinlichkeitsgesteuert (erfordert Kenntnisse über die Anwendung)
- aus der Spezifikation hergeleitet
- gemäß der Faustregel:
  - einige Normalfälle
  - viele Rand- bzw. Extremfälle
  - mehrere Fehlerfälle

Beispiel

var $s : a \dots b$;
Normalfälle: $a \leq x \leq b$
Extremfälle: $x = a, x = b$
Fehlerfälle: $x < a, x > b$

Literatur (zu strukturiertem Test)
J. C. Huang:
An approach to program testing,
ACM Computing Survey 7 (1975,3), S.113-119

Idee (bei strukturiertem Test)
Testdaten so generieren, daß das Programm möglichst vollständig durchlaufen wird.

Abdeckungsbedingungen:

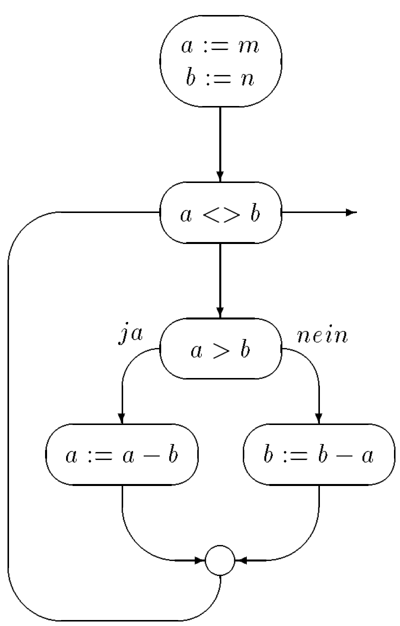
$C0$	Jede Anweisung mindestens einmal durchlaufen
$C1$	Jede Anweisung einmal und jede Bedingung einmal erfüllt und einmal unerfüllt durchlaufen (d.h. jede Kante mindestens einmal durchlaufen)
Ck	$C1$ und jede Schleife wird k–mal durchlaufen
$C\infty$	jeden Pfad im Flußdiagramm einmal durchlaufen

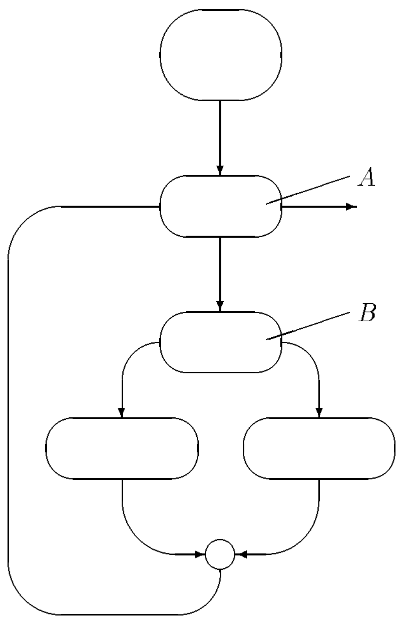
Ein vollständiges Austesten ist nicht erreichbar, da meist eine unendliche Anzahl von Pfaden existiert.

Beispiel ($C1$–Test für größten gemeinsamen Teiler)

$a := m;$
$b := n;$
while $a \neq b$ **do**
  if $a > b$ **then**
    $a := a - b;$
  else
    $b := b - a;$
  fi
end

dazugehöriger Flußgraph





$$\left[\begin{array}{l} a := m \\ b := n \end{array}\right.$$

$A1: \quad m <> n$

$B1: \quad m > n$

$$\left[\, b := n\right.$$

$A2: \quad (m-n) <> n$

$B2: \quad (m-n) \leq n$

$$\left[\, b := n - (m-n) = 2n - m\right.$$

$A3: \quad m - n = 2n - m$

$\quad\multimap 2m = 3n$

Testdaten werden durch Lösung des Ungleichungssystems

$A1 \wedge A2 \wedge A3 \wedge B1 \wedge B2$

geliefert (i.a. unentscheidbar).

Hier: $m = 3$ und $n = 2$

Vorgehensweise

1. Bestimmung einer Menge von Pfaden, die die Abdeckungsbedingung erfüllt.
2. Feststellung der zugehörigen Pfadbedingungen, d.h. der Konjunktion der Teilbedingung pro Test
3. Suchen einer Erfüllung für Pfadbedingungen (häufig Ungleichungssysteme)

Anmerkung

Die Bedingungen eines Pfades erhält man durch symbolische Ausführung.

13.3 Inspizieren

Skizze

Inspizieren

Schreibtischtest
(durch Programmierer)

structured Walkthrough
(durch Team)

Ablauf eines "Structured Walkthrough" (beispielhaft)

1. Einladung (von Entwickler ausgehend: 4-6 Leute, Tester, Entwerfer, Anwender; Management nicht beteiligt)
2. Vorbereitungsgespräche
3. Vorbereitungszeitraum (4-6 Tage)
4. Walkthrough–Gespräche (Moderator führt Vorsitz, 1-2 Stunden, nur Probleme identifizieren, nicht lösen)
5. Nachbereitung

Kapitel 14

Programmieren im Kleinen

(= Programming in the small)

14.1 Schrittweise Verfeinern

Literatur

N. Wirth: *Program Development by stepwise Refinement*,
CACM 14 (1971), S.221-227 oder 26 (1983) S.63-67

Anmerkung

Beim Programmieren im Kleinen ist eine reine Top–Down–Vorgehensweise möglich, weil die Basismaschine (z.B. Pascal,C) dem Programmierer vertraut ist.

Stichworte

- Zerlegung in 3–6 Teile
- Verfeinerungsbezeichnungen können als Kommentare dienen
- enger Zusammenhang zu Pseudocode
- etwa gleichgewichtige Aufteilung

14.2 Strukturierte Programmierung

Feststellung

Strukturierte Programme sind regulär, d.h. sie werden durch

- Sequenzbildung
- Alternativbildung
- Iterativbildung

zusammengesetzt.
Hinzu kommt die Möglichkeit der Abstraktion.

Satz von Böhm–Jacopini

Zu jedem **goto-Programm** (=Programm, indem nur goto-Anweisungen zur Ablaufkontrolle verwendet werden) gibt es ein funktional äquivalent strukturiertes Programm.

Beweisskizze

goto–Programm (stmt = statement):

$l_1 : stmt\_1;$
$l_2 : stmt\_2;$
$\vdots$
$l_k : stmt\_k;$

wobei $stmt\_i$ folgendes darstellen kann:

1. atomare Aktion A
2. **goto** l_j
3. **if** b **then goto** l_j
   else goto l_l

Dieses **goto**–Programm ist äquivalent zu folgendem **while**–Programm:

$z := 1;$
while $1 \leq z$ **and** $z \leq k$ **do**
  case z **of**
    $1 : stmt\_1;$

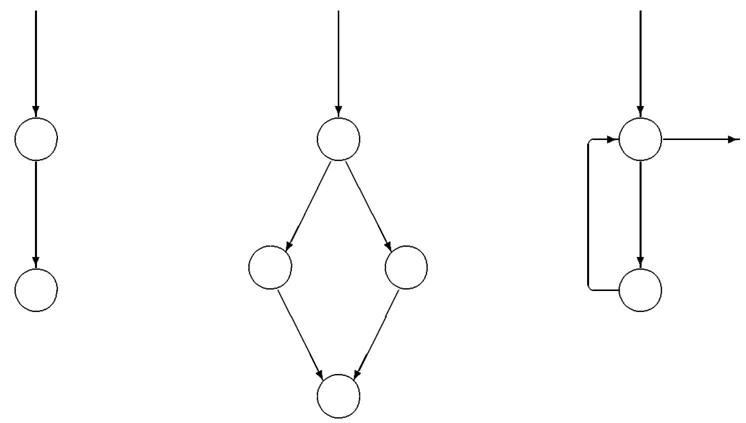
$2 : stmt\_2;$
$\vdots$
$k : stmt\_k;$
end
end

wobei $stmt\_i$:

1. A; $z := z + 1;$
2. $z := j;$
3. **if** b **then** $z := j$
   else goto $z := l$

Anmerkung
Nicht jede Programmstruktur ist durch Sequenzen, Alternativen und Iterationsbildungen nachbildbar.

strukturierte Programmstrukturen:



nicht strukturierte Programmstruktur:

14.3 goto–Debatte

Feststellung
Durch Verwendung von goto–Anweisungen können leicht unübersichliche Programme entstehen (Spaghetti–Programmierung).

möglicher Ausweg: Verbot des **goto**!

Feststellung

Die Strukturen der strukturierten Programmierung sind nicht immer natürlich.

Literatur

D. E. Knuth: *Structured Programming with goto*,
ACM Computing Surveys 6 (1974,7), S.261-302

Beispiel (Suche im Array)

$i := 1;$
while $i \leq n$ **and** $a[i] <> x$ **do**
  $i := i + 1;$
end;
if $i > n$ **then** *"nicht gefunden"*
  else *"gefunden"*;

Erreichen möchte man:

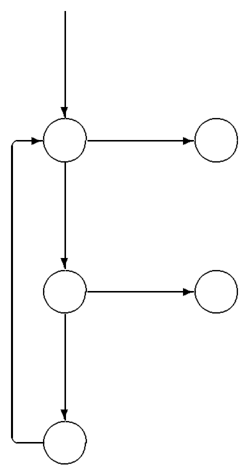
Feststellung

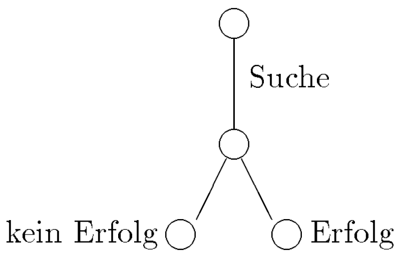
Neuere Programmiersprachen enthalten eingeschränkte Sprungmöglichkeiten. Man sollte deren Verwendung deutlich kennzeichen.

$\boxed{\text{Fest}}$

Ansätze

Zahn–Konstrukt: Schleife mit 2 Ausgängen.




Suche
kein Erfolg
Erfolg

Teil V

Verschiedenes

Kapitel 15

Restliche Phasen

15.1 Integrieren

Alternativen

- bottom up
- top down

in jedem Fall inkrementell

Bottom–Up–Integration
Es sind **Testrahmen** (Testtreiber) zu schreiben, die die entstandenen Bausteine zu Testzwecken kontrolliert aufrufen.

Top–Down–Integration
Es sind Stummel (Dummy–Bausteine, stubs) zu schreiben, die als Platzhalter für vollständige Bausteine dienen.

Feststellung
Erst in dieser Phase sollten Optimierungen stattfinden!

Jackson zur Optimierung:

1. Don't do it!
2. Don't do it yet!

Feststellung
Man sollte zuerst dort optimieren, wo es sich **nachweislich** lohnt (z.B. Programmprofile).

Laut Knuth gilt: Programme verbringen 50% der Zeit in 4% des Codes und 80% im Innersten von Schleifen.

Feststellung

Optimierungen (Tricks) sind besonders gut und deutlich zu dokumentieren.

15.2 Installieren

Feststellung

Die Installation eines Programms in einem Unternehmen erfordert:

- Programmumstellung
- Datenumstellung
- Umstellung von Organisationsabläufen
- Umschulung von Bedienern und Benutzern
- Feinanpassung (Tuning)

Varianten der Installation

- Direktumstellung
- Parallelbetrieb (mit den alten Systemen)

Feststellung

Die Installationsphase wird durch die **Abnahme** (rechtlich wirksam) abgeschlossen.

Abnahmebedingungen sind förmlich festzulegen, z.B.:

- fehlerfreier Betrieb für einen festgelegten Zeitraum
- fehlerfreie Bearbeitung von Testdaten

Tuning

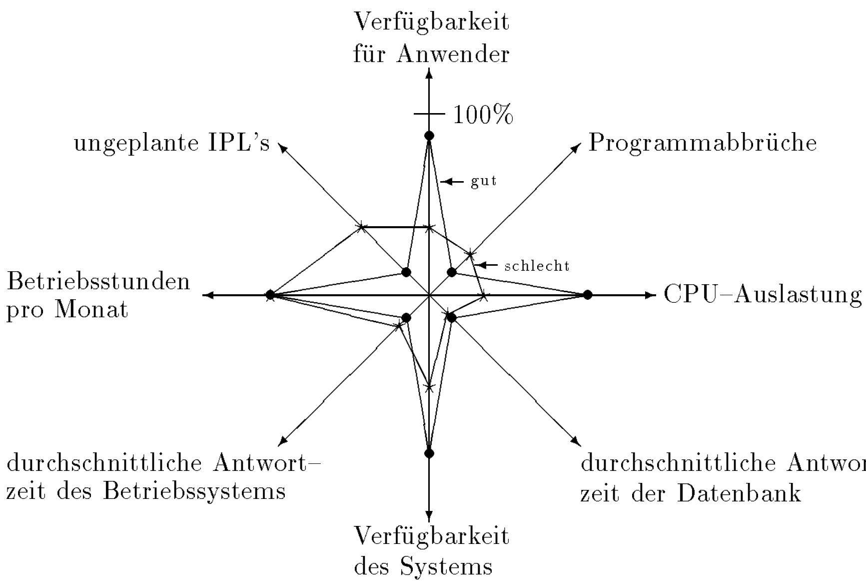
Optimierung in der Zielumgebung durch Anpassung der Parameter, die die Ausführungseigenschaften betreffen.

- Anpassung von Blockgrößen
- Segmentierung von Programmen
- Zuweisung von Puffern an Files/Puffergrößen
- Anpassung von Warteschlangengrößen
- Anpassung der Benutzerschnittstelle (Customizing)

Leistungsmessung

- Zählen von Ereignissen
- Protokolldateien
- Stichprobenverfahren
- Benchmarking = Anwendung des Systems auf "Standarddaten", um die dessen Leistung zu testen.
  - natürlicher Benchmark (d.h. Daten des Anwenders)
  - synthetischer Benchmark (d.h. Daten, die gezielt eine Eigenschaft testen)

Darstellungsmittel (Kiviat–Graph)



15.3 Betreuen

Feststellung

Betreuen besteht aus

- betreiben
- warten (fixing)
- pflegen (changing)

Warten

Ist die **teuerste** Tätigkeit, erfordert die besten Leute und bietet die geringste Befriedigung.

Konsequenz: Hoher Einsatz guter Methoden in den frühen Phasen.

Feststellung

Warten besteht aus

- Fehlererkennung und
- Fehlerbeseitigung

Fehlererkennung und -beseitigung werden unter dem Begriff **Debugging** zusammengefat.

Man muß die **eigentliche Fehlerursache** suchen und nicht nur die Symptome beseitigen.

Hilfsmittel zur Fehlersuche

- Debug–Programme (Debug–Optionen des Übersetzers) möglichst interaktiv
- Tracing, Trace Back
- Speicherauszüge (Dump) mit Interpretationshilfen

Änderungen

Änderungen werden durch Inkremente (Deltas) beschrieben.

Vorteile:

- Klarheit
- Rückgängigmachen wird einfacher
- Übertragen auf andere Installationen wird einfacher
- Ermöglichung der Wiederholung (lokale Modifikation)
- Auswahl einzelner Änderungen

Gelegentlich werden neue Versionen ausgeliefert, die alle Änderungen enthalten.

Feststellung

Neue Versionen werden durch **Regressionstests** überprüft.
Evtl. werden die neuen Versionen bei ausgewählten Benutzern vorab (β–Test) eingesetzt.

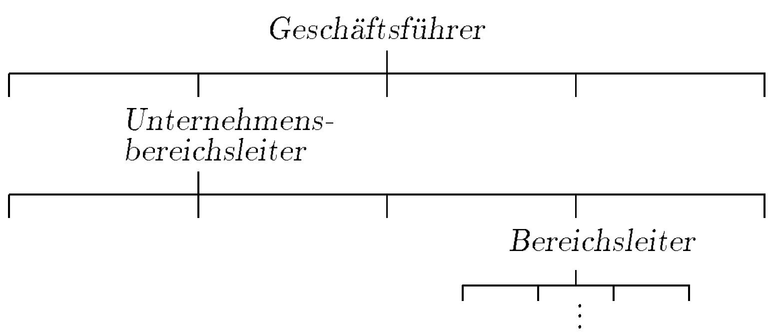
Fest

Kapitel 16

Management und Organisation

16.1 Projektorganisation

Stab–/Linienorganisation



Ebenen (z.B.):
Geschäftsführung
Unternehmensbereich
Bereich
Ressort
Hauptabteilung
Abteilung
Gruppe (zu ca. 6 Leuten)

Die Leiter **erstatten Bericht** an ihre Vorgesetzten. Sie sind in der **Linie**. Sie haben mehrere Mitarbeiter in ihrem **Stab**.

Matrixorganisation

Es gibt Projekte mit eigenen Projektleitern, in denen Personen verschiedener Abteilungen beteiligt sind:

	Abt. 1	Abt. 2	Abt. 3
Proj. 1	**X**, Y		Z
Proj. 2	B, C	**A**	
Proj. 3	D	E	**F**

16.2 Teamorganisation

Chief Programmer Team

Zusammenfassung von 5 – 10 Personen um einen erfahrenen Programmierer, der die Verantwortung und die wichtigsten Aufgaben übernimmt.

Aufbau

1. Chief Programmer
2. Backup Programmer
3. Librarian
4. Team Members

Adaptives Team

- Gerald Weinberg: *The Psychology of Computer Programming*
  van Nostrand, 1971
- Zusammenfassung von 4 – 6 gleichberechtigten Personen
  - hohe Kommunikation
  - gegenseitige Hilfe
  - ego – less programming
  - ähnliche Leistungsfähigkeit
  - klare Aufgabenstellung
- These: Das Team organisiert sich selbst.

16.3 Produktivität

Definition

Produktivität $= \frac{\text{Programmierleistung}}{\text{Zeit}}$ $\quad [\frac{\text{loc}}{\text{Jahr}}]$

loc=lines of code (netto, d.h. ohne Kommentare u. Leerzeilen)

Aufwand = Personenzahl $*$ Zeiteinheit [Personenjahr]

Größenordnungen

Die Produktivität einer Person beträgt etwa 15 – 50 Zeilen pro Tag (im wesentlichen unabhängig von der Programmiersprache)

Konsequenz

Mit Sprachen höheren Niveaus lassen sich daher höhere Leistungen erzielen.

16.4 Zeitliche Planung

siehe Netzplantechnik (DIN 69900)
(WS 90/91 : Effiziente Graphenalgorithmen)

16.5 Dokumentation

Feststellung

- Während des Erstellungsprozesses fallen zahlreiche Einzeldokumente an.
- Zusätzlich (für Wartung und Pflege) ist erforderlich
  - Implementationshandbuch
  - Benutzungshandbuch

Benutzungshandbuch

- beschreibt Benutzungsschnittstelle z.B.:
  - Benutzungsvoraussetzungen (Daten, Namen, ...)
  - Ablauf des Dialogs
  - Eingabesprache
  - funktionales Verhalten
  - Fehlermeldungen (mit Bedeutungen)
    (Art des Fehlers, erwartete Eingaben, geschehene Reaktionen, durchzuführende Maßnahmen)
- evtl. aufgeteilt in:
  - Einführungshandbuch
  - Nachschlagehandbuch
  - Operateurhandbuch
  - Referenzhandbuch

Implementationshandbuch

- enthält Informationen für Wartung und Pflege z.B.:
  - Aufzählung der Module (z.B.Prozeduren)
  - Modulstrukturen (Enthaltensrelation)
  - Benutzte Relation
  - Spezifikationen (Beschreibung der Prozeduren)
  - Globale Größen
  - Typbeschreibungen (Datenstrukturbeschreibungen)
- bei all diesen Dingen, soweit möglich, **graphisch darstellen**.

Literaturverzeichnis

[1] S. Alagic, M. A. Arbib: *The Design of Well Structured and Correct Programs*, Springer, 1978.

[2] Balzert: *Die Entwicklung von Software Systemen*, BI, Mannheim 1982.

[3] F. L. Bauer (ed.): *Software Engineering: An advanced course*, Springer, Berlin 1973.

[4] T. de Marco: *Structured Analysis and System Specification*, Englewood Cliffs, Prentice Hall 1978.

[5] E. Ehrig, B. Mahr: *Fundamentals of algebraic specification*, Springer, EATCS-Serie, Bd. 6, 1985.

[6] C. Gane, T. Sarson: *Structured System Analysis: Tools and Techniques*, Englewood Cliffs, New York. Prentice Hall, 1979.

[7] Gewald, Haake, Pfadler: *Software Engineering*, R. Oldenbourg, München 1979.

[8] D. Gries: *The Design of Programming*, Springer, 1983.

[9] S. L. Hantler, J. C. King: *An Intoduction to Proving the Correctness of Programs*, Computing Surveys 8 (3, 1976), S.331–353.

[10] J. C. Huang: *An Approach to Program testing*, ACM Computing Survey 7 (1975,3), S. 113-119.

[11] C. B. Jones: *Systematic Software Development Using VDM"*, Englewood Cliffs, Prentice Hall 1986.

[12] R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch: *Einführung in Software Engineering*, de Gruyter, Berlin 1979.

[13] D. E. Knuth: *Structured Programming with goto*, ACM Computing Survey 6 (1974,7), S. 261-302.

[14] L. Lamport: LaTeX - *A Document Preparation System*, Addison-Wesley, Amsterdam 1986.

[15] D. L. Parnas: *A technique for Software module spezification with examples.* Comm. ACM 15 (1972), S. 330-336.

[16] D. T. Ross: *Structured Analysis: a language to communicate ideas*, IEEE-SE, 3 (1977,1), S.16-34

[17] J. F. Stay: *HIPO and integrated program design*, IBM Systems Journal, 1976

[18] Stetter: *Software Technologie*, BI, Mannheim 1982.

[19] Gerald Weinberg: *The Psychology of Computer Programming* van Nostrand, 1971

[20] N. Wirth: *Program Development by stepwise Refinement*, CACM 14 (1971), S. 221-227 oder 26 (1983), S. 63-67.

Index